

```
for (i=0 ; i<inumverts ; i++)
```

# Let Run your Neurons

```
i = (lindex > 0)

    r_pedge = &edges[lindex]
    vec = r_pcurrentvertbase[r_pedge->v[0]] position

}
else
{
    r_pedge = &edges[-lindex]
    vec = r_pcurrentvertbase[r_pedge->v[1]] position
}

s = DotProduct (vec, fa->texinfo->vecs[0][1]) + fa->texinfo->vecs[0][3]
s /= fa->texinfo->texture->width

t = DotProduct (vec, fa->texinfo->vecs[1][1]) + fa->texinfo->vecs[1][3]
t /= fa->texinfo->texture->height

VectorCopy (vec, poly->verts[1])
poly->verts[1][3] = s
poly->verts[1][4] = t

s = DotProduct (vec, fa->texinfo->vecs[0][0]) + fa->texinfo->vecs[0][3]
s /= fa->texinfo->texture->width
```

**Team  
LRN**

# Graphics Programming with Direct X 9

## Part I

(12 Week Lesson Plan)

## Lesson 1: 3D Graphics Fundamentals

**Textbook:** Chapter One (pgs. 2 – 32)

### Goals:

We begin the course by introducing the student to the fundamental mathematics necessary when developing 3D games. Essentially we will be talking about how 3D objects in games are represented as polygonal geometric models and how those models are ultimately drawn. It is especially important that students are familiar with the mathematics of the transformation pipeline since it plays an important role in getting this 3D geometry into a displayable 2D format. In that regard we will look at the entire geometry transformation pipeline from model space all the way through to screen space and discuss the various operations that are necessary to make this happen. This will include discussion of transformations such as scaling, rotations, and translation, as well as the conceptual idea of moving from one coordinate space to another and remapping clip space coordinates to final screen space pixel positions.

### Key Topics:

- Geometric Modeling
  - 2D/3D Coordinate Systems
  - Meshes
    - Vertices
    - Winding Order
- The Transformation Pipeline
  - Translation
  - Rotation
  - Viewing Transformations
  - Perspective Projection
  - Screen Space Mapping

**Projects:** NONE

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 5 - 7

## Lesson 2: 3D Graphics Fundamentals II

**Textbook:** Chapter One (pgs. 32 – 92)

### Goals:

Picking up where the last lesson left off, we will now look at the specific mathematics operations and data types that we will use throughout the course to affect the goals discussed previously regarding the transformation pipeline. We will examine three fundamental mathematical entities: vectors, planes and matrices and look at the role of each in the transformation pipeline as well as discussing other common uses. Core operations such as the dot and cross product, normalization and matrix and vector multiplication will also be discussed in detail. We then look at the D3DX equivalent data types and functions that we can use to carry out the operations discussed. Finally we will conclude with a detailed analysis of the perspective projection operation and see how the matrix is constructed and how arbitrary fields of view can be created to model different camera settings.

### Key Topics:

- 3D Mathematics Primer
  - Vectors
    - Magnitude
    - Addition/ Subtraction
    - Scalar Multiplication
    - Normalization
    - Cross Product
    - Dot Product
  - Planes
  - Matrices
    - Matrix/Matrix Multiplication
    - Vector/Matrix Multiplication
    - 3D Rotation Matrices
    - Identity Matrices
    - Scaling and Shearing
    - Concatenation
    - Homogenous Coordinates
- D3DX Math
  - Data Types
    - D3DXMATRIX
    - D3DXVECTOR
    - D3DXPLANE
  - Matrix and Transformation Functions
    - D3DXMatrixMultiply
    - D3DXMatrixRotation{XYZ}
    - D3DXMatrixTranslation
    - D3DXMatrixRotationYawPitchRoll

- D3DXVecTransform{...}
- Vector Functions
  - Cross Product
  - Dot Product
  - Magnitude
  - Normalization
- The Transformation Pipeline II
  - The World Matrix
  - The View Matrix
  - The Perspective Projection Matrix
    - Field of View
    - Aspect Ratio

**Projects:**

Lab Project 1.1: Wireframe Renderer

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 8 - 10

## Lesson 3: DirectX Graphics Fundamentals I

**Textbook:** Chapter Two (pgs. 94 – 132)

### Goals:

In this lesson our goal will be to start to get an overview of the DirectX Graphics pipeline and see how the different pieces relate to what we have already learned. A brief introduction to the COM programming model introduces the lesson as a means for understanding the low level processes involved when working with the DirectX API. Then, our ultimate goal is to be able to properly initialize the DirectX environment and create a rendering device for output. We will do this during this lesson and the next. This will require an understanding of the different resources that are associated with device management including window settings, front and back buffers, depth buffering, and swap chains.

### Key Topics:

- The Component Object Model (COM)
  - Interfaces/IUnknown
  - GUIDS
  - COM and DirectX Graphics
- Initializing DirectX Graphics
- The Direct3D Device
  - Pipeline Overview
  - Device Memory
    - The Front/Back Buffer(s)
    - Swap Chains
  - Window Settings
    - Fullscreen/Windowed Mode
  - Depth Buffers
    - The Z-Buffer / W-Buffer

### Projects:

Lab Project 2.1: DirectX Graphics Initialization

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 8 – 10

## Lesson 4: DirectX Graphics Fundamentals II

**Textbook:** Chapter Two (pgs. 132 – 155)

### **Goals:**

Continuing our environment setup discussion, in this lesson our goal will be to create a rendering device for graphics output. Before we explore setting up the device, we will look at the various surface formats that we must understand for management of depth and color buffers. We will conclude the lesson with a look at configuring presentation parameters for device setup and then talk about how to write code to handle lost devices.

### **Key Topics:**

- Surface Formats
  - Adapter Formats
  - Frame Buffer Formats
- Device Creation
  - Presentation Parameters
  - Lost Devices

### **Projects:**

Lab Project 2.2: Device Enumeration

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 8 - 10

## Lesson 5: Primitive Rendering I

**Textbook:** Chapter Two (pgs. 156 – 191)

### Goals:

Now that we have a rendering device properly configured, we are ready to begin drawing 3D objects using DirectX Graphics. In this lesson we will examine some of the important device settings (states) that will be necessary to make this happen. We will see how to render 3D objects as wireframe or solid objects and also talk about how to affect various forms of shading. Our discussion will also include flexible vertex formats, triangle data, and the DrawPrimitive function call. Once these preliminary topics are out of the way we will look at the core device render states that are used when drawing – depth buffering, lighting and shading, back face culling, etc. We will also talk about transformation states and how to pass the matrices we learned about in prior lessons up to the device for use in the transformation pipeline. We will conclude the lesson with discussion of scene rendering and presentation (clearing the buffers, beginning and ending the scene and presenting the results to the viewer).

### Key Topics:

- Primitive Rendering
  - Fill Modes
  - Shading Modes
  - Vertex Data and the FVF
  - DrawPrimitiveUP
- Device States
  - Render States
    - Z – Buffering
    - Lighting/Shading/Dithering
    - Backface Culling
  - Transformation States
    - World/View/Projection Matrices
- Scene Rendering
  - Frame/Depth Buffer Clearing
  - Begin/End Scene
  - Presenting the Frame

### Projects:

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 5 – 7

## Lesson 6: Primitive Rendering II

**Textbook:** Chapter Three (pgs. 194 – 235)

### Goals:

In this lesson we will begin to examine more optimal rendering strategies in DirectX. Primarily the goal is to get the student comfortable with creating, filling and drawing with both vertex and index buffers. This means that we will look at both indexed and non-indexed mesh rendering for both static geometry and dynamic (animated) geometry. To that end it will be important to understand the various device memory pools that are available for our use and see which ones are appropriate for a given job. We will conclude the lesson with a discussion of indexed triangle strip generation and see how degenerate triangles play a role in that process.

### Key Topics:

- Device Memory Pools and Resources
  - Video/AGP/System Memory
- Vertex Buffers
  - Creating Vertex Buffers
  - Vertex Buffer Memory Pools
  - Vertex Buffer Performance
  - Filling Vertex Buffers
  - Vertex Stream Sources
  - DrawPrimitive
- Index Buffers
  - Creating Index Buffers
  - DrawIndexedPrimitive/DrawIndexedPrimitiveUP
  - Indexed Triangle Strips/Degenerate Triangles

### Projects:

Lab Project 3.1: Static Vertex Buffers  
Lab Project 3.2: Simple Terrain Renderer  
Lab Project 3.3: Dynamic Vertex Buffers

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 8 – 10

## Mid-Term Examination

The midterm examination in this course will consist of 40 multiple-choice and true/false questions pulled from the first three textbook chapters. Students are encouraged to use the lecture presentation slides as a means for reviewing the key material prior to the examination. The exam should take no more than 1.5 hours to complete. It is worth 35% of student final grade.

## Lesson 7: Camera Systems

**Textbook:** Chapter Four (pgs. 238 – 296)

### Goals:

In this lesson we will take a detailed look at the view transformation and its associated matrix and see how it can be used and manipulated to create a number of popular camera system types – first person, third person, and spacecraft. We will also discuss how to manage rendering viewports and see how the viewport matrix plays a role in this process. Once we have created a system for managing different cameras from a rendering perspective, we will examine how to use the camera clipping planes to optimize scene rendering. This will include writing code to extract these planes for the purposes of testing object bounding volumes to determine whether or not the geometry is actually visible given the current camera position and orientation. Objects that are not visible will not need to be rendered, thus allowing us to speed up our application.

### Key Topics:

- The View Matrix
  - Vectors, Matrices, and Planes
    - The View Space Planes
    - The View Space Transformation
    - The Inverse Translation Vector
- Viewports
  - The Viewport Matrix
  - Viewport Aspect Ratios
- Camera Systems
  - Vector Regeneration
  - First Person Cameras
  - Third Person Cameras
- The View Frustum
  - Camera Space Frustum Plane Extraction
  - World Space Frustum Plane Extraction
  - Frustum Culling an AABB

**Projects:** NONE

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 8 - 10

## Lesson 8: Lighting

**Textbook:** Chapter Five (pgs. 298 – 344)

### Goals:

In this lesson we will introduce the lighting model used in the fixed function DirectX Graphics pipeline. We begin with an overview of the different types of lighting (ambient, diffuse, specular, and emissive) that are modeled in real-time games. We will also talk about the specific light types (point/spot/directional) and how to setup their properties and configure the lighting pipeline to use them. This will include some discussion of the role of vertex normals and how to calculate them when necessary. In conjunction with lighting we must also discuss the concept of materials as they define how surfaces interact with the lights in the environment. We will see how to create them and set their properties to produce different results. We will conclude this lesson with a brief discussion of the advantages and disadvantages of using the fixed function vertex lighting pipeline as a means for setting the stage for more advanced lighting models that will be introduced in Part II of this course series.

### Key Topics:

- Lighting Models
  - Indirect Lighting
    - Emissive/Ambient Illumination
  - Direct Lighting
    - Diffuse/Specular Light
- The Lighting Pipeline
  - Enabling DirectX Graphics Lighting
    - Enabling Specular Highlights
    - Enabling Global Ambient Lighting
  - Lighting Vertex Formats and Normals
  - Setting Lights and Light Limits
- Light Types
  - Point/Spot/Directional
- Materials
  - Colors, Specular and Power
  - Material Sources

### Projects:

Lab Project 5.1: Dynamic Lights

Lab Project 5.2: Scene Lighting

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 10 - 12

## Lesson 9: Texture Mapping I

**Textbook:** Chapter Six (pgs. 346 – 398)

### Goals:

In this lesson students will be introduced to texture mapping as a means for adding detail and realism to the lit models we studied in the last lesson. We begin by looking at what textures are and how they are defined in memory. This will lead to a preliminary discussion of mip-maps in terms of memory format and consumption. Then we will look at the various options at our disposal for loading texture maps from disk or memory using the D3DX utility library. Discussion of how to set a texture for rendering and the relationship between texture coordinates and addressing modes will follow. Finally we will talk about the problem of aliasing and other common artifacts and how to use various filters to improve the quality of our visual output.

### Key Topics:

- Texture Memory Pools
  - Texture Formats
    - Validating Texture Formats
    - Surface Formats
- MIP Maps
- Loading Textures
- Setting Textures
- Texture Coordinates
- Sampler States
  - Texture Addressing Modes
    - Wrapping/Mirroring/Bordering/Clamping/MirrorOnce
  - Texture Coordinate Wrapping
  - Texture Filtering
    - Magnification/Minification
    - Point/Bilinear/Trilinear/Anisotropic

### Projects:

Lab Project 6.1: Simple Texturing

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 8 - 10

## Lesson 10: Texture Mapping II

**Textbook:** Chapter Six (pgs. 399 – 449)

### Goals:

This lesson will conclude our introduction to texture mapping (advanced texturing will be discussed in Part II of this series). We will begin by examining the texture pipeline and how to configure the various stages for both single and multi-texturing operations. Then we will take a step back and examine texture compression and the various compressed formats in detail as a means for reducing our memory requirements. Once done, we will return to looking at the texture pipeline and see how we can use transformation matrices to animate texture coordinates in real time to produce simple but interesting effects. Finally, we will conclude with a detailed look at the DirectX specific texture and surface types and their associated utility functions.

### Key Topics:

- Texture Stages
  - Texture Color
  - Texture Stage States
- Multi-Texturing and Color Blending
- Compressed Textures
  - Compressed Texture Formats
    - Pre-Multiplied Alpha
  - Texture Compression Interpolation
  - Compressed Data Blocks – Color/Alpha Data Layout
- Texture Coordinate Transformation
- The IDirect3DTexture Interface
- The IDirect3DSurface Interface
- D3DX Texture Functions

### Projects:

Lab Project 6.2: Terrain Detail Texturing

Lab Project 6.3: Scene Texturing

Lab Project 6.4: GDI and Textures

Lab Project 6.5: Offscreen Surfaces

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 10 – 12

## Lesson 11: Alpha Blending

**Textbook:** Chapter Seven (pgs. 451 – 505)

### Goals:

In this lesson we will examine an important visual effect in games: transparency. Transparency requires that students understand the concept of alpha blending and as such we will talk about various places alpha data can be stored (vertices, materials, textures, etc.) and how what various limitations and benefits are associated with this choice. We will then explore the alpha blending equation itself and look at how to configure the transformation and texture stage pipelines to carry out the operations we desire. We will also examine alpha testing and alpha surfaces for the purposes of doing certain types of special rendering that ignores specific pixels. We will conclude our alpha blending discussion with a look at the all important notion of front to back sorting and rendering, examining various algorithms that we can use to do this. Finally, we will wrap up the lesson with an examination of adding fog to our rendering pipeline. This will include both vertex and pixel fog, how to set the color for blending, and the three different formulas available to us (linear/exponential/exponential squared) for producing different fogging results.

### Key Topics:

- Alpha Components
  - Vertex Alpha – Pre-Lit/Unlit Vertices
  - Material Alpha
  - Constant Alpha + Per-Stage Constant Alpha
  - Texture Alpha
- The Texture Stage Alpha Pipeline
- Frame Buffer Alpha Blending
- Transparent Polygon Sorting
  - Sorting Algorithms and Criteria
    - Bubble Sort/Quick Sort/Hash Table Sort
- Alpha Surfaces
- Alpha Testing
- Fog
  - Enabling Fog and Setting the Fog Color
  - Vertex/Pixel Fog
  - Fog Factor Formulas
    - Linear/Exponential/Exponential Squared

**Projects:**

Lab Project 7.1: Vertex Alpha  
Lab Project 7.2: Alpha Testing  
Lab Project 7.3: Alpha Sorting  
Lab Project 7.4: Texture Splatting

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 10 - 12

## Lesson 12: Exam Preparation and Course Review

**Textbook:** NONE

**Goals:**

In this final lesson we will leave the student free to prepare for and take their final examination. Multiple office hours will be held for student questions and answers.

**Key Topics:** NONE

**Projects:** NONE

**Exams/Quizzes:** NONE

**Recommended Study Time (hours):** 15 - 20

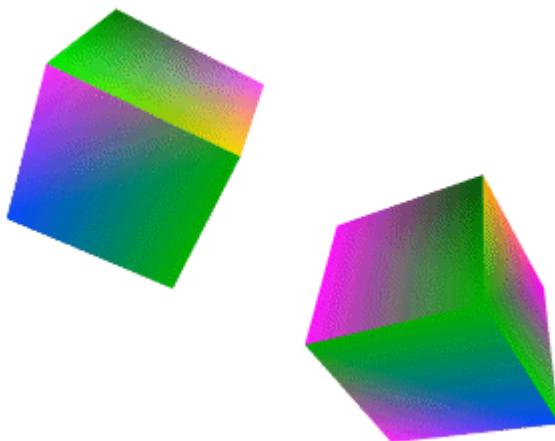
## Final Examination

The final examination in this course will consist of 50 multiple-choice and true/false questions pulled from all of the textbook chapters. Students are encouraged to use the lecture presentation slides as a means for reviewing the key material prior to the examination. The exam should take no more than two hours to complete. It is worth 65% of student final grade.

# Chapter One:

# 3D Graphics Fundamentals

---



© 2003, eInstitute, Inc.

You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.



## Table of Contents

<b>Geometric Modeling</b> .....	4
Geometry in Two Dimensions .....	6
Geometry in Three Dimensions .....	8
Creating Our First Mesh .....	11
Vertices .....	13
Winding Order .....	13
Transformations .....	16
Perspective Projection .....	27
Screen Space Mapping .....	32
Draw Primitive Pseudocode .....	33
<b>3D Mathematics Primer</b> .....	36
Vectors .....	36
Vector Magnitude .....	37
Vector Addition and Subtraction .....	38
Vector Scalar Multiplication .....	41
Unit Vectors .....	42
The Cross Product .....	44
Normals .....	45
The Dot Product .....	46
Planes .....	49
Matrices .....	54
Matrix/Matrix Multiplication .....	55
Vector/Matrix Multiplication .....	58
3D Rotation Matrices .....	61
Identity Matrices .....	61
Scaling and Shearing Matrices .....	62
Matrix Concatenation .....	63
Homogeneous Coordinates .....	63
Quaternions .....	69
<b>D3DX Math</b> .....	70
D3DXMATRIX .....	70
D3DXVECTOR3 .....	71
D3DXPLANE .....	71
D3DXQUATERNION .....	72
D3DX Functions .....	73
<b>The Transformation Pipeline</b> .....	77
The World Matrix .....	77
The View Matrix .....	81
The Perspective Projection Matrix .....	84
Arbitrary FOV .....	87
Aspect Ratio .....	93
Conclusion .....	97



## The Virtual World

Games that use 3D graphics often have several source code modules to handle tasks such as:

1. user input
2. resource management
3. loading and rendering graphics
4. interpreting and executing scripts
5. playing sampled sound effects
6. artificial intelligence

These source code modules, along with others, collectively form what is referred to as the *game engine*. One of the key modules of any 3D game engine, and the module that this course will be focusing on, is the rendering engine (or *renderer*). The job of the rendering engine is to take a mathematical three dimensional representation of a virtual game world and present it as a two dimensional image on the monitor screen.

Before the days of graphics APIs like DirectX and OpenGL, developers did not have the luxury of being handed a fully functional collection of code that would, at least to a certain extent, shield them from the mathematics of 3D graphics programming. Developers needed a thorough understanding of designing and coding a robust 3D graphics pipeline. Those who have worked on such projects previously have little trouble starting to use APIs like DirectX Graphics. Most of the functionality is not only familiar, but is probably something they had to implement by hand at an earlier time.

Unfortunately, novice game developers have a tendency to jump straight into using 3D APIs without any basic knowledge of what the API is doing behind the scenes. Not surprisingly, this often leads to unexpected results and long debugging sessions. 3D graphics programming involves a good deal of mathematics. Without a firm grasp of these critical concepts you will never fully understand nor likely have the ability to exploit the full potential of the popular APIs.

This is a considerable stumbling block for students just getting started with 3D graphics programming. So in this lesson we will examine some basic 3D programming concepts as well as some key mathematics to help create a foundation for later lessons. We will have only one Lab Project in this lesson. In it, we will build a rudimentary software rendering application so that you can see the mathematics of 3D graphics firsthand.

Those of you who already have a thorough understanding of the 3D pipeline may wish to take this opportunity to refresh your memory or simply move on to another lesson.



## Geometric Modeling

During the process of developing a three-dimensional game, artists and modelers will create 3D objects using a modeling package like 3D Studio MAX™, Maya™, or even GILEST™. These models will be used to populate the virtual game world. If you wanted to design a game that took place along the street where you live, an artist would likely create separate 3D models for each house, a street model and sidewalk model, and a collection of various models to represent such things as lamp posts, automobiles or even people. These would all be loaded into the game software and inserted into a virtual representation of the world where each model is given a specific position and orientation.

Non-complex models can also be created programmatically using basic mathematics techniques. This is the method we will use during our initial examples. It will provide you with a better understanding of how 3D models are created and represented in memory and how to perform operations on them. While this approach is adequate for creating simple models such as cubes and spheres, creating complex 3D models in this way would be extraordinarily difficult and unwise.

**Note:** 3D models are often referred to by many different names. The most common are: objects, models and meshes. In keeping with current standard terminology we will refer to a 3D model as a *mesh*. This means that whenever we use the word *mesh* we are really referring to an arbitrary 3D model that could be anything from a simple cube to a complex alien mother ship.

A **mesh** is a collection of polygons that are joined together to create the outer hull of the object being defined. Each **polygon** in the mesh (often referred to as a **face**), is created by connecting a collection of points defined in three dimensional space with a series of line segments. If desired, we can ‘paint’ the surface area defined between these lines with a number of techniques that will be discussed as we progress in this course. For example, data from two dimensional images called **texture maps** can be used to provide the appearance of complex texture and color (Fig 1.1).

The cube without textures    The Mesh of a wooden crate

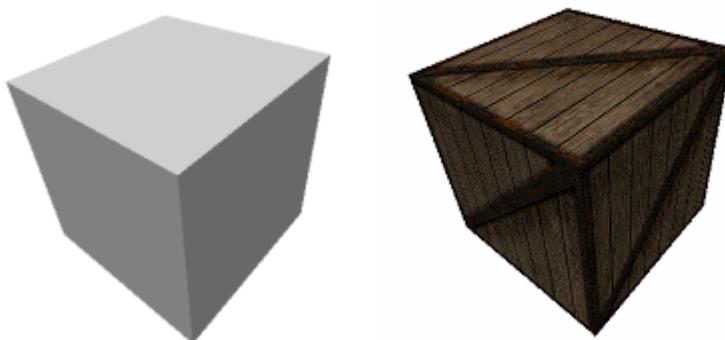
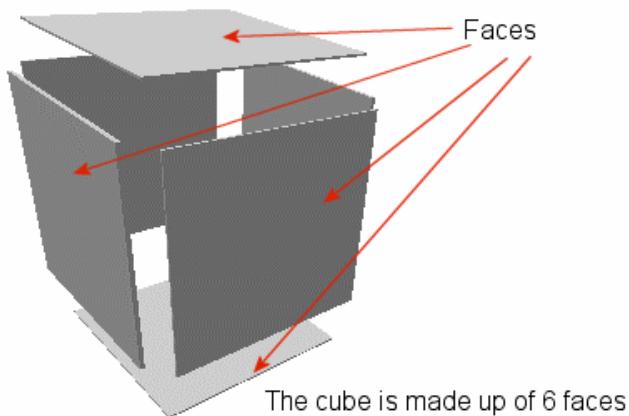


Figure 1.1

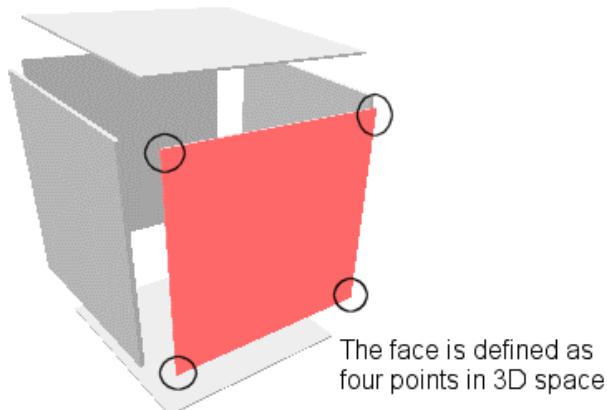
The mesh in Fig 1.1 is constructed using six distinct polygons. It has a top face, a bottom face, a left face, a right face, a front face and a back face. The *front* face is of course determined according to how you are viewing the cube. Because of the fact that the mesh is three dimensional, we can see at most

three of the faces at any one time with the other faces positioned on the opposite side of the cube. Fig 1.2 provides a better view of the six polygons:



**Figure 1.2**

To create a single polygon we will plot a series of points within a 3D coordinate system. The actual shape of the polygon will become clear when we join those points together with lines (Fig 1.3).



**Figure 1.3**

Plotting points within a coordinate system and joining these points together to create more complex shapes is an area of mathematics known as Geometry. We begin by looking at some two dimensional geometry and later move on to three dimensions.

## Geometry in Two Dimensions

A **coordinate system** is a set of one or more *number lines* used to characterize spatial relationships. Each number line is called an axis. The number of axes in a system is equal to the number of dimensions represented by that system. In the case of a two dimensional coordinate system there will typically be a horizontal axis and a vertical axis labeled **X** and **Y** respectively. These axes extend out from the **origin** of the system. The origin is represented by the location (0, 0) in a 2D system. All points to be plotted are specified as offsets along X or Y relative to this origin.

Fig 1.4 shows one example of a 2D coordinate system that we will be discussing again later in the lesson. It is called the *screen coordinate system* and it is used to define pixel locations on our viewing screen. In this case the X axis runs left to right, the Y axis runs from top to bottom, and the origin is located in the upper left corner.

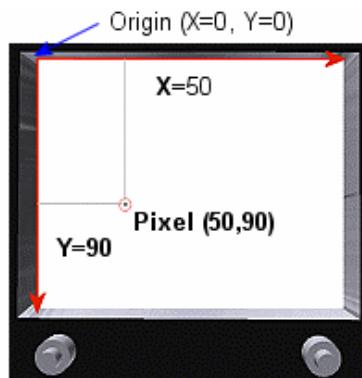


Figure 1.4

Fig 1.5 shows how four points could be plotted using the screen system and how those points could have lines drawn between them in series to create a square geometric shape. The polygon looks very much like one of the polygons in the cube mesh we viewed previously (with the exception that it is viewed two dimensionally rather than three).

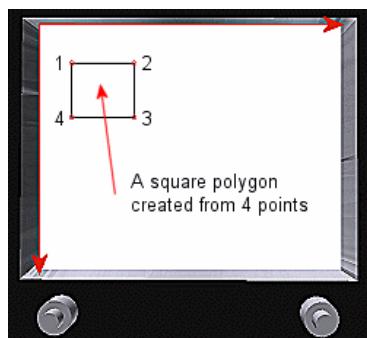


Figure 1.5



We must plot these points in a specific sequence so that the line drawing order is clear. We see that a line should be drawn between point 1 and point 2, and then another between point 2 and point 3 and so on until we have connected all points and are back at point 1.

It is worth stating that this screen coordinate system is not the preferred design for representing most two dimensional concepts. First, the Y values increase as the Y axis moves downward. This is contrary to the common perception that as values increase, they are said to get ‘higher’. Second, the screen system does not account for a large set of values. In a more complete system, the X and Y axes carry on to infinity in both positive *and* negative directions away from the origin (Fig 1.6).

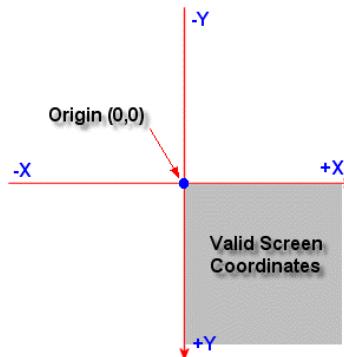


Figure 1.6

Only points within the quadrant of the coordinate system where both X and Y values are positive are considered valid screen coordinates. Coordinates that fall into any of the other three quadrants are simply ignored.

Our preferred system will remedy these two concerns. It will reverse the direction of the Y axis such that increasing values lay out along the upward axis and it will provide the full spectrum of positive and negative values. This system is the more general **(2D) Cartesian coordinate system** that most everyone is familiar with. Fig 1.7 depicts a triangle represented in this standard system:

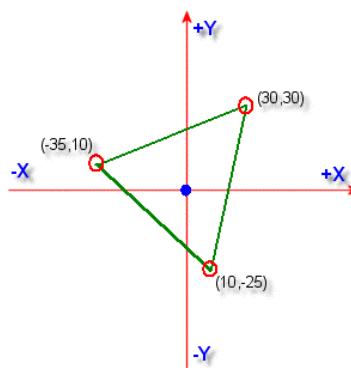


Figure 1.7



## Geometry in Three Dimensions

The 3D system adds a depth dimension (represented by the Z axis) to the 2D system and all axes are perpendicular to one another. In order to plot a point within our 3D coordinate system, we need to use points that have not only an X and a Y offset from the origin, but also a Z offset. This is analogous to real life where objects not only have width and height but depth as well.

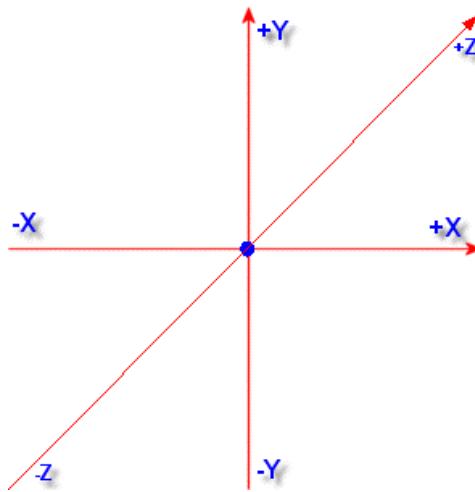


Figure 1.8

Fig 1.8 is somewhat non-intuitive. It actually looks like the Z axis is running diagonally instead of in and out of the page (perpendicular to the X and Y axes). But if we ‘step outside’ of our coordinate system for a moment and imagine viewing it from a slightly rotated and elevated angle, you should more clearly be able to see what the coordinate system looks like (Fig 1.9).

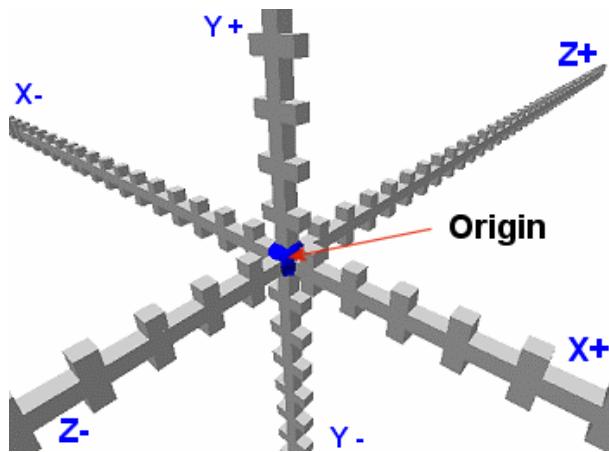


Figure 1.9



There are two versions of the 3D Cartesian coordinate system that are commonly used: the **left-handed system** and the **right-handed system**. The difference between the two is the direction of the +Z axis. In the left-handed coordinate system, the Z axis increases as you look forward (into the page) with negative numbers extending out behind you. The right handed coordinate system flips the Z axis. Some 3D APIs, like OpenGL use a right-handed system. Microsoft's DirectX Graphics uses the left-handed system and we will also use the left-handed system in this course.

## Left and Right Handed Cartesian Coordinate Systems

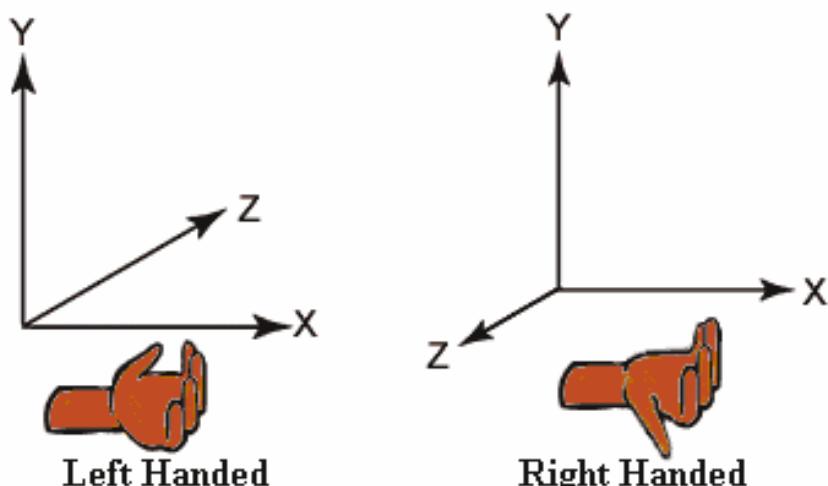


Figure 1.10

**Note:** To remember which direction the Z axis points in a given system:

- 1) Extend your arms in the direction of the positive X axis. (towards the right).
- 2) Turn both hands so that the palms are facing upwards towards the sky.
- 3) Fully extend both thumbs.

The thumbs now tell you the direction of the positive Z axis. On your right hand, the thumb should be pointing behind you, and the thumb on your left hand should be pointing in front of you. This informs us that in a left handed system, positive Z increases in front of us and in a right handed system positive Z increases behind us.

To plot a single point in this coordinate system requires that we specify three offsets from the origin: an X, a Y and a Z value. Fig 1.11 shows us where the 3D point (2, 2, 1) would be located in our left-handed Cartesian coordinate system.

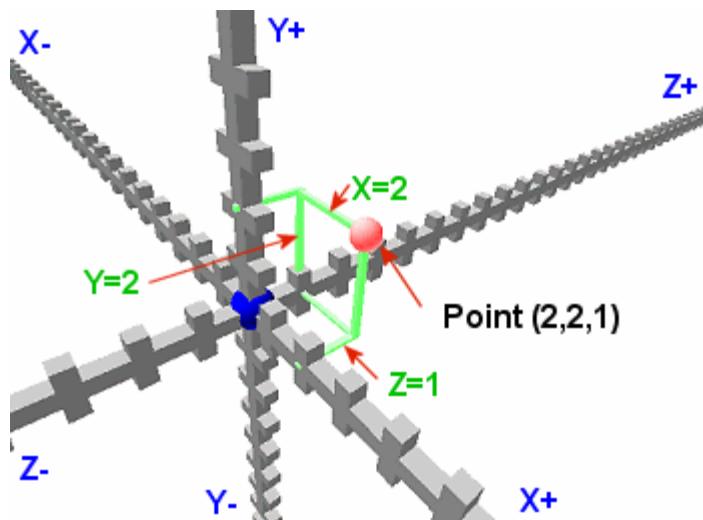


Figure 1.11

A coordinate system has infinite granularity. It is limited only by the variable types used to represent coordinates in source code. If one decides to use variables of type *float* to hold the X, Y and Z components of a coordinate, then coordinates such as (1.00056, 65.0234, 86.01) are possible. If variables of type *int* are used instead, then the limit would be the whole numbers like (10, 25, 2). In most 3D rendering engines variables of type *float* are used to store the location of a point in 3D space. A typical structure for holding a simple 3D position looks like this:

```
struct 3Dpoint
{
    float x;
    float y;
    float z;
};
```

## Creating Our First Mesh

A mesh is a collection of polygons. Each polygon is stored in memory as an ordered list of 3D points. In Fig 1.12 we see that in order to create a 3D cube mesh we would need to specify the eight corner points of the cube in 3D space. Each polygon could then be defined using four of these eight points. The following eight points define a cube that is  $4 \times 4 \times 4$  where the extents of the cube on each axis range from  $-2$  to  $+2$ .

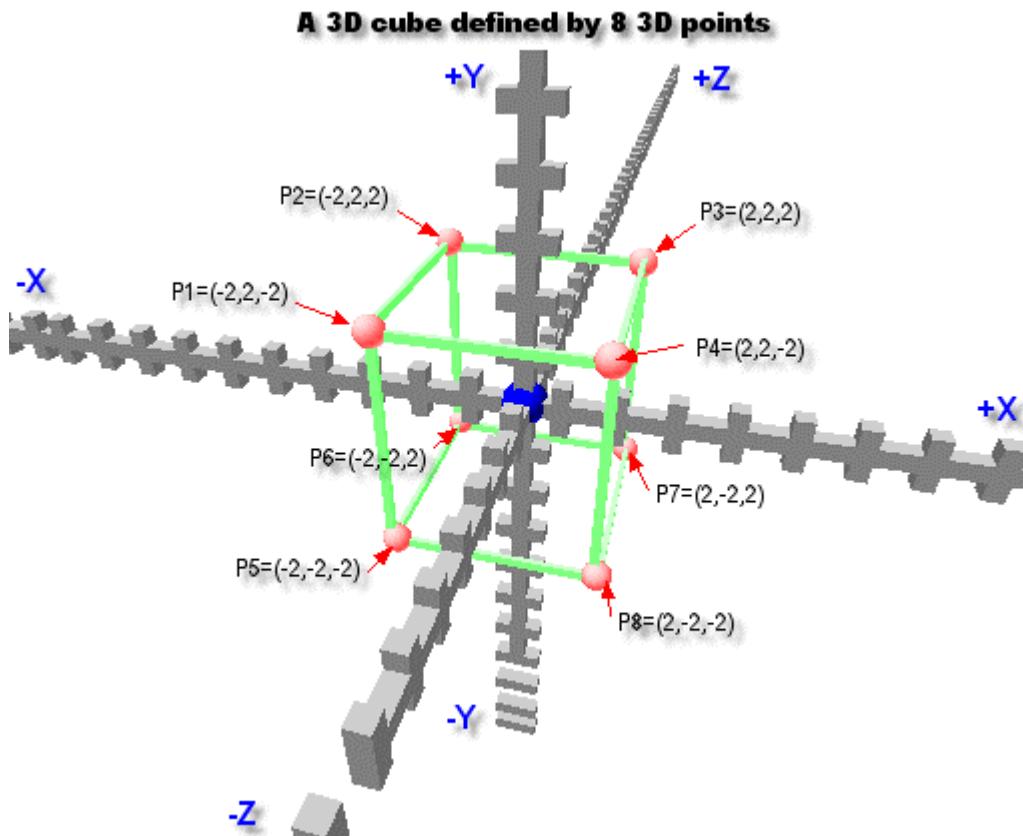


Figure 1.12

We have labeled each of the 3D points P1, P2, P3, etc. The naming order selected is currently unimportant. What is significant is the order that we use these points in to create the polygons of the cube. The front face of the cube would be made up of points P1, P4, P8 and P5. The top face of the cube would be constructed from points P1, P2, P3 and P4. And so on. You should be able to figure out which points are used to create the remaining polygons.

Notice that the center of the cube  $(0,0,0)$  is also the origin of the coordinate system. When a mesh has its 3D points defined about the origin in this way it is said to be in **model space** (or **object local space**). In model space, coordinates are relative to the center of the mesh and the center of the mesh is also the center of the coordinate system. Later we will ‘transform’ the mesh from model space to



**world space** where the coordinate system origin is no longer the center of the mesh. In world space all meshes will coexist in the same coordinate system and share a single common origin (the center of the virtual world).

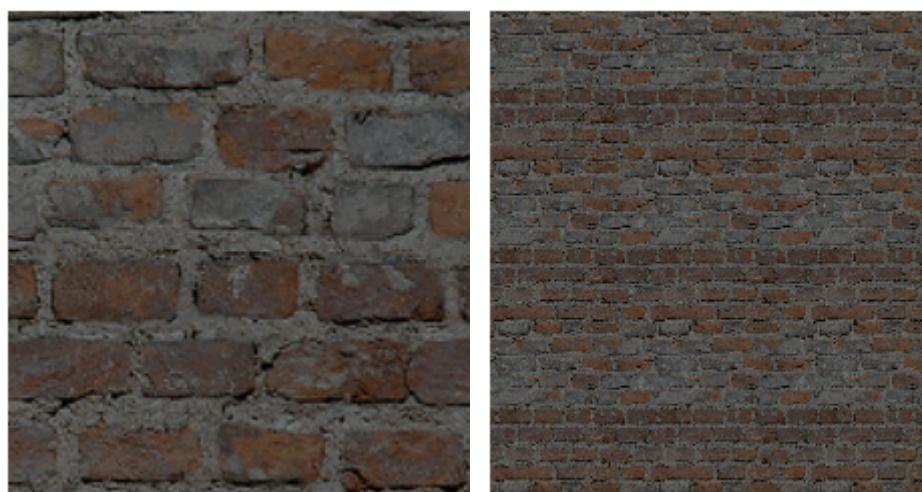
Very often you will want to rotate an object around its center point. For example you might want a game character to rotate around its own center point in order to change direction. We will cover the mathematics for rotating an object later in the lesson, but for now just remember that in order to rotate a mesh you will have to rotate each of the points it contains. In Fig 1.12, we would rotate the cube 45 degrees to the right by rotating each of the eight corner points 45 degrees around the Y axis. When we rotate a point in a coordinate system, the center of rotation will always be at the origin of the coordinate system.

---

### Note: Game World Units

It is up to you, the developer, working with your artists to decide game unit scale. For example, you may decide that 1 unit = 1 meter and ask your artist to design 3D meshes to the appropriate size to make this appear true. Alternatively you might decide that 1 unit = 1 kilometer and once again, create your geometry to the appropriate size. It is important to bear in mind that if you choose such a small scale, you may encounter floating point precision problems.

A mesh could be 4x4x4 units like our cube or even 100x100x100 and look exactly the same from the viewer's perspective. It depends on factors like how fast the player is allowed to move and how textures are applied to the faces. In the next image you can see two identically sized polygons with differently scaled textures. The polygon on the right would probably look much bigger in the game world than the one on the left. As long as all the objects in your world are designed to a consistent scale relative to each other, all will be fine.





## Vertices

The **vertex** (the plural of which is *vertices* or *vertexes* depending on your locale) is a data structure used to hold 3D point data along with other potential information. From this point on we will refer to each point that helps define a polygon in a mesh as a vertex. Therefore, we can say that our cube will have 24 vertices because there are 6 polygons each defined by 4 vertices ( $6 \times 4 = 24$ ).

If you examine our Lab Project for this lesson (LP 1.1), you will see our vertex structure looks like:

```
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
};
```

## Winding Order

3D models will not usually be created programmatically but will be created within a modeling package such as GILEST™ or 3D Studio Max™. This allows us to create scenes with thousands or even millions of polygons. Very high polygon counts often correlate to a reduction in application performance due to the increased volume of calculations that need to be performed when drawing them. As a graphics developer you will use a number of techniques to keep the number of polygons that need to be drawn in a given frame to a minimum. Certainly you would not want to render polygons that the user could not possibly see from their current position in the virtual world. One such optimization discards polygons that are facing away from the viewer; this technique is called **back face culling**. It is assumed that the player will never be allowed to see the back of a polygon. You should notice in our example that regardless of the direction from which you view the cube, you will only be able to see three of the six faces at one time. Three will always be facing away from you. For this reason, 3D rendering engines normally perform a fast and cheap test before rendering a polygon to see if it is facing the viewer. When it is not it can be discarded.

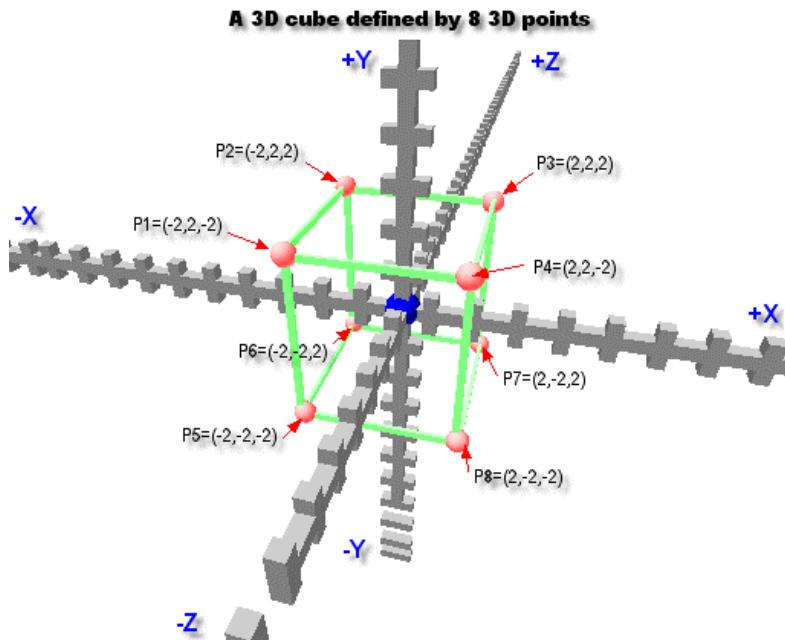


Figure 1.13

Using Figure 1.13 as a reference you should be able to see how each vertex of every face is one of the eight 3D positions of the cube stored in our code. The coordinate P1 is used to create a vertex in the left face, the top face and the front face. And so on for the other coordinates. Also note that the vertices are specified in an ordered way so that lines can be drawn between each pair of points in that polygon until the polygon is finally complete. **The order in which we specify the vertices is significant and is known as the winding order.**

### Clockwise Winding

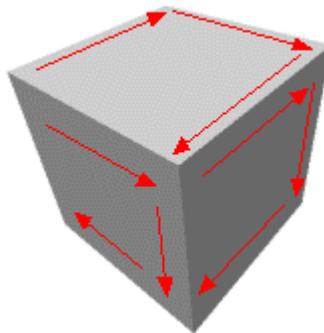


Figure 1.14

So how does one determine which way a polygon is facing? After all, in our cube example, a face is simply four points; we do not provide any directional information.



The answer lies in the order in which we store the vertices within our polygons. If you look at Fig 1.13 and then reference it against the code in LP 1.1, you will notice that the polygon vertices are passed in using a clockwise order.

For example, the front face is made up of points P1, P4, P8 and P5. When viewed from in front of that face this is a clockwise specification. It does not matter which vertex in the face begins the run. We could have created the front face in this order: P8, P5, P1 and P4 and it would still work perfectly because the order remains clockwise. This order is referred to as the polygon **winding order**. In DirectX Graphics, polygons are assumed to have a clockwise winding by default (Fig 1.14) -- although you can change this if desired.

Now look at the back face. It uses the vertex order P6, P7, P3 and P2. This is clearly counter-clockwise so we will not draw it. Of course if we were to rotate the cube so that the back face was now facing us, you would notice that the vertex order would then be clockwise.



## Transformations

### Translation

We can add offsets to the positions of the vertices of a polygon such that the entire polygon moves to a new position in our world. This process is called **translation**. We translate an entire mesh by translating all of its polygons by equal amounts.

In Fig 1.15 we define a 4x4 polygon around the center of the coordinate system (model space). We decided to place our mesh in the virtual game world so that the center of the mesh is at world position (0, 5, 0). If we add this value set to all vertices in the mesh then the center of our mesh is indeed moved to that position.

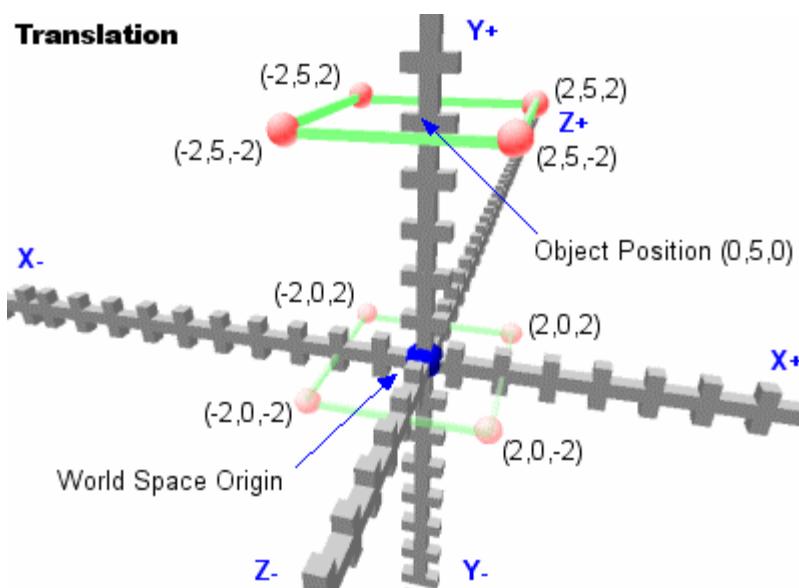


Figure 1.15

In pseudo-code:

```
PositionInWorld.x = 0; PositionInWorld.y = 5; PositionInWorld.z = 0;
for ( Each Polygon in Mesh )
    for ( Each Vertex in Polygon )
    {
        Vertex.x += PositionInWorld.x;
        Vertex.y += PositionInWorld.y;
        Vertex.z += PositionInWorld.z;
    }
}
```

This is a transformation. We are **transforming data from model (relative) space to world (relative) space**. The mesh center (and in turn, its entire local coordinate system) is now positioned at (0, 5, 0) in the game world. You can assign each mesh its own position in the 3D world using this approach.



Note that this is not how we will implement a transformation in code. Rather than altering the polygon data directly we will store the results of the operation in temporary vertices prior to rendering each polygon. We will use a single mesh object defined in model space which never has its data changed. This mesh can be shared by multiple objects types in a process called **instancing** (Fig 1.16).

```
class CObject
{
public:
    CMesh *m_pMesh;
    float PositionX;
    float PositionY;
    float PositionZ;
};
```

Assuming we wanted to have three cubes in our world we would simply create three separate CObject instances. We will specify a position for each object by setting the PositionX, PositionY and PositionZ member variables. The CMesh pointer can point to the same CMesh object in all three instances. For each object in our scene we would do the following prior to rendering:

- a) For each polygon of the mesh referenced by the object
- b) Add the PositionX, PositionY and PositionZ values to the X, Y and Z vertex values.
- c) Store the results in a temporary vertex list.
- d) Render the polygon using the temporary vertices.

```
CMesh *MyMesh; // Pointer to the mesh containing our 4x4 polygon
CObject ObjectA, ObjectB, ObjectC;

ObjectA.m_pMesh = MyMesh;
ObjectB.m_pMesh = MyMesh;
ObjectC.m_pMesh = MyMesh;

ObjectA.PositionX = 0; ObjectA.PositionY = 5; ObjectA.PositionZ = 0;
ObjectB.PositionX = -6; ObjectB.PositionY = 0; ObjectB.PositionZ = 0;
ObjectC.PositionX = 4; ObjectC.PositionY = 0; ObjectC.PositionZ = -5;
```

At the center of Fig 1.16 we see a ghosted image of the model space mesh data. By adding the positional offset of the object to the mesh vertices, we translate the object to the desired position in the 3D world. Notice that it is the **center** of each object that moves to the resulting position. The vertices retain their relationship to that center point. We have effectively moved the **origin** of the model space coordinate system to a new position in the 3D world. Note as well the distinction between a mesh and an object. The mesh is simply the geometry an object uses to represent itself. The object is responsible for maintaining its own position in the 3D world.



## Instances

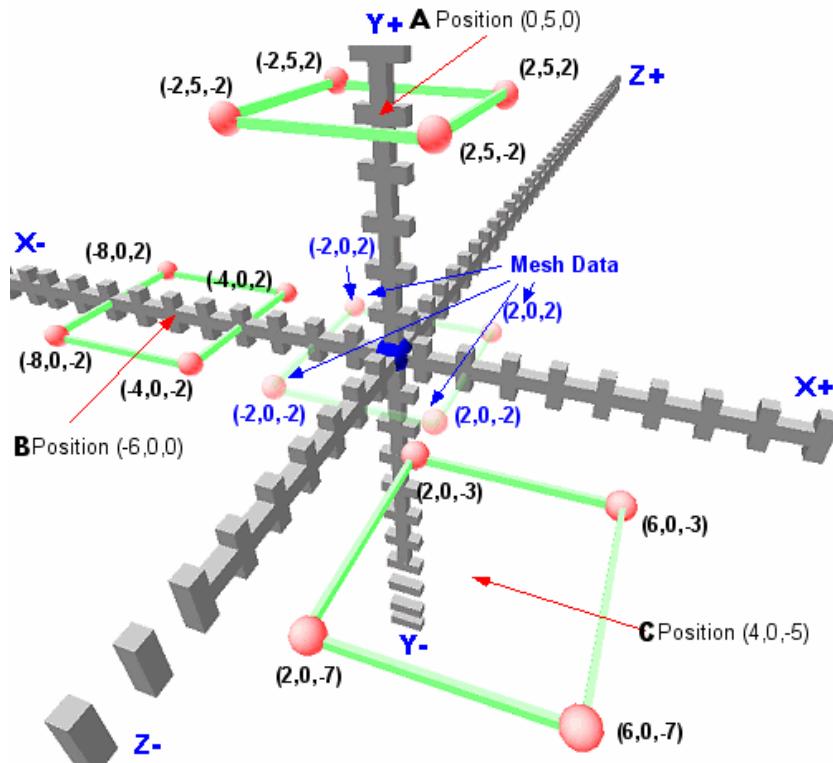


Figure 1.16

The following functions demonstrate how object transformations might occur during each frame so that we can redraw all of the objects in our world. DrawObjects loops through each object, and for each polygon in the mesh, calls the DrawPrimitive function to transform and render it.

```
void DrawObjects ()
{
    // transform vertices from model space to world space
    for ( ULONG i = 0; i < NumberOfObjectsInWorld; i++ )
    {
        CMesh *pMesh = WorldObjects[i]->m_pMesh;
        for ( ULONG f = 0; f < pMesh->m_nPolygonCount; f++ )
        {
            // Store poly for easy access
            CPolygon *pPoly = pMesh->m_pPolygon[f];

            // Transform and render polygon
            DrawPrimitive ( WorldObjects[i] , pPoly )
        }
    }
}
```



```
void DrawPrimitive ( CObject* Object , CPolygon *pPoly )
{
    // Loop round each vertex transforming as we go
    for ( USHORT v = 0; v < pPoly->m_nVertexCount ; v++ )
    {
        // Make a copy of the current vertex
        CVertex vtxCurrent = pPoly->m_pVertex[v];

        // Add world space position to transform to world space
        vtxCurrent.x += Object->PositionX;
        vtxCurrent.y += Object->PositionY;
        vtxCurrent.z += Object->PositionZ;

        // Do further pipeline transformations here which we have
        // not covered yet but will shortly.
        ...
        ...
        // By this point we will have 2D screen vertices so render
        // to screen which we have not yet Covered.
    }
}
```

The transformation from model to world space occurs during every frame for each polygon that we render. By adjusting the position of an object between frames we can create animation. For example, one might make a space ship move through space by incrementally adding or subtracting offsets from the CObject's PositionX, PositionY and PositionZ variables each frame.

## Rotation

To rotate a set of two dimensional points we will use the following formula on each point of the 2D polygon:

$$\begin{aligned} NewX &= OldX \times \cos(\theta) - OldY \times \sin(\theta) \\ NewY &= OldX \times \sin(\theta) + OldY \times \cos(\theta) \end{aligned}$$

In these equations, **OldX** and **OldY** are the two dimensional X and Y coordinates prior to being rotated. **cos** and **sin** are the standard abbreviation for the cosine and sine trigonometric functions. The theta symbol  $\theta$  represents the angle of rotation for the point specified in **radians** and not **in degrees** (most 3D APIs, including DirectX Graphics, use radians for angle calculations).



**Note:** A **radian** is used to measure angles. Instead of a circle being divided into 360 degrees, it is divided into  $2 * \pi$  radians. Pi is approximately 3.14159 and is equivalent to 180 degrees in the radian system of measurement. Therefore there are approximately 6.28 radians in a full circle. 90 degrees is equivalent to  $\pi / 2$  (1.570796 radians) and so on.

Because many programmers prefer working with degree measurements, a macro can be created that will convert a value in degrees to its radian equivalent:

```
#define DegToRad( x ) ( x *( pi/180 ) )
```

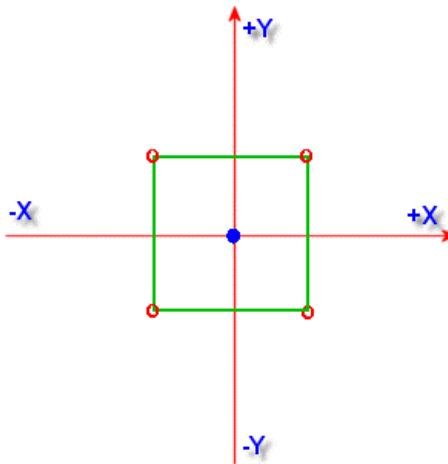


Figure 1.17

We will need to feed each of the four vertices in Fig 1.17, one at a time, through the above rotation formula to receive back our rotated vertices (Fig 1.18). The following code snippet demonstrates this:

```
float angle = DegToRad( 45 );
...
for ( USHORT v = 0; v < pPolygon->m_nVertexCount; v++)
{
    CVertex OldVtx = pPolygon->Vertex[v];
    CVertex NewVtx;

    // Rotate the vertex 45 degrees
    NewVtx.x = OldVtx.x * cos(angle) - OldVtx.y * sin(angle);
    NewVtx.y = OldVtx.x * sin(angle) + OldVtx.y * cos(angle);

    // Vertex is now rotated and stored in NewVtx
    // Use to draw polygon in rotated position
}
...
```

You might think of this rotation as rotating a point around the Z axis. While technically true that we do not see a Z axis in the image, you can contemplate the 2D image in 3D. In this case the Z component



of each point is zero and the Z axis is pointing into the page as it was in the 3D Cartesian system discussed earlier. Fig 1.18 shows the resulting points after rotating the polygon by 45 degrees:

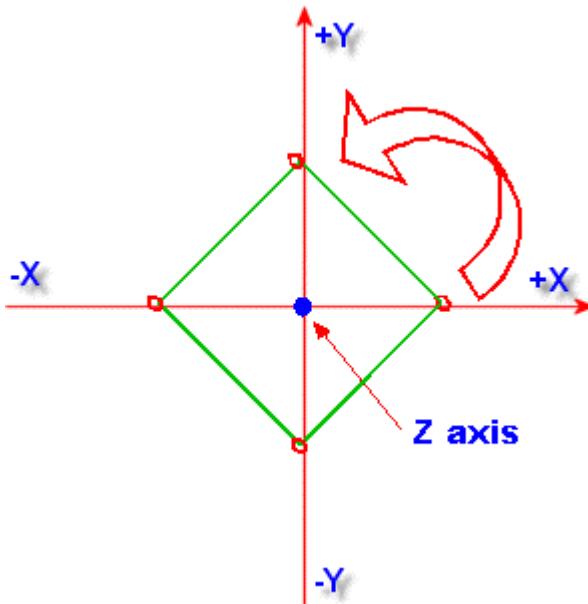


Figure 1.18

The key point to remember is that in a given coordinate system, rotations are relative to the coordinate system origin. You can see in Fig 1.18 that the vertices are rotated about the origin (the blue circle). This is the center of rotation.

Notice that when we rotate a vertex around an axis, the vertex component that matches the axis is unchanged in the result. If we rotate a vertex about the Y axis, only the X and Z values of the vertex are affected. If we rotate about the X axis, only the Y and Z values are affected. If we rotate around the Z axis, only the X and Y values are affected.

The following formulas are used to rotate a 3D point around any of the three principal axes:

#### X Axis Rotation

$$NewY = OldY \times \cos(\theta) - OldZ \times \sin(\theta)$$

$$NewZ = OldY \times \sin(\theta) + OldZ \times \cos(\theta)$$

#### Y Axis Rotation

$$NewX = OldX \times \cos(\theta) + OldZ \times \sin(\theta)$$

$$NewZ = OldX \times -\sin(\theta) + OldZ \times \cos(\theta)$$

## Z Axis Rotation

$$\text{NewX} = \text{OldX} \times \cos(\theta) - \text{OldY} \times \sin(\theta)$$

$$\text{NewY} = \text{OldX} \times \sin(\theta) + \text{OldY} \times \cos(\theta)$$

Because rotations are always relative to the coordinate system origin we have to be careful about the order in which we perform the rotation and the translation operations in our pipeline. Let us imagine that we want to place a mesh into our world at position (0, 5, 0) and that we want it rotated by 45 degrees about the Z axis. We might initially try something like this:

- 1) Apply translation to the vertices to move the object to position (0, 5, 0) in world space.
- 2) Apply 45 degree rotation about the Z axis so it is rolled in world space.

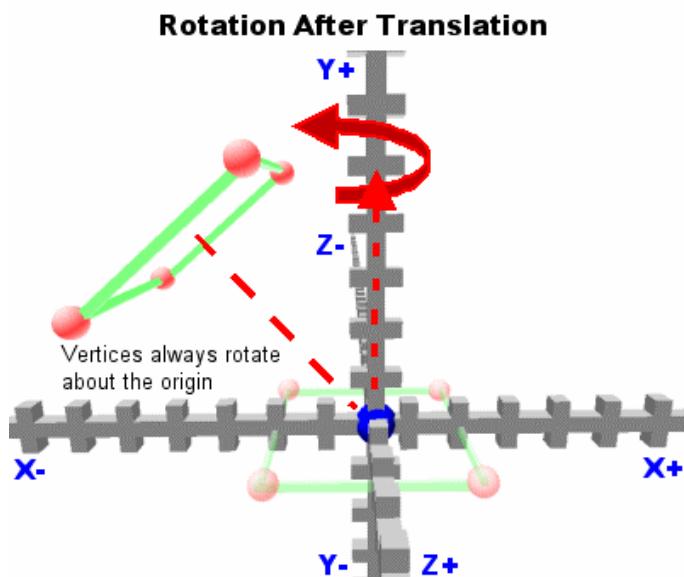
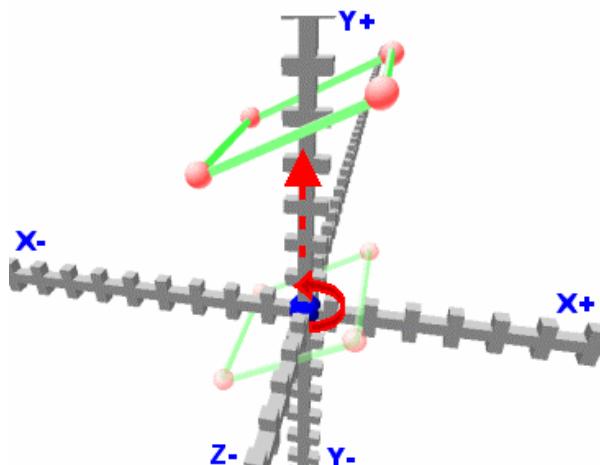


Fig 1.19 might not display what you were expecting. The object was first moved to the world space position (0, 5, 0) and then rotated about the Z axis relative to the world space origin.

More often than not, we want to perform the rotation before the translation. Here the object would first be rotated in model space about its own center point (the model space origin) and then translated to the final position in world space (Fig 1.20).

### Rotation Before Translation



**Figure 1.20**

By performing the rotation transformation first we were able to achieve the expected world space position with a 45 degree roll about the mesh center point. Of course translating before rotating can be useful too. If you had a planet at the coordinate space origin then you might use this approach to make an object rotate around that planet at a constant distance (like an orbit).

We can add some rotation members to our CObject class to allow for object rotations relative to axes:

```
class CObject
{
public:
    CMesh *m_pMesh;

    float PositionX;
    float PositionY;
    float PositionZ;

    float RotationX;
    float RotationY;
    float RotationZ;
}
```

All of these transformations will take place when we render our meshes. They are performed at the ***per-polygon*** level for every frame before those polygons are drawn.



## Cameras

Before we can render anything we must create a virtual camera through which to view our world. All of our world space vertices must then be defined relative to this camera. This requires a new coordinate system called **camera space** (or **view space**) and as we saw earlier, transformations will be required to get our vertices into this new space. We will specify camera properties such as the current position, viewing direction, and field of view (FOV).

We should also be able to move and orient a camera in our world in real-time. This is accomplished in a rather interesting and perhaps not immediately obvious manner:

- 1) When player moves the camera forward, we translate the whole world backward
- 2) When player moves the camera backward, we translate the whole world forward
- 3) When player rotates left around the Y axis, we rotate the entire world right around the Y axis
- 4) And so on...

As you can see, whatever we want our virtual camera to do, we must make the opposite happen to every scene object. This gives the appearance that we are moving through the world when, in fact, it is the world that is moving around us.

A simple camera class might hold only the camera world position and rotation.

```
class CCamera
{
public:
    float PositionX;
    float PositionY;
    float PositionZ;

    float RotationX; // Pitch
    float RotationY; // Yaw
    float RotationZ; // Roll
};
```

One could add input routines to convert mouse or joystick data into rotations for the camera. Moving left on the joystick might store a rotation of 1 degree in the RotationY member to make the camera yaw. If the joystick is pushed forward you might update the position of the camera to make it travel forward along the current heading.



A render loop that includes camera data might look something like the following:

```
void DrawObjects()
{
    for (each object)
    {
        for (Each Polygon in Object)
        {
            DrawPrimitive (Object , Polygon , Camera);
        }
    }
}

void DrawPrimitive (CObject * Object , CPolygon *Poly , CCamera * Cam)
{
    for (each Vertex in Poly)
    {
        // convert polygon to world space (Already discussed)
        Perform any object Rotations on vertices of polygon
        Perform Translation on vertices of polygon to move polygon into world space

        // convert polygon to view space
        Perform inverse camera rotations on vertices of polygon
        Perform inverse camera translations on vertices of polygon

        // Convert polygon to Projection Space
        Not Yet Covered

        // Render 2D polygon
        Not Yet Covered
    }
}
```

If the camera had an X axis rotation of 45 degrees, the following code would rotate all of the vertices of every object in the world -45 degrees about the X axis (i.e. 45 degrees in the opposite direction).

```
if (m_pCamera.RotationX)
{
    VSVertex.y = Vertex.y * cos(-m_pCamera.RotationX) -
                 Vertex.z * sin(-m_pCamera.RotationX);

    VSVertex.z = Vertex.y * sin(-m_pCamera.RotationX) +
                 Vertex.z * cos(-m_pCamera.RotationX);

} // End if X Axis Rotation
```



Y and Z axis rotations would follow along similar lines:

```
if (m_pCamera.RotationY)
{
    VSVertex.x = Vertex.x * cos(-m_pCamera.RotationY) -
        Vertex.z * sin(-m_pCamera.RotationY);

    VSVertex.z = Vertex.x * -sin(-m_pCamera.RotationY) +
        Vertex.z * cos(-m_pCamera.RotationY);

} // End if Y Axis Rotation

if (m_pCamera.RotationZ)
{
    VSVertex.x = Vertex.x * cos(-m_pCamera.RotationZ) +
        Vertex.y * sin(-m_pCamera.RotationZ);

    VSVertex.y = Vertex.x * sin(-m_pCamera.RotationZ) +
        Vertex.y * cos(-m_pCamera.RotationZ);

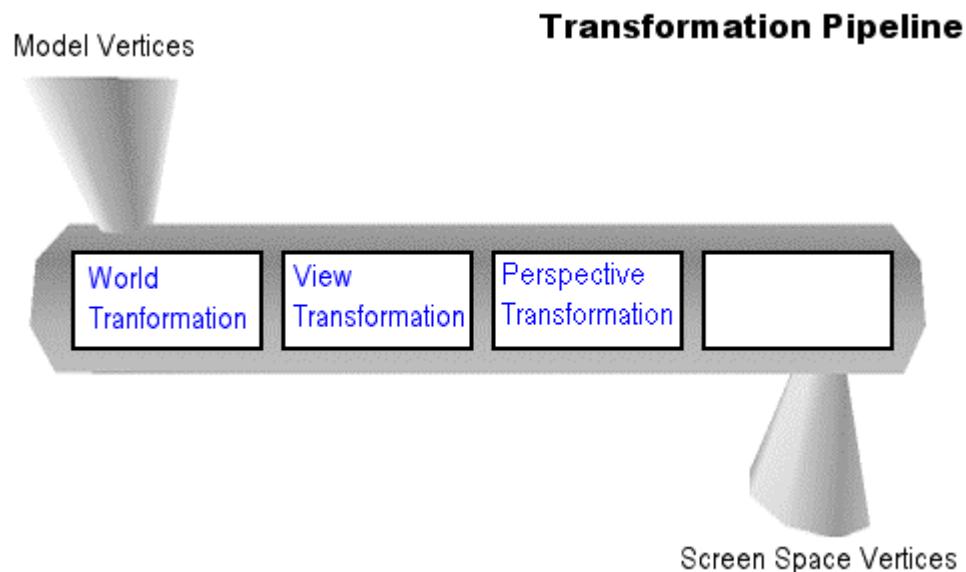
} // End if Z Axis Rotation
```

In the above code, **Vertex** is assumed to be in world space and is being converted into camera space (i.e. view space). The same rotation formulas are used as before with the exception that we are negating the angle passed into the function so that the objects are rotated in the opposite direction. If the camera has a position in the world other than (0, 0, 0) then this would also have to be taken into account which we will look at in a later section.

It is worth pointing out that we devote an entire chapter to camera systems later in the course, so do not be especially concerned if some of these concepts are not immediately obvious to you.



## Perspective Projection



Perspective is an important aspect of how we process distance and scale in the world around us. In real life, as things move further away from us they appear to grow smaller and vice versa. The same will hold true for our scene geometry. As we move the camera away from our meshes, the meshes should ‘shrink’. When we move it closer they should ‘grow’.

Things also tend to move toward the center of your field of view as distance increases and away from the center as distance decreases. Most objects in your field of view will appear to be either left or right or up or down relative to the center of your field of vision. You can refer to these left/right positions using X coordinate values and up/down positions using Y coordinate values relative to that center. In the last section, we discussed converting our vertices to camera space where vertices are offsets relative to a camera coordinate system. We call the distance from the viewer position to any object a: Z coordinate relative to that position (a view space Z coordinate).

As the Z coordinate increases between the viewer and a given mesh, the X and Y coordinates of each vertex in that mesh can be scaled by that Z amount (the distance) to produce the perspective effect:

$$2DX = ViewSpaceX / ViewSpaceZ$$

$$2DY = ViewSpaceY / ViewSpaceZ$$

We divide each vertex view space *x* and *y* components by their view space *z* component. The result is a 2D point in **projection space**.



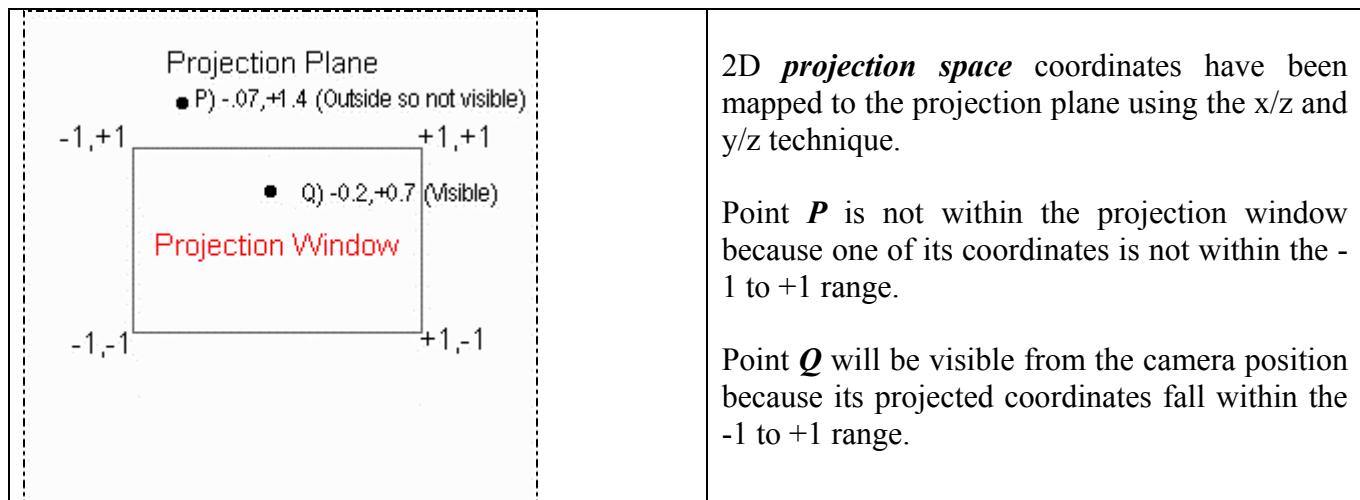
Imagine that we have a coordinate that is 5 units to the right of the camera, 20 units up from the camera and 100 units in front of the camera. This vertex would have a view space coordinate of (5, 20, 100). Performing the perspective projection:

$$\text{projected}X = \frac{\text{ViewSpace}X}{\text{ViewSpace}Z} = \frac{5}{100} = 0.005$$

$$\text{projected}Y = \frac{\text{ViewSpace}Y}{\text{ViewSpace}Z} = \frac{20}{100} = 0.2$$

We end up with the 2D point (0.005, 0.2). Note that these are not screen coordinates (since we know that those must be discrete integer values). These new coordinates are actually called **viewport space coordinates** (sometimes called *clip-space coordinates* or *projection space coordinates*).

The 3D coordinates have been projected onto a 2D infinite plane. On this plane there is a **projection window**. If the x and y values are within this projection window then they are visible to the camera and should be rendered. It is at this point that 3D APIs often perform tests to see if the polygon is facing away from the viewer (back face culling) and may also clip any polygons that are only partially in view.



The projection window is a square 2D window that is 2 units wide and 2 units high with an origin at the center. Thus a projection space point of (0, 0) would map directly to the center of the projection window. Valid coordinates in projection space are in the range of -1 to +1 on both the x and y axes. These are the coordinates generated after dividing by z in the equations shown above.



Both components of the sample projected point ( $x = 0.005$ ,  $y = 0.2$ ) are within the  $-1$  to  $+1$  range so in this particular case the point would be considered within the bounds of the projection window and therefore visible to the camera (within the field of view).

Fig 1.21 shows a side view of the camera in view space prior to the divide by Z operation. Please note that the X axis is assumed to be going into the page and can not be seen and the same logic would also apply to the X coordinate projections.

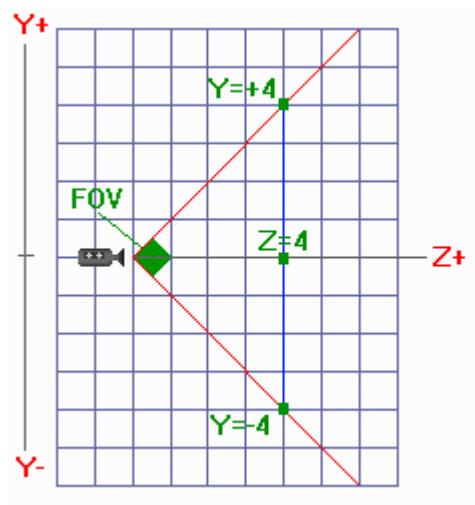


Figure 1.21

As  $Z$  increases,  $Y$  is scaled in direct proportion. Given that the projection window maximum coordinate along the  $Y$  axis is  $+1.0$ , if  $Y = Z$  then our projection formula becomes:

$$\text{projected } Y = \frac{y}{z} = \frac{y}{y} = 1$$

If  $Y = Z$  in view space then that point will be projected at the very top of the projection window. As  $Z$  increases, the maximum  $Y$  point that will fall within the projection window is  $Y = Z$  or  $Y = -Z$ . The same is also true for the  $X$  projection.

So, if at any point  $Y > Z$  or  $Y < -Z$  or  $X > Z$  or  $X < -Z$ , when this point is projected, it will fall outside the  $-1$  to  $+1$  range (and therefore outside the projection window).

If we have a  $Z$  coordinate of  $4$  then the range of  $Y$  coordinates that are visible are  $[-4, +4]$  in view space. The maximum range of  $Y$  values that can be seen at a distance of  $Z=6$  is in the range  $[-6, +6]$ . And so on. (Once again, exactly the same holds true for the  $X$  coordinate projection.)

Thus, for any point in view space where  $(x > -z)$  and  $(x < z)$  and  $(y > -z)$  and  $(y < z)$  that **point is visible**.



When we scale the  $x$  and  $y$  components of the vertex in proportion to  $z$ , we are in effect creating an imaginary view cone that extends out at a 90 degree angle across both the  $x$  and  $y$  axes (45 degrees up and 45 degrees down on the  $Y$  axis and 45 degrees left and 45 degrees right for  $X$ ). Where the two red lines meet in Fig 1.21 (at the camera position) there is a perfect right angle. Any points falling within this cone are considered visible because their divided  $x$  and  $y$  coordinates will fall within the bounds of the projection window.

Thus our virtual camera has a 90 degree field of view because the ratio described above will always produce values that are consistent with this.

Although the 90 degree view cone does not *really* exist, as there is no physical camera in our game world, it is a useful way to think about how functions that convert vertices from 3D to 2D space work. Dividing  $x$  and  $y$  by  $z$  stretches or squashes geometry as it gets closer or further away from the camera respectively. Looking at the view cone in Fig 1.21 we note that the total range between the bottom red cone line and the top red cone line at any given  $z$  position, is mapped into the  $-1$  to  $+1$  range. As  $Z$  increases, a larger portion of the cone is mapped to the  $[-1, +1]$  range and things get squashed more towards the center of the projection window.

Fig 1.22 shows a series of points plotted at the same  $y$  position in view space, each with increasing  $z$ .

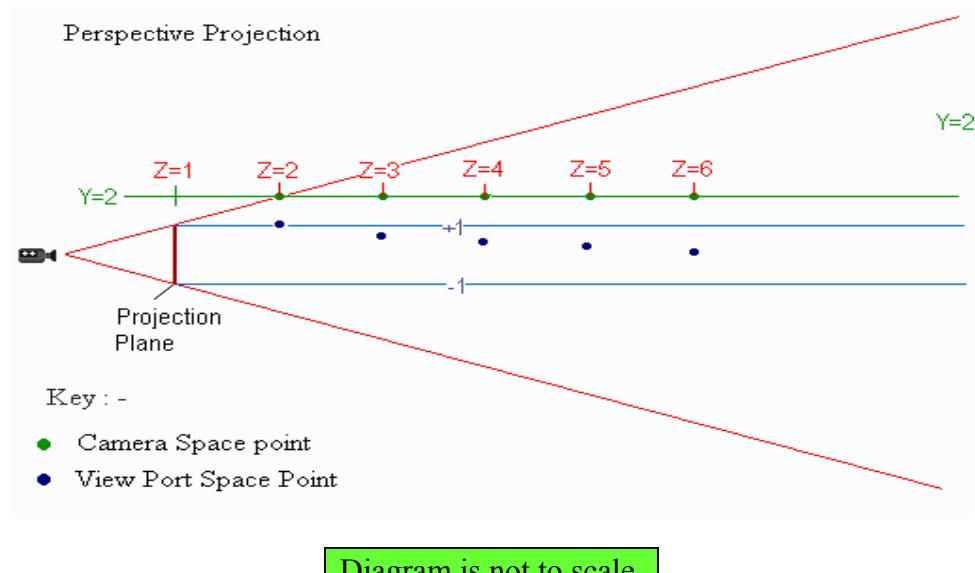


Figure 1.22

The formula squashes the two red cone lines so that they become parallel with each other with a separating distance equal to the size of the projection plane (Fig 1.22). The blue lines show what the cone looks like after it has been squashed into what is essentially a box. Larger  $z$  values produce greater squashing ratios.



In Fig 1.22 there are five points in view space (green circles). Each has a  $y$  value of +2 and increasing  $z$  values are assigned. The effect of our projection formula can be seen when we look at the projected points (blue dots). All of these have been squashed into the  $[-1, +1]$  range. Although the points had identical  $y$  values in *view space*, when mapped to *projection space* they receive different  $y$  values.

In many math textbooks perspective projection formulas are listed as:

$$X_p = \frac{x}{z/d}$$

$$Y_p = \frac{y}{z/d}$$

The problem with the projection formula we have been using is that it **always** projects with a field of view of 90 degrees. We would prefer to use an arbitrary field of view to give complete control over exactly how much of the scene can be viewed by the camera. In order to accommodate this, a new variable is introduced ( $d$ ). This allows the projection window to be moved further from or closer to the camera. Because the size of the projection window remains the same (-1 to +1), moving the projection window further away from the camera reduces the cone size. Moving the projection window nearer to the camera increases the cone size. This new formula allows us to alter the camera FOV in a manner similar to the way a photographer might adjust the lens of his camera to capture more or less of a scene in his photo. In Fig 1.23 you can see why moving the projection window affects the FOV:

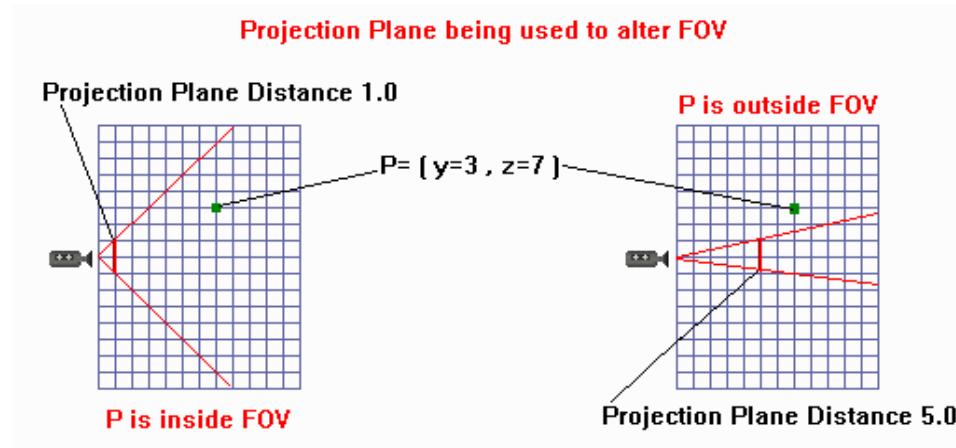


Figure 1.23

The cone is much smaller when the projection window is at a distance of 5 units from the camera than when it is when it is at a distance of 1 unit. This distance is labeled  $d$  in the above formula.

While this technique works quite well, in DirectX Graphics (and in our software renderer) the projection window is always set at a fixed distance of **1.0** unit from the camera. The pipeline performs the  $x/z, y/z$  mapping into 2D space as was the case in our old formula:



$$X_p = \frac{x}{z}$$

$$Y_p = \frac{y}{z}$$

But we can achieve the effect of the  $d$  value using a different strategy. Our code (and DirectX Graphics) will continue to use a 90 degree FOV behind the scenes but will use a projection matrix to deform geometry *prior to* the divide by  $z$  to accommodate the appearance of arbitrary FOV. We will examine the projection matrix in detail later in the lesson. For now we will proceed with a 90 degree FOV.

### Screen Space Mapping

The final stage is finding a screen space pixel coordinate for our projected vertex. Transforming a 2D projection space point to a 2D screen space point requires mapping the  $-1$  to  $+1$  coordinates to the width and height of the current render window. The formula is:

```
ScreenX = projVertex.x * ScreenWidth / 2 + ScreenWidth / 2
ScreenY = -projVertex.y * ScreenHeight / 2 + ScreenHeight / 2
```

Let us assume our window is 640x480 pixels in size and that we have a vertex which has been mapped to  $(0, 0)$  in projection space. This should mean that it is in the center of the screen:

```
ScreenX = 0 * (640/2) + (640/2)
ScreenX = 0 * (320) + (320)
ScreenX = 320
```

Another example would be  $x = -1$  in projection space. It should wind up on the far left hand side of the projection window (and thus the screen):

```
ScreenX = -1 * (640/2) + (640/2)
ScreenX = -1 * (320) + (320);
ScreenX = -320 + 320
ScreenX = 0;
```

The Y value is projected into screen coordinates using the same approach but with one exception. In projection space (as with model space, world space and view space) the Y axis is positive running up and negative running down. In screen space (as we discussed earlier in this lesson) the Y axis would be 0 at the top of the screen and increase toward the bottom. So we will need to invert it by negating the projection space Y coordinate to ensure conformity.

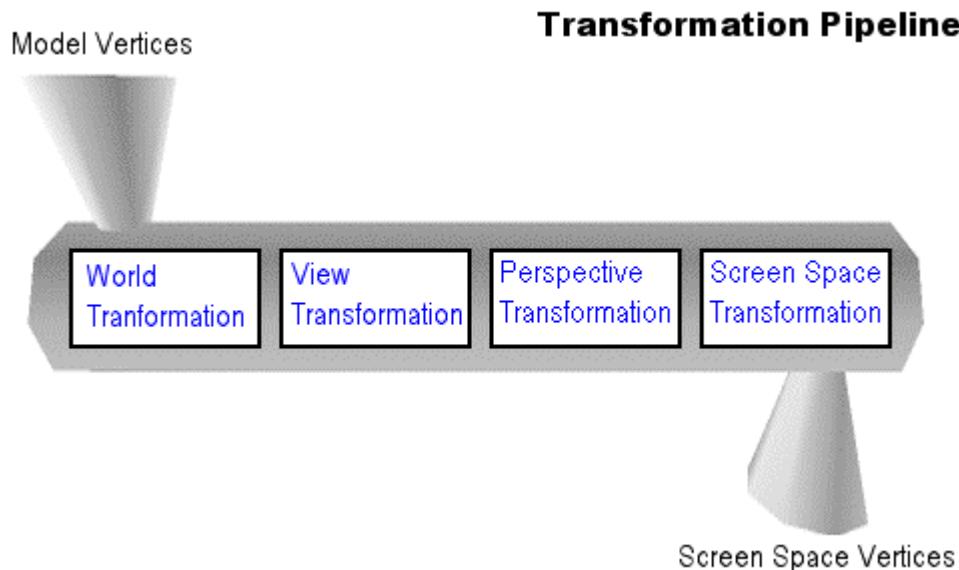
Let us look at an example using a projection Y value of  $Y = 1$  in projection space. We saw earlier that this value was at the very top of the projection window. We need it to be at the top of the screen too:



```
ScreenY = -1 * (480/2) + (480/2)
ScreenY = -1 * (240) + (240)
ScreenY = -240 + 240
ScreenY = 0
```

Once all of the vertices of our polygon are in screen space we can draw lines between each point. The result is a wire frame rendition of our scene geometry.

### Draw Primitive Pseudocode



The pseudo-code to an updated **DrawPrimitive** function follows. In the example, we pass the object we are processing and the polygons we wish to render. The object is needed for its position and rotation information which is necessary to transform the polygons into world space. We pass a pointer to a camera so that we can access the camera position and rotational information in order to do a view space transformation after the world space transformation:

```
void DrawPrimitive( CObject *pObject , CPolygon *pPoly , CCamera *pCamera )
{
    CVertex CurrVertex;
    CVertex PrevVertex;

    // Retrieve object angles;
    float Opitch = pObject->RotationX;
    float Oyaw   = pObject->RotationY;
    float Oroll  = pObject->RotationZ;

    // Retrieve Camera angles
    float Cpitch = pCamera->RotationX;
    float Cyaw   = pCamera->RotationY;
```



```
float Croll = pCamera->RotationZ;

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

    // WORLD SPACE TRANSFORMATION
    // Apply any object rotations if applicable
    if (Opitch) // rotate object about its x axis like pitching up and down
    {
        currVertex.y = currVertex.y * cos(Opitch) - currVertex.z * sin(Opitch);
        currVertex.z = currVertex.y * sin(Opitch) + currVertex.z * cos(Opitch);

    } // End if Pitch

    if (Oyaw) // rotate object about its Y axis like yawing left/right
    {
        currVertex.x = currVertex.x * cos(Oyaw) + currVertex.z * sin(Oyaw);
        currVertex.z = currVertex.x * -sin(Oyaw) + currVertex.z * cos(Oyaw);

    } // End if Yaw

    if (Oroll) // rotate object about its Z axis like rolling left or right
    {
        currVertex.x = currVertex.x * cos(Oroll) + currVertex.y * sin(Oroll);
        currVertex.y = currVertex.x * sin(Oroll) + currVertex.y * cos(Oroll);

    } // End if Roll

    // Now move the vertex into its world space position
    currVertex.x += pObject.PositionX;
    currVertex.y += pObject.PositionY;
    currVertex.z += pObject.PositionZ;

    // VIEW SPACE TRANSFORMATION
    // subtract the camera position from the vertex so its position is relative
    // to the camera with the camera at the origin
    currVertex.x -= pCam->PositionX;
    currVertex.y -= pCam->PositionY;
    currVertex.z -= pCam->PositionZ;

    // if the camera is rotated, rotate the world the opposite way
    // but the only difference
    // from the object rotation is the negated parameter
    if (Cpitch) // rotate camera about its x axis like pitching up and down
    {
        currVertex.y = currVertex.y * cos(-Cpitch) - currVertex.z * sin(-Cpitch);
        currVertex.z = currVertex.y * sin(-Cpitch) + currVertex.z * cos(-Cpitch);

    } // End if Pitch
```



```
if (Cyaw) // rotate cam around its Y axis
{
    currVertex.x = currVertex.x * cos(-Cyaw) + currVertex.z * sin(-Cyaw);
    currVertex.z = currVertex.x * -sin(-Cyaw) + currVertex.z * cos(-Cyaw);

} // End if Yaw

if (Croll) // rotate camera about its Z axis like rolling left or right
{
    currVertex.x = currVertex.x * cos(-Croll) + currVertex.y * sin(-Croll);
    currVertex.y = currVertex.x * -sin(-Croll) + currVertex.y * cos(-Croll);

} // End if Roll

// PERSPECTIVE PROJECTION TRANSFORMATION
// divide x and y by z to project point onto 2D projection
// window in the -1 to +1 range
currVertex.x /= currVertex.z;
currVertex.y /= currVertex.z;

// SCREEN SPACE TRANSFORMATION
// Convert to screen space coordinates
vtxCurrent.x =   vtxCurrent.x * SCREENWIDTH / 2 + SCREENWIDTH / 2;
vtxCurrent.y = -vtxCurrent.y * SCREENHEIGHT / 2 + SCREENHEIGHT / 2;

// 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 between this one and the previous vertex in the loop
DrawLine( vtxPrevious, vtxCurrent, 0 );

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

} // Next Vertex
}
```

After the above function has been called for each polygon of every object in the world we would be left with a 2D visual representation of our 3D world from the point of view of our virtual camera.



## 3D Mathematics Primer

### Vectors

A vector is a mathematical construct that describes a physical point or a direction and magnitude. We can represent a 3D vector using a C++ class:

```
class Vector
{
public:
    float x;
    float y;
    float z;
};
```

Vectors are very important to the 3D graphics programmer. You might have noticed the similarity to 3D Cartesian points. In fact, a 3D point is a vector. To be more precise it is a 3D vector. There are also 2D vectors and so on for other dimensions.

Although many people use the terms ***vector*** and ***point*** interchangeably there is a distinction; a point is always a vector but the reverse is not always true. It depends on how we intend to interpret the values stored in the vector: either as an actual location in space (where the vector does indeed describe an absolute point) or as an indicator of direction with magnitude (which can be used relative to some other point in space).

Point vectors can be defined via a direction from some origin (the origin of our coordinate system) and a magnitude (the distance to travel in that direction). If we travel out from the origin in a given direction for a specified distance we end up at a location described in 3D space.

In Fig 1.24 we see points plotted in a 2D Cartesian coordinate system. Although each point can be described as a collection of offsets from the origin along each major axis, each point also describes a vector from the origin to that point (the green arrows):



## Vectors

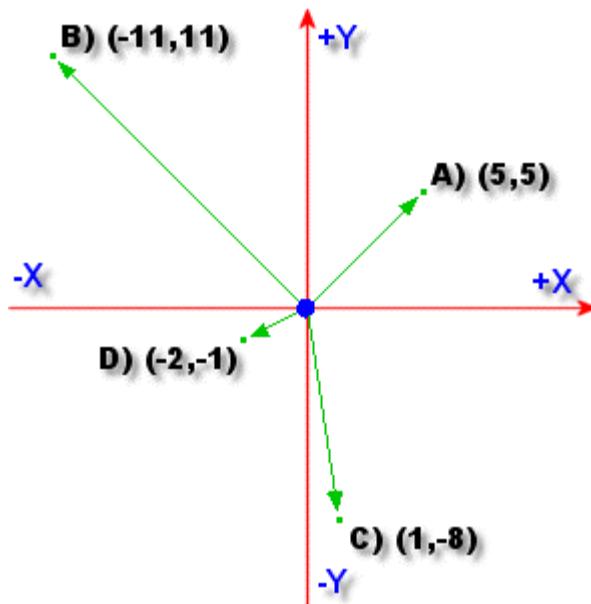


Figure 1.24

The vector  $A<5, 5>$  can be described as a location 5 units right of the origin and 5 units up from the origin. It also describes the line shown by the green arrow which has a definite direction and a length.

### Vector Magnitude

We can use the Pythagorean Theorem to determine the length of a vector. This length is the distance from the origin to the point (the length of the green arrows above).

$$2D\ Vector\ Magnitude = \sqrt{X^2 + Y^2}$$

Everything is identical when working with 3D vectors; we simply have an extra axis. To find the length of a 3D vector we would use the extended formula:

$$3D\ Vector\ Magnitude = \sqrt{X^2 + Y^2 + Z^2}$$

We could write a function that returned the length of a 3D vector like so:

```
float VectorLength3D( CVector * v )
{
    return sqrtf( (v->x * v->x) + (v->y * v->y) + (v->z * v->z));
}
```



If we use the 2D vector A<5, 5>:

```
Length = sqrtf( (5*5)+(5*5) )
        = sqrtf( 50 )
        = 7.0710
```

If we travel a distance of 7.0710 units from the origin and disperse that motion evenly in both the positive X and Positive Y directions (because the x and y vector components are equal in this example) we will arrive at the location (5, 5).

We calculate the length of vector C<1, -8>:

```
Length = sqrtf( (1*1) + (-8*-8) )
        = sqrtf( 65 )
        = 8.06225
```

If we travel from the origin down the positive X axis and the negative Y axis at a ratio of 8:1 for a distance of 8.06225 units we would arrive at location C. The distance is dispersed over the ratio of the X:Y components in a 2D vector or the X:Y:Z components in a 3D vector.

Vector magnitude is represented using two uprights on either side of the vector name:

**Length of C = |C|.**

While we can and will use vectors for representing the vertices of our objects in 3D space, they can also be used for many other tasks in 3D graphics programming, from representing the direction in which the camera is facing, to representing the way that light reflects off a polygon or a vertex. Vectors will be used within collision detection systems and to make objects move around your game world.

### Vector Addition and Subtraction

Vector addition is performed by adding like components together to create a new vector. We can write vector addition using the short hand ( $C = A + B$ ).

```
CVector AddVectors3D( CVector A , CVector B )
{
    CVector C;

    C.x = A.x + B.x;
    C.y = A.y + B.y;
    C.z = A.z + B.z;

    return C;
}
```

To add two 2D vectors together simply remove the addition of the Z components.

Adding two 2D vectors (A and B) we can visualize the resulting vector (C) by taking the tail of B and placing it at the head of A and then drawing a new vector between the tail of A and the head of B (Fig 1.25). The second vector (B) is now relative to first vector (A).

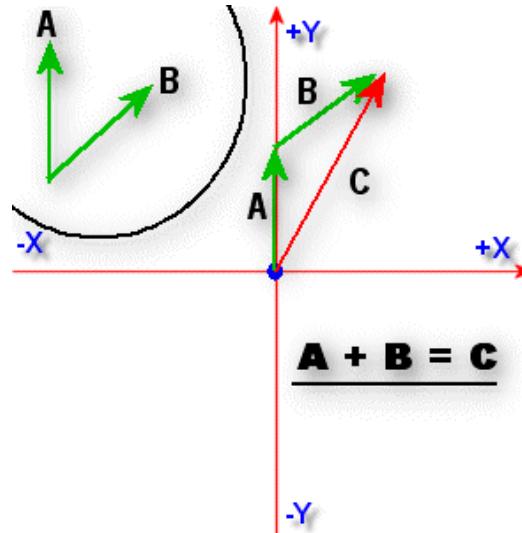


Figure 1.25

The circular inset in Fig 1.25 shows the vectors A and B and their relationship to one another prior to the addition. During addition, A begins at the origin and B is added to this vector. So its tail starts at the tip of A. The resulting vector is the red vector C.

Vector subtraction is similar to addition:

```
CVector SubtractVector3D( CVector A , CVector B )
{
    CVector C;

    C.x = A.x - B.x;
    C.y = A.y - B.y;
    C.z = A.z - B.z;

    return C;
}
```

Because subtracting B from A is the same as negating B and then adding it to A, we could represent this as:

$$\mathbf{C} = \mathbf{A} - \mathbf{B}$$

OR (the negated version)

$$\mathbf{C} = \mathbf{A} + (-\mathbf{B})$$



We can visualize the resulting vector (C) by placing the tail of B at the tip of A as we did with addition. This time we flip (negate) the direction of B so that it is facing in the opposite direction. Fig 1.26 shows the same two 2D vectors A and B. B is subtracted from A to produce the red vector C.

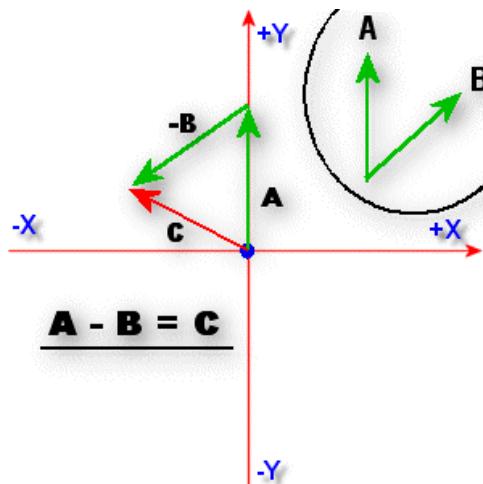


Figure 1.26

Vector subtraction is quite useful. It allows us to gain an understanding of the relationship between the objects in our scene. Let us say, for example, that we have two fighter planes (Fighter Plane A and Fighter Plane B) in our game world. One of them is at position A, and the other at position B. If we subtracted position B from position A, we would end up with a vector that told us both the direction Fighter Plane A would have to fly to get to Fighter Plane B's position as well as the distance between the two by calculating the vector length (Fig 1.27).

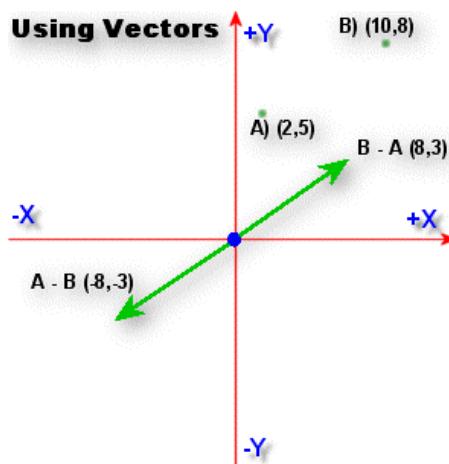


Figure 1.27



In Fig 1.27 there are two points (A and B) representing our fighter planes. If we subtract B from A we would end up with Vector B-A = (8, 3) shown as the green arrow above pointing right and up. We could then go on to calculate the length of the vector as follows

$$\begin{aligned}\text{Magnitude}_{B-A} &= \sqrt{8^2 + 3^2} \\ &= \sqrt{64 + 9} \\ &= 8.544\end{aligned}$$

So the distance between fighter plane A and fighter plane B is 8.544 units. In order for fighter plane A to reach fighter plane B it must travel that distance in a ratio of 8:3 along the positive X and Y axes, respectively.

### Vector Scalar Multiplication

Vectors can be multiplied by scalar values. In this case the scalar is multiplied with each component of the vector. A function that performs scalar multiplication on a 3D vector might look like the following:

```
CVector VectorMultiply3D (CVector A , float scalar)
{
    CVector C;

    C.x = A.x * scalar;
    C.y = A.y * scalar;
    C.z = A.z * scalar;

    return C;
}
```

Fig 1.28 shows the visual effect of multiplying Vector A by 2.0.

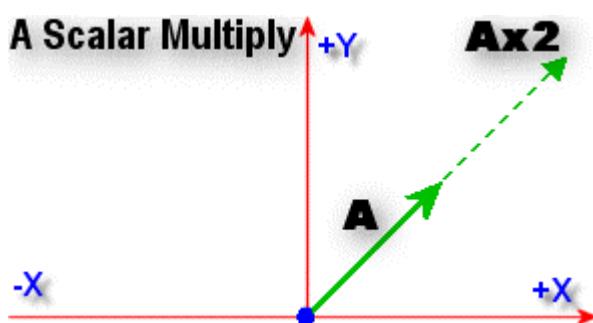


Figure 1.28



## Unit Vectors

A special type of vector that is incredibly useful in 3D graphics programming is the **unit vector**. A unit vector is a vector with magnitude = 1. The process of taking a non-unit vector and making it a unit vector is called normalizing the vector (or **normalization**). This is done by dividing each component of the vector by the overall length of the vector.

$$\begin{aligned}\text{VectorC} &= (8, 3, 10) \\ |\text{C}| &= \sqrt{8^2 + 3^2 + 10^2} \\ |\text{C}| &= 13.152946\end{aligned}$$

$$\begin{aligned}\text{UnitVectorC} &= \left[ \frac{C.x}{|C|}, \frac{C.y}{|C|}, \frac{C.z}{|C|} \right] \\ \text{UnitVectorC} &= \left[ \frac{8}{13.152946}, \frac{3}{13.152946}, \frac{10}{13.152946} \right]\end{aligned}$$

$$\text{UnitVectorC} = (0.60822, 0.2280, 0.7602859)$$

Vector C should now have a length of 1. To prove this let us run the distance calculation on the resulting vector:

$$\begin{aligned}\text{UnitVectorC} &= (0.60822, 0.2280, 0.7602859) \\ |\text{UnitVectorC}| &= \sqrt{0.60822^2 + 0.2280^2 + 0.7602859^2} \\ |\text{UnitVectorC}| &= 0.9999\dots\end{aligned}$$

This is as close to 1.0 as we can generally expect using limited precision floating point math.

Note that while the length becomes 1.0 the directional information remains the same. This is due to the fact that all vector components are scaled equally by the length. Next we see a function that could be called to normalize a vector. It uses one of our earlier functions (*VectorLength3D*) to initially calculate the length of the vector.

```
CVector VectorNormalize3D ( CVector A)
{
    float length = VectorLength3D ( A );
    A.x = A.x / length;
    A.y = A.y / length;
    A.z = A.z / length;

    return A;
}
```

We mentioned that unit vectors can be used for object movement. Let us assume that we have a spaceship facing down the X and Z axes of our world in equal proportions. This direction could be



represented with a single vector (we will call this **DirectionVector**) and might be (1, 0, 1). Imagine that we want to move our space ship forward based on a velocity of 100 world space units per frame.

```
DirectionVector = (1,0,1)
Speed          = 100;

Movement.x = DirectionVector.x * speed;
Movement.y = DirectionVector.y * speed;
Movement.z = DirectionVector.z * speed;

=
Movement.x = 1 * 100 = 100;
Movement.y = 0 * 100 = 0;
Movement.z = 1 * 100 = 100;
```

Is this correct? We said the space ship could travel 100 units so let us check the length of the movement vector:

$$|\text{movement}| = \sqrt{100^2 + 0^2 + 100^2}$$
$$|\text{movement}| = 141.42135$$

That is obviously incorrect as we moved the ship 141 units. The problem is that the direction vector specified (1, 0, 1) is **not** a unit vector. If we calculate the length of that initial direction vector we can see that we would end up with:

$$|\text{DirectionVector}| = \sqrt{1^2 + 0^2 + 1^2} = 1.4142135$$

The error is the result of the ship moving a total of 100 units along the X axis **and** 100 units along the Z axis (which is not the same as moving 100 units diagonally as we would expect). Before we use our direction vector to calculate the new movement vector we must normalize the vector:

$$\text{DirectionVector} = \left[ \frac{1}{1.4142135}, \frac{0}{1.4142135}, \frac{1}{1.4142135} \right] = (0.7071068, 0, 0.7071068)$$

Let us calculate the ship's movement vector again with the normalized direction vector.

```
Movement.x = 0.7071068 * 100 = 70.71068
Movement.y = 0.7071068 * 0 = 0
Movement.z = 0.7071068 * 100 = 70.71068
```

This movement vector (70.71068, 0, 70.71068) gets added to our ship's previous position.

```
NewPosition.x = OldPosition.x + Movement.x;
NewPosition.y = OldPosition.y + Movement.y;
NewPosition.z = OldPosition.z + Movement.z;
```



Let us check our results:

$$|\text{Movement}| = \sqrt{70.71068^2 + 0^2 + 70.71068^2}$$

$$|\text{Movement}| = 100$$

We now have a space ship located at a new position having traveled exactly 100 units from its previous position in the direction of the unit vector. This is the equivalent of moving 70.7 units along the X axis and 70.7 units along the Z axis. If unit vectors are used to represent the direction an object is facing in your game world, using the above technique allows you to easily move that object forward (no matter which direction it is facing).

Object movement is just one of the many uses of unit vectors. Unit vectors are also used to describe the direction your polygons are facing; something which is used extensively during lighting calculations. You will come to discover that unit vectors are seen all the time in 3D graphics programming and we will cover a lot of these situations throughout the coming lessons.

### The Cross Product

The ***cross product*** operation between two vectors results in a third vector perpendicular to the two input vectors. The ‘ $\times$ ’ symbol is used to represent a cross product between two vectors.

#### Input vectors

$$\mathbf{A} = (0, 1, 0)$$

$$\mathbf{B} = (1, 0, 0)$$

#### Cross product calculation

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = ((\mathbf{A.y} * \mathbf{B.z}) - (\mathbf{A.z} * \mathbf{B.y}), (\mathbf{A.z} * \mathbf{B.x}) - (\mathbf{A.x} * \mathbf{B.z}), (\mathbf{A.x} * \mathbf{B.y}) - (\mathbf{A.y} * \mathbf{B.x}))$$

$$\mathbf{C} = ((1*0) - (0*1), (0*1) - (0*0), (0*0) - (1*1))$$

$$\mathbf{C} = (0, 0, -1)$$

The resulting vector C is perpendicular (90 degrees) to vectors A and B. The two green vectors in Fig 1.29 show the input vectors and the resulting vector C is shown in red.

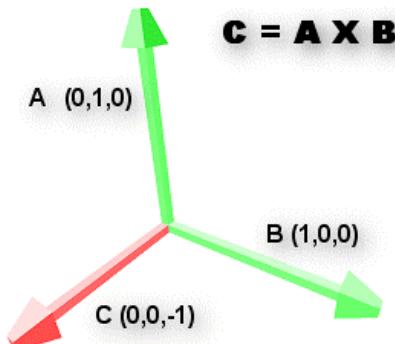


Figure 1.29



In this example we used two unit vectors as input and the vector returned is also a unit vector. The cross product does not require that the input vectors be unit vectors. If the two input vectors are not unit length then the resulting vector will also not be unit length but it will still be perpendicular. If you require a unit length vector then you will need to normalize the resulting vector.

The order in which we pass the vectors into the cross product operation is significant. If we had performed  $B \times A$  instead of  $A \times B$ , the resulting vector  $C$  above would still be perpendicular to the input vectors but would be facing in the opposite direction. Try this out for yourself on paper using the above calculations.

So the cross product works with any two arbitrarily orientated vectors and will always return a vector that is perpendicular to them.

```
CVector VectorCrossProduct (CVector A , CVector B)
{
    CVector C;
    C.x = (A.y*B.z) - (A.z*B.y);
    C.y = (A.z*B.x) - (A.x*B.z);
    C.z = (A.x*B.y) - (A.y*B.x);

    return C;
}
```

## Normals

One very useful application of the cross product is generating what is known as a **surface normal**. A normal is a unit length vector that describes the direction a polygon is facing.

In Fig 1.30 we see a triangular polygon consisting of three vectors ( $v_1, v_2, v_3$ ). If we were to subtract  $v_0$  from  $v_1$  the result would be a vector which describes *Edge 1*. If we do the same again, this time subtracting  $v_0$  from  $v_2$  we get *Edge 3*. The cross product of these two edges yields a vector which, after normalization, is the polygon normal:

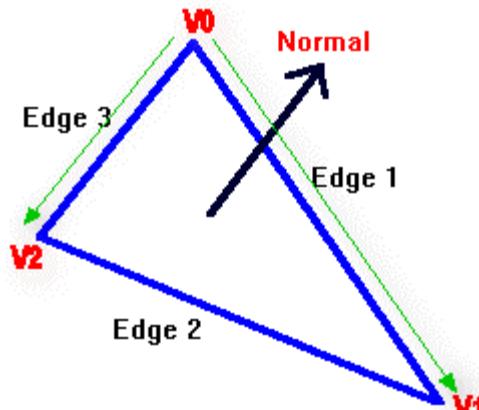


Figure a. Shows a Polygon Normal

Figure 1.30

The following code snippet assumes that the polygon structure has already been initialized with the vertex data and uses some of our previously created vector functions to accomplish this task. This code can also safely cast our polygon vertex structure to a CVector because at this point our vertex structure simply contains an x, y and z position.

```
CVector GeneratePolygonNormal( CPolygon P )
{
    CVector Edge1, Edge2, Normal;

    Edge1 = SubtractVector3D ( Polygon.Vertices[1], Polygon.Vertices[0] );
    Edge2 = SubtractVector3D ( Polygon.Vertices[2], Polygon.Vertices[0] );

    Normal = VectorCrossProduct ( Edge1, Edge2 );
    Normal = VectorNormalize3D ( Normal );

    return Normal;
}
```

If the polygon is rotated, the normal would have to be regenerated in order to correctly describe the new orientation.

### The Dot Product

The • symbol is commonly used to express the dot product (inner product) operation between two vectors. The dot product calculation between two 3D vectors A and B can be expressed as follows:

$$\mathbf{A} \bullet \mathbf{B} = (\mathbf{A.x} * \mathbf{B.x}) + (\mathbf{A.y} * \mathbf{B.y}) + (\mathbf{A.z} * \mathbf{B.z})$$



The results of each component multiply are added to create a single scalar value and *not* another vector. The significance of the result can be appreciated when we look at an alternative formula for the dot product:

$$\mathbf{A} \bullet \mathbf{B} = \cos(\theta) |\mathbf{A}| |\mathbf{B}|$$

The value returned by the dot product of two vectors is equal to the cosine of the angle between those two vectors multiplied by their magnitudes. So we can find the cosine of the angle between two vectors by doing the following:

$$\cos(\theta) = \mathbf{A} \bullet \mathbf{B} / |\mathbf{A}| |\mathbf{B}|$$

When the two vectors are unit vectors then the equation is simplified because the length of both vectors equates to 1. This allows us to eliminate the magnitudes and simplify the procedure:

$$\cos(\theta) = \mathbf{A} \bullet \mathbf{B}$$

Plugging the cosine of the angle into the *acos (inverse cosine)* function, we quickly find the actual angle between the two vectors (expressed in radians).

We can write a generic ‘angle determination’ function which accepts two vectors and returns the angle between them. Unfortunately finding the angle between two vectors in this way involves first finding the length of the vectors. This is not a particularly fast process because it involves three multiplies, three additions and a square root. For this reason we generally try to use *unit vectors* wherever possible because it simplifies and speeds up the calculation. The function below shows how one might implement a dot product procedure:

```
float VectorDotProduct3D (CVector A , CVector B)
{
    return (A.x * B.x + A.y * B.y + A.z * B.z);
}
```

If the two vectors are of unit length, this function will return the cosine of the angle between the two vectors.

If you need to find the angle between two vectors and they are not assured to be unit length then you could write an ‘angle finding’ function which automatically handles the division of the dot product by the vector magnitudes:

```
float FindVectorAngles3D (CVector A, CVector B)
{
    float LengthOfA = VectorLength3D ( A );
    float LengthOfB = VectorLength3D ( B );
    return acos ( (A.x*B.x + A.y*B.y + A.z*B.z) / (LengthOfA * LengthOfB) );
}
```



Fig 1.31 shows how the dot product works when determining the angle between two 2D vectors.

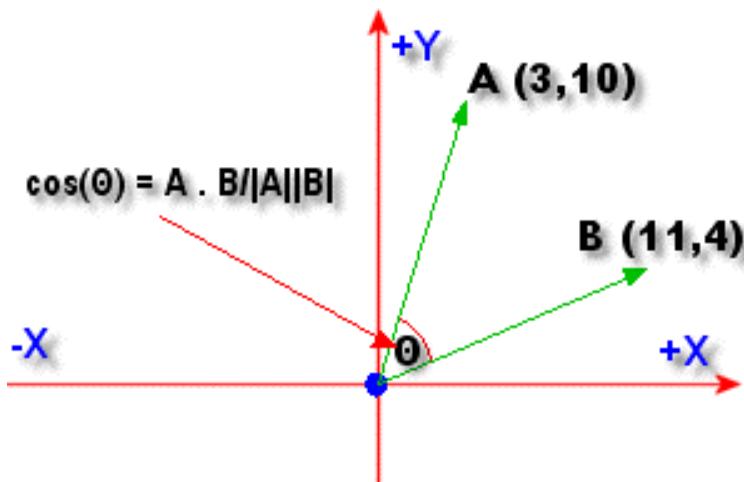


Figure 1.31

$$|A| = \sqrt{3^2 + 10^2} = 10.440306$$

$$|B| = \sqrt{11^2 + 4^2} = 11.704699$$

$$\cos(\theta) = (3*11) + (10*4) / (10.440306 * 11.704699)$$

$$\cos(\theta) = 73 / 122.20063$$

$$\theta = \arccos(0.5973782)$$

$$\theta = 0.93056 \text{ radians}$$

$$\theta = 53.31 \text{ degrees}$$

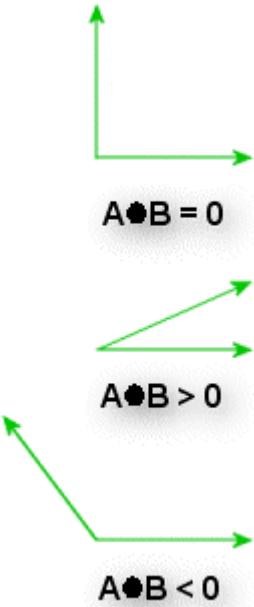
One thing to remember is that the vectors share the same origin during the dot product operation. Think of the process as placing the tail points of each vector at the origin of the coordinate system.

Often we only need to know whether an angle between two vectors is larger or smaller than 90 degrees. If we do not need to know the *actual* angle, then we do not need to use unit vectors or divide by the magnitude. The sign of the result will not change because vector magnitudes are always positive.

There are some important points to note about the dot product between two vectors:

- a. if the angle < 90 degrees the result will be a positive number
- b. if the angle = 90 degrees the result will be zero
- c. if the angle > 90 degrees the result will be a negative number



	<p>This diagram on the left outlines the different relationships between two vectors and the result returned by the dot product operation.</p> <p>If both of the vectors are unit vectors then the result will be the cosine of the angle between the two vectors. You can pass that result into the acos function to determine the actual angle in radians.</p> <p>Remember that both of the source vectors are moved to the origin during the dot product so that their tails share a single point.</p>
---	---

## Planes

A plane is an infinitely thin slice of 3D space that stretches out to infinity in all directions. It is the 3D equivalent of an infinite line in 2D. To visualize a plane, pick up a piece of paper and rotate it to some arbitrary angle (making sure not to bend it). Now imagine that the paper had no edges and in fact went on forever in all directions. Although the plane is infinite it **does** have an orientation in the 3D space.

Rotating your piece of paper to different angles show you an infinite number of different planes. Each orientation change defines a new plane. You could draw a polygon in the center of that piece of paper and you will see that as you rotate the paper the polygon changes orientation too. But it is always on that plane. In fact, polygons are subsets of planes. If you imagine a polygon without any edges so that its area expanded forever in all directions, you would have the plane the polygon is said to lay on.

Planes are useful for many things in 3D graphics development. For example, if we know that a point (say, our camera location) is behind a certain plane then we know that the polygons on that plane are facing away from us and cannot be seen. This allows us to quickly reject polygons that do not require further testing.

Fig 1.32 shows an infinite plane in a 3D Cartesian coordinate system. The red plane is technically infinite but we have taken some liberties to make our plane finite in size for easier viewing. Also depicted is a point that lies on that plane and a vector describing the orientation of the plane (shown in green). This vector is called the **plane normal**. Like the polygon normal discussed earlier in the lesson, the plane normal is also a unit length vector. It describes the orientation of the plane.

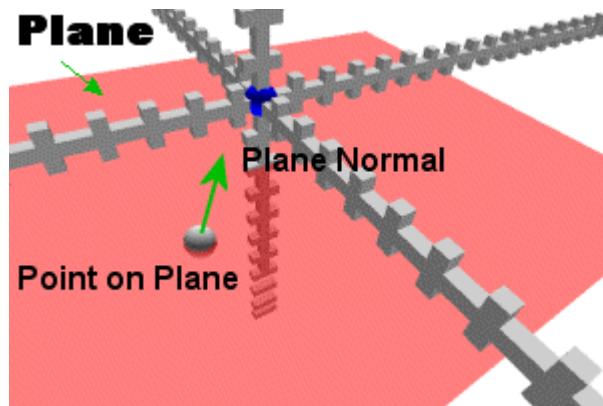


Figure 1.32

A typical class which might be used to store a plane is:

```
class CPlane
{
public:
    CVector PlaneNormal;
    float DistanceToPlane;
};
```

The `PlaneNormal` member variable is a unit length vector that describes the orientation of the plane. The `DistanceToPlane` variable is the distance to the plane as measured from the origin to the closest point on the plane (the black dotted line in the graphic). It can be determined by tracing a line from the origin to the plane in the direction of the plane normal (Fig 1.33).

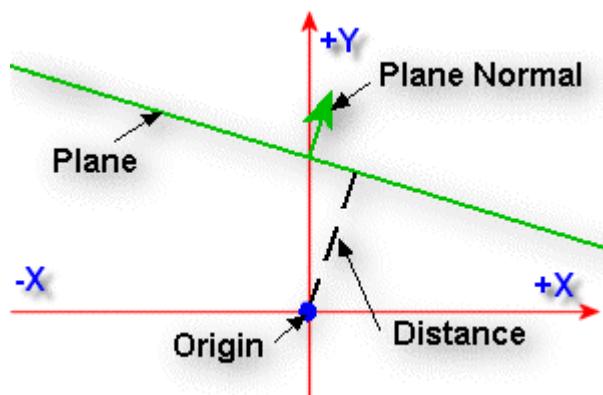


Figure 1.33

Fig 1.33 represents a cross section of the 3D world as if we were looking at the plane from the ‘side’. In two dimensions the plane looks like an infinite line, but it would have an infinite depth coming out and going into the page.

Because the plane normal is facing away from the origin, the origin is said to be *behind* the plane. In this case the distance to the plane will be a *positive* distance value. If the plane is facing the origin then the origin is said to be *in front* of the plane and the distance will be *negative*.

To find the plane on which a polygon lies (and determine the plane normal) we would calculate a normal for the polygon using the cross-product of the edges as described earlier. We could then copy this data directly into the plane structure. To calculate the distance the dot product will be used.

Recall that the dot product of two **unit** vectors is equal to the cosine of the angle between them. However, when one of the vectors is a **non-unit** vector then the outcome of the dot product will equal the cosine of the angle between them multiplied by the length of the non-unit vector.

Figure C. Unit Vector being scaled by Magnitude of non unit vector

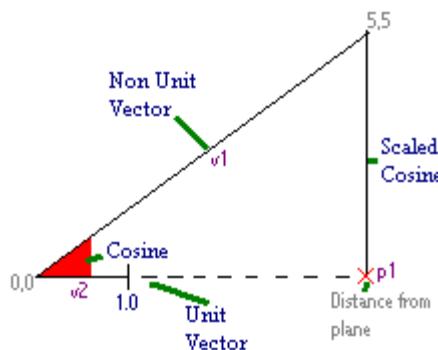


Figure 1.34

In Fig 1.34 the non-unit vector (v1) forms the hypotenuse of a right angled triangle and the unit vector (v2) forms the adjacent leg with a length of 1.0. The cosine is also the length of the adjacent leg of a right-angled triangle.

The result of the dot product of any non-unit vector with a unit vector is the length of the first vector **projected onto the unit vector**. The length of v1 is projected onto v2 in the diagram and results in the length of the adjacent side ((0, 0) to p1). Imagine that the opposite leg of the triangle above is a plane on which the polygon lies and that the (5, 5) coordinate is some point on that plane (any vertex belonging to a polygon will do). v1 is the direction vector from the origin of our coordinate system to the vertex at position (5, 5). Vector v2 is the same as the polygon/plane normal. The result of the dot



product is the shortest distance to the plane ( $p1$ ). Note that this does not tell us the distance to the polygon necessarily, only the distance to the infinite plane on which it lies.

```
CVector PointOnPlane = (5.0, 5.0, 0.0); // Vector from origin to point on plane  
CVector Normal      = (1.0, 0.0, 0.0); // Unit vector (the plane normal)  
  
// p1=distance to the plane  
float DistanceToPlane = VectorDotProduct3D( PointOnPlane, Normal );
```

The result of the dot product is 5. The length of PointOnPlane (v1 in diagram) is:

```
length = sqrtf(5*5,0*0,5*5) ; // we get the answer 7.0710678.
```

We can see in Fig 1.34 that the angle is 45 degrees (it climbs in equal steps along each axis. To be sure this is true we can divide the result of our dot product (which was 5) by the length of the non-unit vector):

```
5 / 7.0710678 = 0.7071067; // gives us the cosine of the angle
```

```
float angle = acosf(0.7071067) = 0.7853981; // 0.78539181 radians = 45 degrees
```

The dot product returns the result we expect by finding the cosine of the angle first which it then multiplies by the length of the non-unit vector.

```
DotProduct = (Cosine of 45 Degrees) * (Length of v1)  
DotProduct = 0.7071067 * 7.0710678  
DotProduct = 5
```

So in order to calculate the distance from the origin to the plane we need two vectors. The first is the plane normal. The second vector is a non-unit vector that starts from the origin and extends to any point known to be on that plane (Fig 1.35). The dot product between these two vectors is our plane distance.

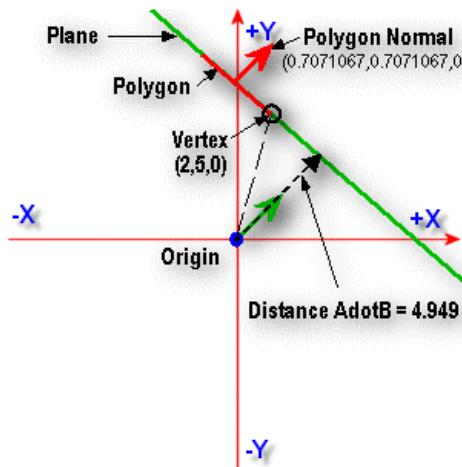


Figure 1.35



We can now write a function that would construct a plane from a polygon. Later we will be using DirectX Graphics helper functions to perform all of these calculations but it is worth understanding the mathematics happening under the hood.

```
CPlane GetPolygonPlane( CPolygon & P )  
{  
    CPlane Plane;  
  
    // Calculate polygon Normal by performing cross product on two of the  
    // polygons edges  
    CVector Edge1 = P.Vertices[1] - P.Vertices[0];  
    CVector Edge2 = P.Vertices[3] - P.Vertices[0];  
    CVector Normal = VectorCrossProduct( Edge1, Edge2 );  
  
    // normalize this so it is unit length. We now have our plane normal  
    Plane.PlaneNormal = VectorNormalize3D (Normal);  
  
    // Perform dot product between ANY vertex in the polygon and the plane normal  
    // to get distance  
    Plane.DistanceToPlane = VectorDotProduct3D( P.Vertices[0], Plane.PlaneNormal );  
    return Plane;  
}
```

Knowing the plane of a polygon also allows us to determine whether it is facing away from the viewer. We can do this test in world space and thus avoid transforming the vertices through the entire pipeline only to be rejected in screen space (where it would have a counter-clockwise vertex winding order).

#### Back Face Removal In World Space

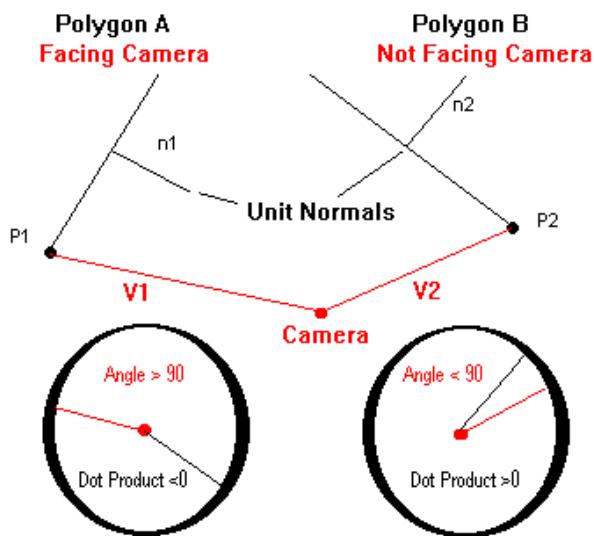


Figure 1.36

Fig 1.36 shows the dot product between the camera position in world space and a polygon plane. We create a vector from the camera position to any point on the polygon plane (any vertex of the polygon will do) and perform the dot product on this vector and the polygon plane normal. This gives us the distance from the camera to the polygon plane.



In the circular inset below polygon A, when the polygon normal and the vector V1 (created by subtracting any vertex in the polygon from the camera's position) have the dot product performed between them the two vectors create an angle that is larger than 90 degrees. Since the  $n1 \bullet V1$  result is a negative number the polygon is facing the camera. This holds true with Polygon B as it is clearly facing away. The two vectors  $n2 \bullet V2$  create an angle smaller than 90 degrees and the result is a positive number.

## Matrices

A matrix is a table of values arranged in rows and columns. The table can be of any dimensions. Below we can see an example of a 3x3 matrix:

$$\begin{bmatrix} 5 & 4 & -1 \\ 0 & 5 & 29 \\ 11 & -7 & 2 \end{bmatrix}$$

Access to the matrix (and indeed all matrices) must be done in a consistent manner. Some math texts use a **row major** addressing approach. A matrix position of [2][3] means that we are referring to the value in [Row 2][Column 3]. In the above example you can see that this is number '29'. Others use a **column major** system where the same [2][3] reference would describe the element in the above table that contains '-7' as its value. Interpreting a matrix element description using the wrong system returns an incorrect result.

**Note:** DirectX Graphics uses the row major system for accessing matrix elements.

The matrix in the above example is a special type of matrix called a **square matrix**. This means that it has as many rows as it does columns. Matrices can be of any size however. The following matrix is an example of a 3x1 matrix, because it has 3 rows but only one column:

$$\begin{bmatrix} 3 \\ 9 \\ 3 \end{bmatrix}$$

The following matrix is a 1x3 matrix:

$$\begin{bmatrix} 3 & 9 & 3 \end{bmatrix}$$

We can replace the numbers in each element with some variables labeled x, y and z:

$$\begin{bmatrix} x & y & z \end{bmatrix}$$



This matrix looks identical to our 3D vector. In fact, you can think of a 3D vector as being a matrix of dimensions  $1 \times 3$ .

A matrix having  $m$  rows and  $n$  columns is referred to as an  $m \times n$  matrix (*order  $m \times n$* ).

A matrix is usually referred to with a capital letter such as ‘M’ or ‘A’. Each element in that matrix has its own address that describes the location of the element using **double suffix notation**. Each address contains three parts: the letter that describes the matrix, the first number indicating the row, and the second number indicating the column:

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

**Note:** In some textbooks and in some code implementations the labeling scheme is zero based for both rows and columns. This means that  $m_{11}$  would be referred to as  $m_{00}$  and  $m_{32}$  would be  $m_{21}$ . If you are converting code from source that uses the  $m_{00}$  based convention, you will need to add ‘+1’ to each label:  $m_{00} = m(0+1)(0+1) = m_{11}$ .

## Matrix/Matrix Multiplication

Two matrices can be multiplied together if and only if they share the same inner dimension.

**Matrix A (3x3) \* Matrix B (3x6). The inner dimensions are A 3x3 \* B 3x6 (OK)**

Above you see that when we line up the four matrix dimensions we get: 3x3 3x6. The inner dimensions of both matrices do indeed match and these matrices can be multiplied. The following matrices could not be multiplied together:

**Matrix A (3x3) \* Matrix B (6x3). The inner dimensions are A 3x3 \* B 6x3 (NOT OK)**

---

**Inner Dimension Rule:** Matrices can only be multiplied when the number of columns in the first is equal to the number of rows in the second.

If two matrices have a matching inner dimension they can be multiplied to create a resulting matrix with dimensions equal to their outer dimensions. A 5x8 \* B 8x16 would result in a matrix of dimensions 5x16.



Matrix multiplication is easier to understand when we look at some reference tables. In the next example, we want to multiply two square matrices A and B:

Matrix A			Matrix B		
a11	a12	a13	b11	b12	b13
a21	a22	a23	b21	b22	b23
a31	a32	a33	b31	b32	b33

We treat each **row** in the Matrix A as a vector and each **column** in Matrix B as a vector. In this example since we are using 3x3 matrices each vector is a 3D vector. Again, vectors can be 4D, 5D,...nD, etc. so this works with any size matrices that can be multiplied. Matrix A consists of 3 row vectors: Vector 1 (a11, a12 ,a13), Vector 2 (a21 , a22 , a23) and Vector 3 (a31, a32 , a33). Matrix B also consists of 3 vectors: Vector 1 (b11, b21, b31), Vector 2 (b12, b22, b32) and Vector 3 (b13, b23, b33). In order to calculate our resultant matrix we need to calculate the value for each element in the output matrix. Our resulting matrix will be called **M** as shown below:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

We begin by calculating the value that will be stored at position **m11**. Because this element is in row 1 and column 1 of the resultant matrix the value stored here will be the dot product • of Vector 1 (1<sup>st</sup> row) in Matrix **A** with Vector 1 (1<sup>st</sup> column) in Matrix **B**. The double suffix notation of the element you are calculating in the resultant matrix describes which rows from Matrix **A** to dot with the columns from Matrix **B**. So the value of **m32** would be calculated like this:

**Blue Row (A) • Green Column (B)**

Since we are calculating the result of address [3][2], we dot the [3] vector of Matrix A with the [2] vector of matrix B. This same multiplication is carried out to compute every cell in the resulting matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} aRow1 \bullet bCol1 & aRow1 \bullet bCol2 & aRow1 \bullet bCol3 \\ aRow2 \bullet bCol1 & aRow2 \bullet bCol2 & aRow2 \bullet bCol3 \\ aRow3 \bullet bCol1 & aRow3 \bullet bCol2 & aRow3 \bullet bCol3 \end{bmatrix}$$

The dot product notation allows us to write the entire multiplication for the resulting 3x3 matrix in a shorthand way. Look at element [1][1] in the resulting matrix. We can write



*aRow1 • bColl*

-or-

*a11 \* b11 + a12 \* b21 + a13 \* b31*

Some source examples do matrix multiplication using only a couple of lines of code. This is possible by nesting for/next loops. We will not do that here. Matrix multiplication should be fast because it may be done hundreds of times per frame. We prefer to avoid the loop logic processing so we will unroll the loops to create a longer but typically faster function. This will also make it easier to see the dot products being performed between the columns and rows. The following source code multiplies two 4x4 matrices together. As we will discuss later, DirectX Graphics works almost exclusively with 4x4 matrices.

```
void MatrixMultiply(MATRIX &result, MATRIX &a, MATRIX &b )
{
    result.m11 = a.m11*b.m11 + a.m12*b.m21 + a.m13*b.m31 + a.m14*b.m41;
    result.m12 = a.m11*b.m12 + a.m12*b.m22 + a.m13*b.m32 + a.m14*b.m42;
    result.m13 = a.m11*b.m13 + a.m12*b.m23 + a.m13*b.m33 + a.m14*b.m43;
    result.m14 = a.m11*b.m14 + a.m12*b.m24 + a.m13*b.m34 + a.m14*b.m44;

    result.m21 = a.m21*b.m11 + a.m22*b.m21 + a.m23*b.m31 + a.m24*b.m41;
    result.m22 = a.m21*b.m12 + a.m22*b.m22 + a.m23*b.m32 + a.m24*b.m42;
    result.m23 = a.m21*b.m13 + a.m22*b.m23 + a.m23*b.m33 + a.m24*b.m43;
    result.m24 = a.m21*b.m14 + a.m22*b.m24 + a.m23*b.m34 + a.m24*b.m44;

    result.m31 = a.m31*b.m11 + a.m32*b.m21 + a.m33*b.m31 + a.m34*b.m41;
    result.m32 = a.m31*b.m12 + a.m32*b.m22 + a.m33*b.m32 + a.m34*b.m42;
    result.m33 = a.m31*b.m13 + a.m32*b.m23 + a.m33*b.m33 + a.m34*b.m43;
    result.m34 = a.m31*b.m14 + a.m32*b.m24 + a.m33*b.m34 + a.m34*b.m44;

    result.m41 = a.m41*b.m11 + a.m42*b.m21 + a.m43*b.m31 + a.m44*b.m41;
    result.m42 = a.m41*b.m12 + a.m42*b.m22 + a.m43*b.m32 + a.m44*b.m42;
    result.m43 = a.m41*b.m13 + a.m42*b.m23 + a.m43*b.m33 + a.m44*b.m43;
    result.m44 = a.m41*b.m14 + a.m42*b.m24 + a.m43*b.m34 + a.m44*b.m44;
}
```



## Vector/Matrix Multiplication

A 3D vector can be treated like a 1x3 matrix. When we multiply a 3D vector with a 3x3 matrix the result is another 1x3 matrix. That is, we get back another 3D vector.

Matrix A

Matrix B

Matrix C

$$[x \ y \ z] \times \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = [aRow1 \bullet bCol1 \ aRow1 \bullet bCol2 \ aRow1 \bullet bCol3] = [X \ Y \ Z]$$

Vector  $\langle x, y, z \rangle$  is transformed into vector  $\langle X, Y, Z \rangle$  by the multiplication.

When a vector is multiplied by a matrix we are actually feeding that vector into an equation and getting back a transformed result. This is very useful because we will need to perform transformations on our 3D vertices. We need to scale them, move them and rotate them to transform them from one coordinate system to another (ex. model space to world space). If we have the equations required to do these operations then we simply need to set up some matrices to hold them.

We will give each object in our game world a matrix to describe its orientation about all three axes and its position in the world. The local space vertices of the object mesh can then be transformed into world space by multiplying each by this matrix. This will clean up our pipeline a fair bit, from a coding perspective, and it will be much faster to execute.

Let us suppose that we have a 2D vector  $\langle x, y \rangle$  that we want to rotate by an arbitrary angle  $\theta$  around the origin to get a transformed 2D vector  $\langle X, Y \rangle$ . Recall from our earlier discussion that we can imagine a Z axis running through the origin for rotation purposes.

### Formula for Rotation around the Z Axis

$$\begin{aligned} X &= x * \cos(\theta) - y * \sin(\theta) \\ Y &= x * \sin(\theta) + y * \cos(\theta) \end{aligned}$$

These input values could be represented by a [1][2] matrix called V.

$$V = [x \ y]$$

Because two values are calculated (x and y) our output vector will be a 2D vector as well (consisting of  $X$  and  $Y$ ). We will call this vector C.

$$C = [X \ Y]$$



Our input vector has 2 columns so we know that our multiplication matrix must have 2 rows (the Inner Dimension rule). Because we need the output matrix C to contain 2 columns, our multiplication matrix must then be a square [2][2] matrix. We will call this matrix M.

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

Then  $V * M = C$  using the above matrices:

$$\begin{bmatrix} V \\ [x \ y] \end{bmatrix} \times \begin{bmatrix} M \\ m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} C \\ X = V \bullet mCol1 \quad Y = V \bullet mCol2 \end{bmatrix}$$

The long-hand form sheds more light on how we might represent our rotation. X and Y below are the X and Y elements of the 2D output vector (matrix C). Remember that column 1 is used for calculating the X component of the output vector and column 2 is used to calculate the Y component in the output vector.

$$\begin{aligned} X &= x * m_{11} + y * m_{21} \\ Y &= x * m_{12} + y * m_{22} \end{aligned}$$

Now look again at our rotation formula:

$$\begin{aligned} x &= x * \cos(\theta) - y * \sin(\theta) \\ y &= x * \sin(\theta) + y * \cos(\theta) \end{aligned}$$

The similarities should be clear. Let us look at the Y calculation first:

$$\begin{aligned} Y &= x * m_{12} + y * m_{22} \quad // \text{Matrix Calculation of } Y \\ Y &= x * \sin(\theta) + y * \cos(\theta) \quad // \text{Rotation Formula for } Y \end{aligned}$$

We can replace **m12** in our matrix with  $\sin(\theta)$  and **m22** with  $\cos(\theta)$ :

$$M = \begin{bmatrix} m_{11} & \sin(\theta) \\ m_{21} & \cos(\theta) \end{bmatrix}$$

The same is also true for X . We calculate X (in vector C) by doing this:

$$x = x * m_{11} + y * m_{21}$$

Compared to our rotation formula of:

$$x = x * \cos(\theta) - y * \sin(\theta)$$



Because the signs are different we can rearrange terms to get:

$$x = x * \cos(\theta) + y * -\sin(\theta)$$

Thus:

$$\begin{aligned} x &= x * m11 + y * m21 && // \text{Matrix Calculation of } x \\ x &= x * \cos(\theta) + y * -\sin(\theta) && // \text{Rotation Formula for } x \end{aligned}$$

We can now replace **m11** with **cos(θ)**, and **m21** with **-sin(θ)**. The final matrix M contains both of our transformations (X and Y):

$$M = \begin{bmatrix} X & Y \\ \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

Matrix M will transform the **x** and **y** coordinates of an input vector to a rotated **X** and **Y** in an output vector. One important benefit here is that we can initialize the matrix once, calling **cos** twice and **sin** twice (or once if we use a local variable) and storing the values in the matrix. Then we could multiply thousands of vectors by this matrix to transform them without having to call **cos** and **sin** to transform every vertex as we did in our earlier code examples.

Rotation is the same in 3D. The only difference is that we use a 3D vector and a 3x3 matrix:

#### Matrix to Rotate 3D Vector V around the Z axis by θ Radians.

$$V = [x \ y \ z] \times M = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = C = [V \bullet mCol1 \ V \bullet mCol2 \ V \bullet mCol3]$$

Very little has changed going to 3D because when a rotation around the Z axis occurs, only the X and Y values of a vector are actually modified by the rotation. This means that we want C[Z] to be the same as V[z]. You can think of the 3<sup>rd</sup> column of the matrix M above, as being the vector that produces the transformed Z component in the output vector.



## 3D Rotation Matrices

### X Axis Rotation

$$NewY = OldY \times \cos(\theta) - OldZ \times \sin(\theta)$$

$$NewZ = OldY \times \sin(\theta) + OldZ \times \cos(\theta)$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

### Y Axis Rotation

$$NewX = OldX \times \cos(\theta) + OldZ \times \sin(\theta)$$

$$NewZ = OldX \times -\sin(\theta) + OldZ \times \cos(\theta)$$

$$\begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

### Z Axis Rotation

$$NewX = OldX \times \cos(\theta) - OldY \times \sin(\theta)$$

$$NewY = OldX \times \sin(\theta) + OldY \times \cos(\theta)$$

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

## Identity Matrices

You might think that because we do not need Z to change during the earlier rotation transformation that we could simply fill the 3<sup>rd</sup> column of M with zeros. This is not so. The Z component in the output vector is computed as follows:

$$z = x*m13 + y*m23 + z*m33$$

If the input vector contained a value of  $z = 10$  then we would want to make sure that this value made it through the z axis rotation transformation unmodified. The output vector must also have  $Z = 10$ . If we had filled the last column of matrix M with zeros, we would have computed the output as follows:

$$z = x*0 + y*0 + z*0 = 0$$

What we really want to do is copy the value into the output vector. By placing a '1' in **m33**, the Z calculation now becomes:

$$\begin{aligned} z &= x * 0 + y * 0 + z * 1 = z \therefore \\ z &= x * 0 + y * 0 + 10 * 1 = 10 \end{aligned}$$



This new column in the matrix is called an ***identity column*** because the value it outputs is the same as its input. Using this knowledge we can create a special type of matrix known as an ***identity matrix*** which is the matrix equivalent of the number 1:

$$\text{Identity Matrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Given a vector  $V=[x\ y\ z]$  and an Identity Matrix  $I$ , by multiplying  $V * I$  we should get a resulting vector  $C[X\ Y\ Z]$  such that  $C = V$ :

$$V = [x\ y\ z] \times I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = C = [x*1 + y*0 + z*0 = x \quad x*0 + y*1 + z*0 = y \quad x*0 + y*0 + z*1 = z]$$

Losing the zeroed out values we are left with:

$$C = [x*1, y*1, z*1]$$

### Scaling and Shearing Matrices

The identity matrix is a matrix that multiplies an input vector by one. We can expand this concept to build a matrix that multiplies vectors by other values as well. The result is a **uniform scaling matrix** that replaces the 1's for some other amount by which you wish to scale the vector. For example, if you wanted to scale all vectors by 10, the scaling matrix looks like this:

$$10X \text{ ScaleMatrix} = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$

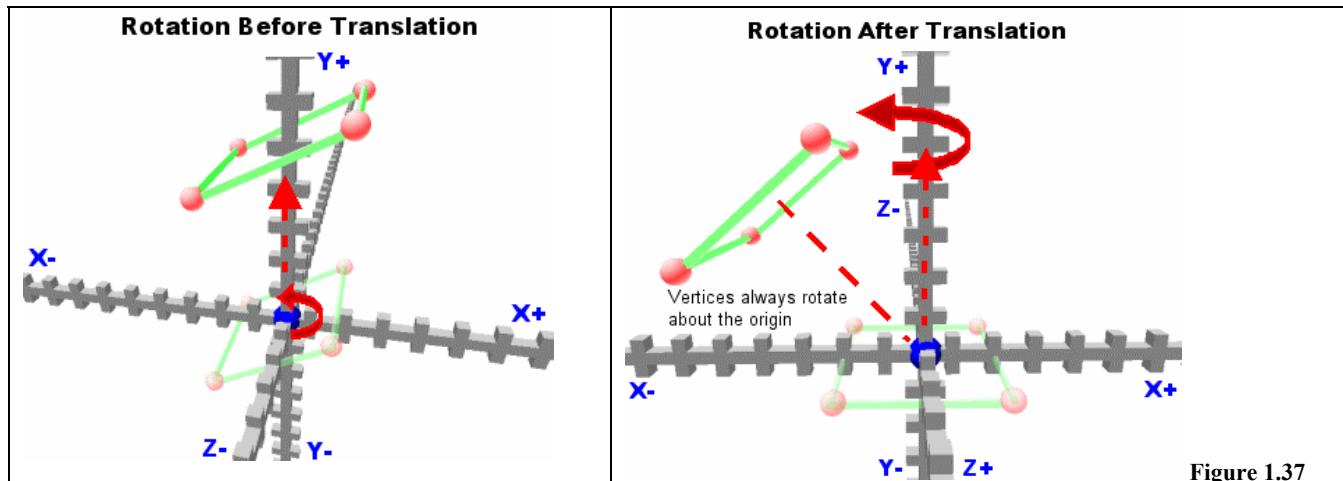
If you multiplied all of the vertices of a mesh by the above matrix, the object would become 10x bigger. Note that we can also create a matrix for **non-uniform scaling** along individual axes. It is called a ***shearing matrix***:

$$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 20 & 0 \\ 0 & 0 & 35 \end{bmatrix}$$

## Matrix Concatenation

Matrix multiplication is **associative** ( $A(BC) = (AB)C$ ). So if matrix A rotates points around the Z axis and matrix B rotates points around the Y axis, they can be combined (**concatenated**) into a single matrix M that does the work of both. Thus for vector V to have both transformations applied to it, rather than doing  $V * A = C$  and then  $C * B = D$ , we will take a different approach. Instead we will do  $M = A * B$  first. This allows us to use  $V * M = D$  and get the same effect as  $V * A = C$ ,  $C * B = D$ . Concatenating matrices like this means that you can have many different matrices, each of which performs its own transformation, and combine them into a final matrix using matrix multiplication. We can now multiply a vector by this final matrix and it is completely transformed in one pass. This is very efficient.

Matrix multiplication is not **commutative**. ( $A * B \neq (B * A)$ ). That is, multiplication order is significant. This should sound familiar. Earlier in this chapter we saw the effects of rotating vertices before translating them and the very different results when we saw when translating first and then rotating. Rotating the mesh before translating it gave the appearance that the object was rotating around its own center point (often the desired effect). When we translated the object into world space first and then follow with a rotation, the vertices were rotated around the new world space origin rather than its own (Fig 1.37). Since we will use matrices to store these types of transformations, the order in which matrix multiplication is performed will be significant to us.



## Homogeneous Coordinates

A vector is multiplied with a matrix by performing a series of row/column multiplications. You might conclude then that matrices can only be used for linear transformations (where the output is proportional to the input). A good example is the identity matrix, which maps the source vector to the output vector. Scaling and rotating are certainly linear transformations as the output represents some



multiple of the input. But what if we wanted to build a matrix that always produced an X component in the output vector of 5 regardless of the input vector?

$$[x \ y \ z] \times \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = [X \ Y \ Z]$$

Ignore the Y and Z components of the output vector for now, and just concentrate on how the X component in the output vector is calculated:

$$X = x * m_{11} + y * m_{21} + z * m_{31}$$

Because  $x$ ,  $y$  and  $z$  are all used to create the resulting  $X$  component, there is no way to fill in an element of our matrix that would always result in  $X = 5$ .

So, if our object is supposed to be positioned at world space coordinate (50, 70, 10) we would need to build a matrix that *translates* all the vertices in its mesh by 50 along the x axis, 70 along the y axis and 10 along the z axis.

It would seem the only solution would be to handle the linear transformations first using a 3x3 matrix and then translate that result separately like so:

$$\mathbf{V}' = \mathbf{MV} + \mathbf{T}$$

Where  $\mathbf{V}$  and  $\mathbf{V}'$  represent the input and output vectors respectively,  $\mathbf{M}$  is a 3x3 rotation matrix, and  $\mathbf{T}$  is a translation vector (a 1x3 matrix).

While this approach would certainly work, we would much prefer to get the job done with a single matrix multiplication. But how do we build a matrix that will **add** 50 to the X, 70 to the Y and 10 to the Z of the input vector, regardless of the vector's initial input value? In the matrix above you should be able to see that this is just not possible. Take a look at column 1 in the matrix, which is responsible for the output of the X component. There is no way to substitute  $m_{11}$ ,  $m_{21}$  or  $m_{31}$  for any value that would simply ADD 50 to the x value for example. A solution is required and that solution is homogeneous coordinates.

Although human beings have difficulty visualizing more than three dimensions, in mathematics many dimensions can and do exist. These 'hidden' dimensions (specifically the fourth dimension in this case) provide us with an interesting mathematical method for solving the translation problem. Have a look at a four dimensional vector. A 4D vector is perhaps very much like you would expect (a 1x4 matrix):

$$[x \ y \ z \ w]$$



The 4D vector above has an  $x$ ,  $y$  and  $z$  component just like our 3D vector and it also has a fourth component labeled  $w$ . If we divide a 4D vector by its own  $w$  component like so we can map back into 3D space:

$$[x/w \quad y/w \quad z/w \quad w/w]$$

In fact any 4D vector where  $w=1$ , maps directly to 3d space:

$$[x/1 = x \quad y/1 = y \quad z/1 = z \quad w = 1]$$

This type of vector is known as a **homogeneous coordinate**. The operation of dividing  $x$ ,  $y$  and  $z$  by  $w$  is called **homogenizing** the coordinate. This projects a vector from dimension  $N$  to dimension  $N-1$ . Homogeneous coordinates do not apply to 4D vectors only. They exist in every dimension N. If we wanted to homogenize a 3D vector  $V$ :

$$\mathbf{v} = (\mathbf{x}, \mathbf{y}, \mathbf{z})$$

We can do this by dividing  $x$  and  $y$  by  $z$  like so:

$$\mathbf{v} = (\mathbf{x}/\mathbf{z}, \mathbf{y}/\mathbf{z}, \mathbf{z}/\mathbf{z})$$

This resulting vector  $R$  looks like:

$$\mathbf{r} = (\mathbf{x}, \mathbf{y}, 1)$$

The operation projected the vector from 3D space into 2D space. You may recall this technique from our earlier discussion. Dividing the  $X$  and  $Y$  components of a 3D point by the  $Z$  component was the formula we used to perspective project a 3D point onto a 2D plane located at  $z = 1$ .

So an  $N$  dimensional vector can be **homogenized** by dividing its components by the **Nth** component of that vector. This projects the vector from dimension  $N$  to dimension  $N-1$ .

Using a homogeneous 4D vector we can ignore the  $w$  component since it always equals 1. To be sure, there will be times when we work with 4D coordinates where the  $w$  component does not equal 1, and we will discuss those cases later in the course.

How does this help us solve the problem of representing a non-linear translation transformation in a matrix? The key is found in the idea that  $w = 1$ . Below, we see a 4D vector  $V$  multiplied with a 4x4 identity matrix  $I$  to create resulting 4D vector  $R$ . Notice that a new column is needed to compute  $w$  in the matrix. If  $w=1$  (as we know it does) then multiplying by this matrix also results in an output vector where  $w=1$ .



$$V = [x \ y \ z \ w=1] \times I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = R[X \ Y \ Z \ W=1]$$

With this knowledge we can create a new 4x4 matrix storing the equation for rotation around the Z axis:

#### **Matrix for Rotation around the Z Axis**

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix is essentially the same as it was before with the exception being that the last two columns now have been set to identity columns. Remember that we only want the  $x$  and  $y$  values of the input vector to be affected by the rotation operation. The  $z$  and  $w$  values of the input vector should be copied over into the output vector unchanged. The new identity column above would copy the  $w$  component from the input vector into the  $W$  component of the output vector unchanged as expected:

$$W = x * 0 + y * 0 + z * 0 + w * 1 = w$$

Note as well that we have added an extra row to our matrix. This is done firstly in order to allow us to multiply vector[1][4] with the matrix[4][4] (the inner dimension rule). It will also allow us to represent translation in our matrices. Take a look at the two matrices below, which demonstrate the multiplication of our vector V with a matrix M:

$$V[x \ y \ z \ 1] \times M \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} = R[X \ Y \ Z \ 1]$$

To calculate element X in output vector R:

$$X = Vx*m_{11} + Vy*m_{21} + Vz*m_{31} + 1*m_{41}$$

We know that  $w=1$  so the last portion of the calculation for  $X$  will always be “+  $1*m_{41}$ ”. In other words, whatever value we put into element  $m_{41}$  will be used as an addition operation:



The same also holds true for both the y and z columns in the matrix. Any value we store in m42 will be directly added to the y component of the input vector V and any value we store in element m43 will be added to the z component of input vector V. So the fourth row, along with the homogeneous coordinate, lets us represent translation in a matrix because together they give us the benefit of a required addition operation. You should think of the fourth row as a separate section of the matrix that does not scale the input vector like the upper 3x3 portion does. It will be used to add or subtract values to the components of the input vector.

With this knowledge, we can now create a matrix that would translate a vector **TX** along the X axis, **TY** along Y the axis and **TZ** along the Z axis like so:

**Translation Matrix**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ TX & TY & TZ & 1 \end{bmatrix}$$

For example: TX=200, TY=0 and TZ=-50. Input vector **V=( 200 , 70 , 500 )**. We will need to represent this vector as a 4D homogeneous coordinate so **V=( 200 , 70 , 500 , 1 )**.

$$V = [200 \ 70 \ 500 \ 1] \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 200 & 0 & -50 & 1 \end{bmatrix} = R = [X \ Y \ Z \ 1]$$

Calculating vector R ourselves:

$$Rx = 200 * 1 + 70 * 0 + 500 * 0 + 1 * 200 = 200 * 1 + 1 * 200 = 400 \ (Rx=400)$$

$$Ry = 200 * 0 + 70 * 1 + 500 * 0 + 1 * 0 = 70 * 1 = 70 \ (Ry=70)$$

$$Rz = 200 * 0 + 70 * 0 + 500 * 1 + 1 * -50 = 500 * 1 + 1 * -50 = 450 \ (Rz=450)$$

$$Rw = 200 * 0 + 70 * 0 + 500 * 0 + 1 * 1 = 1 * 1 = 1 \ (Rw=1)$$

$$\therefore R = [400 \ 70 \ 450 \ 1]$$

Our point has been successfully transformed by the matrix. This 4x4 translation matrix can be combined with other matrices. We can concatenate a rotation matrix and a translation matrix into a single matrix and pump all of our vectors through it.



### Matrix for Z Axis Rotation and Translation

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ TX & TY & TZ & 1 \end{bmatrix}$$

We can hardcode the fact that input vector W components will always equal 1 in a function. This is a shortcut we can safely make for a function that works explicitly with 3D vectors and matrices that consist of translations, rotations and scaling. Below, we see a function called VectorMatrixMultiply that takes a 3D vector and stores it as a 4D vector internally in order to carry out the calculation. The result is then homogenized to make sure we return a 3D vector. This is why we are dividing x, y and z by w. This is a handy function for multiplying the mesh's model space vertices to world space or view space when we know that we want a resulting vector where w = 1.

```
BOOL VectorMatrixMultiply( VECTOR3D& vDest, VECTOR3D& vSrc, MATRIX& mat)
{
    FLOAT x = vSrc.x*mat.m11 + vSrc.y*mat.m21 + vSrc.z*mat.m31 + mat.m41;
    FLOAT y = vSrc.x*mat.m12 + vSrc.y*mat.m22 + vSrc.z*mat.m32 + mat.m42;
    FLOAT z = vSrc.x*mat.m13 + vSrc.y*mat.m23 + vSrc.z*mat.m33 + mat.m43;
    FLOAT w = vSrc.x*mat.m14 + vSrc.y*mat.m24 + vSrc.z*mat.m34 + mat.m44;

    // Prevent Divide by 0 case.
    if( fabsf( w ) < g_EPSILON ) return FALSE;

    // Homogenize the coordinate.
    vDest.x = x / w;
    vDest.y = y / w;
    vDest.z = z / w;

    return TRUE;
}
```

Some matrix functions *seem* to multiply a 3D vector with a 4D matrix but we know this can not be done because of the inner dimension rule. Of course, what these functions are doing is similar to what we are doing here: explicitly treating the 3D vector as a 4D vector where w always equals 1. There are times when you do not want a function to homogenize the result and therefore the more generic 4D vector/4D matrix multiplication would be used.

For each component calculated in the output vector we *add* the 4<sup>th</sup> row values from the matrix. We then homogenize the 4 components back into 3D space by **dividing by w**. This allows us to return a 3D space vector. You could alter this code to return a 4D vector instead where the divide by w could be performed elsewhere. Another point to note is that if you know for sure that the 4<sup>th</sup> column (the w column) of the matrix being multiplied is an identity column, then the resulting w will still equal 1 and the divisions by w are not needed. This is often the case when dealing with vertex positions.



## Quaternions

A quaternion is a complex number defined by a 3D directional vector  $\mathbf{v} = \langle x, y, z \rangle$  and a scalar value  $s = w$ . A unit length quaternion provides an alternative format for storing rotation data. The direction vector  $\mathbf{v}$  represents an arbitrary axis around which rotation will occur. It is stored as a unit vector that has been scaled by the sin of the half-angle  $\theta$  (the rotation angle in radians). The scalar value  $w$  completes the rotation data and is defined as the cosine of the half-angle  $\theta$ .

$$\begin{aligned}\mathbf{q} &= (s, \mathbf{v}) = [w, \langle x, y, z \rangle] \\ \mathbf{q} &= [\cos(\theta / 2), |\mathbf{v}| * \sin(\theta / 2)]\end{aligned}$$

Quaternions provide a number of advantages over using matrices to store rotation information. The most obvious benefit is that they occupy less memory (4 floats versus 9 floats). Additionally, concatenation of quaternions is actually a bit faster than matrix concatenation. While vertices cannot be transformed by a quaternion directly, quaternions can be quickly converted to matrix format to accomplish this purpose. As we will see later in the course, they are also a better choice for interpolating between rotations.

To convert a quaternion to 3x3 rotation matrix form we will use the following formula:

$$\mathbf{M} = \begin{bmatrix} 1 - 2y^2 - 2x^2 & 2xy + 2wz & 2xz - 2wy \\ 2xy - 2wz & 1 - 2x^2 - 2z^2 & 2yz + 2wx \\ 2xz + 2wy & 2yz - 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

As a quaternion is a complex number, it is composed of both a real and an imaginary part. For our purposes in this course we do not need to be familiar with complex numbers to use quaternions. Students in the Game Mathematics course will have an opportunity to explore the quaternion and its operations in detail. The formulae above are all we will need to understand how to create and use quaternions in our applications, since we will wind up converting all of our quaternions to matrix form to perform rotation transformations.



## D3DX Math

The DirectX9 SDK ships with a DirectX Graphics helper API called the Direct3D Extensions (D3DX). Among just some of its components it includes numerous mathematical structures and functions that will be of value to us. We can include this functionality in our source code modules by adding `#include <d3dx9.h>` near the top of the source file. Most D3DX classes have overloaded operators and various constructors to make operations easy and intuitive. LP 1.1 will be using D3DX only for its math functions and its matrix and vector structures. This will be advantageous since D3DX math functions may take advantage of CPU capabilities like MMX™ or 3DNow™ when available. As the course progresses we will use D3DX for many other important tasks.

### D3DXMATRIX

DirectX Graphics works exclusively with 4x4 matrices (16 floats). Matrix elements in this class can be accessed in two ways: via a 4x4 array or by using the double suffix notation we are already familiar with. Each member of the matrix can be accessed using the `.row_column` method. This means that we can assign a value to the 3<sup>rd</sup> row and the 2<sup>nd</sup> column using the following code:

```
D3DXMATRIX Mat;  
Mat._32 = f; //f = float value
```

With operator overloading we can perform matrix multiplication, addition and subtraction:

```
D3DXMATRIX mat1, mat2, mat3;  
  
mat3 = mat1 * mat2;      // matrix multiplication  
mat3 = mat1 + mat2;      // matrix addition  
mat3 = mat1 - mat2;      // matrix subtraction
```

There are two constructors worthy of mention. The first initializes the matrix using another matrix passed as a parameter. The second allows us to pass in each of the 16 float values we want placed in each element of matrix. Their definitions are shown below:

#### Constructor 1

```
D3DXMATRIX( CONST D3DMATRIX& );
```

#### Constructor 2

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



## D3DXVECTOR3

The D3DXVECTOR3 stores 3D vectors (3 floats). There are structures for 2D (two floats) and 4D (four floats) vectors also. All contain overloaded operators and constructors for easy initialization. This means we can perform addition, subtract, multiplication, etc. with the standard operators as shown below.

### Construction

```
D3DXVECTOR3 MyVector    ( 20.0f,    50.0f, -10.0f);
D3DXVECTOR3 YourVector  (  0.0f, -200.0f,   35.0f);

MyVector = MyVector + YourVector ;           // Vector Addition
MyVector = MyVector * 5;                     // Vector scalar multiplication
MyVector = MyVector - YourVector;            // Vector subtract
MyVector = (YourVector*2) + MyVector;         // Combination
```

We can access and modify the individual vector elements x, y, or z as follows:

```
MyVector.x += 10.0f;
MyVector.y = YourVector.y;
```

Of course with a D3DXVECTOR2 structure there are only x and y member variables and with the D3DXVECTOR4 structure we have x, y, z and w member variables. The latter can be used to store homogeneous coordinates.

## D3DXPLANE

D3DX also provides a structure for holding plane information (four floats). The first three floats will store the plane normal (x, y and z components). The fourth float will be assigned the distance to the plane from the origin.

```
D3DXPLANE MyPlane;
MyPlane.a = Normal.x;
MyPlane.b = Normal.y;
MyPlane.c = Normal.z;
MyPlane.d = dist;      // Distance to plane from origin.
```

We can make use of the constructor for easy initialization.

```
D3DXPLANE MyPlane(Normal.x, Normal.y , Normal.z , DistanceToPlane);
```

D3DX has many helper functions that can be used to create planes. Using the D3DXPlaneFromPointNormal function you could, for example, create a plane simply by passing in the plane normal and any point known to be on the plane. In the case of using a polygon's plane, we could pass in a plane normal and any one of the polygons vertices.



```
D3DXPlaneFromPointNormal( D3DXPLANE* pOut, CONST D3DXVECTOR3* pPoint,
                           CONST D3DXVECTOR3* pNormal);
```

We pass in a pointer to a D3DXPLANE structure that will receive the final plane and also pointers to two 3D vectors, the plane normal and a point known to be on the plane.

If you do not have access to the polygon normal and want it calculated on your behalf, use the D3DXPlaneFromPoints function. This function can be used to create a plane from any three points known to be on the plane. For example, if you were creating a plane for a polygon, you could pass in three of the polygon's vertices. D3DX would calculate the plane normal and the distance for you, returning the information via the D3DXPLANE structure passed in the pOut parameter.

```
D3DXPlaneFromPoints( D3DXPLANE* pOut, CONST D3DXVECTOR3* pV1,
                      CONST D3DXVECTOR3* pV2, CONST D3DXVECTOR3* pV3 );
```

## D3DXQUATERNION

While a quaternion can be a difficult concept to grasp due to its origins as a complex number, using quaternions is made quite simple through D3DX.

The D3DXQUATERNION structure stores 4 floating point components (x,y,z,w). As mentioned earlier, we will convert quaternions to matrix form before using them for rotation operations. For example, to create a quaternion to store a 45 degree rotation around the X axis, we can do the following:

```
float angle = 0.785f;

D3DXVECTOR3 axis(1,0,0);

axis *= (float)sin(angle/2.0f);

D3DXQUATERNION q;
q.x = axis.x;
q.y = axis.y;
q.z = axis.z;
q.w = (float)cos(angle/2.0f);
```

Before converting the quaternion to a rotation matrix we will want to make sure it is normalized first. D3DX provides a function for this purpose called D3DXQuaternionNormalize. Continuing our example above:

```
D3DXQUATERNION normalized_q;
D3DXQuaternionNormalize(&normalized_q, &q);
```

Now that we have a unit quaternion we can proceed with matrix conversion. D3DX also provides a function for this purpose called D3DXMatrixRotationQuaternion:



```
D3DXMATRIX rotation_matrix;
D3DXMatrixRotationQuaternion (&rotation_matrix, &normalized_q);
```

We now have a rotation matrix constructed from our quaternion. As it turns out, the quaternion math we have looked at uses the right-handed coordinate system. Since DirectX Graphics uses a left-handed system we will need to invert the matrix as a final step before using it in a transform operation. Given the orthogonal nature of our rotation matrix, we can opt for the faster transpose call:

```
D3DXMatrixTranspose(&rotation_matrix, &rotation_matrix);
```

A faster alternative is to invert the quaternion before conversion to matrix form. This operation will conjugate the quaternion and renormalize it, so you could replace the normalization call above and forego the matrix transpose:

```
D3DXQUATERNION normalized_q;
D3DXQuaternionInverse (&normalized_q, &q);

D3DXMATRIX rotation_matrix;
D3DXMatrixRotationQuaternion (&rotation_matrix, &normalized_q);
```

## D3DX Functions

D3DX provides a function to multiply two 4x4 matrices:

```
D3DXMATRIX* D3DXMatrixMultiply( D3DXMATRIX* pOut,
                                CONST D3DXMATRIX* pm1,
                                CONST D3DXMATRIX* pm2 );
```

The function takes the addresses of the two 4D matrices to be multiplied and the address of a matrix which will receive the result of the operation. The multiplication will take advantage of any hardware (CPU) features or optimizations available.

D3DX provides three functions for building specific **rotation** matrices:

```
D3DXMatrixRotationX (D3DXMATRIX* pOut, FLOAT Angle);
D3DXMatrixRotationY (D3DXMATRIX* pOut, FLOAT Angle);
D3DXMatrixRotationZ (D3DXMATRIX* pOut, FLOAT Angle);
```

The functions accept a pointer to a D3DXMATRIX structure and float values that describe the amount of rotation (in radians) we require about that particular axis. For example, if we want to build a matrix that rotates vectors 1.3 radians about the world Y axis we can do the following:

```
D3DXMATRIX RotationMatrixY;
D3DXMatrixRotationY ( &RotationMatrixY , 1.3 );
```



When the function returns, the matrix passed via the pOut parameter will contain the correct values. In this case, the matrix returned internally would look like so:

$$\begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

There is also a function to create a **translation** matrix for positioning our objects in the world:

```
D3DXMatrixTranslation ( D3DXMATRIX* pOut, FLOAT x, FLOAT y, FLOAT z );
```

If we wanted to translate our mesh so that it was positioned at (10, 40, 50) in world space we could do the following:

```
D3DXMATRIX TranslationMatrix;
D3DXMatrixTranslation (&TranslationMatrix , 10 , 40 , 50);
```

Using these functions, we can give each object its own world matrix. When that object is rendered for each frame, its vertices are multiplied by this matrix to transform it into world space. For example, let us say that we have an object that we want to be rotated 2 radians about the Z axis and positioned at (10, 50, 2) in world space. We could build a matrix that would perform this operation by first building the translation matrix, and then combining it with a rotation matrix in order to generate the concentrated matrix. The code to do this is shown below:

```
CObject Object; // assumed to have two members, a mesh and a world matrix
D3DXMATRIX RotMat, TransMat;

// Build the matrices
D3DXMatrixTranslation ( &TransMat , 10, 50, 2);
D3DXMatrixRotationZ ( &RotMat , 2 );

// Set the combined matrix as the object's world matrix
Object.WorldMatrix = RotMat * TransMat;
```

The object now has a single world matrix which completely describes its orientation and position within the 3D world. Using that matrix to transform the vertices into world space, we can move the object around the world simply by altering the matrix values. We might have a function that is called every time the left arrow key is pressed, that builds a rotation matrix around the Y axis by 0.2 radians and then multiplies this with the object's current world matrix. This would cause the object to rotate each time by a further 0.2 radians.

Note that in the above code the matrix multiplication order is significant. Here we are rotating the object about its local origin first and then translating the object into its final world space position.



Reversing the order of the multiplication would produce a translation into the world space position followed by a rotation about the world origin.

The next function allows you to specify the rotations about the X, Y and Z axes with a single call. This would otherwise have to be built using three separate rotation matrices about each of the X, Y and Z axes respectively and then multiplying each of them together.

```
D3DXMatrixRotationYawPitchRoll( D3DXMATRIX* pOut, FLOAT Yaw, FLOAT Pitch,
                                FLOAT Roll );
```

To build a single matrix that rotates 1 radian about the X axis, 2 radians about the Y axis and 0.5 radians about the Z axis and then positions our object at (100,50,-20), we would use the following code:

```
D3DXMATRIX OrientationMat , TranslationMat;

D3DXMatrixRotationYawPitchRoll( &OrientationMat , 2 , 1 , 0.5);
D3DXMatrixTranslation      ( &TranslationMat , 100 , 50 , -20 );

Object.WorldMatrix = OrientationMat * TranslationMat;
```

If you maintain three floats (Yaw, Pitch, Roll) for each object, these can be altered in response to user input and used to build the object's new orientation matrix each time it needs to be updated.

The D3DX library also has functions that allow us to multiply a vector with a matrix. We will need to do this multiply on each of the mesh's vertices using the object's world matrix. There are three functions that concern us and each behaves somewhat differently:

1. 

```
D3DXVec3TransformCoord( D3DXVECTOR3* pOut, CONST D3DXVECTOR3* pV,
                           CONST D3DXMATRIX* pM );
```

This function multiplies a 3D vector with a 4x4 matrix. As we know from our earlier discussion, the function treats the input vector as a homogenous 4D vector in the form (x, y, z, 1). The 4D vector is multiplied by the 4x4 matrix which creates another 4D vector. This function takes care of homogenizing the resulting vector back into 3D space.

This is the function we will use to multiply our object vertices by our object world matrix.

2. 

```
D3DXVec3TransformNormal(D3DXVECTOR3* pOut, CONST D3DXVECTOR3* pV,
                           CONST D3DXMATRIX* pM );
```

This function is provided when the result vector needs to be normalized. For example, let us say that we have a polygon facing down the positive Z axis. The normal for this polygon equals (0, 0, 1). If the polygon were rotated 45 degrees about the Y axis, the normal would also have to be updated. We can



rotate normal vectors just as we do ordinary vectors as long as the unit vector remains a unit vector after the matrix multiplication.

When the matrix contains translation information (which will most likely be the case with the object's world matrix) then the normal will also be translated. As a result, its tip would no longer necessarily be one unit from the origin. So we want to ignore the bottom row of the matrix which stores the translation and only multiply the normal using the upper 3x3 section storing the orientation. The D3DX function does just this.

```
3. D3DXVec3Transform ( D3DXVECTOR4* pOut, CONST D3DXVECTOR3* pV,  
CONST D3DXMATRIX* pM);
```

This function takes a 3D input vector and a matrix and returns a 4D vector. The input vector is treated as a 4D vector in the form (x, y, z, 1). This output vector is in the form (x, y, z, w) where w does not equal 1. Unlike D3DXVec3TransformCoord, this function does not homogenize the result by dividing x, y and z by w. You may need this function if you are required to use a matrix where the fourth column is not an identity column.

The D3DX library also provides functions for performing normalization of vectors, dot products, cross products and functions for returning the length of a vector. Some of these are listed below.

#### Cross Product of two 3D vectors

```
D3DXVec3Cross ( D3DXVECTOR3* pOut, CONST D3DXVECTOR3* pV1,  
CONST D3DXVECTOR3* pV2);
```

Returns a vector perpendicular to A and B in pOut result.

```
D3DXVECTOR3 Result, A, B;  
D3DXVec3Cross( &Result , &A , &B );
```

#### Dot Product of two 3D vectors

```
D3DXVec3Dot ( CONST D3DXVECTOR3* pV1, CONST D3DXVECTOR3* pV2 );
```

Returns cosine of the angle between A and B scaled by vector magnitudes.

```
D3DXVECTOR3 A, B;  
float CosAngle = D3DXVec3Dot( &A , &B );
```

#### 3D vector Magnitude

```
D3DXVec3Length( CONST D3DXVECTOR3* pV );
```

Returns the length of the passed vector.

```
D3DXVECTOR3 A;  
float Length = D3DXVec3Length ( &A );
```

#### Normalizing a 3D vector

```
D3DXVec3Normalize ( D3DXVECTOR3* pOut, CONST D3DXVECTOR3* pV);
```

[www.gameinstitute.com](http://www.gameinstitute.com) Graphics Programming with DX9



Takes a vector pV and makes it Unit Length.

```
D3DXVECTOR3 A;
D3DXVec3Normalize( &A , &A );
```

The functions that return a vector result allow an output vector to be specified. This vector can be used to specify a vector other than the one used for input. This is useful if you do not want the normalized vector to overwrite the input vector. However, in the above example we have passed vector A as both the input and the output, therefore normalizing vector A and storing the result back in vector A.

## The Transformation Pipeline

In our first Lab Project (LP 1.1), three key matrices are used. These matrices combine to perform the initial phase of the transformation pipeline from model space to projection space. After a polygon has passed through each of these three matrices its vertices are ready to be scaled from 2D projection space to 2D screen space as discussed earlier. We will now cover each of the three matrix types, their use, and some interesting facts about them.

### The World Matrix

Each object in our scene will have a world matrix. The world matrix is used to position, scale and orient the object in world space. The first thing our pipeline will do is multiply each of the polygon's vertices with the current object's world matrix. This will transform the polygon from model space into world space. By applying new rotations and translations during each frame of our game, we can animate a 3D object. Our object structure looks like this:

```
class CObject
{
    CMesh      *pMesh;
    D3DXMATRIX WorldMatrix;
};
```

If an object's world matrix has been set as an identity matrix, then we know that the object will not be translated or rotated at all, it will positioned in the world at position (0, 0, 0) and is assumed to face straight down the positive Z axis. Let us look at an identity matrix again for a moment:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{array}{l} LocalXAxis \\ LocalYAxis \\ LocalZAxis \\ Position \end{array}$$

Ignoring row 4 and column 4 for the moment, we can see that the first three rows are actually unit vectors which are identical to that of the world X, Y and Z axes. The third row for example is a vector

of  $(0, 0, 1)$  which is a unit vector describing the world Z axis. You should think of these three rows as the object's local coordinate system. They describe the orientation of the model space X, Y and Z axes in relation to the world space X, Y and Z axes. We can see that the local coordinate system exactly matches the world space coordinate system when using an identity matrix.

In Fig 1.38, the X and Y rows in the identity matrix are unit vectors pointing in the same direction as the world axes. We know that the input vector will be unchanged by this matrix since the matrix used is constructed with identity columns.

### World Matrix (Identity)

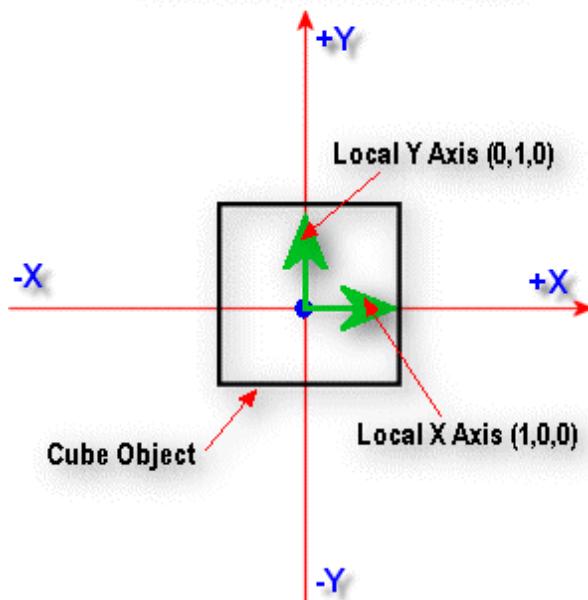


Figure 1.38

Regardless of whether the world matrix of an object is an identity matrix or not, we can still think of the first three rows of the matrix as unit vectors describing the local coordinate system x, y and z axes.

Let us see what happens when we combine our identity initialized world matrix with a Z axis rotation matrix. In the next example we will build a rotation matrix that rotates our points by 45 degrees (0.785398 radians).



$$\begin{array}{l}
 \text{Identity} \quad \quad \quad \text{45 degree Z rotation} \\
 \left[ \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] X \left[ \begin{array}{cccc} \cos(45) & \sin(45) & 0 & 0 \\ -\sin(45) & \cos(45) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right]
 \end{array}$$

=

$$\begin{array}{l}
 \text{Identity} \quad \quad \quad \text{45 degree Z rotation} \quad \quad \quad \text{New Rotated World Matrix} \\
 \left[ \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] X \left[ \begin{array}{cccc} 0.707106 & 0.707106 & 0 & 0 \\ -0.707106 & 0.707106 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] = \left[ \begin{array}{cccc} 0.707106 & 0.707106 & 0 & 0 \\ -0.707106 & 0.707106 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \begin{array}{l} \text{LocalXAxis} \\ \text{LocalYAxis} \\ \text{LocalZAxis} \\ \text{Position} \end{array}
 \end{array}$$

We see in Fig 1.39 that the axes of the local space coordinate system are now rotated 45 degrees:

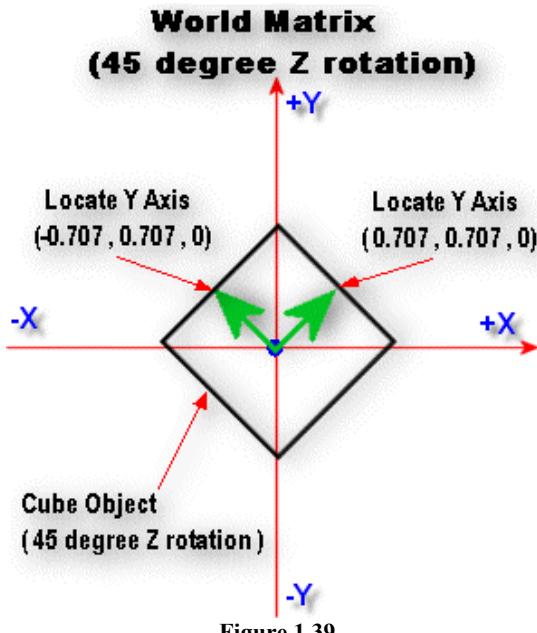


Figure 1.39

If we have a unit vector describing the way an object is pointing, we can use it to update its position in the world (by moving along this vector). Our object matrix contains all of the information we need to move it along a local axis even though it is now in world space. For example, the 3<sup>rd</sup> row of the matrix is referred to as the ***look vector***. It is a unit vector describing the way the object is facing. It is actually the model space Z axis and it retains exactly the same relationship to the model in world space as it did in model space. If we want to move our object a certain distance forward, we can use the 3<sup>rd</sup> row of the matrix to do this:



```
// move spaceship forward
void MoveForward( float distance )
{
    D3DXVECTOR3 LookVector;

    // Extract the Look vector (local z axis) from the world matrix
    LookVector.x = SpaceShip.WorldMatrix._31;
    LookVector.y = SpaceShip.WorldMatrix._32;
    LookVector.z = SpaceShip.WorldMatrix._33;

    SpaceShip.WorldMatrix._41 += LookVector.x * distance;
    SpaceShip.WorldMatrix._42 += LookVector.y * distance;
    SpaceShip.WorldMatrix._43 += LookVector.z * distance;
}
```

MoveForward extracts the look vector from the 3<sup>rd</sup> row of the matrix and then scales it by the forward distance we wish to move. We add the resulting vector to the current position -- which we know is stored in the 4<sup>th</sup> row of the matrix. Because LookVector is a unit vector, the distance is dispersed over the X, Y and Z axis in their correct proportions and the world matrix now contains the new world space position. This new position is exactly *distance* units from its previous position in the direction of the look vector. So whatever the orientation of the object in world space, we now have the means to move it forward in the desired direction. If MoveForward was passed a negative distance value, it would move the object backwards in world space.

We also have the object local Y axis (called the **up vector**) in the second row of the matrix and the local X axis (called the **right vector**) in the first row. This means that we could, for example, make our spaceship strafe left or right by using a function that uses the first row of the matrix. A negative distance value would cause the object to move left instead of right:

```
// Strafe spaceship left or right
void MoveStrafe( float distance )
{
    D3DXVECTOR3 RightVector;

    // Extract the Right vector (local x axis) from the world matrix
    RightVector.x = SpaceShip.WorldMatrix._11;
    RightVector.y = SpaceShip.WorldMatrix._12;
    RightVector.z = SpaceShip.WorldMatrix._13;

    // update position in matrix
    SpaceShip.WorldMatrix._41 += RightVector.x * distance;
    SpaceShip.WorldMatrix._42 += RightVector.y * distance;
    SpaceShip.WorldMatrix._43 += RightVector.z * distance;
}
```

With these two examples it should be no problem for you to write a third function called MoveUpDown.



The world matrix can be summarized as follows:

### World Matrix

Right Vector.x	Right Vector.y	Right Vector.z	0
Up Vector.x	Up Vector.y	Up Vector.z	0
Look Vector.x	LookVector.y	Look Vector.z	0
Position.x	Position.y	Position.z	1

This matrix will serve as our entire world transformation module. Multiplying our object vertices with a world matrix will convert those vertices from model space to world space:

$$\text{WorldSpaceVertex} = \text{ModelSpaceVertex} * \text{WorldMatrix}$$

### The View Matrix

The next task is to transform our world space vertices to view space (relative to some virtual camera position). The camera orientation and position information can also be stored in a single matrix. We refer to this matrix as the **view matrix**.

The view matrix works a little differently than our world matrix. As we saw earlier, in order to transform vertices into view space, we have to perform the **opposite** operations on them. When the camera is rotated to the right, we need to rotate the vertex left. If the camera is moved forward, we need to move the vertex backwards, and so on. In order to accomplish this we will use the **inverse matrix**.

Let us assume that we have three vectors describing the Up, Look and Right vectors of the camera, and that we also have a camera position in our 3D world.

// Assumed to be later initialized to meaningful values...

```
D3DXVECTOR3 CLook; // Camera Look Vector  
D3DXVECTOR3 CRight; // Camera Right Vector  
D3DXVECTOR3 CUp; // Camera Up Vecrtr  
D3DXVECTOR3 CPos; // Camera World space Position
```

If we were to build a standard local to world matrix for the camera it would look like so:

$$\begin{bmatrix} CRight.x & CRight.y & CRight.z & 0 \\ CUp.x & CUp.y & CUp.z & 0 \\ CLook.x & CLook.y & CLook.z & 0 \\ CPos.x & CPos.y & CPos.z & 1 \end{bmatrix}$$

However this matrix would **not** have the desired effect. In fact it would take a vertex that is already in view space and transform it so that the result is back in world space! This can actually be handy in



certain situations we will encounter later, but it is not what we need at the moment. We need to use the inverse of this matrix:

$$\begin{array}{c} \text{View Matrix} \\ \left[ \begin{array}{cccc} CRight.x & CUp.x & CLook.x & 0 \\ CRight.y & CUp.y & CLook.y & 0 \\ CRight.z & CUp.z & CLook.z & 0 \\ -(CPos \bullet CRight) & -(CPos \bullet CUp) & -(CPos \bullet CLook) & 1 \end{array} \right] \end{array}$$

This is the matrix we will use to convert vertices from world space into view space. The virtual camera in our game can be represented using this single matrix. D3DX has a function that will take a 4x4 matrix and invert it:

```
D3DXMatrixInverse(D3DXMATRIX* pOut, FLOAT* pDeterminant, CONST D3DXMATRIX* pM);
```

The mathematics involved in inverting arbitrary matrices can be complex and is covered in detail in the Game Mathematics course here at the Game Institute. For our purposes, we can simply pass in a transformation matrix and set the determinant value to NULL (as we will not need it). Also notice that the output matrix need not be the same as the input matrix.

There will be times when you will need to call the above function to invert a matrix but it is certainly not the way we would recommend creating the view matrix each time the camera moves and the view matrix needs to be updated. Storing the camera position and orientation as a normal transformation matrix and calculating the inverse each time the camera moves is an expensive operation. Instead, the camera is usually managed by having Up, Right, Look and position vectors and building the view matrix manually; inserting the vectors into the matrix as shown above. The matrix will only need to be rebuilt when the camera moves or rotates. We could for example replace our previous CCamera class with a new one that looked like this:

```
class CCamera
{
public:
    D3DXVECTOR3 LookVector;
    D3DXVECTOR3 Up Vector;
    D3DXVECTOR3 RightVector;
    D3DXVECTOR3 Position;
}
```

At the start of your application you might set the camera to its correct starting position and set the look, up and right vectors so they are aligned with the world axes using vectors (0,0,1) , (0,1,0) and (1,0,0) respectively. Then when the player presses an arrow key, you could rotate the vectors with a rotation matrix that rotates them so they are now facing in a new direction. Finally the view matrix would be rebuilt by inserting these vectors manually into the view matrix.



In LP 1.1 we will not be moving the camera and we will be leaving the view matrix set as an identity matrix. This means that the camera can be visualized as being at position (0,0,0) in the world, with its local coordinate system aligned with the world axes so that it is looking down the positive Z axis.

### **View Matrix**

Right Vector.x	Up Vector.x	Look Vector.x	0
Right Vector.y	Up Vector.y	Look Vector.y	0
Right Vector.z	UpVector.z	Look Vector.z	0
- (Position • RightVector)	- (Position • UpVector)	- (Position • LookVector)	1

There are D3DX matrix functions which aid in the setting up of a view matrix. We will discuss using the view matrix in Chapter 4 in much more detail. We will see how to get it to behave like a first person shooter game camera or even a space ship game camera.

A D3DX view matrix helper function of interest to us is shown below. It takes a camera position in world space, a point in that we want the camera to look at, and a vector describing the UP vector of the camera (often <0, 1, 0> at startup) and builds the matrix for us:

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

The parameters are shown below:

**pOut** - The address of a D3DXMATRIX structure that will contain the calculated view matrix.

**pEye** – World space position of the camera, referred to here as the eye point

**pAt** - World space position that we want the camera to be looking at

**pUp** - Orientation of the camera up vector

It should be noted that the ‘LH’ at the end of the function call is short for ‘Left Handed’. This function builds a view matrix suitable for a left handed coordinate system, which DirectX Graphics (and we) will use. D3DX does contain a right handed version of the function called D3DXMatrixLookAtRH so make sure that you do not accidentally call the wrong one.

**Note:** Inverting a matrix produces the opposite effect of a normal transformation matrix. Thus, by inverting an object’s world matrix, we get a matrix that would transform world space vectors into model space vectors.

This technique is often used in collision detection routines where you may have to check each vertex of a mesh against a world space bounding box or a bounding sphere. In these cases it is much cheaper to back transform a single world space sphere into model space and perform the test there than to transform every vertex in the mesh into world space and then test.

If you think of a transformation matrix as transforming points from one space to another, you can think of the inverse of that matrix as performing a canceling or reversing transformation into the original space.



Remember that we can multiply matrices together to create a single combined transformation matrix that will transform any vectors as if they had been multiplied by all the original matrices. We could combine each object's world matrix with the current view matrix prior to rendering that object, and thus transform all vertices from model space to view space with one vector/matrix multiplication. This saves us a fair amount of work as each vertex would otherwise need to be multiplied by the object world matrix, and then again by the view matrix.

```
D3DXMATRIX ComboMatrix = Object.WorldMatrix * ViewMatrix;  
  
D3DXVECTOR ViewSpaceVertex = ModelSpaceVertex * ComboMatrix;
```

In LP 1.1 we will not do this. At this point we would like to keep the World, View and Projection matrices separate to better demonstrate each stage of the pipeline.

### The Perspective Projection Matrix

We spent a good deal of time earlier in this lesson discussing how to project a 3D view space point into a 2D projection space point. You may recall that the resulting point was in the  $-1$  to  $+1$  range on both the X and Y axes. This point was later mapped to screen space. You may also recall that the formula we used to perspective project the 3D point into a 2D projection space coordinate was simply:

$$2D\text{ ProjectionPoint } X = \frac{\text{ViewSpace}X}{\text{ViewSpace}Z}$$

$$2D\text{ ProjectionPoint } Y = \frac{\text{ViewSpace}Y}{\text{ViewSpace}Z}$$

As a point gets further away from the camera it is scaled down (and vice versa). This provides the illusion of perspective. Recall that the one important characteristic of this formula is that it always uses a 90 degree FOV. This means that the camera can always see 45 degrees to the left, and 45 degrees to the right, and on the other axis, 45 degrees up and 45 degrees down. We could visualize this as a view cone spreading out from the camera origin at an angle of 90 degrees.

This is exactly how DirectX Graphics (and our own software code) perspective projects a 3D view space point. But having no choice other than a 90 degree FOV is simply not acceptable to us. First of all, a 90 degree FOV does not usually look particularly good. Most developers prefer to use a 45 to 65 degree FOV. This is not technically correct because humans have a wider FOV than that in real life, but it looks correct in the game. Second, the monitor screen is not square and we usually have more pixels horizontally than we do vertically. This means we should really have a wider FOV left and right than we do Up and Down. If we do not, then the scene will look squashed because we are doing a SQUARE projection onto a rectangular monitor screen. If your application is running in a perfectly square window, then no squashing or distortion will appear, but usually, we like our games to run in full screen resolution such as 800x600 or 1024x768 which are rectangular video modes.



If all of this is true, then we would appear to have a problem. DirectX Graphics calculates the perspective projection using the 90 degree FOV formula we saw earlier ( $x/z$  and  $y/z$ ) yet we wish to use arbitrary FOV.

In order to combat this problem we can multiply view space vectors by a third matrix prior to the divide by z (the perspective projection process). This matrix will distort the geometry in our world in a controlled manner so that the illusion of an arbitrary FOV is maintained. We are still doing a 90 degree unit projection, but because the vertices have been deformed, we can control whether or not they fit within the 90 degree FOV.

The vertices of a mesh will be multiplied with this new projection matrix *after* the vertices have been converted to view space. This means that we can write the complete transformation from model space vertex to projection space vertex as:

***ProjectedVertex = ModelVertex \* (WorldMatrix \* ViewMatrix \* ProjectionMatrix);***

At this point we can map the  $[-1, +1]$  range of the vertex along the x and y axes into the range of the current screen resolution as shown earlier in the lesson.

**Note:** It is perhaps odd that it is called a 'projection matrix' since it does not project the vertices at all. Rather it swells or shrinks their position values prior to a perspective divide. You could say, then, that the projection matrix is a matrix that **prepares** 3D vertices for projection to 2D.

Refer back to the section on perspective projection if you need to. It is important that you understand why the divide by z performs a 90 degree projection if you are to understand this next section.

Let us start our analysis of the projection matrix with an identity matrix and build up from there. We know that the projection matrix is a  $4 \times 4$  matrix and that it will output a  $1 \times 4$  vector. The input vector will be a homogeneous 4D coordinate in the form of  $(x, y, z, 1)$  as we have already discussed. As with all of the matrices we have used up until this point, the W column of the matrix is an identity column. Thus the output vector will also be in the form of  $(x, y, z, 1)$  and we can discard the w component.

### Projection Matrix

$$V = [x \ y \ z \ 1] \times M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = P[X \ Y \ Z \ W=1]$$

The above projection matrix does absolutely nothing. Because it is an identity matrix, output vector  $P$  will be identical to input vector  $V$ . Once vector P has been calculated we could simply do  $x/z$  and  $y/z$  to calculate the new 2D projection space position of the vector. This would scale the geometry using a 90 degree FOV projection.



Note that the projection matrix is the last point at which we have control over the vertex in the DirectX Graphics fixed-function transformation pipeline. We will pass DirectX Graphics a World matrix, a View matrix and a Projection matrix and call the DrawPrimitive function to render polygons. DirectX Graphics will multiply our vertices with the three matrices and will then take care of performing the perspective divide on the resulting vector returned from the projection matrix. It will eventually remap the coordinate to a screen space coordinate. The software renderer in LP 1.1 will mimic this behavior to a certain extent. Therefore we will set up the projection matrix the same way it will need to be set up when using DirectX Graphics.

The first problem we must address with our matrix is that DirectX Graphics requires that the  $w$  component of the output vector be equal to the  $z$  component of the input vector *after* the projection matrix multiply ( $W=z$ ). The reason is that DirectX Graphics uses the  $w$  component of the output vector for other calculations (depth-based fog, color interpolation, W-Buffer). It may seem more intuitive to copy the input  $z$  component into the output  $Z$  component and use that, but as you will see later on, we need the  $Z$  value of the output vector to hold specialized information intended for something called a Depth Buffer. Copying the  $z$  component of the input vector into the  $W$  component of the output vector is no big deal, and we can alter our matrix quite easily to ensure that this is so:

### Projection matrix

$$V = [x \ y \ z \ 1] \times M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = P[X \ Y \ Z \ W = z]$$

By adjusting the 4<sup>th</sup> column of the projection matrix so that the 1 is no longer in the 4th row but is now in the 3<sup>rd</sup> row, the  $W$  component of the output vector will be calculated as follows:

$$W = x*0 + y*0 + z*1 + 1*0 = z$$

This matrix has correctly copied over the  $z$  component of the input vector  $V$  into the  $W$  component of the output vector  $P$ . So if  $V = (20, 40, 105, 1)$ , then  $P = (20, 40, 105, 105)$ .

We said before that to move a vertex from view space to 2D projection space (where the divide by  $z$  happens) we simply do:

$$2D\text{ProjectionSpace}_x = \frac{\text{ViewSpace}.x}{\text{ViewSpace}.z}$$

$$2D\text{ProjectionSpace}_y = \frac{\text{ViewSpace}.y}{\text{ViewSpace}.z}$$



And now we have a new coordinate space. This space is the space the vertex is in **after** it has been multiplied by the projection matrix but **before** it has been projected into 2D projection space (the divide by z). This new space is referred to as ***Homogeneous Clip Space***. The only current difference between view space and homogeneous clip space is that in homogeneous clip space we have copied the z component into w and we have:

$$\begin{aligned}\text{ViewSpaceVector} &= (x, y, z, 1) \\ \text{HomogeneousClipSpaceVector} &= (x, y, z, z)\end{aligned}$$

Because the w component holds the z value and because the z value of the output vector will later hold something else, DirectX Graphics (and our software engine) does its perspective projection using this formula:

$$2D\text{ProjectionSpace}_x = \frac{\text{HomogenousClipSpace}.x}{\text{HomogenousClipSpace}.w}$$

$$2D\text{ProjectionSpace}_y = \frac{\text{HomogenousClipSpace}.y}{\text{HomogenousClipSpace}.w}$$

The formula remains totally unchanged, only now the *z* value is in W instead of Z.

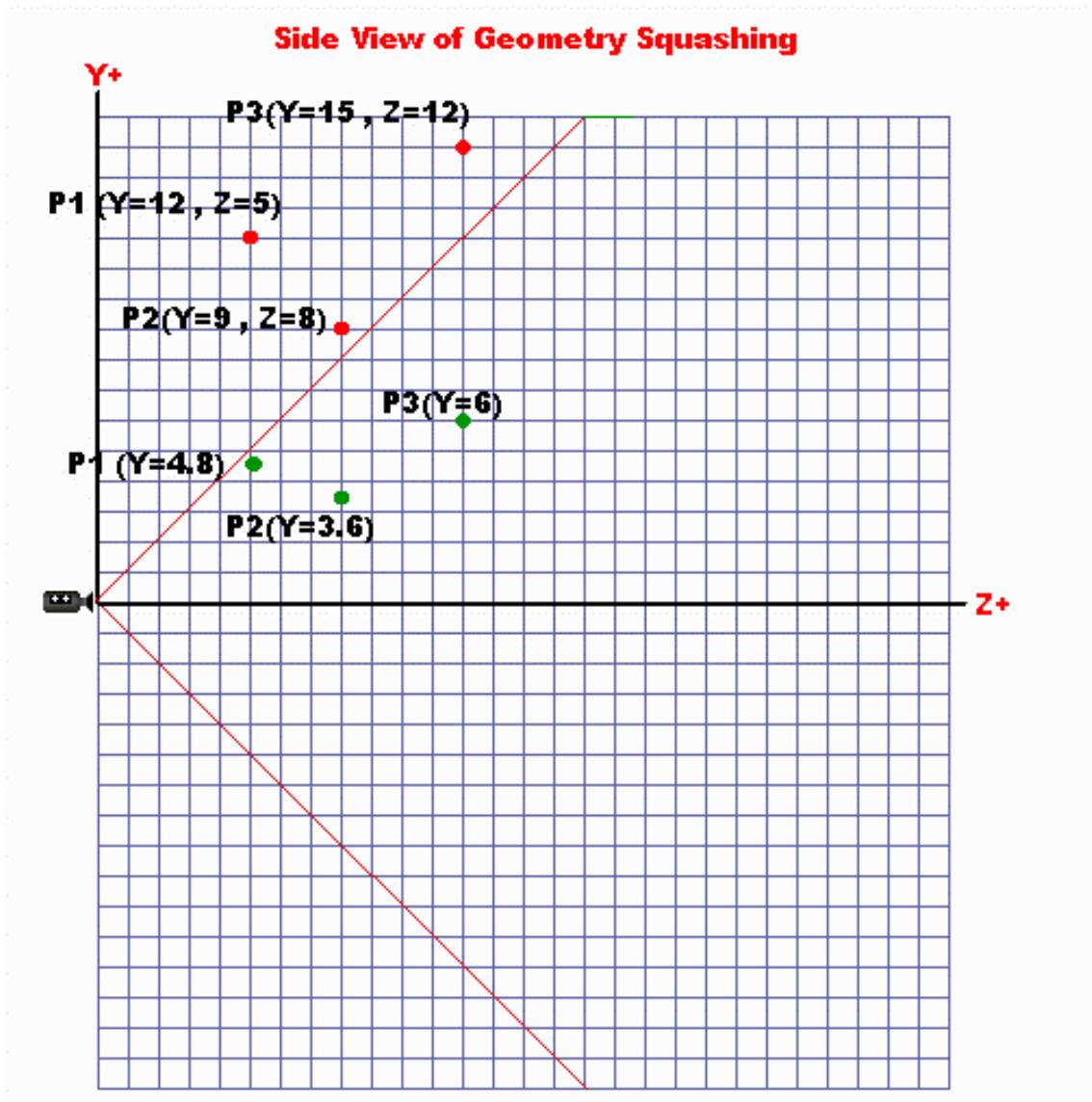
So if we set up our projection matrix correctly, it will output a 4D vector **P** like so:

$$\mathbf{P}(X, Y, Z=\text{Depth Buffer value}, W=z)$$

Ignore the Depth Buffer value for now. We will cover it in our next lesson. For now we are only interested in finding out how the X and Y columns of the projection matrix can be used to deform geometry to give an arbitrary FOV. The W column is already taken care of; it simply copies over the input z component into w. So let us now look at what we should do with columns 1 and 2 of the projection matrix.

### Arbitrary FOV

If you take a look at the first two columns of the projection matrix, you see that for *x* and *y*, it is really like a 2x4 scaling matrix. At the moment it is simply scaling  $x*I=x$  and  $y*I=y$  -- which is why these are not altered. But by changing the values in elements **m11 and m22** we can scale the *x* and *y* values prior to the divide by w. In effect, we still perform a 90 degree FOV projection ( $x/w$  and  $y/w$ ), but we can use the **m11** and **m22** elements in the matrix to scale (squash or enlarge) geometry so that it falls either in or out of the 90 degree FOV projection cone. This is what allows us to have any FOV we desire. To understand this concept, take a look at Fig 1.40. It shows us squashing geometry into the view cone that would otherwise not be rendered.



**Figure 1.40**

Noting that the z value of the input vector is simply copied over into the W value of the output vector, any given x or y point in space will only be mapped inside the projection window if the following is true:

$$\begin{aligned}-W < X < W \\ -W < Y < W\end{aligned}$$

In Fig 1.40 there are three view space points (red dots) labeled **P1**, **P2** and **P3**. These points are outside the 90 degree view cone because the **Y** value of each of these points is greater than the **Z** value. However, if we were to multiply each Y value by, say, 0.4, the Y values would be smaller than their z counterparts (green dots in the above diagram). This means that these points would be projected onto



the projection window when the divide by  $w$  is calculated. If we do this to *all* vertices in our scene, we can squash as much or as little geometry into our 90 degree view cone as we want.

In the previous example, we multiplied our  $y$  values by 0.4 to create a wider FOV. But we can also scale the geometry up as well. For example, if we were to scale each vertex by 1.5,  $Y$  values that did fit within the 90 degree FOV originally would be increased and would leave the projection matrix greater than  $W$ . These points would not fall within the projection window and this would simulate a narrower FOV.

Scaling the  $x$  and  $y$  values is easy. The first two columns of our projection matrix look just like a scaling matrix. Therefore, in  $m11$  and  $m22$  where we currently have a value of 1.0, we can replace these values with other values that will increase or decrease  $x$  and  $y$  input vector values.

### Projection Matrix

$$M = \begin{bmatrix} 0.4 & 0 & 0 & 0 \\ 0 & 0.4 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$Y$  is calculated as follows:

$$Y = x * 0 + y * 0.4 + z * 0 + w * 0$$

This simplifies to:

$$Y = y * 0.4.$$

The same is true for the  $x$  coordinate.  $X$  is calculated as follows: -

$$X = x * 0.4 + y * 0 + z * 0 + w * 0$$

This simplifies to:

$$X = x * 0.4$$

At this point we can now scale the geometry, calculate the depth buffer output value and copy  $z$  into  $W$ , all by performing one vector/matrix multiplication.

## Co-Tangent and User Defined FOV

In the last example we used an arbitrary value of 0.4 in the **m11** and **m22** elements of the matrix to scale the geometry. This provided the appearance of a wider FOV because geometry was scaled down. However gaining precise control over the FOV settings requires a trigonometric function: the co-tangent.

Cosine, Sine, and Tangent are functions that return the ratio of two sides of a right triangle. For example the Tan function returns the ratio between the Opposite side of a triangle and the Adjacent side of a triangle.

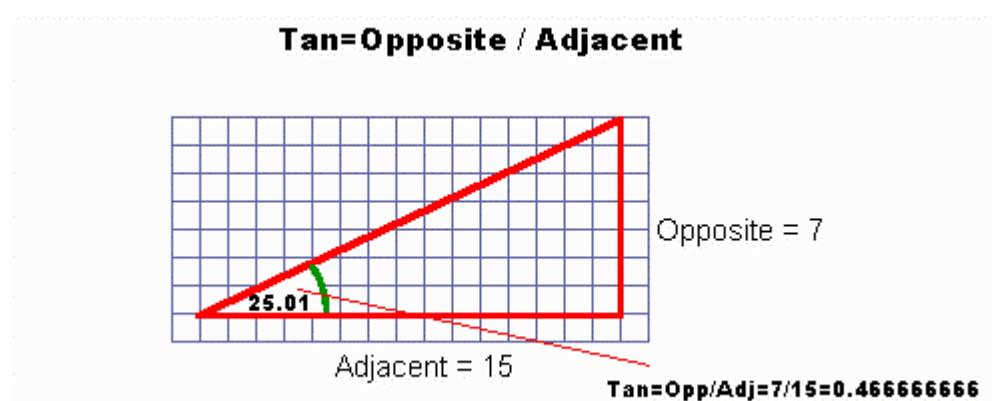


Figure 1.41

In Fig 1.41, the length of the Opposite side of the right triangle is 7 and the length of the Adjacent side of the triangle is 15. When we divide the opposite side by the adjacent side we get a result of 0.466666666. This ratio is called the **Tangent** (or **Tan** for short). The tangent is always calculated by dividing opposite by adjacent.

All angles of a right angled triangle are mapped to a specific tangent value that describes the ratio between the opposite and adjacent sides. For example, let us say that we know the triangle has an angle of 25.01689345 degrees and we also know the length of the Adjacent leg of the triangle. If we wanted to figure out how long the opposite leg was, we could punch in the angle on our calculator (25.01689345) and then press the Tan button (which in this case would return 0.466666666). This value describes the ratio of the opposite leg to the adjacent. Thus to find the length of the opposite leg:

$$\text{Opposite} = \text{Tan}(25.01689345) * 15 = 0.466666666 * 15 = 7 \text{ (approx, actually } 6.9999999999999\text{)}$$

If we have the lengths of both the opposite side and the adjacent side, but we do not know the angle value, we can use the inverse tangent *atan* (*i.e.*  $\text{Tan}^{-1}$ ). First we calculate the tangent:

$$\text{Tan} = \text{Opposite}/\text{Adjacent} = 7/15 = 0.466666666$$



Punch in this tangent value and press the **atan** key and the calculator will return the angle for that tangent (which in our case we already knew was 25.01689345 degrees or 0.436627159 radians).

If we take a side-on look at our view cone, and split it down the middle, you can see that for any  $z$  value along the Z axis in view space, we do indeed have a right angled triangle.

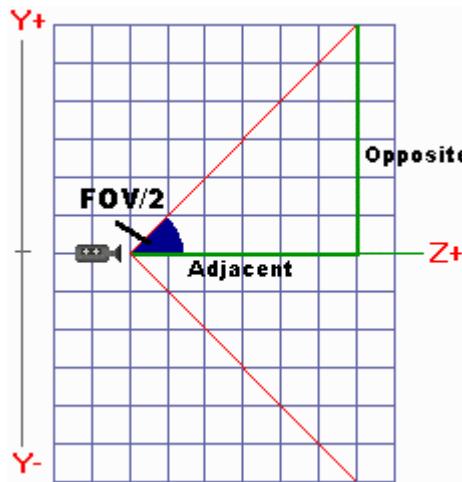


Figure 1.42

The opposite side of the triangle is represented by the Y value and the Adjacent leg is represented by the Z value. The same would also be true in a top-down view of the view cone, where the X axis would represent the Opposite leg.

In Fig 1.42, the Opposite side of the Triangle is at a distance of  $Z=+6$ . (It should be noted that for any  $Z$  value, the ratio (tangent) between the Opposite and Adjacent would remain the same and the angle would remain the same). Note that the angle of the triangle is  $FOV/2$ . Also notice that the Opposite and Adjacent sides have the same lengths as each other.

$$\frac{\text{Opposite}}{\text{Adjacent}} = \frac{6}{6} = 1$$

If you type 1.0 into your calculator and press the atan function you will be returned an angle of 45 degrees. Recall that when the Y value at any point is equal to Z, then the FOV is 90 degrees. If we change this relationship we could come up with a value that we could put into our projection matrix (**m11** and **m22**) to scale the geometry.

Let us suppose that we want a FOV smaller than the default 90 degree projection carried out by the divide by w projection (say 60 degrees). Logically we would want a value that would increase our X and Y values so that the geometry which was just inside our 90 degree FOV is pushed outwards. This simulates a smaller FOV since we should not see as much of our scene as we would be able to see with a 90 degree FOV.



What happens if we use the tangent function to calculate the ratio for us?

$$\tan(30) = 0.577350269$$

That is clearly not correct. Multiplying our  $x$  and  $y$  values by 0.577350269 would actually make the values smaller and would squash even more geometry into the view cone. It is the opposite effect that we want. In order to get the correct ratio to scale our  $x$  and  $y$  values we need to use the **co-tangent** function:

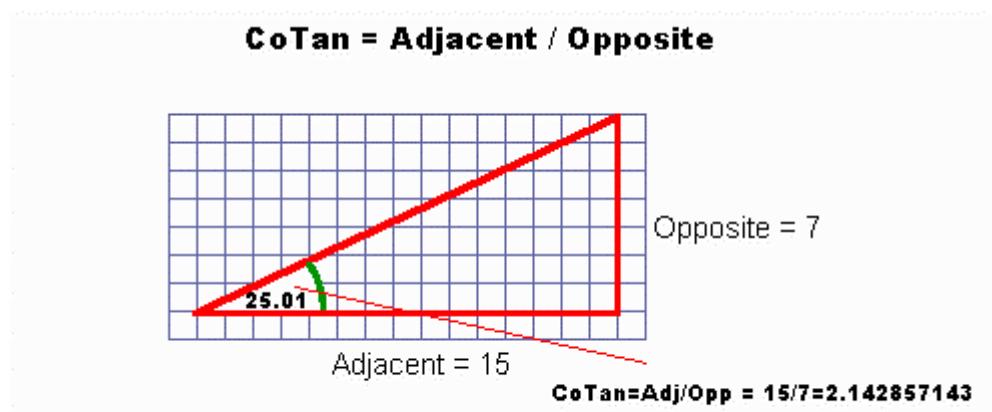


Figure 1.43

The co-tangent in our example ratio is 2.142857143. This value is exactly what we need to multiply by the opposite side in order to make it equal to the adjacent side:

$$7 * 2.142857143 = 15$$

More specifically, this is the value that we need to create an opposite side length such that the triangle is forced into becoming a 45 degree triangle (where both the opposite side and the adjacent side have lengths of 15).

So this is the value we need to multiply by our  $x$  and  $y$  values in order to simulate a 50.02 FOV (twice the angle above for the full view cone). Since the Co-Tangent function is not implemented in many programming languages or on many calculators we can use trig functions to figure it out:

$$\text{co-tan} = \frac{1}{\tan(\theta)} = \frac{\cos(\theta)}{\sin(\theta)}$$

If we want a FOV of 60 degrees, we can scale the  $x$  and  $y$  values in the projection matrix by filling out the **m11** and **m22** elements of our matrix as follows.

```
m11 = 1 / tan(60/2)  
m22 = 1 / tan(60/2)
```



or

```
m11 = cos(60/2) / sin(60/2)
m22 = cos(60/2) / sin(60/2)
```

Notice above that the FOV (60 degrees) is divided in two (30 degrees) because the trigonometry functions use one half of the view cone.

$$\theta = 1.047197551 \text{ radians (} 60 \text{ degrees)}$$

$$M = \begin{bmatrix} 1/Tan(\frac{\theta}{2}) & 0 & 0 & 0 \\ 0 & 1/Tan(\frac{\theta}{2}) & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

We now have a projection matrix that will scale geometry according to any arbitrary FOV. The 4<sup>th</sup> column simply copies the input  $z$  value into  $W$  of the output vector for the divide by w. The 3<sup>rd</sup> column maps the input z value into a value that can be used by the DirectX Graphics depth buffer. We are just about finished.

### Aspect Ratio

When projected into 2D space, we get back a value between -1 and +1 in both the x and y dimensions for any point inside the FOV. This is the coordinate system of the Projection Window. The next task is to convert those projection coordinates to valid screen coordinates that can be rendered on the display. In order to calculate the final screen coordinates, we do something like this:

```
ScreenX = Vector.x * ViewportWidth / 2 + ViewportLeft + ViewportWidth / 2
ScreenY = -Vector.y * ViewportHeight / 2 + ViewportTop + ViewportHeight / 2
```

**ScreenX** and **ScreenY** are screen space coordinates. In a resolution of 800x600, **ScreenX** is in the range of 0 to 800 and **ScreenY** is in the range of 0 to 600.

**Vector.x** and **Vector.y** are the clip space coordinates on the projection window (in the range -1 to +1) and are the results of the divide by w.

**ViewportWidth** and **ViewportHeight** are the dimensions of the viewable area on screen. For example, in a full screen window of 800x600, these values would be 800 and 600 respectively.



**ViewportLeft** and **ViewportTop** should be set to zero for full screen windows, or should contain the top left coordinates of the view window if you only wish to render to a view port that covers part of the screen.

The projection window coordinates range from -1 to +1 in both the x and y dimensions and thus the window is square ( $2 \times 2$  in size). However, monitor screens are generally not square. Most are rectangular (usually wider than they are higher). This is also true for the most common video modes: 800 x 600, 640 x 480, 1024 x 768. These are all video modes that have more pixels horizontally than they do vertically. This presents us with a problem. Suppose we have a polygon in front of the camera that is a perfect square. This will be projected onto the projection window as a perfect square also. However, when the projection window coordinates are mapped to screen coordinates, they will be stretched to take up the extra width of the video mode. This means that the user of your application will see the square as a rectangle (Fig 1.44).

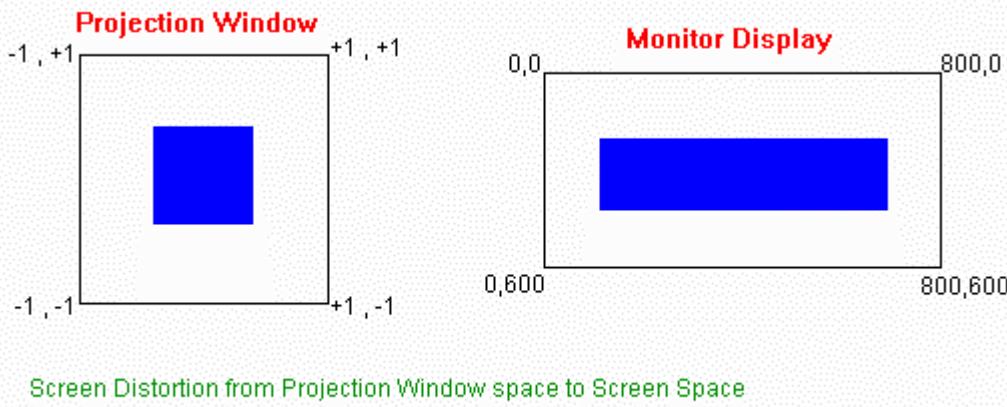


Figure 1.44

In order to counter this unwanted effect we will set a different FOV in the X dimension of our matrix (**m11**). By increasing the FOV in the X dimension, we scale the input  $x$  values in our projection matrix down. This means a square in camera space will be squashed in X onto the projection window such that when the projection window is stretched into screen coordinates, the resulting rectangle is stretched back into a square shape (Fig 1.45).

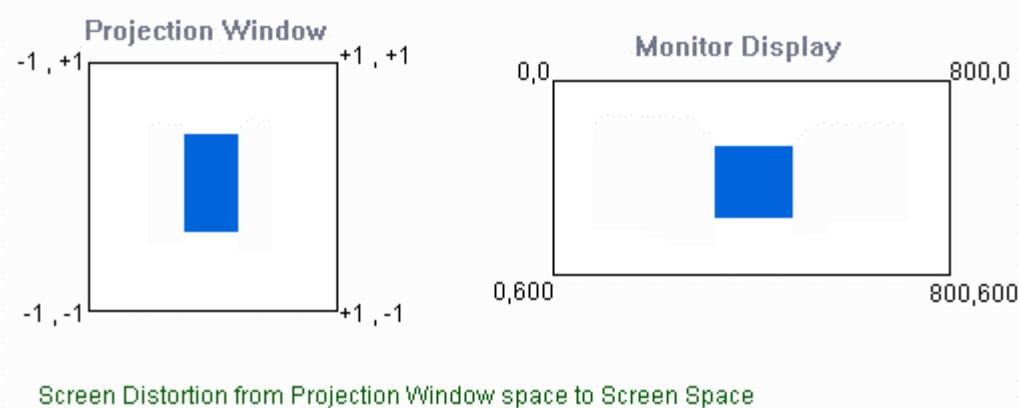


Figure 1.45

If we can measure the ratio of Screen Width to Screen Height that our application is using, and set the FOV for the x axis (**m11**) in our projection matrix accordingly, we get a wider FOV along the x axis. This is logical; if the monitor is wider in the x dimension than it is in the y, we should be able to see more in the x dimension, and therefore have a wider FOV in the x dimension. In order to correct the problem, we must first measure the ratio of screen distortion. This ratio is nearly always referred to as the **Aspect Ratio**, and can be calculated like so:

$$\text{Aspect Ratio} = \frac{\text{Width}}{\text{Height}} = \frac{1024}{768} = \frac{800}{600} = \frac{640}{480} = 1.33333333$$

Notice how the aspect ratio is the same for all the standard full screen video resolutions (1.3333333). If you are not using a standard video mode, or are using a viewable area that is not the full screen, Width and Height in the above equation refers to the width and height of the view in which port you are rendering (in screen coordinates).

With this aspect ratio, we can adjust the **m11** element of our matrix to correct for screen space distortion by setting up the matrix as follows:



### Projection Matrix with 60 Degree FOV and Aspect Ratio Correction

$\theta = 1.047197551$  (60 Degree FOV)

$$M = \begin{bmatrix} 1/\tan(\frac{\theta}{2}) & 0 & 0 & 0 \\ \frac{\text{AspectRatio}}{1/\tan(\frac{\theta}{2})} & 0 & 0 & 0 \\ 0 & 1/\tan(\frac{\theta}{2}) & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

When we specify a FOV of 60 degrees, the FOV is only 60 degrees with respect to the Y axis. It is  $\text{ATAN}(\text{TAN}(\frac{60}{2}) \times 1.333333) \times 2 = 75.1781788$  degrees with respect to the X axis.

**Note:** In some source implementations you might see Aspect Ratio calculated as  $\frac{\text{Height}}{\text{Width}} = 0.77777777$ . These implementations will MULTIPLY element **m11** with 0.77777777 instead of DIVIDING **m11** by 1.33333333.

After that somewhat lengthy discussion on setting up a projection matrix, you will be glad to know that you can set-up a projection matrix easily with a single call to a D3DX function:

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

We pass to this function the address of a matrix that will store the final matrix, a FOV for the Y axis, and an aspect ratio (ViewportWidth / ViewportHeight). The matrix returned will be calculated in the way that we have just described.

The two parameters at the end of the parameter list in the above function (zn and zf) are used to configure the 3<sup>rd</sup> column of the projection matrix to scale the Z value of the input vector into a range that can be used by the DirectX Graphics depth buffer. We will not be using a depth buffer in our first lab project so we can leave this discussion until the next chapter when we use DirectX Graphics to render our geometry.



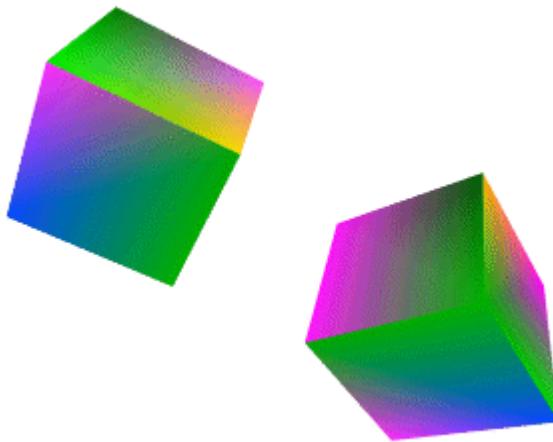
## Conclusion

The key points from this lesson are the core processes involved in transforming objects from model space to world space to view space to eventual screen coordinates. We also learned that 3D models are constructed from polygons and that each polygon is made up of a number of vertices. Finally we covered a good deal of crucial mathematical techniques that will be invaluable as we progress through the course. At this point it is recommended that you enroll in the Game Mathematics course to continue to reinforce this mathematics knowledge as well as learn new techniques. The two courses can now be taken in parallel since the core math you will need for this course has been covered.

# **Workbook Chapter One:**

## **3D Graphics Fundamentals**

---



© 2003, eInstitute, Inc.

**You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.**

## 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](http://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.

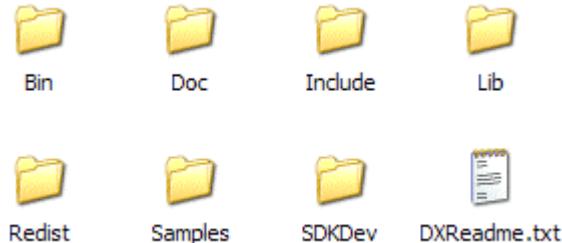
Let us first cover setting up the DX9 SDK for your compiler. The following examples are for Microsoft's Visual C++ 6 compiler. If you are using a different compiler then you will have to interpret and translate the following instructions for use with your particular system.

The first thing you will need to do is to visit the Microsoft website ([www.microsoft.com](http://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:

C:\dx9sdk



### 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**

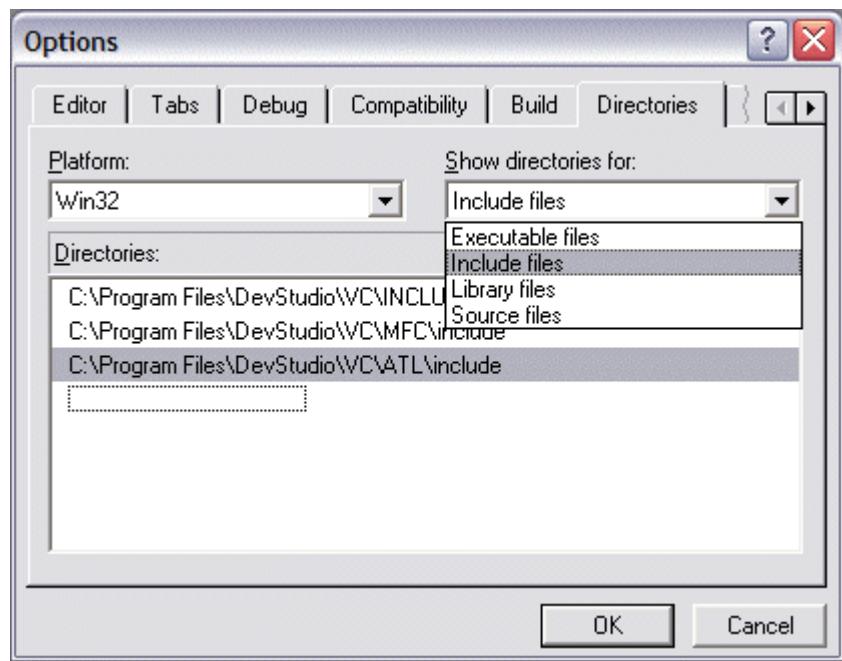
This directory contains the runtime install applications that are automatically installed with the SDK. They are English language only and contain both debug and retail DirectX 9.0 system components. You can switch between the retail and debug versions of the runtime via the DirectX Control Panel component (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.

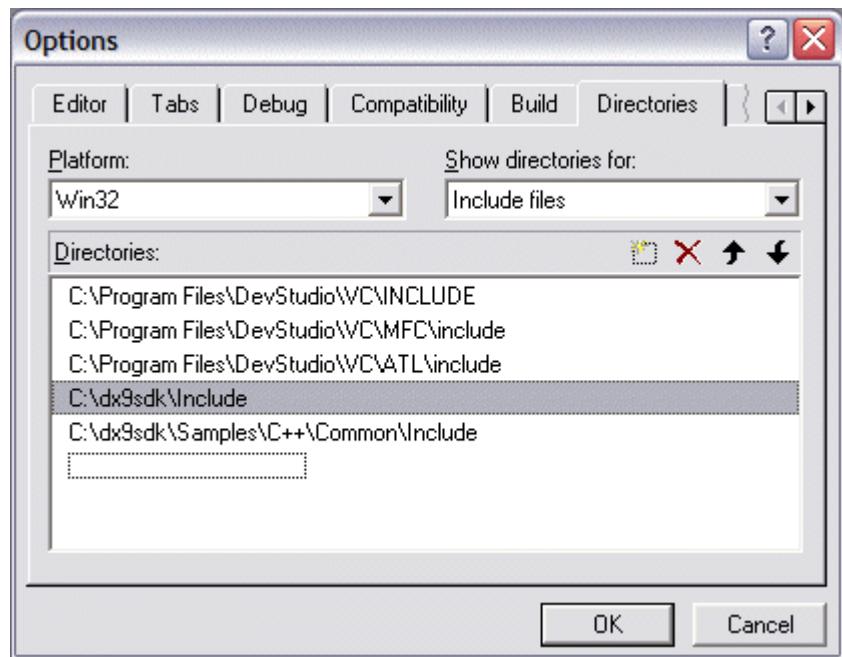
## **Setting up the Build Environment for DirectX9**

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:



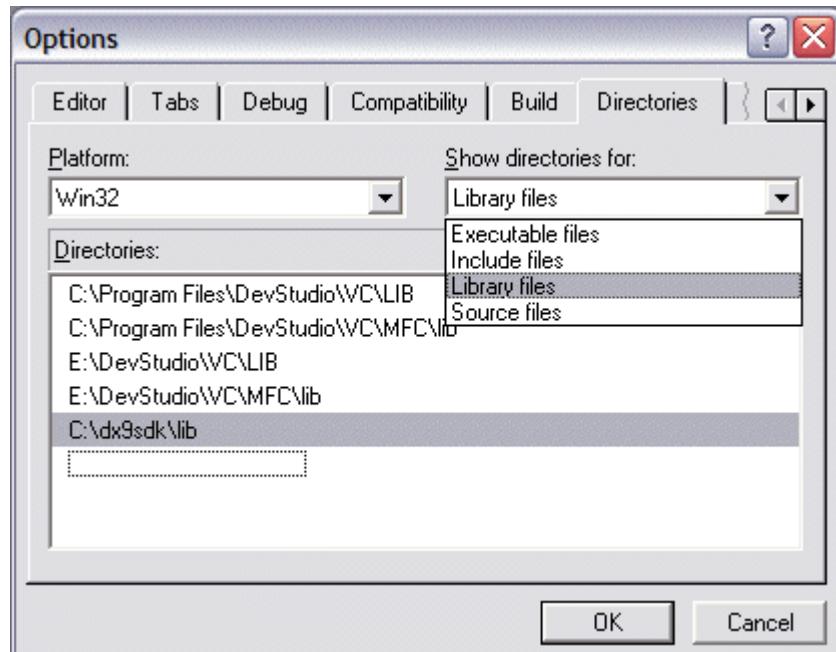
In the above image we have added two folders to the list. The first one (described earlier) is the folder in which the primary include files are contained. These are required for building a DirectX application.

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:**

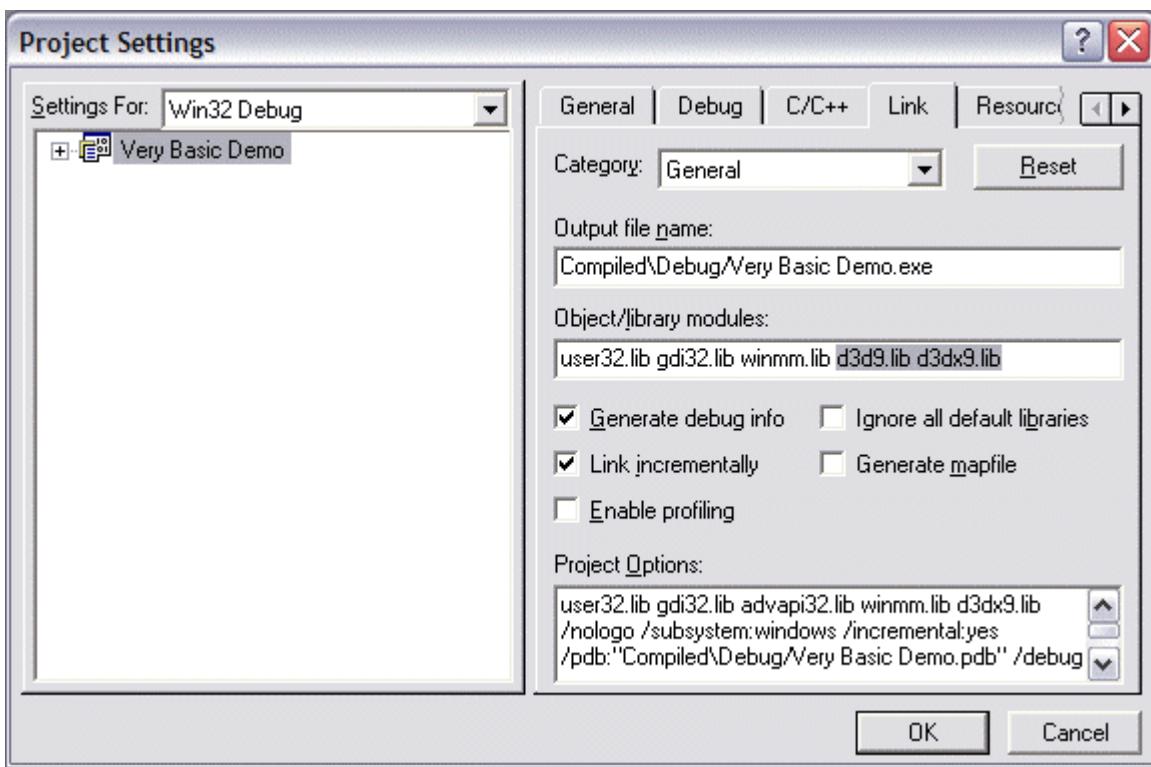
The search path list is processed in order from top to bottom. This is important to note if you have a previous version of the SDK installed and you have a path to those folders in the list. If it is higher up in the list and some of those files share the same name as the ones in the dx9sdk, those files 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 show below:



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;
}

```

## The CMesh Class

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\_hdclFrameBuffer**

### **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.cbcClsExtra = 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	Up Vector	Look 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.

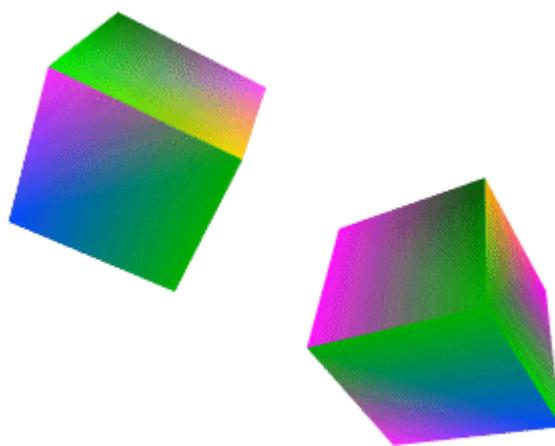
## Further Reading:

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

# Chapter Two:

# DirectX Graphics Fundamentals

---



© 2003, eInstitute, Inc.

You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.



## Table of Contents

DirectX Graphics Overview .....	3
Initializing DirectX Graphics.....	5
The Direct3D Device .....	8
Pipeline Overview.....	10
Device Memory .....	11
Frame Buffers .....	12
Refresh Rate.....	12
The Front Buffer .....	14
Swap Chains.....	14
Screen Settings.....	15
Fullscreen Mode.....	16
Windowed Mode.....	17
Depth Buffers.....	18
The Z-Buffer .....	20
The W-Buffer.....	23
Surface Formats .....	24
Adapter Formats.....	25
Frame Buffer Formats.....	25
Creating a Device.....	26
Presentation Parameters .....	32
Format Selection .....	44
Lost Devices.....	47
Filled Polygons .....	49
Shaded Polygons.....	50
Vertices in DirectX Graphics.....	52
The Flexible Vertex Format.....	56
Planar Polygons .....	60
The DrawPrimitive Functions.....	62
The DrawPrimitiveUP Function .....	63
The Rendering Pipeline.....	74
Device States.....	75
Frame Buffer and Depth Buffer Clearing .....	83
Beginning and Ending Scenes .....	84
Presenting the Frame.....	84
Conclusion .....	85



## DirectX Graphics Overview

DirectX Graphics provides a unified programming interface for multimedia development with integrated support for hardware acceleration when available. Since even the most moderately priced PCs on the market typically include hardware acceleration for 3D graphics, most end-user systems can be counted on to meet minimum requirements. Driver support for the DirectX Graphics API exists for practically every video card sold since the mid-90s. If hardware acceleration is not present on an end user system, DirectX Graphics provides software based emulation with full support for optimized CPU instruction sets (like MMX or 3DNow).

When card manufacturers ship their latest hardware, they release a small high-speed software layer called a **device driver** along with it. Driver software acts as an interpreter, taking requests from the OS and turning them into native instructions the hardware can execute. As newer versions of the OS are released, the manufacturer can release new drivers to maintain compatibility. Device drivers are generally fast and stable and tend to improve with time. Hardware manufacturers like nVidia and ATI are constantly working on optimizing their device drivers and you should check their websites' driver downloads sections periodically to ensure optimal application performance.

Most hardware manufacturers package a DirectX Graphics compliant device driver called the **Hardware Abstraction Layer (HAL)**. When a HAL is found on the current system, it indicates that the graphics hardware has hardware accelerated support for at least some DirectX Graphics functionality. DirectX Graphics can talk to hardware in a consistent way because the HAL takes care of translating requests into the native instruction set for the 3D hardware. Some adapters provide only hardware accelerated polygon rasterization. When this is the case, DirectX Graphics will transform and light polygons in software and then pass them to the HAL for rasterization. DirectX will shift the entire process to the HAL when **transformation and lighting (T&L)** support is available.

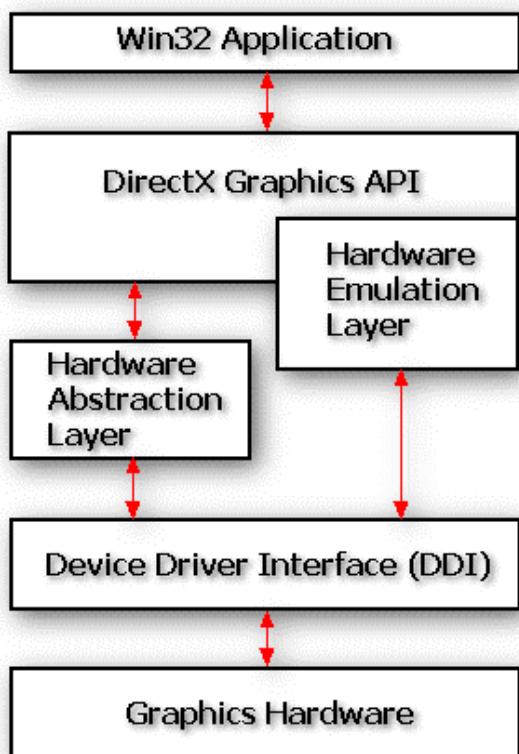
One of the first things your application will need to do is determine whether or not a HAL is present on the current system. If a HAL is available (which is likely the case), then you will generally prefer it to use software emulation. If a HAL is not provided, then the graphics adapter has no DirectX Graphics support. In this case you can choose to use the DirectX Graphics **Hardware Emulation Layer (HEL)**. When you use the HEL, all transformation, lighting and rasterization of polygons is done on the CPU. The DirectX Graphics software emulation module is called the **Reference Rasterizer**. It emulates all of the DirectX Graphics features but is not viable for commercial purposes due to performance constraints.

The reference rasterizer is useful for testing DirectX features when development hardware does not support all of the DirectX Graphics features your game will use. If you were developing an application that used bump mapping, and your test hardware did not support bump mapping, you could use the reference rasterizer to test your code. This ensures that users who have hardware that supports bump mapping can still enjoy it in your game.



Features supported in the HAL vary widely across video hardware. Our application must be flexible enough to ensure that it does not attempt to use features that are not available while taking advantage of those that are.

Fig 2.1 shows the relationship between the application and hardware layers:



If no HAL exists on the system, or if the application has decided not to make use of it, then DirectX Graphics will emulate all functionality in software using the HEL (reference rasterizer).

The reference rasterizer is slow but is useful for testing features not supported in hardware on your development machine.

The HAL does not provide emulation of DirectX features when a feature is not supported by the underlying hardware.

**Figure 2.1**

When graphics hardware supports the entire transformation and rendering pipeline, this frees up the system CPU to handle other game tasks like AI and physics. Most users will have cards capable of hardware rasterization, but not everyone will have full T&L support. The latest cards from nVidia (the GeForce™ family) and more recently ATI (the Radeon™ family) support the transformation and lighting of vertices in hardware.

The DirectX Graphics environment must be initialized appropriately to ensure that your application can take advantage of the best features available on a given system. Failure to properly initialize DirectX Graphics could result in significant performance loss or even total software failure. This will be the main focus of the first part of this chapter.

Using DirectX Graphics to draw 3D shapes will be our focus in the second part of this lesson.



## Initializing DirectX Graphics

The IDirect3D9 object interface provides access to core DirectX Graphics functionality. Creating this interface is typically one of the first things our application will do during initialization. DirectX Graphics contains a global function to handle creation:

```
IDirect3D9 *Direct3DCreate9(UINT SDKVersion);
```

This is how it would be called from our code:

```
LPDIRECT3D9 pD3D;  
pD3D = Direct3DCreate9( D3D_SDK_VERSION );
```

The function accepts a single unsigned integer parameter. The integer identifier D3D\_SDK\_VERSION is defined in the d3d9.h header file and ensures that the application is built with the correct header file versions. The function creates the Direct3D9 COM object, increases its reference count, and returns an IDirect3D9 interface to the object. Direct3DCreate9 is the only global non-COM function that DirectX Graphics provides (excluding D3DX). All other functionality will be accessed using COM methods either directly or indirectly through the IDirect3D9 interface. A short COM tutorial can be found included with this chapter for those who are not familiar with the model.

IDirect3D9 exposes methods that allow the application to query the hardware capabilities of the current system. This interface is also used to create the Direct3DDevice9 object and retrieve a pointer to the IDirect3DDevice9 interface. The IDirect3DDevice9 interface provides the functionality our application will use most of the time.

In order to create a proper Direct3DDevice9 object, we will need to know the capabilities of the hardware installed on the system. For example, cards like the Voodoo 1™ and the Voodoo 2™ are 3D accelerators with no 2D support. As a result they exist alongside another graphics card which provides that 2D functionality. So there may be two physically separate 3D hardware accelerated devices on the system. Since we can only use one of them, which one do we choose? If we choose incorrectly our application is not likely to perform as well as it should. We may wind up using the CPU when there was hardware acceleration available on the video card.

The IDirect3D9 interface provides functions for querying the number of graphics adapters installed on the system as well as functions for querying the capabilities of each of those adapters. So the main purpose of this object is to gather information that we can use to create the most optimal Direct3DDevice9 object possible on an end user system. Some of the key functions of this interface are shown below. This is not a complete list, but it does provide the core functionality we will need in this lesson:

```
UINT    GetAdapterCount      (VOID);  
HRESULT GetDeviceCaps       (UINT Adapter,   D3DDEVTYPE DeviceType,  D3DCAPS9* pCaps);  
UINT    GetAdapterModeCount (UINT Adapter , D3DFORMAT Fomat );  
HRESULT GetAdapterDisplayMode(UINT Adapter,   D3DDISPLAYMODE* pMode );  
HRESULT CheckDeviceType    (UINT Adapter,   D3DDEVTYPE CheckType,  D3DFORMAT DisplayFormat,
```



```
HRESULT EnumAdapterModes           D3DFORMAT BackBufferFormat,    BOOL Windowed );
(HRESULT Adapter,      D3DFORMAT Format,     UINT Mode,
D3DDISPLAYMODE* pMode);
HRESULT CreateDevice               (UINT Adapter,      D3DDEVTYPE DeviceType,   HWND hFocusWindow,
DWORD BehaviorFlags,
D3DPRESENT_PARAMETERS* pPresentationParameters,
IDirect3DDevice9*** ppReturnedDeviceInterface);
```

The structures and enumerated types used as parameters will be covered later in the lesson. For now we will briefly explore some of these functions so that we can begin to understand system capability querying.

**GetAdapterCount** – This function returns the number of physical display adapters available on the current system. The value returned will usually be 1; indicating only one display adapter is installed. The first graphics card installed is typically referred to as the **primary display adapter**.

While it is true that only one adapter will exist on the vast majority of systems, we still want our code to handle cases where more than one is present. Although we could choose to ignore the other adapters and simply use the first adapter found, we risk not selecting the most capable adapter available. The Enumeration class that we will build in our final Lab Project for this lesson will let the user choose which adapter they wish to use.

**GetAdapterDisplayMode** – This function returns the current display mode of the adapter identifier. Each adapter on the system is assigned an integer index between 0 and AdapterCount – 1. This is referred to as the **adapter ordinal**. If you need to find out information about the current display mode, the D3DDISPLAYMODE returned will include this information (resolution, color bit depth, and so on). If the adapter is the primary display adapter currently being used to display the Windows desktop, then the display mode returned will be equal to the resolution and color depth you have your desktop set to.

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

The D3DDISPLAYMODE structure contains the width and height (in pixels) of the current display mode, the monitor refresh rate, and the display surface pixel format.

**EnumAdapterModes** – In DirectX Graphics, there are a number of formats that describe how image pixels are represented in memory. The D3DFORMAT enumerated type contains all of the formats currently supported by DirectX. When we create our game, we will want it to run in a variety of different video resolutions given the wide range of hardware capability across the marketplace. People with low-end machines might need to run our game in a resolution of 640x480 for better performance while users with high-end machines can run 1600x1200. This function allows us to request a list of video resolutions available for a given pixel format.



For example, let us assume that there is one adapter installed in the system (the primary display adapter) and that we desire a display mode with a 16 bit color format of **D3DFMT\_R5G6B5** (5 bits for red, 6 for green, and 5 for blue in every pixel). We could use the following code to find out if this format is supported by the adapter:

```
D3DDISPLAYMODE           Mode;
UINT                   AdapterOrdinal = 0;
D3DFORMAT              Format = D3DFMT_R5G6B5;
LPDIRECT3D9            pD3D;

pD3D = Direct3DCreate9( D3D_SDK_VERSION );
if (!pD3D) return FALSE;

UINT NumberOfModes = pD3D->GetAdapterModeCount (Adapter, Format);

if (!NumberOfModes) return FALSE;

for (UINT I=0; I < NumberOfModes; I++)
{
    pD3D->EnumAdapterModes( Adapter, Format , I , &Mode);
    FormatModeList->push_back(Mode);
}
```

In this example we tested for **D3DFMT\_R5G6B5** format support (generally available on most cards). We use an adapter ordinal of 0 (the number of the default adapter) and do not iterate through all adapters on the system. Next we create the Direct3D9 object and use one of its member functions to query the number of video modes the adapter supports for that pixel format. For example, the adapter may support 640x480, 800x600 and 1024x768 video modes -- all using the **D3DFMT\_R5G6B5** format. If this was the case, then the number of modes returned would be 3. More recent hardware may support many more modes than this (sometimes going to resolutions beyond 2000 pixels in a single dimension). Of course, in a commercial application we would not look for one particular format. We will write some code later in the lesson that will search all formats available. If the desired video mode format is not available, we will try another until we find the best match.

If the number of modes returned is zero, the graphics card does not support this color format. This is not unusual as there are a number of 16 bit color formats available and it may use one of the others instead. In a real application we would continue to test other possible 16 bit formats until we found a suitable match.

Next, we loop through the number of available modes and call the `IDirect3D9::EnumAdapterModes` function. This function parameter list includes the adapter ordinal, the desired pixel format, and the number of modes we wish to retrieve. For each format an adapter supports, there is a list of display modes containing 0 to modecount - 1 elements. This function asks for details of the display mode at a given index in that list (the third input parameter 'I' above). Details are returned in the `D3DDISPLAYMODE` structure whose address is contained in the last parameter. This structure will contain the width and height of the mode, the format itself (which we passed in) and the refresh rate. Note that it is quite possible that many of the modes returned have identical width, height, and format settings, and differ only with respect to refresh rate. This reflects



the wide range of capabilities present on current monitors. Each is copied into an STL vector called FormatModeList and at the end of the loop the vector will contain all display modes available for the D3DFMT\_R5G6B5 color format on that system. A brief STL vector tutorial is included in the Appendices to this chapter if you are unfamiliar with its usage.

**Note:** You should always use the format returned in the D3DDISPLAYMODE structure from EnumAdapterModes to create your device object. Although we pass in the format that we wish to have modes enumerated for, the format returned in the D3DDISPLAYMODE structure is not always guaranteed to be the same for certain 16 bit formats. The formats, D3DFMT\_X1R5G5B5 and D3DFMT\_R5G6B5 are two commonly supported 16 bit pixel formats. In some cases an adapter will only support one or the other. The EnumAdapterModes function will return the version that the hardware supports in the D3DDISPLAYMODE structure. So if you enumerated all modes for D3DFMT\_X1R6G5B5 but the graphics card only supported D3DFMT\_R5G6B5 then the latter format would have its modes enumerated. This is the only case where this is true. For all other formats the function will not succeed if the explicit format passed is not supported by the graphics adapter.

## The Direct3D Device

Once we have used the IDirect3D9 object to gather information about the current system, we will create a device object based on that information. The device object can be thought of as a black box that encapsulates the transformation pipeline, rendering to the frame buffer, pixel blending, depth testing and texture mapping -- using hardware acceleration when available.

In many respects the device object is a 3D engine. At a very basic level, we tell it to render a polygon by passing a collection of vertices to the `IDirect3DDevice9::DrawPrimitive` function. This is very much like the way we passed vertices into our software transformation and rendering code in Lab Project 1.1. The vertices are passed through a series of computations to arrive at the screen representation of the polygon. Unlike our simple software rendering demo, the polygons rendered by the device can have lighting effects applied to them, multiple textures blended onto them, and even have several color blending operations done at the per-pixel level to allow for transparency.

The device is also a state machine that can be controlled through member functions (e.g. `IDirect3DDevice9::SetRenderState`). These states control the way the device transforms and color blends our polygons onto the screen. Any state that is set will remain set until we unset it or set it to something else. If we set the device to wireframe render mode for example, every polygon drawn will be rendered in wireframe until we set the render state to some other value (such as solid fill mode).

Using the device states to control the transformation and lighting of vertices is referred to as **fixed function pipeline** rendering. There will be times however when even all of the many render states available do not provide the results you want. Beginning with DirectX 8, Microsoft exposed the rendering pipeline to the developer using something called programmable shaders. Shaders allow the developer to create small code modules for transforming and lighting vertices and coloring pixels



instead of using the fixed function pipeline. We call this the **programmable pipeline**. Shaders will be covered in detail during the next course in this series.

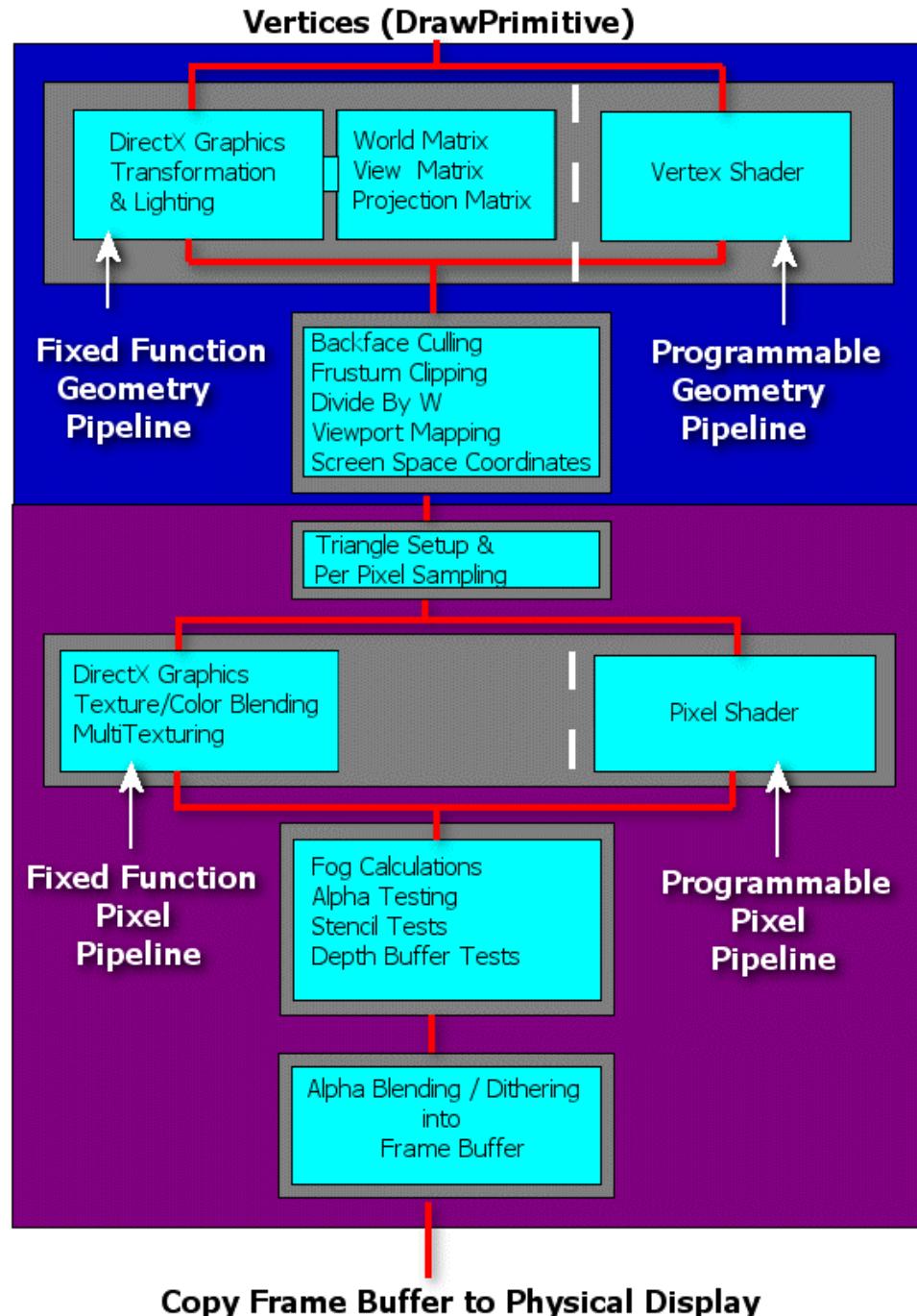


Figure 2.2



Fig 2.2 provides a representation of the device object and the software modules that it contains. The device is divided into two main sections: the vertex processor (BLUE) and the pixel processor (PURPLE).

## Pipeline Overview

Vertices are sent to the device using the `DrawPrimitive` function(s). As we did in Lab Project 1.1, we will pass in world, view, and projection matrices so that the device can perform the necessary transformations. We set each matrix using the `IDirect3DDevice9::SetTransform` function prior to rendering an object. The function takes as its first parameter a member of the `D3DTRANSFORMSTATETYPE` enumerated type. This tells the device which of the matrices is being passed (projection, view or world). The second parameter is a pointer to the matrix itself.

```
HRESULT SetTransform(D3DTRANSFORMSTATETYPE State, CONST D3DMATRIX* pMatrix);
```

At application startup we might create a projection matrix and send it to the device as follows:

```
m_pD3DDevice->SetTransform( D3DTS_PROJECTION, &m_mtxProjection);
```

Each frame we can create a view matrix which would contain the position and orientation of the virtual camera in our world. Before rendering any objects we would use the `SetTransform` function again to set the device view matrix:

```
m_pD3DDevice->SetTransform( D3DTS_VIEW, &m_mtxView);
```

Finally, before we render each object's polygons, we send the object world matrix to the device:

```
for(I=0; I<NumberOfObjects; I++)
{
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i].m_mtxWorld );
    m_pD3DDevice->DrawPrimitive ( All Object[I] Polygons); //pseudo function call
}
```

Polygons are transformed by these matrices in a manner similar to what we saw in our software demo.

Once the device has transformed (and lit – see Chapter 5) the vertices, it performs backface culling (if enabled) to remove polygons facing away from the viewer. It then performs the divide by w to perspective project the vertices into 2D projection space coordinates in the range of -1 to 1.

The transformed vertices now enter the pixel pipeline. The device will set up the outline of the polygon in screen space and then draw that polygon one scan line at a time -- and ultimately one pixel at a time. Once the device has interpolated the depth and color values for a pixel (using a weighted interpolation between the vertices and their depth and color values), the pixel is sent through the rest of the pixel pipeline where it may have texture and/or fog effects applied that alter its color. The pixel depth value



is then tested against the depth buffer (and possibly the stencil buffer if one is being used) to see if it is closer to the camera than a previously rendered pixel. If so, it enters the final phase where it may be color or alpha blended with a pixel already stored in the frame buffer. If not, it is discarded.

Do not be too concerned if this description is a little overwhelming. We will be dealing with every element described above as we progress through this course.

### Device Memory

The device owns and maintains memory for a number of important data storage buffers. The frame buffer (and usually the depth buffer) memory will be created when we create the device. Device memory can also be allocated by our application for assets like texture images and mesh geometry. These memory buffers are referred to as **device resources**. Having these resources available in device memory provides maximum speed on T&L hardware. Fig 2.2 shows the memory buffers owned by the device for an application that uses four vertex buffers (perhaps to hold the vertices for four different meshes), four textures, a depth buffer, and a frame buffer.

### The Device

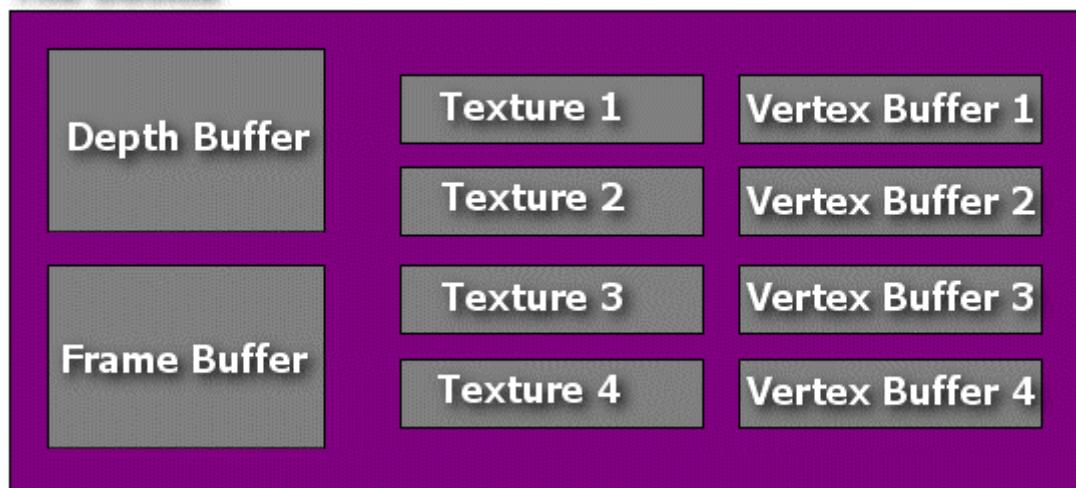


Figure 2.3

Although it is likely that your application will use all of the memory buffers types in Fig 2.3, only a frame buffer is required. Later we will see how our application can configure the device to control where resources are stored.



## Frame Buffers

The frame buffer (or **back buffer**) is a memory buffer where the image of our 3D scene is rendered prior to displaying the output on the screen. This approach allows us to minimize or even avoid certain artifacts that may occur if we rendered directly to the screen buffer. We discussed some of these artifacts in Chapter 1. We saw that a frame buffer was critical because proper scene rendering required that we erase the prior frame image before displaying the new one. If we were to try to do this on the physical display, the user would see the image flicker as it was erased and then redrawn at high speeds. The frame buffer solved this problem by clearing and rendering to an off-screen memory buffer. Only after the scene was completely rendered did we copy it to the screen and replace the existing image.

## Refresh Rate

The speed at which the monitor screen repaints itself is referred to as the **refresh rate**. Refresh rate is measured in Hertz (1/sec). A refresh rate of 60 Hz means the monitor repaints itself 60 times per second. The higher the refresh rate, the more rapidly the monitor can react to changes in the image rendered to the screen. There is typically a block of memory on every video card that is used to map images directly to the monitor screen. When the screen is repainted by the electron gun, it gets information about how it should be painted directly from this display memory. When changes are made within this address space, it changes the image seen by the viewer.

The electron gun inside the monitor starts at the top left corner of the display. Each line of the monitor display is called a **scan line** and is refreshed as the electron gun moves left to right. At the end of each scan line, the electron gun is moved to the beginning of the next line to repeat the process until the entire screen has been refreshed.

### The Vertical Retrace Period

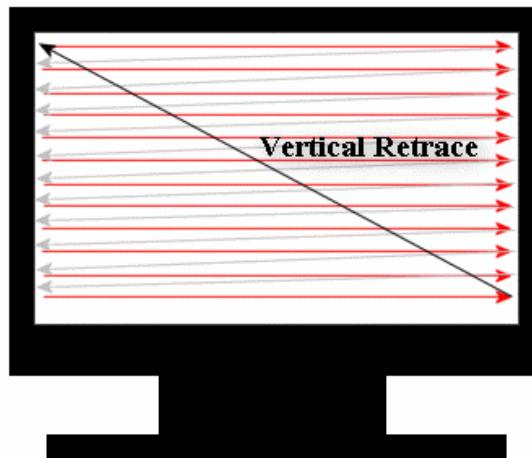
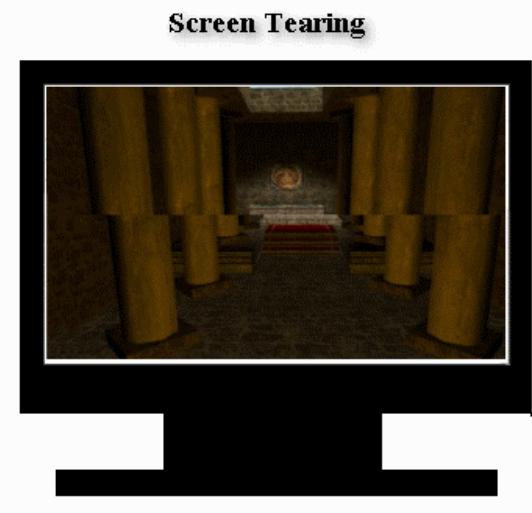


Figure 2.4



If we copy the frame buffer to the display memory while the electron gun is halfway through repainting the screen, the new image will be displayed only on the bottom half of the screen for a fraction of a second. This is because the top part of the display has already been repainted by the electron gun using the image that was previously in display memory, while the new image is used for the second part of the repaint. Although this corrects itself very quickly due to high refresh rates, it is still noticeable to the viewer. We call this visual artifact **tearing**.



**Figure 2.5**

In Fig 2.5 the current frame in the frame buffer is slightly skewed to the right with respect to the previous frame's camera setting (currently being displayed in display memory). When the frame buffer is copied to display memory and the electron gun is only half way through a repaint, the bottom half of the screen is updated with the new image. The top half of the screen will not be updated until the next repaint.

Referring back Fig 2.4 we notice that there is a time at which the electron gun reaches the bottom right corner of the screen and has to stop repainting and return to the top left corner for the next repaint. During this time the electron gun is not painting the screen, so this will be an ideal time for us to copy the frame buffer to display memory. This period of time during which the gun retraces from bottom right to top left is called the **vertical retrace** period (sometimes called the **vertical blank**). While the vertical retrace time is indeed quite short, we can be assured with modern hardware that we can copy the entire frame buffer to the display memory within that time block to prepare for the next monitor repaint.

We will tell the device that we want to synchronize our frame buffer with the vertical retrace period. This is called VSYNC. When given a command to present the frame buffer, the device will wait until the vertical retrace starts before it performs the copy operation from the frame buffer to display memory.



**Note:** Some commercial games allow the user to disable VSYNC in order to increase the responsiveness of the game and increase the frame rate slightly. This usually comes at the cost of visual artifacts such as screen tearing. When VSYNC is disabled in such games, the frame buffer is copied to display memory as soon as the scene is rendered and no waiting for the vertical retrace occurs.

### The Front Buffer

The display memory used by the electron gun to repaint the monitor is sometimes called the **front buffer**. DirectX Graphics enforces the use of a frame buffer by denying the application access to the front buffer -- all rendering must be done to the frame buffer. When the frame buffer is complete, we call `IDirect3DDevice9::Present` to tell the device object to copy or promote the frame buffer to display memory.

It should be noted that while you cannot directly access or alter the image in the front buffer, the `IDirect3DDevice9` interface does have a function called `IDirect3DDevice9::GetFrontBuffer`. This function will return a *copy* of the image in the front buffer only. Altering this returned image will not alter display memory. This can be useful for taking a screen shot of your application.

**Note:** `IDirect3DDevice9::GetFrontBuffer` is the only way to take a screen shot of an anti-aliased scene.

### Swap Chains

It is possible to create more than one frame buffer for a device. When more than one frame buffer is used, this is called a **swap chain**. Consider a scenario where your application tells the device to present the frame buffer to display memory. The device may have to wait until the vertical retrace period before it can present the frame buffer. Your application will essentially wait for the all clear signal to render the next frame; which it cannot do until the current frame buffer has been presented. If a swap chain is used, you can continue to render the next frame into the next frame buffer in the swap chain. This can speed things up under certain circumstances and may even smooth out erratic frame rates, but it comes at a cost. At high screen resolutions (especially in 32 bit color) each frame buffer can take up a considerable amount of precious video memory. This is memory often best reserved for resources that need to be accessed frequently by the device (like textures or vertex buffers).

The process of using two frame buffers (plus the front buffer) is called **triple buffering**. The more typical approach uses just one frame buffer and is called **double buffering**. We will be using double buffering for most of the demo applications in this course. DirectX allows swap chains with as many as four frame buffers.

Fig 2.6 shows the relationship between the front buffer, the frame buffer, and the physical display. All polygon rendering is done through the device to the frame buffer. When we have finished rendering

the frame, we tell the device to present the frame buffer to the user. The device then takes the current image in the frame buffer and puts it into the front buffer when the next vertical retrace period starts. If we have disabled synchronization with the vertical retrace period, then the device will put the frame buffer image into the front buffer immediately. Each time the monitor is repainted, it takes the information about what to display from the front buffer. Notice that even though the frame buffer and the front buffer are both located in memory on the video card, sometimes referred to as **local video memory**, only the front buffer memory is used to repaint the physical display:

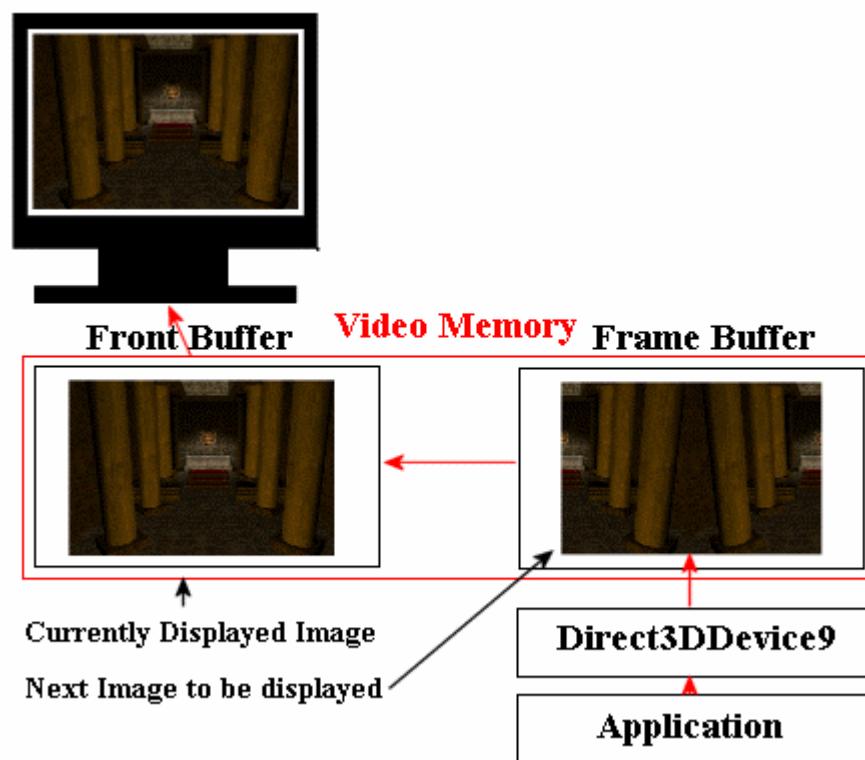


Figure 2.6

## Screen Settings

When we create a Direct3DDevice9 object at the start of our application, we also have to choose a windowing mode to operate under. Most commercial games use a fullscreen mode. In fullscreen mode, the 3D image covers the entire display area. Alternatively, windowed mode games run alongside other applications on your desktop. As you will discover for yourself, this mode is critical during the development phase of your application.



## Fullscreen Mode

When we create the device object, we query the current hardware to see which fullscreen video modes it supports. Once we select a resolution and color depth (or let the user choose from a list) we create a fullscreen device that physically puts the graphics hardware into this video mode. The Windows desktop will no longer be visible and the front buffer will take up the entire screen.

In fullscreen mode, the frame buffer created for the device must be exactly the same size and color depth as the front buffer. If we choose to create our device so that it operates in a video mode of 640x480 in 16 bit color (640x480x16), the frame buffer should also be created to these specifications.

In this mode the device can perform a fast presentation from the frame buffer to the front buffer using a technique called **flipping** -- a feature available on virtually all current graphics hardware. Flipping essentially amounts to a pointer swap. The video card has two pointers; one to the current frame buffer and the other to the front buffer. The monitor is repainted by the image pointed to by the front buffer pointer. Drawing commands issued by the application to the device take place in the area of video memory pointed to by the frame buffer pointer. Once we have rendered the scene in the frame buffer and it is ready for presentation, the device (in a double buffer system) just swaps the pointers. Now the front buffer pointer points at the old frame buffer and the frame buffer pointer points at the old front buffer. This is much faster than the alternative which is called **blitting**, where every pixel would need to be copied between the buffers.

Once the swap has taken place, the new frame buffer replaces the old front buffer and all drawing commands are directed to the current frame buffer (the old front buffer). When the next image is complete the device will once again swap the two pointers. Fig 2.7 shows this concept in action over two consecutive frames:

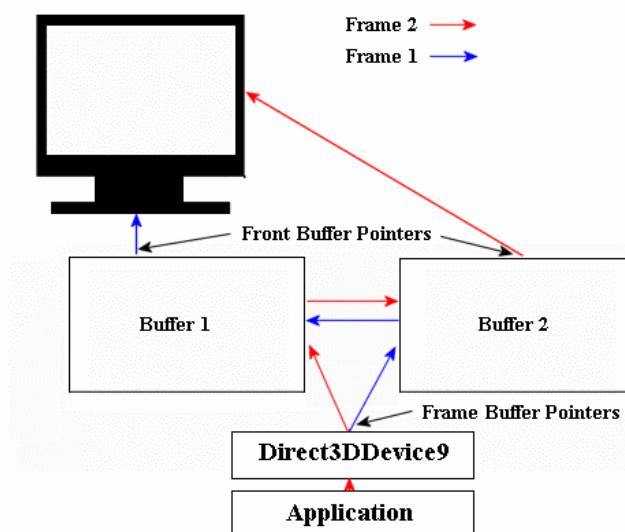


Figure 2.7



The blue arrows show the buffer arrangement during frame one. The device draws directly to Buffer 2 (the current frame buffer). Buffer 1 is the front buffer and contains the image currently being displayed on the monitor. When the device is told to present the image in the frame buffer, the pointers are switched so that Buffer 2 is now the front buffer. Its contents (the image we just rendered) are painted by the electron gun. When we render frame two, the pointers have been swapped. The device now draws directly to Buffer 1 while Buffer 2 is used as the front buffer. When the device is told to present the frame buffer, the pointers are switched again, and Buffer 1 is promoted to the front buffer, with Buffer 2 becoming the frame buffer for the next frame. And so it goes for the lifetime of the application.

### Windowed Mode

In windowed mode the desktop is not hidden and it shares the current video mode with other applications that may be running, including yours. Thus the video mode cannot be changed. Flipping cannot be used because the front buffer is mapped directly to the client area of the application window. In windowed mode, the frame buffer is copied to the client area pixel-by-pixel each time we present the scene. Although this blitting process is handled by the device, it is likely to be slower than flipping.

Movement of the application window by the user is handled automatically by DirectX Graphics. However, until the release of DirectX 9.0, resizing the window was not. Until now, when the user resized the window (WM\_SIZE) our message handler would need to tell the device to rebuild its swap chain so that the frame buffers matched the new dimensions of the front buffer. We may still decide to do this anyway, but it is no longer a requirement.

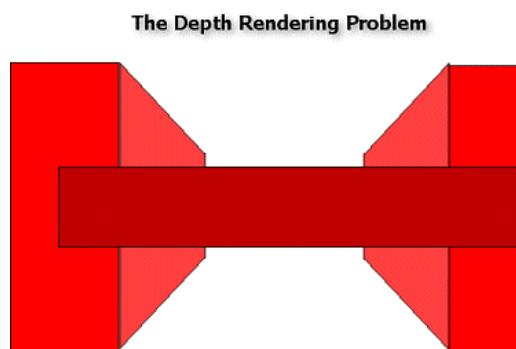
As long as the device is in windowed mode, the frame buffer(s) does not have to be the same size or color format as the front buffer. DirectX Graphics will automatically shrink or expand the frame buffer image to fill the front buffer, which in this case is the window client area. The same is true with color depth. It is now possible in windowed mode to have a 32 bit frame buffer even when the desktop (and therefore your front buffer) is in 16 bit color mode. DirectX Graphics will perform the color conversion when the image is copied from the frame buffer to the front buffer. Because all of this conversion and resampling will be slower, you should still try to keep the formats and sizes matched up for optimal performance. Note that the above features are only true when running in windowed mode. In fullscreen mode the frame buffer must be the same size and format as the front buffer.

When we create our windowed mode device in our lab projects, we will use the current desktop display mode for our frame buffer. This makes environment setup much easier for windowed mode applications as we will see in Lab Project 2.1.



## Depth Buffers

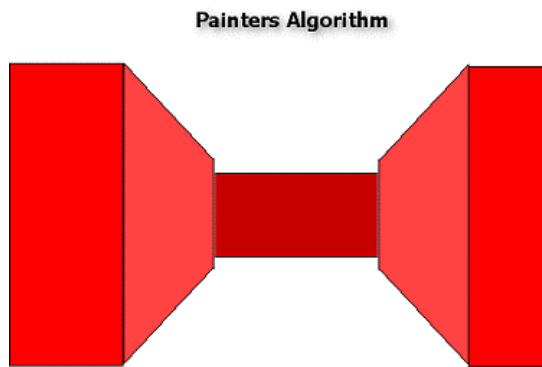
One of the trickiest parts of creating a 3D game used to be making sure that the polygons in the scene were rendered in such a way that polygons nearer to the camera were rendered on top of polygons further away. While this is not an issue when rendering in wireframe mode, when we use filled (solid) polygons this is a very significant problem. Polygons cannot just be rendered in any random order without potentially damaging the integrity of the scene. In Fig 2.8 we see an example of polygons forming a corridor section of a game world viewed from the player location. If we rendered the wall polygons in no particular order (perhaps just using the order they were stored in the mesh), we might render the wall furthest from the camera last:



**Figure 2.8**

The horizontal dark red polygon is supposed to be forming the back wall of the passage where it meets in a T-junction. It should be partially obscured by the nearer polygons to give the illusion that it is further away. We cannot simply define our meshes so that the polygons are ordered correctly because the drawing order will depend on the viewing angle and position of the player. These values will change as the player moves around the world.

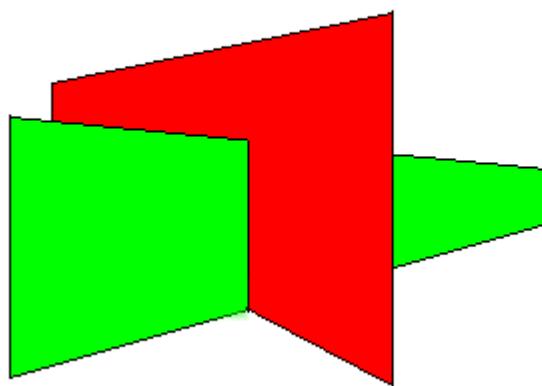
One way to solve this problem is with a technique called the **Painter's algorithm** (Fig 2.9). The polygons in the scene are sorted into a back to front ordered list prior to rendering. Polygons further from the camera are rendered first and polygons close to the camera are rendered last, drawing over the distant polygons. This is similar to how a painter builds up the scene on his canvas; painting background objects first, followed by foreground objects.



**Figure 2.9**

The Painter's algorithm worked well for the above case but it is not suitable for the complex worlds we expect in commercial games today. Sorting all of the visible polygons before rendering would seriously diminish performance if thousands of polygons or more were visible on the screen at once. Many polygons will be rendered only to be overdrawn by nearer polygons. We will also have difficulties choosing a sorting criterion and would have to settle for an approximation that can be applied to the whole polygon. For example, we could use the nearest vertex position in the polygon and calculate its distance from the camera and use that to sort polygons. Or we could try to find an average distance using all of the vertices' distances from the camera. No matter what criterion we decide to use, it will not suffice in all situations (Fig 2.10).

### A Sorting Paradox



**Figure 2.10**

If we render the green polygon in Fig 2.10 first, then the portion of the red polygon that is supposed to be behind it will be rendered in front. If we render the red polygon first, then the portion of the green polygon that is supposed to be obscured by the red polygon will be rendered in front. The Painter's algorithm cannot resolve this.



The solution is to work with smaller units. Eventually these polygons will need to be rendered at the per-pixel level. While a particular vertex might not be behind another polygon, when pixels are interpolated across the polygon from one vertex to another, the pixel itself might be obscured because a closer polygon has already had its pixels rendered there. Ideally the current pixel would not be rendered in this case. So we need a per-pixel test that allows us to figure out whether a given pixel should be rendered or whether a pixel that is closer to the viewer has already been rendered in that location in the frame buffer.

### The Z-Buffer

The most popular depth solution creates a memory buffer that is the same size as the frame buffer. Instead of each buffer location holding a pixel color, it will store the interpolated Z depth value for each corresponding pixel in the frame buffer. This technique is known as Z-Buffering and the memory buffer itself is referred to as a Z-Buffer.

In Chapter 1 we discussed the projection matrix transformation. We saw that it takes a vertex from view space to homogenous clip space prior to the divide by  $w$ . It is possible to ensure that when the vertex Z value is output from the projection matrix and divided by  $w$ , it ends up in the range [0.0, 1.0]; where 0.0 represents a vertex very close to the viewer and 1.0 represents a vertex at the furthest possible point from the viewer. This is not a pure distance value mind you. It is simply the view space Z component of each vertex mapped to the range [0.0, 1.0]. This will suffice however because the sorting problem is a view space problem. When the device renders a polygon, it will perform a linear interpolation between the Z values stored at each vertex to produce a Z value for each pixel. This Z value provides us with a relative distance from the viewer to each pixel that we render.

Before we render our scene we will clear the Z-Buffer to the maximum Z distance that can be stored. For example, let us say that the Z-Buffer is a BYTE array. Each element can hold a number between 0 and 255. So in this case we will set every element in the buffer to 255 (the maximum depth value).

Next we render our polygons. After the polygon is transformed into screen space, we calculate the Z component for each pixel based on an interpolation of the Z values stored at each vertex in the polygon. Once we have the pixel depth value, we compare it against the corresponding value stored in the Z-Buffer. Every pixel in the frame buffer has a corresponding entry in the Z-Buffer describing its distance from the viewer.

If the value already stored in the Z-Buffer is smaller than the depth value of the current pixel about to be rendered, then it means another pixel has already been rendered at this location in the frame buffer that is closer to the viewer than the one we are currently about to draw. In this case we discard the current pixel and move on to the next one.

If the depth value of the pixel we are about to render is smaller than the corresponding value in the Z-Buffer, then the pixel we are about to render is closer to the viewer than any we have previously



rendered in that position up to this point. So we should render the current pixel and overwrite the pixel residing in that frame buffer location. After we do this, we store the current depth value in the corresponding Z-Buffer location overwriting the depth value that was previously there.

In Fig 2.11 we see a low-resolution frame buffer and depth buffer. We used a 5 bit Z-Buffer where each value falls between 0 and 16. Before the scene is rendered, the Z-Buffer is cleared so that every location contains the maximum depth value of 16. Then we render our polygons:

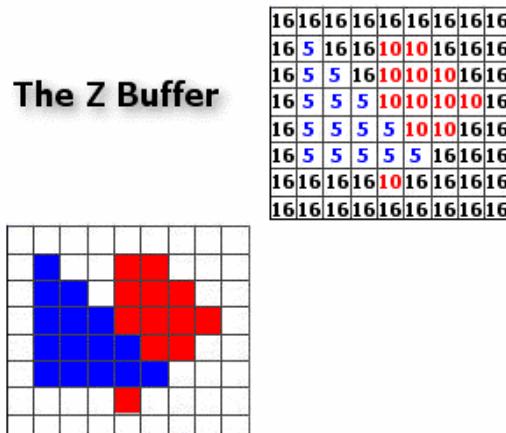


Figure 2.11

Because we are doing per-pixel tests using the Z-Buffer values, rendering order no longer matters. If we rendered the red triangle in Fig 2.10 last and tried to write a pixel where the blue triangle already had a pixel, the Z-Buffer test would fail because a 5 would already be stored at that location. Since this is less than the depth of the red pixel (10) we would be about to render, the red pixel would be discarded.

Per-pixel tests in software are very expensive simply because there are going to be so many of them. Fortunately, virtually all 3D graphics cards support Z-Buffers in hardware and our applications can use them without any performance concerns. The DirectX Graphics device object will handle depth testing for us automatically. We simply instruct it to create a Z-Buffer when it creates the frame buffer at application startup and activate the appropriate render state. When we render our polygons, the device will record the depths of each pixel in the Z-Buffer and perform the per-pixel depth tests at high speeds.

Our application must query and select a Z-Buffer format supported by the current hardware and tell the device to use it. We also have to make sure that we setup the 3<sup>rd</sup> column of our projection matrix so that it generates a proper Z value for each vertex. We will discuss this exact process a little later in the lesson.



## Z-Buffer Inaccuracy

Graphics hardware usually supports 16, 24, or 32 bit Z-Buffers and sometimes all three. But it is worth discussing 16 bit in particular because it presents us with some real problems that we will need to solve.

The Z value for each vertex -- and eventually each pixel -- is the result of our projection matrix multiply and the divide by w. This gives each screen space vertex a depth value between 0.0 and 1.0. In code, this is a floating point value and is thus 32 bits wide. In order to fit 32 bit floats into 16 bit Z-buffer entries, two bytes of the float have to somehow be discarded. The clear consequence is the loss of a significant amount of precision.

Let us assume that we need 32 bits to store values with four decimal places and 16 bits to store values with only 2 decimal places. The problem becomes clear if we consider two hypothetical pixels from separate polygons:

32 bit depth values:

Pixel A = 0.1025

Pixel B = 0.1029

16 bit truncated depth values:

Pixel A = 0.10

Pixel B = 0.10

The 16 bit values lost the last two digits in the truncation and both A and B now equal 0.10. The Z-Buffer can no longer tell which pixel should be obscuring the other. If B was rendered after A, it would pass the Z-Buffer test and overwrite A, even though it should not do so. This loss in precision results in unattractive rendering artifacts. Unfortunately, on hardware where only a 16 bit Z-buffer is available, this is mostly unavoidable.

There is another problem with the Z-Buffer. When we calculate the depth value for each vertex in the projection matrix, we need some way to provide DirectX Graphics a Z depth value between 0.0 and 1.0 that it can use for rendering the polygon and interpolating per-pixel depths. We cannot simply hand it the view space Z value input into the projection matrix because this will eventually get divided by w when the vertex is homogenized. As the W component of the vertex output from the projection matrix multiplication is always equal to the Z value that was input, this equates to:

$$\text{Depth } Z = z / w$$

$$\text{Depth } Z = z / z$$

$$\text{Depth } Z = 1$$

As  $W=Z$  after the projection matrix multiply, the depth value has to be something other than W when it leaves the projection matrix multiply. Otherwise the depth value will always be 1.0. We will discuss



later how we setup the third column of the projection matrix to generate this depth value so that after it is divided by w, it ends up in the 0.0 to 1.0 range depending on its distance from the camera.

The unfortunate and unavoidable problem is that the third column multiply of the projection matrix followed by the divide by w will not linearly map the depth value to the 0.0 to 1.0 range. In fact, most of the time, the first 10 percent of the scene will be mapped to the 0.0 to 0.9 range. That is, 90 percent of the Z-buffer's precision is used up in the first 10 percent of the viewing distance. As a result, all of the depth values for the remaining 90 percent of the scene will be mapped to fractional values between 0.9 and 1.0. This does not present as significant a problem with 32 bit floating point numbers since there is enough precision between 0.9 and 1.0 to generate thousands of unique depth values. 16 bit Z-Buffers do not fare nearly as well, as you might expect. Appendix A at the end of this lesson explores this issue in greater depth.

To be fair, for non-complex scenes, or at least in scenes where all of the objects are relatively close to the camera, a 16 bit Z-Buffer will probably suffice. But for modern game scenes that have many polygons at medium and far distances from the camera, a 16 bit Z-Buffer is insufficient.

Fortunately, most graphics cards that have been released in the last few years support either 24 or 32 bit Z-Buffers. 24 bits usually provide more than an adequate amount of precision to represent all of our depth values accurately. Cards with 32 bit Z-Buffers often allow us to use the last 8 bits for another function entirely, since the first 24 would meet our depth testing needs.

### The W-Buffer

Some graphics cards support a depth buffer variation known as a W-Buffer. W-Buffers use the same per-pixel comparison technique and the same physical video memory buffer as a Z-Buffer. The W-Buffer differs in the way that it calculates the depth values for each vertex, and ultimately each pixel.

When a view space vertex is multiplied with the projection matrix, we end up with an output vertex where W is equal to the Z component of the input vector ( $W = Z_{view}$ ). W-Buffers use the reciprocal value for depth testing:

$$\text{Depth} = 1/w$$

This provides a more distributed linear mapping than for Z-Buffers. However, using a W buffer can still produce artifacts when many of the objects in the scene are close to the camera. Contrast this with the Z-Buffer which has 90 percent of its precision in that range. Nevertheless, the W-Buffer has a lot more precision available for objects in the middle to far distance range from the viewer. The choice of whether to use a Z-Buffer or W-Buffer depends on whether your objects are dispersed evenly over the view distance (use a W- Buffer) or whether your objects are typically going to be close to the camera (use a Z-Buffer).



Because most cards now support 24 bit Z buffers the need for W buffers is not as great. This is fortunate since W buffers are not as widely supported on modern hardware as Z buffers. However, if your application does find itself on a system where only a 16 bit Z Buffer is available, a W buffer (if available) can often produce better results.

The device object manages W buffer calculations for our application just as it does the Z buffer. We will generally only need to check for support and then specify our preference when creating the depth buffer.

## Surface Formats

A **surface** is an object that stores image data. For example, both the frame buffer and depth buffer are physically stored as a surface. Textures are stored as surfaces as well. We carry out per-pixel operations on a surface object by acquiring an IDirect3DSurface9 interface. Surfaces come in a variety of sizes and color bit depths and not all surface formats are supported by all hardware. One of the trickiest tasks when initializing the environment is making sure that:

- The frame buffer surface is created by the device in a format that the hardware supports
- We create a Z-Buffer surface that the hardware supports
- We load our textures into surfaces whose format and type the current hardware supports.

The enumerated type D3DFORMAT contains the surface formats supported by DirectX Graphics. Many graphics cards will indeed support a great number of these formats in hardware, but some formats might not be supported. For example, older cards such as the Voodoo™ 1 and 2 supported only 16 bit colors. None of the 32 bit color formats were available to developers targeting those platforms.

When we create the device at application initialization time, we must tell it the format of the frame buffer(s) we would like constructed. This format must be one that is supported by the hardware. Our environment setup routines will need to obtain a list of supported surface formats on the current hardware and make sure that the frame buffer, depth buffer, and textures are created using only these formats.

Table 2.1 lists the image surface formats we will use in the early stages of the course. These are the formats most commonly supported on modern cards.

**Table 2.1 Common D3DFORMATS**

<b>D3DFMT_R8G8B8</b>	24-bit RGB pixel format with 8 bits per channel.
<b>D3DFMT_A8R8G8B8</b>	32-bit ARGB pixel format with alpha, using 8 bits per channel.
<b>D3DFMT_X8R8G8B8</b>	32-bit RGB pixel format, where 8 bits are reserved for each color.
<b>D3DFMT_R5G6B5</b>	16-bit RGB pixel format with 5 bits for red, 6 bits for green, and 5 bits for



blue.

<b>D3DFMT_X1R5G5B5</b>	16-bit pixel format where 5 bits are reserved for each color.
<b>D3DFMT_A1R5G5B5</b>	16-bit pixel format where 5 bits are reserved for each color and 1 bit is reserved for alpha.
<b>D3DFMT_A4R4G4B4</b>	16-bit ARGB pixel format with 4 bits for each channel.
<b>D3DFMT_X4R4G4B4</b>	16-bit RGB pixel format using 4 bits for each color.
<b>D3DFMT_A2B10G10R10</b>	32-bit pixel format using 10 bits for each color and 2 bits for alpha.
<b>D3DFMT_A8B8G8R8</b>	32-bit ARGB pixel format with alpha, using 8 bits per channel.
<b>D3DFMT_X8B8G8R8</b>	32-bit RGB pixel format, where 8 bits are reserved for each color.

Just to ensure complete understanding of what these formats represent, let us examine the format D3DFMT\_R8G8B8. This might look familiar if you have worked with COLORREFS in Win32. Each pixel on the surface is represented by 24 bits (3 bytes). Each byte can hold a value between 0 and 255 that describes the intensity of the color. If all three bytes were set to 255, then the pixel would be full white. If the second byte was set to 255 and the first and third were set to 0, then the pixel would be bright green.

The format A8R8G8B8 is a 32 bit format where Red, Green and Blue values each receive a byte of storage space. The A stands for **alpha** and is used to measure pixel opacity; it also consumes one byte per-pixel. If the device has **alpha blending** enabled, then when a pixel is rendered into the frame buffer, its alpha value will be used to determine how its color blends with any pixel color currently in that location. In this format the Alpha value would range from 0 to 255 as fully transparent to fully opaque respectively. We will discuss alpha values and transparency in detail in Chapter 7.

## Adapter Formats

When we create a fullscreen device we must choose a format to put the adapter into. This is the format of the front buffer and can only be one of the following:

D3DFMT\_X1R5G5B5  
D3DFMT\_R5G6B5  
D3DFMT\_X8R8G8B8

This is useful because we know that all video cards will at least support one of these three modes. Notice that the front buffer cannot use a format with an alpha channel.

## Frame Buffer Formats



Since windowed mode applications share the desktop, the front buffer must use the format that the adapter is already using. This actually makes setting up the environment for a windowed mode device significantly easier. The frame buffer has no such requirements. In windowed mode, the format and resolution of the frame buffer does not have to match the format of the adapter mode (the front buffer). The device will handle the color conversion between the two when they differ.

There are a number of formats that we can use for the frame buffer:

```
D3DFMT_X1R5G5B5  
D3DFMT_R5G6B5  
D3DFMT_X8R8G8B8  
D3DFMT_A8R8G8B8  
D3DFMT_A1R5G5B5  
D3DFMT_A2R10G10B10
```

Not all of the above formats are guaranteed to be supported by all video cards, so when we setup our device we will need to make sure that we select a valid format. With the exception of the last mode in the list, you should notice that the only difference is that the back buffer supports modes that add an alpha channel to the pixel. You will not often need a frame buffer to have an alpha pixel format. Often, you will simply match the front buffer and back buffer pixel formats exactly.

For fullscreen devices, the formats and resolutions must match, with one exception: the back buffer can still have an alpha channel even though the physical display does not. The rule is that the alpha mode must match the non-alpha mode counterpart (with the placeholder ‘X’ value). If we set the display mode of the adapter to 32 bit, the front buffer format will be `D3DFMT_X8R8G8B8`. This means we can have a back buffer format of either `D3DFMT_X8R8G8B8` or `D3DFMT_A8R8G8B8`. Likewise, if we were in 16 bit mode `D3DFMT_X1R5G5B5`, we could create a back buffer in either `D3DFMT_X1R5G5B5` or `D3DFMT_A1R5G5B5`.

## Creating a Device

Let us now examine the process of creating a device in DirectX Graphics. We begin by looking at the `IDirect3D9` method that provides this functionality.

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

### *UINT Adapter*

This is the adapter ordinal that the device will be created for. Usually there is only one graphics adapter on the system. The primary display adapter is the adapter with an ordinal of 0 (or `D3DADAPTER_DEFAULT`).

### *D3DDEVICETYPE DeviceType*



This parameter defines whether we will create a hardware accelerated device or a slower, software emulated one. For this parameter we pass in one of the `D3DDEVICETYPE` enumerated types:

```
typedef enum _D3DDEVTYPETYPE {
    D3DDEVTYPETYPE_HAL = 1,
    D3DDEVTYPETYPE_REF = 2,
    D3DDEVTYPETYPE_SW = 3,
    D3DDEVTYPETYPE_FORCE_DWORD = 0xffffffff
} D3DDEVTYPETYPE;
```

**D3DDEVICETYPE\_HAL** – the HAL device is typically our preference since it uses the hardware acceleration features on the adapter. If there is no 3D accelerated graphics adapter on the current system, then the request to create a HAL device will fail. We will be left with no choice but to create a HEL device, provide our own renderer and forego DirectX, or exit the application. If a HAL device is created successfully then it means that the hardware has at least some 3D capability. This may be hardware triangle rasterization or it may be the entire transformation and lighting pipeline too. We will have to check the capabilities of the HAL to make sure it supports the functionality we require.

**D3DDEVICETYPE\_REF** - If a HAL device cannot be created then the Hardware Emulation Layer (HEL) is our remaining choice. Outside of feature testing, the reference rasterizer is not viable for commercial applications. Even simple scenes might render at as few as 1 or 2 frames per second. The HEL is really intended for hardware manufacturers and hardware engineers to ensure that their hardware performs correctly. For Example, video card makers can test their development boards against the reference rasterizer to check that their card is not rendering polygons brighter or darker than they should be. The reference rasterizer has helped to maintain image consistency across the variety of different video cards.

Since the reference rasterizer is considered to be of no use for commercial purposes, it is not even enumerated by DirectX Graphics when it is installed. You must manually go to the DirectX properties applet in the Windows control panel and enable it via a check box on the Direct3D properties page if you wish to use it. Our applications will try to create a HEL device if no HAL is found so you should enable this check box. This will be especially important later on in the next course in this series when we cover features that your hardware may not support.

**D3DDEVICETYPE\_SW** – Because of a lack of a commercially usable software device within DirectX Graphics, Microsoft provides developers with the ability to produce pluggable software devices. This allows developers to ship their applications with the ability to run on machines without hardware acceleration. From the application's perspective, it is still using a single unified API. The Driver Development Kit (DDK) can be used to create such software devices. Once the devices are installed and registered with the operating system, they can be enumerated and created as part of DirectX Graphics. DirectX Graphics will pass application requests to the software device driver and the software device will perform the actual task. Unlike the HEL device, software devices will probably not support the entire set of DirectX



Graphics functionality. Certain techniques may also be too processor intensive to run in software.

Creating a software device yourself is a complex task that requires a strong understanding of the processes involved. You are essentially writing your own IDirect3DDevice9 object. Most games no longer offer the choice of running in software mode and require 3D graphics cards. Developers often feel that it is simply not worth the effort when most PCs have 3D hardware acceleration. Some of the very latest games even require vertex transformation and lighting in hardware too.

#### *HWND hFocusWindow*

The window to which the device object will be linked is referred to as the **focus window**. This will most often be the parent window of your application (such as your main application frame window). DirectX traps and dispatches certain messages to and from this window when the device is created, and toggled between fullscreen and windowed modes. Interestingly, the focus window is not necessarily the window where the frame buffer will be rendered. We will discuss this a little later in the chapter. In most cases, passing in the HWND of our main window application will suffice. Also note that if the device will potentially be toggled from windowed to fullscreen mode, the focus window must be a top level window. This is a window that has the WS\_EX\_TOPMOST flag set. If this is not the case, the device will fail to be created in fullscreen mode or fail to be switched to fullscreen mode from windowed mode.

#### *DWORD BehaviourFlags*

There are three mutually exclusive behavior flags that we can use when creating the device to request the maximum level of hardware support. Device creation will fail if the level of hardware acceleration we request is not available. We can then try again using the next best level of hardware acceleration until we eventually find one that is supported. At least one of the following flags must be stated. They are listed in order of desirability.

### **D3DCREATE\_HARDWARE\_VERTEXPROCESSING**

Try to create a device that performs transformation, lighting and rasterization on the video card. This is the maximum level of hardware support that we can request. If we request a D3DDEVICETYPE\_HAL device type and a HAL is present on the system, we can try specifying this flag to create a T&L accelerated device.

If the device creation call fails, we can try the D3DCREATE\_SOFTWARE\_VERTEXPROCESSING flag next. If that device is successfully created then this means that the HAL can perform rasterization in hardware but the transformation and lighting of vertices will be done in software. This is slower, but still acceptable in most cases as a next best option. Very few games at the time of writing require T&L capable video cards, although that will likely change in the future.



Specifying the `D3DCREATE_HARDWARE_VERTEXPROCESSING` flag whilst trying to create a device of type `D3DDEVICETYPE_REF` will succeed, since the reference rasterizer does emulate a hardware device. However, this will not speed up the reference rasterizer in any way. While you may be able to create a HAL device that supports hardware vertex processing, this does not always mean that all vertex processing will be done on the hardware. For example, a video card may only support vertex transformation and not lighting. In this case the driver will perform the lighting calculations in software using the host CPU.

`IDirect3D9` provides a function called `IDirect3D9::GetDeviceCaps` that will retrieve information about device capability. Since this is part of the `IDirect3D9` interface, it can be called to query a device without having to create the device first:

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

We pass the adapter ordinal and the type of device we wish to learn about. We also pass a pointer to a `D3DCAPS9`. This structure contains all of the capability information for the device.

```
D3DCAPS9 DevCaps;
pD3D9->GetDeviceCaps (D3DADAPTER_DEFAULT, D3DDEVICETYPE_HAL, &DevCaps);
```

We will examine this structure in detail throughout this lesson. Our primary interest right now is a DWORD field called `VertexProcessingCaps`. The bits in this field indicate the level of hardware vertex processing supported by the device:

```
D3DVTXPCAPS_DIRECTIONALLIGHTS
Device supports directional lights.  

D3DVTXPCAPS_LOCALVIEWER
Device supports local viewer.  

D3DVTXPCAPS_MATERIALSOURCE7
Device supports selectable vertex color sources.  

D3DVTXPCAPS_POSITIONALLIGHTS
Device supports positional lights (including point lights and spotlights).  

D3DVTXPCAPS_TEXGEN
Device can generate texture coordinates.  

D3DVTXPCAPS_TWEENING
Device supports vertex tweening.  

D3DVTXPCAPS_NO_VSDT_UBYTE4
Device does not support the D3DVSDT_UBYTE4
```

Do not worry about what these flags actually mean for now. We are currently focused on understanding device capability querying only. For example, we could query the `D3DCAPS9` structure to determine whether vertex tweening was supported. This is an advanced technique used to create an intermediate mesh from two or more input meshes. We simply use a bitwise AND operation to do the test:



```
if (DevCaps.VertexProcessingFlags & D3DVTXCAPS_TWEENING)
{
    //Tweening is supported
}
```

If your application needed hardware tweening support and the above test failed, then you would not be able to use hardware vertex processing when creating the HAL device. You would instead choose software vertex processing where the tweening could be done on the CPU.

### **D3DCREATE\_MIXED\_VERTEXPROCESSING**

If a device supports hardware vertex processing but does not support the capabilities that we require, we can attempt to create a device that supports both software and hardware vertex processing. In this case, our application can dynamically switch between the two vertex processing modes.

Continuing the tweening example discussed previously, we could use hardware vertex processing to transform and light the vertices of objects that do not need to be tweened. This affords them maximum hardware acceleration. When we need to render our tweened objects we could switch the device into software vertex processing mode so that the transformation, lighting, and tweening of those vertices would be carried out on the CPU by DirectX Graphics.

### **D3DCREATE\_SOFTWARE\_VERTEXPROCESSING**

If the CreateDevice function has failed to create a HAL device using the flags just discussed, or if you are creating a device of type `D3DDEVICETYPE_REF` or type `D3DDEVICETYPE_SW`, then you will need to pass this flag. All calculations to transform and light vertices will be done by DirectX Graphics on the CPU. If you created a HAL device but were unable to create it with any other flag but this one, it means that the 3D graphics card supports 3D accelerated rasterization only.

It is possible to specify this flag even when the device supports hardware vertex processing. This would force the transformation and lighting to be done by DirectX Graphics instead of the GPU. This may be necessary if the hardware does not support the vertex processing capabilities you require.

The device behavior flags discussed above are mutually exclusive. However, there is another flag that can be combined with `D3DCREATE_HARDWARE_VERTEXPROCESSING` to create a device designed for optimal performance:

### **D3DCREATE\_PUREDEVICE**

If this flag is used with the `D3DCREATE_HARDWARE_VERTEXPROCESSING` type and device creation is successful then it means that the HAL supports a **pure device**.



Recall that the device object is a state machine and that we are able to change state by calling certain functions like SetRenderState or SetTransform. Our application can also query the device to retrieve its current state. For example, we might ask the device to return the contents of its current world matrix:

```
D3DXMATRIX mMat;  
pDevice->GetTransform(D3DTS_WORLD, &mMat);
```

Although our application will generally set these device states to begin with, it is certainly easier not to have to store and maintain state data in persistent variables. Querying back the state data from the device as we need it may be convenient, but it adds overhead. The driver has to ensure that state data can be returned at any time.

When we choose to create a pure device, we are telling the driver that we have no intention of querying the device for such states. The result is that we can no longer use most of the device query functions. This allows the driver and hardware to work more efficiently at a cost of denying the application convenient access to the current state of the device. Generally, this is not a major concern since our application is responsible for setting the states anyway. It is easy enough to store these states in persistent variables that our application can read and update whenever we update the state of the device.

The following code might be used at application startup to create a device. It starts out requesting the maximum level of hardware support and reduces those requirements until it is able to successfully create the best device possible.

```
D3DPRESENT_PARAMETERS d3dpp;  
IDirect3D9 *pD3D;  
IDirect3DDevice9 *pDevice;  
  
// First of all create our D3D Object  
pD3D = Direct3DCreate9( D3D_SDK_VERSION );  
if (!pD3D) return false;  
  
// Try creating a HAL pure hardware device first  
if( FAILED( pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVICETYPE_HAL ,hWnd,  
                                D3DCREATE_HARDWARE_VERTEXPROCESSING | D3DCREATE_PUREDEVICE,  
                                &d3dpp, &m_pDevice ) )  
{  
  
    // Pure device failed so try just hardware device with T&L acceleration  
    if (FAILED(pD3D->CreateDevice(D3DADAPTER_DEFUALT, D3DDEVICETYPE_HAL,hWnd,  
                                    D3DCREATE_HARDWARE_VERTEXPROCESSING , &d3dpp , &m_pDevice)))  
{  
  
        // nope, lets try a software vertex processing hardware device  
        // for accelerated rasterization  
        if (FAILED(pD3D->CreateDevice(D3DADAPTER_DEFUALT, D3DDEVICETYPE_HAL,hWnd,  
                                        D3DCREATE_SOFTWARE_VERTEXPROCESSING , &d3dpp ,  
                                        &m_pDevice)))  
{  
  
            // last resort is the reference rasterizer  
            if (FAILED(pD3D->CreateDevice(D3DADAPTER_DEFUALT, D3DDEVICETYPE_REF,hWnd,
```



```
        D3DCREATE_SOFTWARE_VERTEXPROCESSING , &d3dpp ,
        &m_pDevice)))
    {
        // We couldn't even create a HSL device, something is wrong and
        // your dx app will not run on this machine
        return FatalError;
    } // End Reference Rasterizer
} // End Hal - Software VP
} // End Hal - Hardware VP
} // End Hal - Pure Device
```

## Presentation Parameters

The fifth parameter in the `CreateDevice` function is the address of a `D3DPRESENT_PARAMETERS` structure. It is used to pass information such as the video mode we wish to use (in fullscreen mode only), the width, height, and pixel format of the back buffer, and settings such as which window we wish to render to in windowed mode.

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

### ***BackBufferWidth / BackBufferHeight***

These fields inform the device of the dimensions of the desired frame buffer. They are interpreted based on whether we are going to create a fullscreen or a windowed device.

In fullscreen mode the frame buffer must match the resolution of the physical display mode and these field values must match one of the supported fullscreen video modes enumerated using `IDirect3D9::EnumAdapterModes`. When we create the device, DirectX Graphics will change the current video mode of the hardware so that it matches this resolution, and then it creates a frame buffer of the same size. This allows flipping to be used.

In windowed mode, our application is not allowed to change the video mode resolution since the desktop and other applications are using it. However, there is no need to match the back buffer and front buffer sizes as in fullscreen mode.

If we set these values to 0 in windowed mode, the device will automatically create a frame buffer to match the resolution of the client area of the window it is attached to. This window is represented by



the handle passed in the hDeviceWindow field and is not necessarily the same as the focus window passed in to the `CreateDevice` function. This is the approach we will take in Lab Project 2.1, our first demo in this lesson.

### ***BackBufferFormat***

This field specifies the pixel format for the frame buffer. In fullscreen mode this format will set the video mode for the adapter. If we specify a 32 bit `D3DFMT_A8R8G8B8` format as the back buffer, then the device will change the video mode to use a matching format. Because the front buffer cannot use an alpha channel, this will put the adapter into `D3DFMT_X8R8G8B8` color mode with the resolution specified in BackBufferWidth and BackBufferHeight. This assumes of course that this display mode is supported by the adapter. If the back buffer format does not correspond to one of the supported adapter display modes then device creation will fail. The enumerator class we develop in Lab Project 2.2 will use the `IDirect3D9::CheckDeviceType` function to build a list of modes that can be used with the device on the current hardware. This function allows us to check whether or not a particular back buffer format can be used with a particular adapter mode on both windowed and fullscreen devices.

Since we are using windowed mode in Lab Project 2.1, we can simply use the same format for the frame buffer as the current display mode because we know it will be supported; the adapter is in that mode already.

### ***BackBufferCount***

This field allows you to create a device with more than one frame buffer (double or triple buffering). Valid values are between 0 and 4. 0 is treated the same as 1 since there must always be at least one frame buffer.

If you specify a BackBufferCount that is larger than the number of buffers that can be created on the hardware, the call to create the device will fail and this field in the structure will be filled in with the maximum number of frame buffers that can be created. This allows for subsequent call to `CreateDevice`, passing the now amended structure to resolve the problem.

For our demonstrations, and indeed for most applications, one back buffer will suffice.

### ***MultiSampleType / MultiSampleQuality;***

More recent video hardware has support for multisampling video modes that remove the jagged edges of polygons that are especially visible in lower resolutions. When we enumerate our available video modes, we can record whether the hardware can perform multisampling in that video mode. If so, we have the option to create the device so that it uses it. We will cover multisampling in later lessons so we will set these fields to 0 for now. This informs the device that we do not wish to use available multisampling capabilities.

### ***SwapEffect***

The `D3DSWAPEFFECT` enumerated type describes how the device should transition frame buffer content to the front buffer:

#### **D3DSWAPEFFECT\_FLIP**



In the case of fullscreen rendering, presenting the frame buffer is done very quickly by swapping the frame buffer and front buffer pointers. When we have more than one frame buffer, the swap chain is rotated each time we present the scene. After the flip, the current frame buffer becomes the front buffer and the current front buffer is sent to the back of the swap chain.

**Note:** When we use a flip in windowed mode, the effect of hardware flipping is emulated using pixel copying (blits) between surfaces. The behavior of the swap chain frame buffers is the same from the application perspective. For example, in a double buffered device, after the presentation, the frame buffer will hold the contents of the previous front buffer and vice versa. Using flip in a windowed system carries processing overhead and may consume video memory. This is especially a concern when using D3DSWAPEFFECT\_FLIP with a windowed swap chain of two or more buffers.

Let us take one quick example of a device with two frame buffers (triple buffering). We will render a different shape to each buffer and then repeatedly flip through them:

### Flipping with a Swap Chain (Triple Buffering)

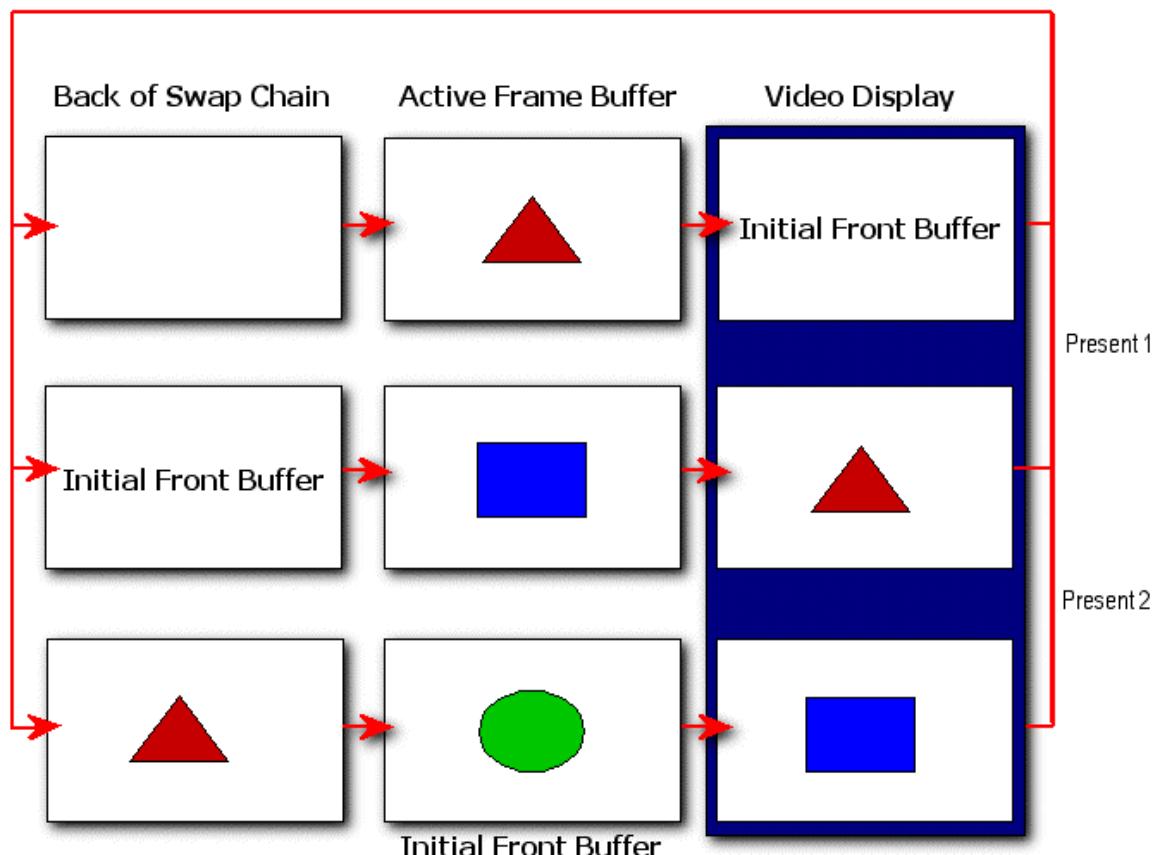


Figure 2.12

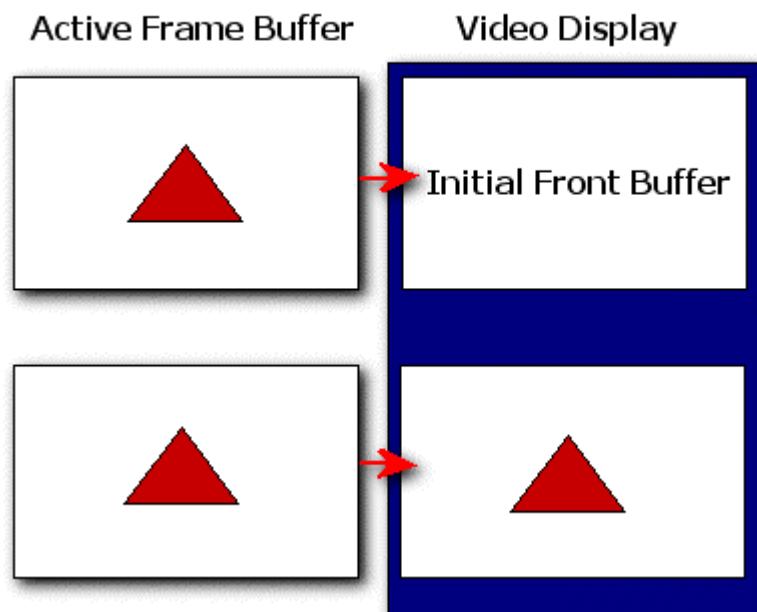


To create the device for Fig 2.12 we specify a BackBufferCount of 2. The result is two frame buffers and a front buffer. Assume that all three are initially blank. Next we render a triangle. The device will automatically render to the active frame buffer, which is initially the first of the two frame buffers created. When the image is complete, we call the `Present` function and the pointers are flipped (or copied in windowed mode). The frame buffer now becomes the front buffer, the front buffer becomes the second frame buffer, and the second buffer becomes the active frame buffer.

Next we render a square and present the scene again. The front buffer which currently has the triangle is sent to the back of the frame buffer queue, the new frame buffer with the square becomes the new front buffer, and the initial front buffer has now become the next active frame buffer. If we were to render a circle next and then call the `Present` function in a loop, the image on the screen would switch between a triangle, a square and a circle over and over again.

#### **D3DSWAPEFFECT\_COPY**

`D3DSWAPEFFECT_COPY` causes the contents of the frame buffer to be copied to the front buffer when the scene is presented. In windowed mode this is performed by doing a blit of all pixels in the frame buffer into the front buffer. In full screen mode, the copy may be performed in hardware using copies, flips, or a combination of the two to emulate the behavior:



**Figure 2.13**



When a copy is performed, the contents of the frame buffer are unaltered by the presentation. This is in contrast to flipping where the frame buffer holds the image that was previously in the front buffer (in a double buffer arrangement). This is important to remember if your application needs to read pixels back from the frame buffer after the scene has been presented. This setting makes sense for windowed mode applications since they are going to perform copies anyway and emulating the flip comes with some overhead. In fullscreen mode, unless our application requires an unaltered post-presentation frame buffer, flipping should be used. Copying in fullscreen mode is slower and may carry additional video memory overhead.

**Note:** D3DSWAPEFFECT\_COPY can only be used for devices with one frame buffer (BackBufferCount = 0 or 1). Device creation will fail if you try to create it using D3DSWAPEFFECT\_COPY and more than one frame buffer.

### **D3DSWAPEFFECT\_DISCARD**

This setting lets the device choose the best method to use (flipping or copying) based on the current video and window modes. This will generally mean that flipping is used with a fullscreen device and copying is used with a windowed device, but this is not guaranteed. Thus our application should not make any assumptions about the state of the frame buffer after the screen presentation. When using D3DSWAPEFFECT\_DISCARD we will always treat the frame buffer as an uninitialized memory buffer requiring that we render over the entire surface. In fact, the DirectX Graphics debug runtime will automatically fill the contents of a presented frame buffer with random data to discourage you from making such assumptions when using this swap effect.

#### ***HWND hDeviceWindow***

This parameter is often confused with the focus window HWND in the CreateDevice call. In most cases these will be the same and if you leave this parameter set to NULL, the device will use the focus window passed into CreateDevice as the device window also. But there are differences between the two and in some cases we may want to use a focus window separate from our device window. First let us examine what the device window is used for in both fullscreen and windowed modes:

#### **Windowed Mode**

In windowed mode this is the HWND of the window that will have its client area used as the front buffer. This device window is treated like any other windowed application. For example, messages from the mouse or keyboard will be sent to this window's WndProc function when it has focus. It can be minimized and maximized just like any other desktop window. If the window is moved, then the device will automatically track the positional changes such that the presentation happens at the new position of the client area. Resizing (WM\_SIZE messages) of the window, although not required, should be handled separately by our application and we will discuss this later in the lesson.

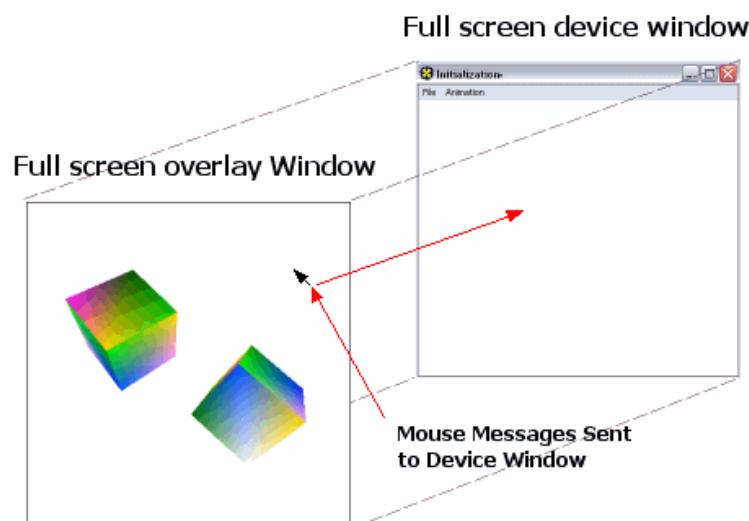


### Fullscreen Mode

When the device is in fullscreen mode it gains exclusive access to the screen and the desktop is no longer visible. It is as if the device has created its own window without any caption or borders and has overlaid the entire desktop. This overlay has the dimensions of the video mode stated in the BackBufferWidth and BackBufferHeight fields.

However, this overlay window is not a real window. The desktop is still active (behind it) and handling messages from mouse and keyboard input. So although this device window is not actually rendered to, and is in fact not visible, the device resizes it so that it takes up the entire display. It also changes its Z order so that it is always atop all other desktop windows. This ensures that any mouse and keyboard input is correctly sent to the device window. This avoids accidentally clicking the mouse on a window belonging to another application -- invisible behind the overlay window -- and changing the focus to that other application. The WndProc of this device window will receive the mouse and keyboard messages as well as other window messages. Although this window is not actually visible, it will have a one-to-one mapping with the physical overlay window that the device is using to render.

### **Device Window in Full Screen mode**



**Figure 2.14**

Fig 2.14 shows the device window that would be created by our application in fullscreen mode. It has a caption, a border and a menu. Regardless of the initial size of this window, when we create the device in fullscreen mode and pass this HWND as the device window, the device will alter the dimensions of the window so that it takes up the entire video resolution. This window will not actually be visible, so all rendering will be done using an overlay window. Windows messages such as keyboard or mouse events will be sent to the device window. So unlike the windowed mode scenario where the device window was used for rendering, in fullscreen mode the device window is merely used as a message collector.

Imagine a scenario where the window was initially set so that it was 400x400 in size. Now consider what would happen if the device did not resize it to take up the entire desktop. When the mouse was clicked on the overlay window, it may not actually be situated over the (invisible) device window. We might accidentally select another application's window, or perhaps click an empty area on the desktop, or in a worst case scenario, drag items into the recycle bin!

Fig 2.15 shows what would happen if the device did not resize the device window. While rendering is unaffected, the position of the mouse is not actually over the device window and mouse messages will not get to our device window's WndProc function.

### What would happen if there was not a resize

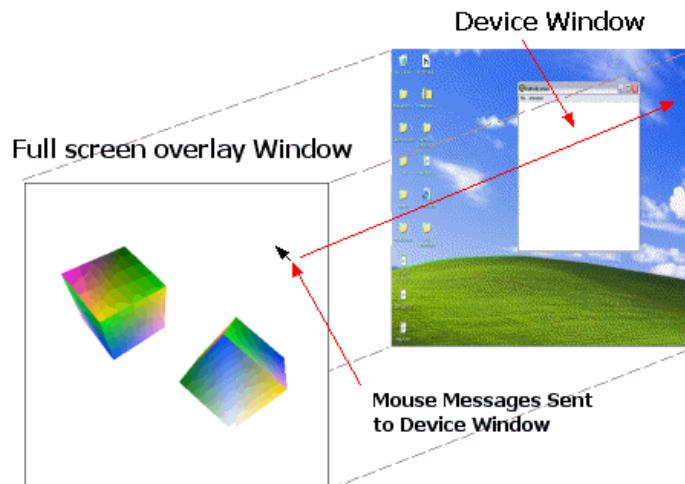


Figure 2.15

So in order for everything to work correctly, we must make sure that we create our device window as a top level window if we intend to use fullscreen mode.

There is something else to watch out for when using a device in fullscreen mode. Although the device will resize the window to take up the entire dimensions of the chosen video mode, this does not always mean that the *client area* of the window is resized to the full extent of the video mode. In Fig 2.16 we can see that the device window contains a border, a caption bar and a menu. When the window is resized, the menu and border still remain. Thus the client area will not have a one-to-one pixel mapping with the overlay window. At the top of the window in this example, about 10% of the overlay window covers up the caption bar and menu of the device window. If a mouse button was clicked in this area, the client area would not receive the message and we might instead be selecting a menu item or clicking on the caption bar:



## Device Window in Full Screen mode

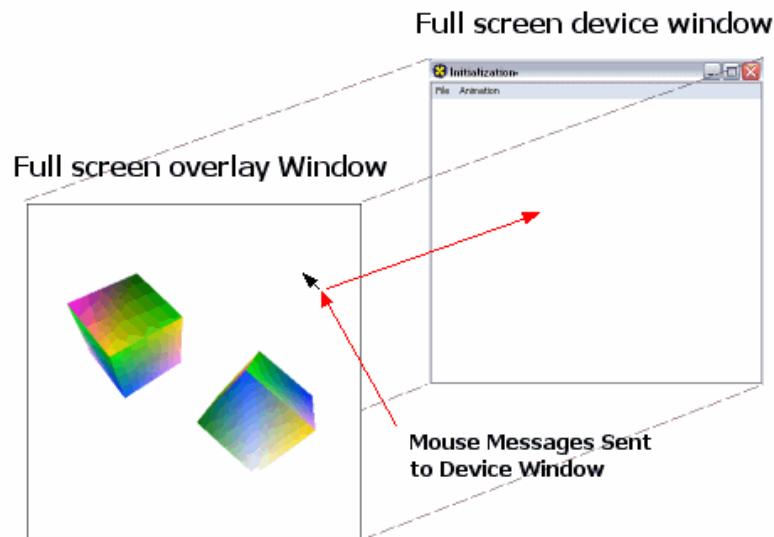


Figure 2.16

The solution is to create a device window without caption, menus, or borders so that the window contains only a client area.

There will be times when you want your application to toggle between fullscreen and windowed modes. There are two options. First, we can change the current operating mode of the device by calling the `IDirect3DDevice9::Reset` function. So if we are moving to fullscreen mode, we could alter the style of the window such that the menu, caption bar and borders are removed. Another option is to create two device windows at application startup. The first has a border and caption for windowed mode. The second has no caption or borders, and we can pass this one into the Reset function when going to fullscreen mode.

**Note:** The focus window can only be specified when the device is initially created. If we create a device in windowed mode and use the window for both the focus window and the device window, changing to fullscreen mode later may cause a problem if we are not careful. In order for the device to transition to fullscreen mode, the focus window must be a top level window. If our windowed rendering window is not a top level window then we have no way of changing the focus window without destroying the device and creating a new one. Therefore, it is common practice to make the application main frame window the focus window even when this window will never be used for rendering. For applications that use multiple fullscreen devices, such as a multi-monitor system, only one device should use the focus window as the device window. All others should have unique device windows. Otherwise, behavior is undefined and applications will not work as expected.



### ***Windowed***

This parameter tells the device creation function whether we wish to create the device in windowed or fullscreen mode. If we set it to TRUE then the device will be created in windowed mode and the device window client area will be the front buffer. If set to FALSE then the device will be created in fullscreen mode where the video mode will be set by the BackBufferWidth and BackBufferHeight fields.

### ***EnableAutoDepthStencil***

This Boolean variable instructs the device creation function whether or not we wish it to create and attach a depth buffer surface. If it is set to TRUE, the function will create the depth buffer (Z-Buffer or W-Buffer) using the depth buffer surface format specified in the next parameter. If the depth buffer surface is created successfully, it will be automatically attached to the device frame buffer. Any pixels rendered by the device will also have their depths tested and recorded in the depth buffer. If the device is reset (perhaps to resize it or alter it to a different video mode), it will automatically destroy the current depth buffer, create a new one that matches the new frame buffer size and attach it as the current depth buffer. This auto management feature makes using depth buffers very convenient.

If it is set to FALSE, a depth buffer will not be created and the application will be responsible for creating a depth buffer surface and attaching it to the device, if it needs one. The application would also be responsible for managing the destruction and recreation of the buffer if the device is reset.

### ***AutoDepthStencilFormat***

If the EnableAutoDepthStencil parameter above is set to TRUE, then this field should hold the `D3DFORMAT` describing the format of the depth surface our application requires. Unlike the surface formats used for textures and frame buffers, there are special `D3DFORMAT` types for use with depth buffers.

Table 2.2 contains the depth buffer surface formats available in DirectX 9.0. If the hardware supports Z-Buffering (as nearly all do these days) then at least one of the `D3DFORMATs` listed will be available for use. It is possible that some 3D hardware may support many of these depth buffer formats. If this is the case then our application will have to choose which one is best for our application to use. As we will discuss



## 32 bit formats

### D3DFMT\_D32

A 32 bit surface where each pixel can hold 32 bits of depth information. Provides a significant range of depth granularity.

### D3DFMT\_D24X8

24 bits of this 32 bit surface can be used to hold depth information. 8 bits are unused.

### D3DFMT\_D24S8

24 bits of this 32 bit surface are used for depth information with 8 bits being used to hold stencil buffer values.

### D3DFMT\_D24X4S4

A 32 bit surface with 24 bits of each pixel being used to hold depth values and 4 bits of each pixel being used to hold stencil information. 4 bits are unused.

## 16 bit formats

### D3DFMT\_D16

Each pixel in this surface can hold 16 bits of depth information. 16 bit surfaces can suffer Z-Buffer artifacts. A 24 bit Z-Buffer minimum is desirable.

### D3DFMT\_D15S1

Only 15 bits are used for depth information with 1 bit reserved for use by a stencil buffer. Z-Buffer artifacts are exacerbated with only 15 bits of accuracy.

### D3DFMT\_D16\_LOCKABLE

A special type of 16 bit depth buffer surface that can be locked. This allows us to read/write to the surface directly through a pointer. This is the only depth buffer surface format that is allowed to be locked.

functions to check which depth buffer formats can be used with the selected frame buffer format.

Some older graphics cards only support 16 bit depth buffer formats. These are not ideal but if that is all we have available then we will have to use them. It is also possible that even if a particular card supports 16 and 32 bit depth buffers and frame buffers, it may require that the bit depth of the depth buffer matches the bit depth of the frame buffer. Therefore, if we have a 32 bit capable card but we have a full screen device running in 16 bit color mode (a 16 bit frame buffer), the hardware may insist that we also use a 16 bit depth buffer. Fortunately, most of the recent graphics cards support the D3DFMT\_D24X8 depth buffer format. 24 bit depth buffers provide us with more than adequate depth granularity so this format will suit our purposes for any demonstration we see in this lesson.

Table 2.2 Depth/Stencil Formats

As we will discuss in the next course in this series, many 3D graphics cards provide support for **stencil buffers**. Stencil buffers are used to mask areas of the frame buffer we do not wish rendered to. They share the same physical memory as a depth buffer. A 32 bit depth buffer format may have 24 bits used for storing pixel depth information and the remaining 8 bits for stencil information.

Our application will have to ensure that it selects a depth buffer format that the 3D hardware is capable of supporting. The Direct3D9 object provides member

functions to check which depth buffer formats can be used with the selected frame buffer format.



### ***DWORD Flags***

This parameter allows the application to specify how the frame buffer and depth buffer should be treated. The two flags that are of importance to us now are listed below:

**D3DPRESENTFLAG\_LOCKABLE\_BACKBUFFER** – If this flag is specified, the device will create the frame buffer such that it can be locked and modified. When we lock a surface (calling the `IDirect3DSurface9::Lock` method) a pointer to the surface pixels is returned. This allows us to modify the frame buffer at the pixel level or to read back pixel colors from the frame buffer. On some hardware, creating a frame buffer with this flag can incur a performance cost. The cost may be the result of the device maintaining a system memory copy of the frame buffer so that it is reachable by the application. Even if this is not the case, the act of locking the frame buffer itself is an expensive operation and should be avoided if possible. Frame buffers are created such that they are not lockable by default so this flag is required if your application needs lock permissions (which will not usually be the case).

**D3DPRESENT\_DISCARD\_DEPTHSTENCIL** – If the device was created with a depth buffer, then setting this flag may improve performance. If this flag is not set then the device object will maintain the integrity of the depth buffer information after the scene has been presented to the front buffer. If the application does not clear the depth buffer before rendering the next scene, then the depth buffer will still hold the per-pixel depth information from the last render. Sometimes this can be useful, but it is usually not required.

On some hardware the depth buffer data is swizzled to a proprietary format for rendering. If this is the case and the flag was not set, the driver would need to make sure that the original depth buffer information is restored afterwards so that the data appears unchanged. This might require an expensive copy operation.

We will generally set this flag because our applications will clear the depth buffer before rendering each frame. The DirectX debug runtime will enforce discarding by filling the depth buffer with a constant value after the scene has been presented.

### ***UINT FullScreen RefreshRateInHz***

This field lets the application specify the refresh rate for fullscreen devices. In windowed mode, this value must be zero since we will need to use the refresh rate used by the current adapter running the desktop. Setting this field to `D3DPRESENT_RATE_DEFAULT` allows the device to choose a refresh rate. This is typically the approach that our applications will use.

### ***UINT PresentationInterval***

This field allows the application to specify the rate at which the frame buffer is presented to the front buffer. For fullscreen devices we normally want to synchronize the presentation with the vertical retrace period of the monitor to avoid tearing artifacts. However, there are several other options. We can choose to present the buffer immediately without waiting for the vertical retrace or we could have the device wait for more than one retrace to occur before presenting the scene. Table 2.3 lists the possible values (defined in `d3d9.h`) that can be passed.



#define	Description
D3DPRESENT_INTERVAL_DEFAULT	The device creation function will automatically select a compatible presentation synchronization scheme.
D3DPRESENT_INTERVAL_ONE	The device will wait until the vertical retrace period before copying/flipping the frame buffer to the physical display. This avoids tearing and essentially locks the frame rate to that of the monitor's refresh rate.
D3DPRESENT_INTERVAL_TWO	The device will wait for every second vertical retrace period before the frame buffer is copied/flipped to the front buffer. This will essentially limit the presentation rate to $\frac{1}{2}$ the monitor's refresh rate.
D3DPRESENT_INTERVAL_THREE	The device will wait for every third vertical retrace period before the frame buffer is copied/flipped to the front buffer. This will essentially limit the presentation rate to $\frac{1}{3}$ the monitor's refresh rate.
D3DPRESENT_INTERVAL_FOUR	The device will wait for every fourth vertical retrace period before the frame buffer is copied/flipped to the front buffer. This will essentially limit the presentation rate to $\frac{1}{4}$ the monitor's refresh rate.
D3DPRESENT_INTERVAL_IMMEDIATE	The device will perform the copy/flip immediately regardless of the current position of the electron gun.

**Table 2.3 Presentation Intervals**

Not all of these modes are supported on all hardware in all video modes, but you can safely assume that at least three are:

- D3DPRESENT\_INTERVAL\_DEFAULT
- D3DPRESENT\_INTERVAL\_IMMEDIATE
- D3DPRESENT\_INTERVAL\_ONE.



## Format Selection

The `IDirect3D9::CheckDeviceFormat` function allows us to test whether a specific surface pixel format is compatible with a device in a specific display mode. This function will be used by our **FindDepthStencilFormat** function (see Lab Project 2.1) to check the various depth buffer formats against the current device on the current adapter. We have to do this because it is possible an adapter that supports 32 bit depth buffers might only support 16 bit depth buffers in 16 bit color mode. So it is not enough to know what depth buffer formats the hardware supports. We also have to know which ones are supported in a particular display mode.

`CheckDeviceFormat` can also be used to check whether or not a certain texture format is supported by the device in the requested display mode or whether a texture surface can be rendered to directly by the device. For now however, we simply wish to use it for determining the best depth buffer format available.

```
HRESULT IDirect3D9::CheckDeviceFormat(
    UINT Adapter,
    D3DDEVTYPE DeviceType,
    D3DFORMAT AdapterFormat,
    DWORD Usage,
    D3DRESOURCETYPE RType,
    D3DFORMAT CheckFormat
);
```

### ***UINT Adapter***

The adapter ordinal for the adapter we are checking the format against.

### ***D3DDEVTYPE DeviceType***

The device type we are checking against. In our code this will either be a HAL device (`D3DDEVTYPE_HAL`) or the reference device if no 3D hardware acceleration is found.

### ***D3DFORMAT AdapterFormat***

The display mode the adapter will be placed into. This is the format for which a compatible depth buffer format must be found for our application. In Lab Project 2.1 for example, this will be the display mode currently being used by the desktop (returned by `IDirect3D9::GetAdapterDisplayMode`). In a fullscreen application we would pass in the display mode format that we are intending to put the hardware into.

### ***DWORD Usage***

A depth buffer surface is a special type of device resource. Internally, it is a block of memory just like any other surface (a texture or the frame buffer for example) but by specifying a USAGE flag we inform DirectX Graphics what we intend the resource to be used for. In this example, we are trying to



find an image surface format that can be used for a depth buffer. In this case we use the `D3DUSAGE_DEPTHSTENCIL` flag.

### ***D3DRESOURCETYPE rType***

The `CheckDeviceFormat` function is used for checking the availability of many resource types so we must specify the resource type we are inquiring about.

```
typedef enum _D3DRESOURCETYPE {
    D3DRTYPE_SURFACE = 1,           D3DRTYPE_VOLUME = 2,
    D3DRTYPE_TEXTURE = 3,          D3DRTYPE_VOLUMETEXTURE = 4,
    D3DRTYPE_CUBETEXTURE = 5,       D3DRTYPE_VERTEXBUFFER = 6,
    D3DRTYPE_INDEXBUFFER = 7,       D3DRTYPE_FORCE_DWORD = 0x7fffffff
} D3DRESOURCETYPE;
```

As you might have guessed, the `D3DRTYPE_SURFACE` type is the one we need for the depth buffer.

### ***D3DFORMAT CheckFormat***

The final parameter allows us to specify our desired surface format. We looked at a table earlier that specified the available 16 and 32 bit depth buffer formats that DirectX Graphics supports. Since we will not use a stencil buffer at this point in the course, we will choose one of the standard depth buffer formats: `D3DFMT_D32`, `D3DFMT_D24X8` or `D3DFMT_D16`.

To avoid the artifacts described earlier in the lesson, our code will first test for a 32 bit depth buffer. If that fails, we will try a 24 bit depth buffer and fall back to a 16 bit buffer as a last resort.

To check for 32 bit depth buffer support:

```
if (SUCCEEDED( m_pD3D->CheckDeviceFormat(AdapterOrdinal, DevType, Mode.Format,
                                             D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE,
                                             D3DFMT_D32 )))
{
    return D3DFMT_D32;
}
```

The above code checks the adapter, device and display mode for 32 bit depth buffer support. If it is supported, the function returns successfully and we execute the code in the braces. If the function did not succeed, we would try again but change the final parameter to a 24 bit format and so on until we were successful. Notice that when the function succeeds the format is returned back to the caller where it will be placed into the `D3DPRESENT_PARAMETERS` structure for the call to `CreateDevice`.

There is one last thing we must do before we accept the format. In windowed mode, DirectX 9.0 permits devices where the frame buffer and the front buffer have different surface formats. So we could use a 32 bit frame buffer with a 16 bit front buffer. We will need to know whether our requested depth buffer format will work with the current front buffer/frame buffer arrangement. The following



code calls an additional function from the `IDirect3D9` interface called `CheckDepthStencilMatch` to make this verification:

```
if (SUCCEEDED( m_pD3D->CheckDeviceFormat(AdapterOrdinal, DevType, Mode.Format,
                                              D3DUSAGE_DEPTHSTENCIL , D3DRTYPE_SURFACE ,
                                              D3DFMT_D32 )))

{
    if (SUCCEEDED( m_pD3D->CheckDepthStencilMatch(AdapterOrdinal, DevType,
                                                    Mode.Format, Mode.Format, D3DFMT_D32 )))
    {
        return D3DFMT_D32;
    }

} // End if 32bpp Available
```

The parameter list to the `CheckDepthStencilMatch` function are (in order) adapter, device type, adapter format (i.e. front buffer format), render target format, and depth buffer format. While this may seem redundant given the previous function, there is a difference. As we will discover in the next course in this series, although the frame buffer is initially the render target when the device is created -- meaning all rendering is done on the frame buffer surface -- we will have the ability to change render targets to other surfaces (like textures for example).

The following code example is taken from Lab Project 2.1. It demonstrates the process just discussed.

```
D3DFORMAT CGameApp::FindDepthStencilFormat(ULONG AdapterOrdinal, D3DDISPLAYMODE Mode, D3DDEVTYPE
DevType )
{
    // Test for 32bit depth buffer
    if (SUCCEEDED( m_pD3D->CheckDeviceFormat(AdapterOrdinal, DevType, Mode.Format,
                                                D3DUSAGE_DEPTHSTENCIL , D3DRTYPE_SURFACE , D3DFMT_D32 )))
    {
        if (SUCCEEDED( m_pD3D->CheckDepthStencilMatch ( AdapterOrdinal, DevType, Mode.Format,
                                                        Mode.Format, D3DFMT_D32 )))
            return D3DFMT_D32;
    } // End if 32bpp Available

    // Test for 24bit depth buffer
    if (SUCCEEDED( m_pD3D->CheckDeviceFormat(AdapterOrdinal, DevType, Mode.Format,
                                                D3DUSAGE_DEPTHSTENCIL , D3DRTYPE_SURFACE , D3DFMT_D24X8
)))
    {
        if (SUCCEEDED( m_pD3D->CheckDepthStencilMatch ( AdapterOrdinal, DevType, Mode.Format,
                                                        Mode.Format, D3DFMT_D24X8 )))
            return D3DFMT_D24X8;
    } // End if 24bpp Available

    // Test for 16bit depth buffer
    if (SUCCEEDED( m_pD3D->CheckDeviceFormat(AdapterOrdinal, DevType, Mode.Format,
                                                D3DUSAGE_DEPTHSTENCIL , D3DRTYPE_SURFACE , D3DFMT_D16 )))
    {
        if (SUCCEEDED( m_pD3D->CheckDepthStencilMatch ( AdapterOrdinal, DevType, Mode.Format,
                                                        Mode.Format, D3DFMT_D16 )))
            return D3DFMT_D16;
    } // End if 16bpp Available

    // No depth buffer supported
    return D3DFMT_UNKNOWN;
}
```



## Lost Devices

It is possible that the device object may be placed into a ‘lost’ state while the application is running. Consider an application running in fullscreen mode where the device has frame buffers, texture surfaces, and vertex buffers stored in video memory. The device knows precisely where these resources are located when they need to be accessed. Now imagine that the user decides to ALT+TAB the focus to another application. The application would be forced back into windowed mode so that another application on the desktop could assume the dominant role. At this point, the new focus application might require access to video memory. Because it has OS focus, its requests for video memory will take precedent and the application resources still occupying video memory are not guaranteed to be preserved. The textures, vertex buffers, and frame buffers may all need to be deleted to create space for memory requests from the focus application. As such, the device pointers now point to resources which no longer exist. Even if we ALT+TAB again to return focus to our application, the memory that was previously being used by our application has been corrupted. It is at this point that the device is said to be in a lost state. When a device is lost we cannot perform normal operations with that device object. Only two functions will be valid at this stage: one to test if the device is lost, and the other to ‘reset’ it if it is.

The following function call enables us to determine the state of the device:

```
HRESULT hRet = m_pD3DDevice->TestCooperativeLevel();
```

This function returns one of two possible results: **D3DERR\_DEVICELOST** or **D3DERR\_DEVICENOTRESET**.

### **D3DERR\_DEVICELOST**

This result indicates that the device memory is still not available as it may be still in use by another application that has focus. Under these circumstances no rendering should be done. The application will need to continually poll the device using the TestCooperativeLevel function until it returns D3DERR\_DEVICENOTRESET. At this time we will be able to reset the device.

### **D3DERR\_DEVICENOTRESET**

This indicates that the memory resources have been handed back to the device. However, caution is in order. When a device is lost, it loses exclusive access to the memory for its resources. Memory handed back to the device is invalid and the previous resource data should be regarded as corrupt.

Once we receive the D3DERR\_DEVICENOTRESET return code, we can reset the device as follows:

```
m_pD3DDevice->Reset( &m_D3DPresentParams );
```

Resetting a device entails passing a **D3DPRESENT\_PARAMETERS** structure as was done when the device was initially created. This tells the device how to rebuild its frame buffer(s), which rendering window to use, and so on. This is similar to recreating the device from scratch. Technically speaking, we are not recreating the device; we are simply instructing it to recreate its resources (textures, frame buffers, etc.). Although we can use different presentation parameters when resetting a device, this is not



usually the approach we will take. Our preference is to return the device back to the state it was in before the loss occurred. Note that this applies to states as well. Lost devices also lose the render and transform states that the application may have set previously. All states return to their default conditions when the device is reset. This is why it is good practice to put our initial device render states in a separate function.

It is important to note that certain textures, vertex buffers, and other resources will need to be reconstructed when a lost device is finally recovered. We will examine these scenarios as we begin to use these resources later in the course.

## Common Causes of Lost Devices

When the user switches focus (ALT+TAB) to another application from a fullscreen application, the device is automatically lost because it no longer has the exclusive access to the video memory that it needs. The application will be minimized on the task bar. The device will remain lost (TestCooperativeLevel will continue to return D3DERR\_DEVICELOST) until the application is maximized again, giving it the focus. At this time, TestCooperativeLevel will return D3DERR\_DEVICENOTRESET and we can reset our device.

Other possible examples might include minimizing the application or shifting focus to another application when running in windowed mode. On some machines (such as our test machine) this does not cause device loss, but this may not always be the case on other hardware.



## Primitive Rendering 101

Now that we know how to set up a DirectX Graphics environment, let us try to use some of the core features. The rest of our discussion will focus on how to render polygons, change render states, and how to use the device to transform our vertices.

### Filled Polygons

In Chapter 1 we constructed a wireframe renderer. While it was useful for understanding the transformation of vertex data from world space to screen space, it is unlikely that we will be using DirectX Graphics to render our scenes only in wireframe. Generally we want our polygons to be filled with color. Let us briefly examine the different **fill modes** that set the polygon drawing strategy for the device.

#### *Point*

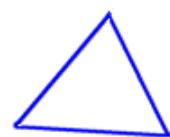
In point fill mode, the device renders each transformed vertex as a point on the screen and does not connect them. Point mode might be useful for tasks like generating a low-quality star field for a space game but is obviously not a fill mode you will likely use very often. Below we see an example of a triangle rendered in point mode.



We can control the color of individual vertices by adding color data to our vertex structure (in addition to the positional data). We will discuss this in detail later in the lesson.

#### *Wireframe*

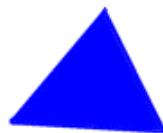
In wireframe mode, one-pixel thick lines are rendered between the screen space vertices. We saw this technique in Lab Project 1.1. The color of the line can be modified by using a color stored in each vertex structure. The following is a triangle rendered in wireframe mode:





## Solid

Solid rendering is the mode you will use most. In this mode the device renders the outlines of the polygon and paints every pixel inside the wireframe outline to provide a solid appearance. Once again the color we store at each vertex can control how the inside area of the polygon gets rendered. Below we see an example of a triangle rendered in solid fill mode.



We configure our device to render with a particular fill mode using the `IDirect3DDevice9::SetRenderState` function and passing in `D3DRS_FILLMODE` and the desired fill mode. The first parameter is a `D3DRENDERSTATETYPE` enumerated type and tells the device which render state we wish to change. This same function is used to change all render states. In this case, we are changing the current fill mode. The second parameter describes the new fill mode. Every polygon rendered following a call to any one of these functions will be rendered using that fill mode until the state is changed.

```
pDevice->SetRenderState (D3DRS_FILLMODE , D3DFILL_POINT);  
pDevice->SetRenderState (D3DRS_FILLMODE , D3DFILL_WIREFRAME);  
pDevice->SetRenderState (D3DRS_FILLMODE , D3DFILL_SOLID);
```

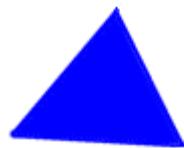
## Shaded Polygons

We are not limited to just using a single color to fill polygons or draw lines. We can instruct the device to render the surface of a polygon using an interpolation between colors stored in the vertex structures. This can be used to generate smooth coloring effects. DirectX Graphics supports both Flat and Gouraud shading. Shade modes and fill modes are not mutually exclusive and will be used together to create the desired effect. The following code shows us how to set one of the two shade modes.

```
pDevice->SetRenderState (D3DRS_SHADEMODE , D3DSHADE_FLAT);  
pDevice->SetRenderState (D3DRS_SHADEMODE , D3DSHADE_GOURAUD);
```

### Flat Shade Mode

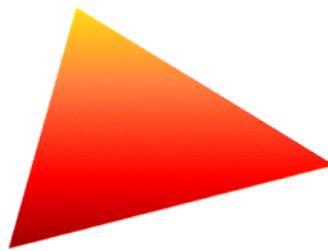
Flat shading applies a single color to the entire polygon. In flat shade mode the device uses the color stored at the first vertex in the triangle to color the entire triangle. If you had a triangle where the first vertex was blue, the second vertex was red and the third vertex was green, only the color of the first vertex (blue) would be used to color the entire triangle. The other colors would be ignored. The next image shows a flat shaded polygon. To render a polygon in this manner, the device would be set to use the solid fill mode and flat shade mode device render states.



In wireframe fill mode and flat shade mode, the color of each line in the polygon will be the color stored in the first vertex.

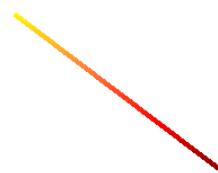
### **Gouraud Shade Mode**

When each pixel in the triangle is rendered using Gouraud shading, the color will be calculated by performing a linear interpolation of the three vertex colors weighted by the position of the pixel in relation to each vertex. For example, if we had two vertices that had red color components of 0.2 and 0.8 and the pixel being rendered was exactly half-way between those two vertices, the red component of that pixel would be 0.5. The following triangle has one yellow vertex at the top and two red vertices at the bottom:



As we will see later, Gouraud shading helps to cover up sharp edges between adjacent polygons and makes the mesh appear more rounded.

Gouraud shading also works in wireframe mode. The color of the line between each vertex making up the edge of the polygon will be determined through the same interpolation process:



The line above consists of two vertices. The top vertex contains a yellow color and the bottom vertex contains a red color. We thickened the line so that it is several pixels wide for ease of viewing but keep in mind that in DirectX Graphics, line thickness in wireframe rendering mode is always one pixel.

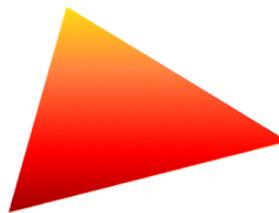


## Vertices in DirectX Graphics

As we learned in Chapter 1, 3D worlds are made up of a collection of polygons, each of which represents a collection of vertices. We also learned that vertices can hold more than just positional information.

```
struct Vertex
{
    float x;
    float y;
    float z;
};
```

When the device renders a polygon and Gouraud shading is enabled, the color stored at each vertex is interpolated across the face of the polygon for each pixel. This smoothly blends the color from one vertex into the next:



In the triangle above, the top vertex in the face holds a yellow color and the bottom two vertices hold slightly different shades of red. Each pixel has its color calculated as a function of its position relative to the three vertices.

So we can store a color at each vertex. In this case, we are looking at the *diffuse* color of the polygon and our structure now looks like this:

```
struct Vertex
{
    float x;
    float y;
    float z;
    DWORD diffuse;
};
```

It may seem strange that we used a DWORD to hold color information but this is in fact how DirectX Graphics represents colors. We will often see DirectX code where colors are defined as D3DCOLOR.

### **D3DCOLOR diffuse;**

D3DCOLOR is actually a typedef for a DWORD (see d3d9types.h):

### **typedef DWORD D3DCOLOR**



Colors are stored in the DWORD as ARGB (alpha/red/green/blue) using a byte for each color component. We can use the D3DCOLOR\_ARGB macro to pass in 4 byte values and have the packed DWORD (D3DCOLOR) returned:

```
#define D3DCOLOR_ARGB(a,r,g,b)  
( (D3DCOLOR) (((a)&0xff)<<24) | (((r)&0xff)<<16) | (((g)&0xff)<<8) | ((b)&0xff) )
```

The macro simply shifts the input bytes values into there respective positions inside the DWORD.

In DirectX Graphics, colors are always represented as 32 bit DWORDs even if the device is in 16 bit or 24 bit video modes. The device will handle any conversions that need to take place as well as the quantization of 32 bit color values into 16 bit color values.

Another macro allows us to ignore the alpha component and deal with colors as RGB values:

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

The resulting alpha component will be set to 255. This means that it is completely opaque.

**Note:** There will be times (especially when dealing with lighting) when we will need to specify colors as a series of floats (one float each for A, R, G and B). In this case we will use the D3DCOLORVALUE structure:

```
typedef struct _D3DCOLORVALUE {  
    float r;  
    float g;  
    float b;  
    float a;  
} D3DCOLORVALUE;
```

**Note:** Each component above has a value in the range to 0.0 to 1.0 (instead of 0 – 255). These values will be converted back into DWORD values for the final render.

Colors are not the only thing we can store in our vertex structure. We may also want to texture our polygons. In order for polygons to have textures applied, each vertex must store a new pair of coordinates. You can think of these two coordinates (generally referred to by U and V) as the X and Y coordinates of the pixel in the texture where the vertex is mapped. Once we give each vertex a set of UV coordinates, the device can interpolate the pixels of the texture across the polygon surface between the vertex coordinates. All of this will be examined in detail in chapter 6.

```
struct Vertex  
{  
    float x, y, z;  
    DWORD diffuse;  
    float u, v;  
};
```

It might seem odd to have both a color and a texture applied to the same polygon. After all, if a texture is mapped to a polygon, wouldn't the color of the texture pixels determine the color of the polygon pixels?

Not necessarily, although we certainly could do it that way. What we will do instead is instruct the device to blend the interpolated diffuse color of each pixel with the texture pixel computed via interpolation of the UV coordinates and use that single color result for our frame buffer image. Fig 2.17 shows an example of this. We have a polygon with a texture applied to it where the vertices on the left edge of the polygon have darker diffuse colors than those on the right.

### Vertices with diffuse color component and texture map applied



Figure 2.17

As we will see in later lessons, a single polygon can have multiple textures assigned to it. When the polygon is rendered, each pixel in the polygon has its color blended from a series of textures (possibly including the diffuse color as well). If we wanted a polygon which had three textures and a diffuse color, we would give it three sets of texture coordinates:

```
struct Vertex
{
    float x;
    float y;
    float z;
    DWORD diffuse;
    float u1;
    float v1;
    float u2;
    float v2;
    float u3;
    float v3;
};
```

To make our objects shiny, we can store another color value called **specular** at each vertex. The specular color we specify determines the color used for surface highlights:



**Figure 2.18**

The sphere on the left has no specular component. The sphere on the right has a white specular color at each vertex. DirectX Graphics will calculate the specular component based on the location of the vertices relative to light sources in the scene and the position of the camera. We will cover specular lighting in detail in chapter 5.

```
struct Vertex
{
    float x;
    float y;
    float z;
    DWORD diffuse;
    DWORD specular;
    float u1
    float v1;
};
```

DirectX Graphics allows us to place lights in our scene to enhance realism. Vertices closer to light sources will be lit more brightly than those that are further away. The effects are even more compelling when combined with an algorithm like Gouraud shading. While we will discuss lighting in detail in chapter 5, a brief discussion will be helpful to illustrate the next concept.

In order for the device to light our vertices we must place first light sources in the scene. Additionally, the device must know whether or not the polygon is facing the light source. Polygons facing away from the light source should obviously not be lit. Because lighting is done at the vertex level and not the face level, we must provide information about the orientation of each vertex. We can do this by storing a normalized vector at each vertex.

A **vertex normal** is a normalized vector stored at each vertex describing the direction the vertex is facing. When the device lights a vertex it will measure the angle between the vertex normal and the direction vector from the vertex to the light source. Vertex color will be scaled based on this angle. A vertex pointing right at a light source will be lit at full intensity while a vertex rotated at some angle away from the light source will have its color scaled down appropriately.



So if our application intends to use the DirectX Graphics internal lighting pipeline, then it will need a vertex structure that contains this normal.

The following vertex structure contains position, a vertex normal vector, a diffuse color and a specular color.

```
struct Vertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Normal;
    DWORD Diffuse;
    DWORD Specular;
};
```

Notice that we used a D3DXVECTOR3 rather than three floats this time. We can access any individual float component of the positional data by using Position.x, Position.y, and Position.z (likewise for the Normal vector components).

The real point of this discussion is that our applications will use vertices in a number of different ways. We may want to render some polygons using DirectX Graphics lighting and a single set of texture coordinates and others without lighting but with three sets of textures, and so on. The question is, how will we tell DirectX Graphics what our vertices look like so that it knows what to expect when we pass them into the IDirect3DDevice9::DrawPrimitive function? The answer is the Flexible Vertex Format.

### The Flexible Vertex Format

We can inform DirectX Graphics about the components it can expect to find in our vertices by calling the following function:

```
IDirect3DDevice9::SetFVF(DWORD fvf);
```

The DWORD will be some combination of Flexible Vertex Format flags. Some of the more common flags are seen in the next table:

Common FVF Flags	Description
D3DFVF_XYZ	The vertex is untransformed and will need to be multiplied by the world, view and projection matrices to transform it into screen space. The structure will contain a 3D vector describing its model (or world) space position.
D3DFVF_XYZRHW	The vertex will not need to be transformed or lit. The positional information contained within the vertex is specified in screen coordinates.



<b>D3DFVF_NORMAL</b>	The vertex contains a normal vector that describes its orientation. If lighting is enabled, this normal is used in lighting calculations to scale the intensity of the light in relation to the orientation of the vertex to the light.
<b>D3DFVF_DIFFUSE</b>	The vertex has a diffuse color component. If lighting is enabled, this color is scaled by the lighting calculations (and the color of lights effecting the vertex) to create a final diffuse color. If lighting is not enabled and no normal is specified, the diffuse color is considered to be the final output diffuse color used to render the polygon.
<b>D3DFVF_SPECULAR</b>	The color of specular highlights that should be reflected by this vertex. If lighting is enabled and a normal is specified, this value is scaled based on the light sources in the scene and the position of the camera in relation to the object and the light. If lighting is disabled this value is considered to be the final specular color used at the rasterization stage.
<b>D3DFVF_TEX0 through D3DFVF_TEX8</b>	DirectX Graphics supports vertices with up to 8 sets of texture coordinates. We can check the MaxSimultaneousTextures member of the D3DCAPS9 structure returned from the IDirect3D9::GetDeviceCaps function to inquire about the device texture blending capabilities. Although many 3D graphics cards will only support 2 to 4 textures being blended simultaneously, this does not limit the ability to store 8 texture coordinates in a single vertex. You may wish to store the texture coordinates in the vertex and render the polygon several times using different sets. This will be covered later in the course when covering multi-texturing.

**Note:** The IDirect3DDevice9 interface has a function called GetFVF() which retrieves the currently set vertex format for the device. Remember that the device is a state machine. Once you call SetFVF with a vertex format, the device will expect that vertex format in all future calls to the DrawPrimitive functions until you call SetFVF again to specify another vertex format.

Vertex flags that are only valid for pre-transformed vertices (vertices specified in screen coordinates) are highlighted in blue, while flags that are only valid for untransformed vertices are highlighted in yellow.

These are mutually exclusive flags. The yellow flags cannot be used with the blue flag. This would be like informing the device that the positional information of the vertex is untransformed and transformed at the same time.

Similarly we would not use the D3DFVF\_XYZRHW flag with the D3DFVF\_NORMAL flag because the first flag states that we are using vertices that have already been transformed. When we specify screen space vertices, the vertices do not pass through the transformation and lighting pipeline. Since the normal is only used for lighting calculations, we would not need to pass it. Flags that are not highlighted can be used with both untransformed and transformed vertices, although they have different implications depending on which of the two is being used.



Let us have a look at some examples. In this first example, we will create a structure that holds positional information, a diffuse color, and one set of texture coordinates. This vertex would be used to specify vertices in model space or world space. They will need to be transformed by the fixed-function pipeline into screen coordinates. Because we have specified a diffuse color and no normal, we will indicate that we do not want the pipeline to light the vertices and that the diffuse color should be used explicitly in the rendering process. This demonstrates that we can enable or disable functionality by choosing only the components we need. In this case, we are choosing the transformation capabilities of the device but not the lighting module.

```
struct Untransformed_Prelit_Textured_Vertex
{
    D3DXVECTOR3 Position; // untransformed model space vertex position
    DWORD Diffuse; // vertex color
    float u; // x texture coordinate
    float v; // y texture coordinate
};
```

Notice that although we can leave out the components we do not wish to use, the components that we do use must appear in the order that they are listed in the table. Diffuse must come after position and so on. To tell the device what to expect from our vertices:

```
m_pDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0))
```

Notice that the last flag is not found in the above table. That is because it is not a flag, it is actually a macro. It tells the device how many floats the texture coordinate set contains. D3DFVF\_TEXCOORDSIZE2(0) informs the device that the first set of texture coordinate in this vertex (index 0) is two floats in size (the typical size). We will see later on in the course that there will be times when we need to use 1D or even 3D texture coordinates and this macro will allow us to specify that. Keep in mind that at this point in the course that you are not expected to understand how texture coordinates work, only that a vertex may need to contain them.

The next example vertex invokes both the transformation and the lighting module of the device. This time we will need to supply a vertex normal. When we render polygons containing vertices of this type, the device will transform the vertices by the device's currently set world, view and projection matrices. It will use the vertex normal to calculate its orientation from any lights placed in the scene which will be used to scale the diffuse and specular colors. This type of vertex is referred to as an untransformed and unlit vertex since it needs the device to transform and light it before rendering it. Also note that it does not contain any texture coordinates, so a texture will not be applied to this polygon when it is rendered.

```
struct Untransformed_Unlit_Vertex
{
    D3DXVECTOR3 Position; // untransformed model space position
    D3DXVECTOR3 Normal; // unit vector orientation vector
    DWORD Diffuse; // color reflected from diffuse lighting
    DWORD Specular; // color of specular highlights reflected
```



```
};
```

To use this vertex format we would need to call SetFVF with the following flags:

```
m_pDevice->SetFVF( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE |  
D3DFVF_SPECULAR);
```

Notice again that the flags are specified in the order that they appear in the table (ignoring omitted flags) and that the vertex structure itself retains this ordering as well.

Our lab projects in this lesson will not use texturing or the lighting module. We will specify our vertices in model coordinates and render polygons using Gouraud shading. Therefore, we will need to store a color at each vertex. This means that we will need only two components, a position and a diffuse color component:

```
struct CVertex  
{  
    float x;  
    float y;  
    float z;  
    DWORD Diffuse;  
};
```

Since our application uses only one vertex type we can simply call SetFVF as soon as the device is created and leave this state set for the life of the application:

```
m_pD3DDevice->SetFVF( D3DFVF_XYZ | D3DFVF_DIFFUSE );
```

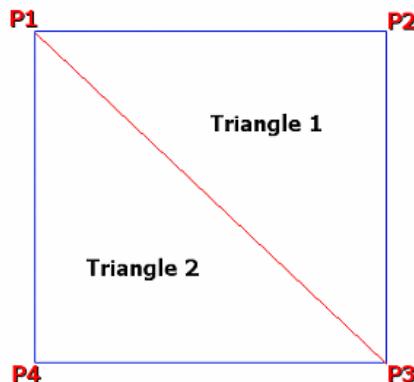
**Note:** When an application requires many different FVF types, it is preferable to #define the flags and give them meaningful names, making it easier to read. For example:

```
#define MyUnLitVertex D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE  
#define MyPreLitVertex D3DFVF_XYZ | D3DFVF_DIFFUSE  
  
m_pDevice->SetFVF(MyUnLitVertex)  
// Render un-lit meshes here  
  
m_pDevice->SetFVF(MyPreLitVertex);  
// Render pre-lit meshes here  
// Present scene here
```

In Lab Project 1.1, our spinning cubes were made up of six faces each with four vertices per face. We were able to render those faces (called **quads** due to the four sided nature of the polygon) once we had transformed the vertices into valid screen coordinates. In DirectX Graphics (and virtually all 3D API's commonly available) you are limited to rendering only two types of primitives: lines or triangles. A line is specified using two vertices which define starting and ending points in either 3D space or screen space. A triangle is constructed from three vertices defined in either 3D space or screen space. We will cover using DirectX Graphics for 2D rendering using screen space vertices later in the course. At this

point in time we will concentrate only on primitives defined in 3D model space or world space. To render a quad, we must construct it using two triangles as shown below:

**A Quad made up of Triangles**



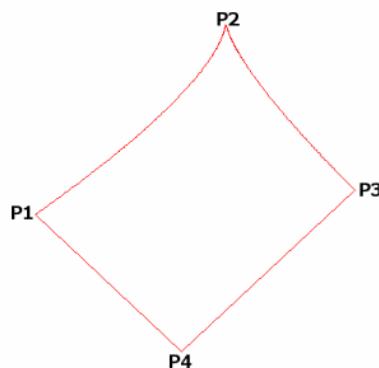
**Figure 2.19**

Triangle 1 contains three vertices (P1, P2 and P3) and Triangle 2 also has three vertices (P1, P3 and P4). Let us briefly discuss exactly why it is that we are limited to triangle rendering.

### Planar Polygons

As we discussed in Chapter 1, if all of the points of a polygon are on a single plane, we can take any two edges in that polygon and perform a cross product to generate a normal vector for the entire polygon. This normal can be used to determine whether or not the polygon is facing away from the camera. Many mathematical operations performed in 3D graphics programming are simplified (and thus made fast enough for real-time use) when the assumption is made that all vertices in a polygon are on the same plane. If the plane is facing away from us then so are polygons that share the plane. Plane calculations are used for back-face culling, collision detection, object picking and even color interpolation. If we were allowed to generate polygons where all of the vertices did not share the same plane, then these mathematical operations would fail to return the correct results.

Let us consider an example. Imagine that we were trying to find out whether a point was behind or in front of a polygon. We usually do this by using the polygon plane and classifying that point against that plane. If the point is behind the plane then the point is behind the polygon. Fig 2.20 shows a quad where three of the points share a plane but one of the vertices has been lifted off of the plane. This is similar to laying a piece of paper flat on a desk, and then picking up one of its corners. P2 does not lie on the same plane as P1, P3 and P4:

**A Twisted Quad****Figure 2.20**

What is the plane normal of the quad in Fig 2.20? If we perform a cross product on edges  $(P1 - P2)$  and  $(P1 - P3)$  we would get a very different result than if we used edges  $(P1 - P3)$  and  $(P1 - P4)$ . The answer is that they are both wrong because the polygon does not exist on a single plane. We could have a situation where a point is classified as being in front of the plane (because it is in front of points  $P1$ ,  $P3$  and  $P4$ ) when it is actually behind point  $P2$ .

Of course, we know that the vertices of a triangle are always co-planar. If you move a vertex to a different position, the entire triangle is rotated or pivoted onto a new (but single) plane. So, in choosing triangles, DirectX Graphics can be sure that when it is dealing with the vertices of its primitives, they will always exist on a single plane. Fig 2.20 does not accurately depict the situation. We had to bend ends  $P1-P2$  and  $P2-P3$  to represent the fact that the vertex  $P2$  is raised off the plane. Of course, this is not actually the case since polygon edges are always straight. With this in mind, you should be able to carve the above quad into two triangles. Each will exist on different planes to be sure, but nevertheless they will have co-planer vertices when taken individually. Note that we can still store our polygons as N-sided convex polygons (squares, hexagons, octagons, etc.) as long we are sure to deconstruct them into triangles prior to passing them to the device for rendering.



## The DrawPrimitive Functions

The IDirect3DDevice9 interface defines the following primitive rendering functions.

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

DrawPrimitive is used to draw polygons when the vertices are stored in a device resource called a vertex buffer. Vertex buffers are blocks of memory allocated by the device that we use to store vertex data. We will discuss vertex buffers in the next chapter.

```
HRESULT DrawPrimitiveUP(D3DPRIMITIVETYPE PrimitiveType, UINT PrimitiveCount,  
                        const void *pVertexStreamZeroData,  
                        UINT VertexStreamZeroStride);
```

DrawPrimitiveUP is the function that we will use to render the two cubes in our first demo for this lesson (Lab Project 2.1). When we use this function to render polygons we will pass in a pointer to an array of vertices much like we did in our software based rendering demo. This is not the optimal way to render polygons in DirectX Graphics and we will learn why this is so in the next lesson. For now however, it will suit our purposes because it is very easy to use. The ‘UP’ appended to the end of the function name stands for ‘User Pointer’ because the vertices are maintained by the application (via a pointer to a vertex array) and not in a device owned vertex buffer.

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

DrawIndexedPrimitive allows us to make certain optimizations based on the fact that vertices from different faces might share the same 3D space position and properties. In our cube example we created 24 vertices (four for each face) when technically they describe only eight unique positions in 3D space. Many of the faces, such as the top face and the front face for example, used the same vertices. There were three vertices at each corner of the cube, one belonging to each face that shared that corner point. This is wasteful because we wind up transforming and lighting 24 vertices when we could just operate on 8. Using the DrawIndexedPrimitive function we pass a device resource called an index buffer along with our vertex buffer. It is filled with indices into the vertex buffer describing which vertices make up each face. This allows us to reuse the same vertex in each of the three faces in our cube and speeds things up considerably.

```
HRESULT DrawIndexedPrimitiveUP(D3DPRIMITIVETYPE PrimitiveType,  
                             UINT MinVertexIndex,  
                             UINT NumVertexIndices, UINT PrimitiveCount,  
                             const void *pIndexData, D3DFORMAT IndexDataFormat,  
                             const void* pVertexStreamZeroData,  
                             UINT VertexStreamZeroStride );
```



This is behaviorally the same as the DrawIndexedPrimitive function, only it allows us to pass in pointers to system memory allocated vertex and index arrays, rather than device allocated vertex and index buffers.

### The DrawPrimitiveUP Function

DrawPrimitiveUP is the function we will use in this lesson to render our polygons:

```
HRESULT DrawPrimitiveUP(D3DPRIMITIVETYPE PrimitiveType, UINT PrimitiveCount,
                        const void *pVertexStreamZeroData,
                        UINT VertexStreamZeroStride);
```

We will discuss the parameter list slightly out of order to clarify certain concepts.

#### ***UINT VertexStreamZeroStride***

This parameter represents the size of our vertex structure (a single vertex). It tells the device how big each vertex is so that it knows how far to advance the pointer to access the next vertex in the array. The size should match the size that would result given the FVF definition.

#### ***void \*pVertexStreamZeroData***

This is the pointer to our array of vertices. The first demo in this lesson will call DrawPrimitiveUP for each polygon in each cube. During each call, this pointer points to a single face consisting of four vertices. Later, we will learn how to render many triangles simultaneously with a single function call.

#### ***UINT PrimitiveCount***

This value describes how many primitives we intend to render from the vertex array. This value depends on the D3DPRIMTIVETYPE described next.

#### ***D3DPRIMITIVETYPE PrimitiveType***

The D3DPRIMTIVETYPE tells the device how to interpret the vertex data passed in and how it should be used to render triangles. The primitive types defined by DirectX Graphics are as follows:

#### **D3DPT\_POINTLIST**

The D3DPT\_POINTLIST primitive type informs the device that the vertex data should be treated as a list of points to be rendered, not as a list of triangles. The vertices pass through the transformation and lighting pipeline and have their vertex colors calculated just like normal vertices, but at rendering time they are treated as individual points to be drawn on the screen. The following code demonstrates rendering our cube faces as a point list:

```
for ( ULONG f = 0; f < pMesh->m_nPolygonCount; f++ )
{
    CPolygon * pPolygon = pMesh->m_pPolygon[f];
```

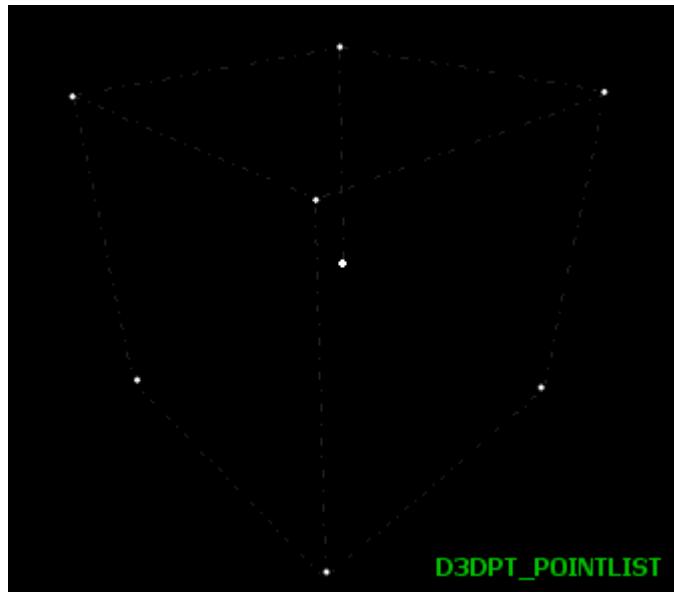


```
// Render the primitive  
m_pD3DDevice->DrawPrimitiveUP(D3DPT_POINTLIST,pPolygon->VertexCount,  
                                pPolygon->m_pVertex, sizeof(CVertex) );  
} // Next Polygon
```

With `DrawPrimitiveUP`, the second parameter describes how many primitives we wish to draw. When using `D3DPT_POINTLIST` to render points, each vertex is a primitive and thus the number of primitives is equal to the number of vertices to be rendered.

We do not have to render all of the primitives contained in the vertex array. Of course, when using a Point List primitive type, the primitive count cannot exceed than the number vertices contained in the vertex array or the call will fail.

Fig 2.21 demonstrates the results of the previous code. Faint gray lines were added to help you see the original cube shape. Only the white dots would actually be rendered during the call.



### PrimitiveCount = NumberOfVertices

Figure 2.21

### D3DPT\_LINELIST

The device treats the vertex array as a collection of vertex pairs when using line lists. Each pair defines a start and end point in 3D space (or screen space if using pre-transformed vertices). During rendering the device will draw a straight line between each pair of points. As with the



point primitive, the vertices that pass through the pipeline can have colors, shading, and even textures applied. One limitation is that line thickness is limited to a single pixel.

Fig 2.22 shows how an array of six vertices would be rendered by DrawPrimitiveUP using the D3DPT\_LINELIST primitive type. Since each line is defined by two vertices, the maximum number of primitives that can be rendered is equal to NumberOfVertices / 2. In this example, six vertices would describe three separate lines.

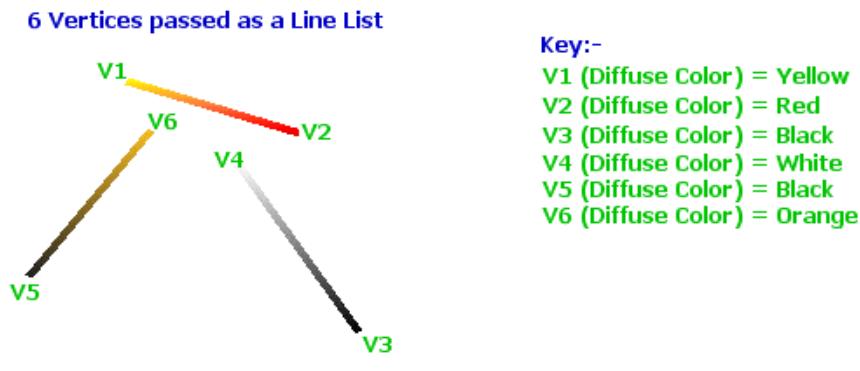


Figure 2.22

Note that the vertices also contain their own colors and that Gouraud shading smoothly blends the colors of the two vertices across the length of the line. The image is not accurate since we widened the lines beyond one pixel for easier viewing.

Although we do not have to render all of the lines passed into the function, the PrimitiveCount parameter should not exceed the total number of vertices in the array divided by two:

**PrimitiveCount = NumberOfVertices / 2**

Rendering a series of connected lines using the D3DPT\_LINELIST primitive type will require vertex duplication since vertices are paired. Fig 2.23 shows five connected line segments. Ten vertices would be required ( $5 \times 2 = 10$ ):

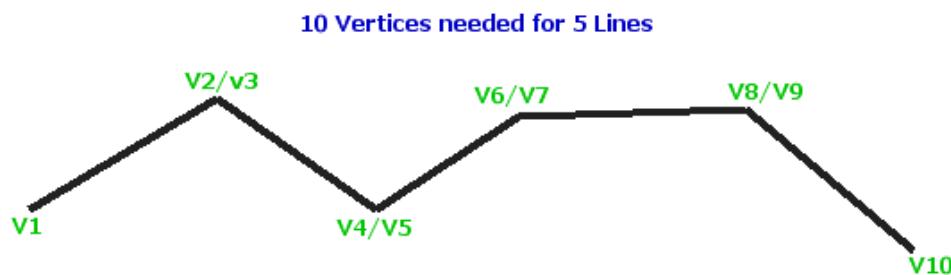


Figure 2.23



Although this approach works correctly, it is inefficient. The end point of line 1 (v1) is in the same position as the start point of line 2 (v3) and so on. Not only will this be a less than optimal use of memory (especially if we were rendering a significant number of line segments) but if the line segments use untransformed vertices, duplicates sharing the same positions would still need to be transformed individually.

The code to render the five line segments just mentioned is shown next. It assumes that `m_pLineVertexArray` is a pointer to an array of type `CVertex` large enough to hold the ten vertices:

```
// There are 10 vertices in our linked line because it is made up of 5 separate lines  
UINT LineVertexCount = 10  
  
// Render the primitives  
m_pD3DDevice->DrawPrimitiveUP( D3DPT_LINELIST , LineVertexCount / 2 ,  
                                m_pLineVertexArray , sizeof(CVertex) );
```

### D3DPT\_LINESTRIP

The `D3DPT_LINESTRIP` primitive informs the device that the lines are connected. This eliminates the need for duplicated vertices. During rendering, the device uses the end vertex of the previous line as the start vertex of the next line and so on for each line rendered.

If we have two line segments to draw, v1 to v2 and v2a to v3 and v2 and v2a are duplicates, we can pass in vertices v1, v2, and v3 and the device will automatically render the first line between v1 and v2 and the next line between v2 and v3. This allows us to do remove duplicated vertices and conserve memory and means that the vertex position v2 only has to be transformed and lit once by the pipeline.

Fig 2.24 illustrates the same five line segments using the `D3DPT_LINESTRIP`. Only six vertices are required to render the five line primitives. The primitive count for a line strip is:

**PrimitiveCount = NumberOfVertices-1**

**Only 6 vertices needed for 5 lines**

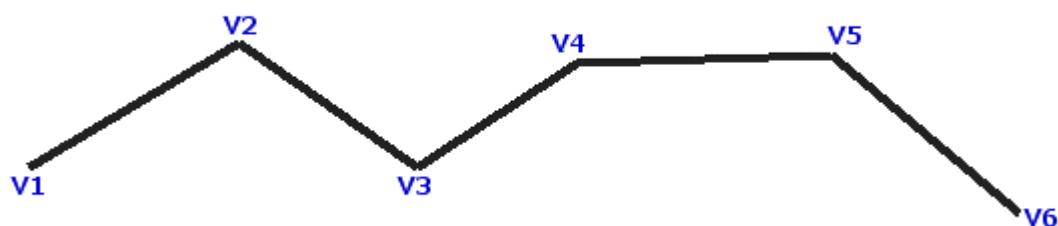


Figure 2.24

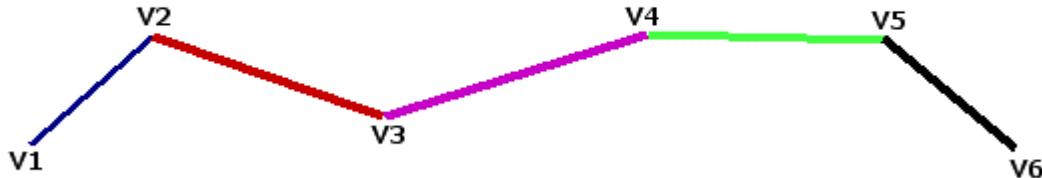
In this example, the device would use v2 as both the end point for line 1 and would reuse it as the start vertex for line two. The same holds for the other vertices that are both the start and end vertices of neighboring line primitives. The following code demonstrates how we would render this example:

```
// There are 6 vertices uses to describe our 5 lines
UINT LineVertexCount = 6

// Render the primitives
m_pD3DDevice->DrawPrimitiveUP( D3DPT_LINELIST , LineVertexCount +1 ,
                                m_pLineVertexArray , sizeof(CVertex) );
```

Although line strips are more efficient for rendering connected lines, they cannot be used if the connected line segments require different properties (such as a different color):

### **When a strip won't work**



**Figure 2.25**

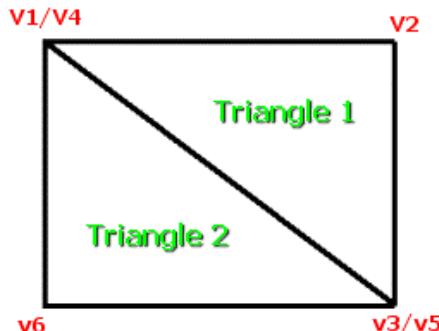
In Fig 2.25 v2 is the start and end point of lines 1 and 2 respectively. In order to make line 1 blue, both of its vertices must have a blue color component. In order for line 2 to be red, both of its vertices must have red color components. Because v2 is shared by both lines 1 and 2 and there is no way to simultaneously store both colors in the vertex, a D3DPT\_LINELIST primitive type with duplicate vertices at each line intersection must be used. Each line would have its own copy of the vertex in the same position but with the correct color.

### **D3DPT\_TRIANGLELIST**

When we use the D3DPT\_TRIANGLELIST, the vertex array is expected to have three vertices for each primitive. If a vertex array had nine vertices, it would be capable of producing three triangle primitives. Vertices [v1, v2, v3] would be used for triangle 1, [v4, v5, v6] for triangle 2 and vertices [v6, v7, v8] for triangle 3. The primitive count can be calculated as follows:

$$\text{PrimitiveCount} = \frac{\text{NumberOfVertices}}{3}$$

The vertex array must contain a vertex count that is a multiple of 3. Fig 2.26 depicts a quad stored as a list of 6 vertices:



**Figure 2.26**

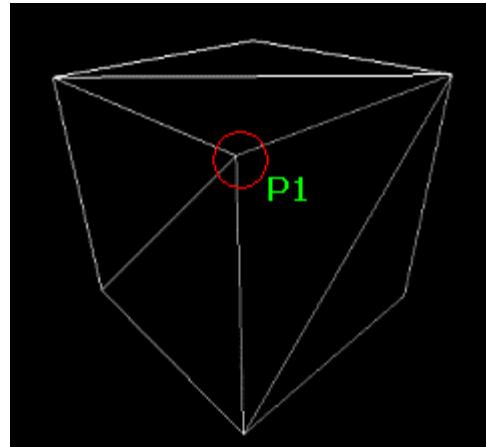
Triangle 1 in this example would be made up of v1, v2 and v3 and triangle 2 would be made up of v4, v5 and v6. The vertex array passed into DrawPrimitiveUP would be arranged as follows.

**v1 , v2 , v3 , v4 , v5 , v6**

The device will treat each group of three vertices as a separate triangle for rendering. Note once again the duplicate vertices problem. Vertex 4 in Triangle 2 has exactly the same position property as vertex 1 used in Triangle 1. The same is true for v3 and v5. This is unavoidable when using the triangle list primitive. The vertices for each triangle can be defined in any order as long as a clockwise winding order is maintained for display.

As we saw with the line primitive, there may be times when there is no choice but to do triangle list rendering and accept the duplicated vertex problem. Different properties such as color or texture coordinate would be examples of why we might need to take this approach.

This situation only becomes more difficult when one thinks of duplicated vertices within a more complex mesh. Fig 2.27 depicts a triangle list representation of a cube. Each face is rendered as two triangles. This amounts to storage for six vertices rather than the four used in our software demo.



**Figure 2.27**

The red circle highlights a corner in the cube where three faces meet. We see that four vertices share the same position (look at the triangles) and each will be sent through the transformation and rendering pipeline. Consider the implications of a game world made up of thousands of polygons. Storing and rendering this world as a triangle list can more than double or triple the amount of vertices needing to be processed.

The main advantage of triangle lists is that they are relatively easy to work with. For example, if we had a large mesh consisting of thousands of triangles, we could render the whole lot with one call to DrawPrimitiveUP. We simply pass in the vertex data for the entire mesh. This is much more efficient than calling DrawPrimitiveUP for every individual triangle despite the duplicated vertices.

The following code shows how we could render each face of a cube using triangle lists. The code assumes that each face of the cube now stores 6 vertices. The two duplicates are needed to represent the two unique triangles from which the face is composed.

```
for ( ULONG f = 0; f < pMesh->m_nPolygonCount; f++ )
{
    CPolygon * pPolygon = pMesh->m_pPolygon[f];

    // Render the primitive
    m_pD3DDevice->DrawPrimitiveUP ( D3DPT_TRIANGLELIST, pPolygon->m_nVertexCount / 3,
                                      pPolygon->m_pVertex, sizeof(CVertex) );
}

// Next Polygon
```

In this example we render each face (two triangles) with its own call to DrawPrimitiveUP. Alternatively, we could have designed our cube mesh structure to hold all of the vertices in one large array rather than each face having its own pointer to vertex data. Had this been the case then we could have rendered the entire cube with one call to DrawPrimitiveUP. This would



have been a more efficient solution but is not ideal for our cube meshes given how they are currently stored.

**Note:** It is important to understand that duplicated vertices are not always undesirable. In fact, sometimes they are absolutely necessary. If we wanted each face in a cube to be different colors, then each face would need four unique vertices not shared by any other faces. We could modify the face color by altering the color components of the vertices without concern for affecting neighboring faces sharing the same vertex.

The cube is actually a good example of a possible situation where you might desire duplicated vertices between faces. This is because we usually texture the faces of our meshes. If we wanted each face of the cube to have a different texture applied (or use different portions of the same texture) then we would need to give each face its own unique vertices with their own unique set of texture coordinates. Note that this does not mean that we need to duplicate vertices within a single face. It would be much more efficient to store a single cube face as four vertices instead of six. We will examine how this can be accomplished later in this section.

### D3DPT\_TRIANGLESTRIP

Triangle strips are one of the most efficient primitive types available. This is particularly true when many duplicated vertex positions exist between adjacent triangles. Triangle strips are analogous to line strips. Strips use the first three vertices in the array to render the first triangle. For every triangle thereafter, the strip uses the last two vertices of the previous triangle and the next vertex in the list to create the next triangle. This eliminates duplicate vertices. 3D cards are often optimized for triangle strip rendering. Fig 2.28 demonstrates how the vertex array is used to construct triangles for rendering:

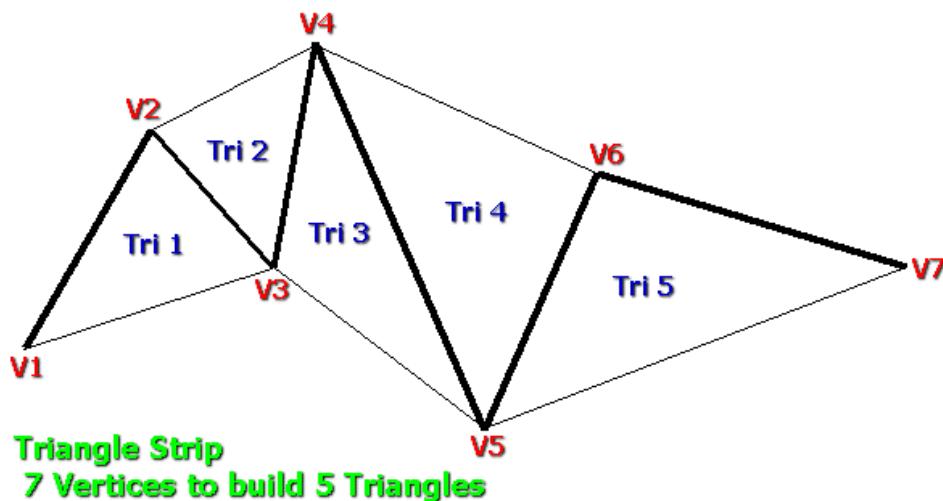


Figure 2.28

Fig 2.28 shows that we can pass 7 vertices to render 5 primitives. This ratio is very efficient. If we had used a triangle list instead, 15 vertices would have been necessary to achieve the same



result. Triangle strips cut this requirement roughly in half by exploiting connected triangles that share edges (and therefore vertices) with neighboring triangles.

Let us quickly step through the render process for the above set. The device renders the first triangle using the first three vertices in the vertex array: V1, V2, and V3. It then processes vertex V4 and creates the second triangle using vertices V2, V4, and V3. Moving to the next vertex (V5) the device builds Tri 3 from V3, V4, and V5. Triangle 4 is rendered using vertices V4, V6, V5, and the pattern repeats until the strip is complete.

Be sure to note the vertex order used in the strip. For example, Triangle 2 was built using V2, V4 and V3 rather than the order the vertices were passed in (V2, V3, and V4). Recall that backface culling is performed by taking the winding order of the vertices of a polygon into account. If a triangle strip did not swizzle the order of the vertices in every second triangle in the list, those triangles would have an counter-clockwise winding order and would be back face culled by the device, and never rendered. If Triangle 2 was built using V2, V3 and V4, it would create a triangle with a counter-clockwise winding order and would be incorrect. The device takes this into account when rendering your triangles as strips and automatically adjusts the order of the second and third vertex in every second triangle. When we define our strip, every second triangle should have a counter-clockwise winding order since the device will automatically flip it to be clockwise during rendering. We calculate the primitive count parameter to be passed into the DrawPrimitiveUP as:

$$\text{PrimitiveCount} = \text{NumberOfVertices} - 2$$

Fig 2.29 shows a quad represented as a triangle strip. It looks similar to the quad diagram using a triangle list with the exception of the counter-clockwise winding order for the complete face and the use of 4 vertices rather than 6.

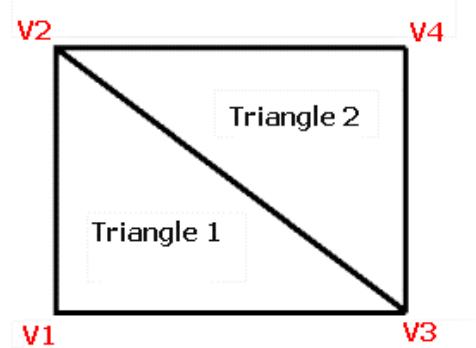


Figure 2.29

Only triangle 1 has a clockwise winding, triangle 2 does not. Please take time to review the diagrams above as strips are often a confusing concept for newcomers to 3D graphics programming.

It should also be noted that if we wanted each triangle to be rendered using a different color, then a triangle strip would not be the appropriate choice since the two triangles share two common vertices. Altering the color of one of these vertices would affect the color interpolated across both triangles. For this effect, a triangle list should be used instead.

### D3DPT\_TRIANGLEFAN

When using triangle fans, the first three vertices in the array are used to create the first triangle. For every other triangle, the first vertex in the array, the last vertex of the previous triangle, and the next vertex in the array are used. We pass the vertices to our cube face as four vertices in a clockwise winding order and it will automatically be rendered as two triangles by DirectX Graphics. There are no duplicated vertices within the face itself. Fig 2.30 demonstrates the concept.

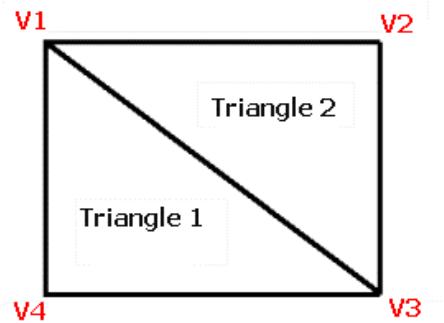


Figure 2.30

We pass in a vertex array with the vertices V1, V2, V3, and V4 arranged in a clockwise order. The device uses V1, V2, and V3 to create the first triangle. V1, V3, V4 are then used to render the second triangle. The first vertex in the list is used as the first vertex for all triangles in the list. Fig 2.31 should make the concept clear.

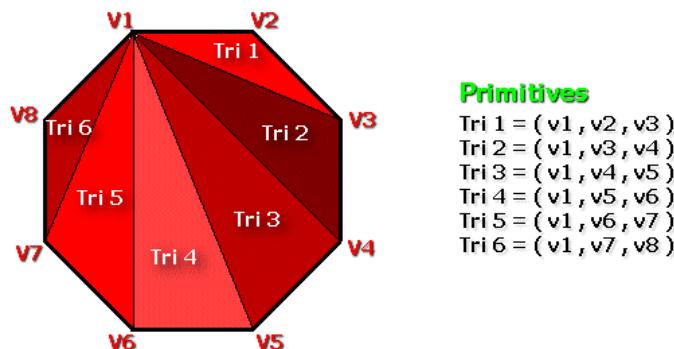


Figure 2.31



We render the entire face as a triangle fan by passing in the ordered vertex array. The octagon in Fig 2.31 would be broken down into six separate triangles for rendering. In the diagram, the triangles are colored for easier viewing only. Since vertices are shared, we recognize that properties such as color must also be shared if we desire a single color for the polygon. If we wanted the triangles to have individual colors, a triangle list would be required.

The diagram shows the pattern used by the device when constructing the triangles. V1 is used in all six triangles. For every triangle but the first, the second vertex in each triangle is the vertex that was the third vertex in the previous triangle.

This is an ideal primitive type when dealing with data stored as convex N-gons as it does not suffer from duplicate vertices. It will be a good choice for rendering the faces of our cubes, and is the type we will use in Lab Project 2.1.

The primitive count for a triangle fan can be calculated as:

### **PrimitiveCount = NumberOfVertices – 2**

The following code could be used to render the faces of our cube using triangle fans:

```
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
```

Rendering polygons is very easy in DirectX Graphics. We just need to make sure that we calculate the primitive count correctly and use a primitive type that is compatible with the way our geometry is stored. In our next lesson we will examine other members of the DrawPrimitive family of functions, as well as how to use vertex buffers and indexed primitives to eliminate duplicate vertices.



## The Rendering Pipeline

We now understand how to define vertices and how to render them using the DrawPrimitiveUP function. It is time to bring these concepts together and examine what happens to the vertex when one of the DrawPrimitive functions is called. Let us assume that we are using a mesh made up of untransformed, pre-lit (i.e. colored) vertices. The vertex structure might look like the following:

```
struct CVertex
{
    float x, y, z;
    DWORD color;
};
```

We can describe this vertex structure using the following flexible vertex format flags:

```
#define MyPreLitVertex D3DFVF_XYZ | D3DFVF_DIFFUSE
```

Before rendering the mesh we will tell the device object to expect this type of vertex:

```
m_pDevice->SetFVF ( MyPreLitVertex );
```

The device object maintains three state matrices used to transform vertices into screen space coordinates. From our discussion in the last chapter we know that these matrices are the world, view, and projection matrices.

When we call DrawPrimitiveUP, the device checks the current FVF flags. If it finds the D3DFVF\_XYZ flag, it multiplies each vertex in the array (or subsection of the array) with the current World, View, and Projection matrices to produce homogeneous clip space coordinates. Tasks such as clipping and back face culling follow, and then the device performs the divide by w. At this point, vertices that are visible are inside the -1 to +1 range on the x and y axes of the projection window. The device maps these vertices into the range of the viewport to produce screen space coordinates. Had we used the D3DFVF\_XYZRHW flag instead, the device would understand that there is no need to transform the vertices by these matrices as they are already in screen space. These flags allow us to directly control which parts of the transformation pipeline we want to use.

Thus all we must do before we render a mesh is make sure that the world, view, and projection matrices are setup correctly and sent to the device. We can set all three of these matrices using the SetTransform method of the IDirect3DDevice9 interface, specifying the matrix we want to set:

```
D3DXMATRIX mtxWorld, mtxView, mtxProjection  
  
//build World, View, and Projection matrices with correct information here  
...  
//whenever we need to update one of the device matrices  
//we can use one of the following transform states to  
m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_mtxWorld);  
m_pD3DDevice->SetTransform( D3DTS_VIEW, &m_mtxView);  
m_pD3DDevice->SetTransform( D3DTS_PROJECTION, &m_mtxProj);
```

[www.gameinstitute.com](http://www.gameinstitute.com) Graphics Programming with DX9



As discussed in Chapter 1, the projection matrix is often set once at application startup. The view matrix will need to be updated whenever the position of the camera changes (typically once per frame if the camera is moving). The world matrix will normally need to be set before rendering each mesh in the scene.

The following code snippet from Lab Project 2.1 renders two cube objects. It assumes that the view and projection matrices have already been sent to the device. Note that the world matrix is set for each object and that we render each face of each cube as a triangle fan.

```
// Loop through each object (there are two cubes)
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
```

Hopefully you will find that this is a lot easier to follow than the rendering code we wrote in Chapter 1. In that project we had to manually multiply each vertex by the various matrices and transform them into screen space ourselves.

## Device States

The device object is a state machine and when we set a state inside the device (such as turning lighting on or off), it remains in effect until it is unset or modified to some other state. There are four main state groups:

- Render States
- Transform States
- Texture Stage States
- Sampler States

The IDirect3DDevice9 interface exposes four functions used to alter the states within these four categories. We will ignore the latter two for now as these will be covered in chapter 6 and focus only on the render state and transform state groups.



## Render States

We can set a render state using the following function exposed by the IDirect3DDevice9 interface:

```
HRESULT SetRenderState(D3DRENDERSTATETYPE State, DWORD Value);
```

The first parameter is one of the members of the `D3DRENDERSTATETYPE` enumerated type and the second parameter is a `DWORD` whose meaning depends on the render state specified in the first parameter.

The `D3DRENDERSTATETYPE` enumerated type has a significant number of entries. We will explain each render state only as we cover it in the text. As we move forward in the course, at the end of each chapter you will find an appendix with a listing of any new states introduced during the lesson.

**Note:** The device includes a function called `GetRenderState` that allows the application to retrieve the current device setting for a given state. We pass the render state we wish to inquire about and the address of a `DWORD` variable that will be filled with that current state inside the device:

```
HRESULT GetRenderState(D3DRENDERSTATETYPE State, DWORD *pValue);
```

**Note:** `GetRenderState` should not be called if you are using a **pure device**. A pure device eliminates the overhead resulting from maintaining an internal structure of render states to return information to the `GetRenderState` function. This improves application performance. When using a pure device your application must retain its own copy of the current state settings if it requires access to this information.

## Z – Buffering

After we have created and attached a Z-Buffer to the device, we need to tell the device that we wish to use it when rendering. As we will discover later in the course, there will be times when we will want to render some objects with the Z-Buffer and some without it. That is why there is a render state that allows the application to toggle it on and off:

```
m_pDevice->SetRenderState(D3DRS_ZENABLE , D3DZB_TRUE);
```

The `D3DRS_ZENABLE` member of the `D3DRENDERSTATETYPE` enumerated type specifies that we wish to alter the current state of the device Z-Buffer. The device expects the second parameter to be a member of the `D3DZBUFFERTYPE` enumerated type:

```
typedef enum _D3DZBUFFERTYPE
{
    D3DZB_FALSE = 0,
    D3DZB_TRUE = 1,
    D3DZB_USEW = 2,
    D3DZB_FORCE_DWORD = 0x7fffffff
} D3DZBUFFERTYPE;
```



### **D3DZB\_FALSE**

This disables the Z-Buffer so that no per-pixel depth testing is performed. It is the default state of the device if a Z-Buffer was not automatically created during device creation. Since our applications will specify automatic Z-Buffer creation during device initialization (we set the **EnableAutoDepthStencil** member of the **D3DPRESENT\_PARAMETERS** structure to **TRUE**) , this will not be the default state of the device.

### **D3DZB\_TRUE**

This enables the device Z-Buffer for per-pixel depth testing. This state change will only succeed if a Z-Buffer has been created and attached to the device swap chain (frame buffer(s)). This is the default state of the device if the Z-Buffer created at device creation time used the **EnableAutoDepthStencil** member of the **D3DPRESENT\_PARAMETERS** structure. Otherwise, the default is **D3DZB\_FALSE**.

### **D3DZB\_USEW**

Some 3D graphics adapters support the use of a W-Buffer. The W-Buffer uses the same memory as the Z-Buffer but calculates the per-pixel depth values differently. When we enable W-Buffer, the device uses the reciprocal of W (1/W) where W is the value output from the projection matrix. This is equal to the view space Z component of the input vertex. W-Buffers provide a more linear mapping of depth values and eliminate artifacts caused by 16 bit Z-Buffers.

In order to use this parameter type, our application must ensure that the adapter supports W buffering by checking the **RasterCaps** member of the **D3DCAPS9** structure to see if the **D3DPRASTERCAPS\_WBUFFER** flag is set.

```
D3DCAPS9 Caps;
// Caps was filled out in the InitDirect3D function by calling IDirect3D9::GetDeviceCaps
if (Caps.RasterCaps & D3DPRASTERCAPS_WBUFFER)
    m_pDevice->SetRenderState ( D3DRS_ZENABLE , D3DZB_USEW ); // Use W Buffer
else
    m_pDevice->SetRenderState ( D3DRS_ZENABLE , D3DZB_TRUE ); // Use Z buffer
```

This is generally something we will do only when a 16 bit Z-Buffer is the only option available.

## **Lighting**

This next render state allows us to enable or disable the device's internal lighting pipeline. In our initial applications, we will disable lighting since our vertices do not have the required vertex normal. Lighting will be covered in Chapter 4. To enable/disable lighting we use the following respective render states:

```
m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
```



The **D3DRS\_LIGHTING** member of the **D3DRENDERSTATETYPE** enumerated type tells the device that the second parameter will set the state of the internal lighting module. Lighting is enabled by default so if we do not require lighting, then we must explicitly disable it.

## TRUE

This is the default state of the device. Vertices that use lighting must include a vertex normal. Lighting calculations are done by taking the angle between the vertex normal and the light direction vector to establish the angle between the vertex and the light. We scale the light's effect on that vertex using a dot product of those two vectors. If the vertex normal is absent, a dot product result of 0 will be used.

## FALSE

Disables the lighting module of the device. Our current application will use this render state.

**Note:** All render states can be changed at any time, even in the middle of rendering a scene. For example, we could disable lighting and render some pre-lit polygons, then enable lighting and render some unlit polygons. Pre-lit polygons have no vertex normal and include a color at each vertex. Unlit polygons contain a vertex normal and require the device to light them before rendering.

## Shading

Our applications will use the Gouraud shading model so that the colors stored at each vertex in the face are linearly interpolated across the surface of that face. There are a few shading models available in DirectX Graphics and they are set using the following render state:

```
m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADEMODE );
```

When setting the **D3DRS\_SHADEMODE** render state, the second parameter should be a member of the **D3DSHADEMODE** enumerated type:

```
typedef enum _D3DSHADEMODE
{
    D3DSHADE_FLAT = 1,
    D3DSHADE_GOURAUD = 2,
    D3DSHADE_PHONG = 3,
    D3DSHADE_FORCE_DWORD = 0x7fffffff
} D3DSHADEMODE;
```

Although there are three choices listed (**D3DSHADE\_FLAT**, **D3DSHADE\_GOURAUD**, and **D3DSHADE\_PHONG**), only flat and Gouraud shading modes are currently supported.

## **D3DSHADE\_FLAT**

When this shade mode is set, the diffuse and specular colors of the first triangle vertex are used and no interpolation is done between vertex colors. Diffuse and specular colors stored in other vertices within the same triangle are ignored. Vertex alpha however *is* interpolated across the surface as we will see in Chapter 7. Note that the first vertex in the triangle is selected, not the first vertex in the entire polygon.



Since a cube face is made using two triangles, vertex 0 will be used to color triangle 1 and vertex 3 will be used to color triangle 2 (using the triangle fan example). The first vertex in a given triangle can be defined for the different primitive types as follows.

- For a triangle list, the first vertex of the triangle  $i$  is  $i * 3$ .
- For a triangle strip, the first vertex of the triangle  $i$  is vertex  $i$ .
- For a triangle fan, the first vertex of the triangle  $i$  is vertex  $i + 1$ .

### **D3DSHADE\_GOURAUD**

When a triangle is rendered with Gouraud shading, the colors of all vertices in the triangle are used to calculate the final color of a pixel within that triangle by using a linear interpolation between all three vertices. The distance from a pixel to a vertex is a weight value for the vertex color contribution to the pixel. This is the default shade mode when the device is created. To set the Gouraud shading mode in code:

```
m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
```

### Dithering

In 16 bit color mode the range of colors is significantly less than those available in 32 bit color mode (65,535 vs. 16,000,000 or so). There will be times when 16 bit color modes cannot accurately produce the shade of a certain color your application may require. If dithering is enabled, it creates the color by using two colors at alternating pixel positions across the surface. For example, let us imagine that the color yellow was unavailable. If dithering was enabled then the triangle would be rendered using alternating red and green pixels. Because the pixels are so close together the human eye perceives the triangle as yellow. While dithering can be useful in these situations, it can result in a grainy appearance on high resolution monitors. Dithering is disabled by default when the device is initially created. We enable dithering using the following respective render states:

```
m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE, TRUE );
m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE, FALSE );
```

### Back Face Culling

DirectX Graphics can check the winding order of triangles that have passed through the World, View, and Projection matrices and remove them from further consideration when their vertices are ordered in a counter clockwise fashion. This ordering indicates that the camera is looking at the back of the polygon. Our application can set the back face culling behavior using the **D3DRS\_CULLMODE** render state and specifying a member of the **D3DCULL** enumerated type as the second parameter:

```
m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL );
```

```
typedef enum _D3DCULL
{
```



```
D3DCULL_NONE = 1,  
D3DCULL_CW = 2,  
D3DCULL_CCW = 3,  
D3DCULL_FORCE_DWORD = 0x7fffffff  
} D3DCULL;
```

### **D3DCULL\_NONE**

This sets the device so that no back face culling is done. The triangle orientation is not tested and it is rendered as if it had two sides. If the camera was moved behind the triangle, the viewer will still be able to see it.

### **D3DCULL\_CW**

When this state is set, triangles with a clockwise winding order in relation to the camera are considered to be facing away from the camera. This mode is useful when using geometry ported from OpenGL engines. OpenGL uses a right-handed Cartesian coordinate system where the faces have a counter-clockwise winding order.

### **D3DCULL\_CCW**

This is the default culling state for the device and is the mode that we will use throughout this course. In this mode triangles that have a counter-clockwise winding order with relation to the camera are considered to be facing away from the camera and are not rendered. We generally set this state once at environment setup:

```
m_pD3DDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
```

## **Transformation States**

We set the device transform states to control how vertices are transformed from model space into screen space. The device maintains three state matrices (along with a few others that we will discuss in later lessons) that are used for this process.

### **The World Matrix**

Before rendering each object in our scene we will set the object world matrix as the current world matrix for the device as follows:

```
m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxWorld );
```

We use the SetTransform function with `D3DTS_WORLD` as the first parameter to inform the device we are setting the world matrix. The second parameter is the address of the object world matrix (a `D3DXMATRIX` structure) for the object to be rendered. This matrix holds the position and orientation of the object in the 3D world.



The device world matrix will typically have to be changed many times per frame. In our first few applications we will be rendering two cube objects. Each will have its own world matrix which must be set prior to rendering. We will set object 1's world matrix and render its polygons, then we will set object 2's world matrix (which overwrites the previous world matrix setting of the device) and then render its polygons. This is a critical point to understand. The device has only one world matrix. Before you render an object you will send its world matrix to the device. That matrix will remain the device world matrix until replaced with another world matrix as shown below:

```
m_pD3DDevice->SetTransform( D3DTS_WORLD, &Object1->mtxWorld );
Object1->Render();

m_pD3DDevice->SetTransform( D3DTS_WORLD, &Object2->mtxWorld );
Object2->Render();
```

## The View Matrix

The application maintains a view matrix to describe the camera position and orientation in the virtual world. World space vertices are multiplied by this matrix to transform them into view space relative to the camera. When the position or orientation of the camera changes, we need to build a new view matrix and send it to the device using the following transform state:

```
m_pD3DDevice->SetTransform( D3DTS_VIEW, &mtxView );
```

## The Projection Matrix

The projection matrix describes the FOV of the camera and is used to convert the camera relative coordinates into homogenous clip space. We set the device projection matrix using the following transform state:

```
m_pD3DDevice->SetTransform( D3DTS_PROJECTION, &mtxProj );
```

Once this matrix is set, the device transformation pipeline setup is complete. From now on, whenever our application calls one of the DrawPrimitive functions, each vertex will be multiplied by the device state matrices to be transformed from model space to homogeneous clip space.

## **HomogeneousVertex = ModelSpaceVertex \* World \* View \* Projection**

At this point, the device will do various clipping tests and perform back face culling. Then the divide by W maps the vertex onto the projection window where the vertices are in the range of -1 to +1 in both the x and y dimensions. Finally, the coordinates are converted into screen coordinates and used to rasterize the triangles.

### GetTransform

Just like the SetRenderState function, the SetTransform function also has a counterpart called GetTransform in the IDirect3DDevice9 interface. It can be called to query the current device world, view, or projection matrix:

```
D3DXMATRIX mtxWorld, mtxView, mtxProj;

m_pDevice->GetTransform( D3DTS_WORLD, &mtxWorld );
m_pDevice->GetTransform( D3DTS_VIEW, &mtxView );
m_pDevice->GetTransform( D3DTS_PROJECTION, &mtxProj );
```

**Note:** As with GetRenderState, GetTransform does not work if you are using a pure device. The application must maintain copies of the matrices if access is required after sending them to the device.

Fig 2.33 shows how the SetRenderState and SetTransform functions are used to alter the states of the device object. These states remain in their current condition until set to new conditions.

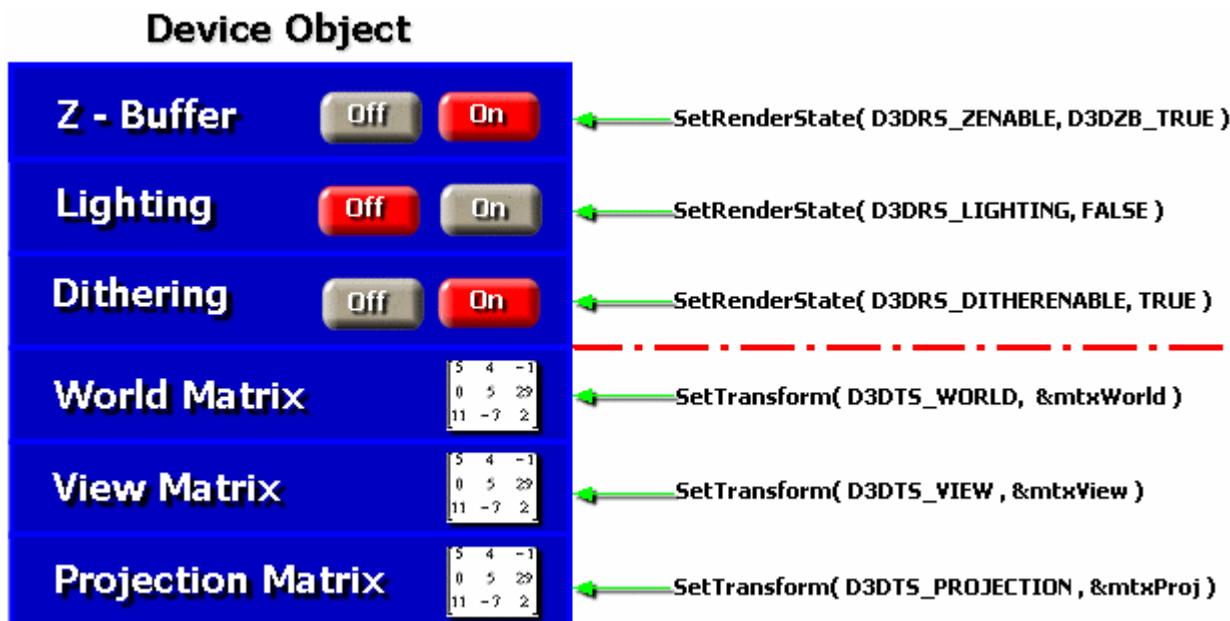


Figure 2.33



## Frame Buffer and Depth Buffer Clearing

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

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

### ***DWORD Count***

It is possible to clear only portions of the frame buffer (and Z-Buffer) rather than the entire surface. Our application can pass an array of one or more D3DRECT structures indicating the desired areas to be cleared. If the second parameter to this function is not NULL then this value will indicate the number of D3DRECT structures pointed to by pRects.

### ***D3DRECT pRects***

If the first parameter (Count) is not 0, this parameter will point to the start of an array containing the D3DRECT structures describing areas of the frame buffer or depth buffer the application want cleared. If the entire frame buffer (and Z-Buffer) is to be cleared, this parameter will be set to NULL.

### ***DWORD Flags***

This parameter is a combination of flags that tell the device which surfaces to clear. We can choose to clear the frame buffer, the depth buffer, and/or the stencil buffer by combining the following flags. Note that at least one of the following flags must be used and that these flags are not mutually exclusive:

**D3DCLEAR\_STENCIL:** Clear the stencil buffer to the value in the *Stencil* parameter. We are not using a stencil buffer at this time so the Stencil parameter will be set to zero.

**D3DCLEAR\_TARGET:** Clear the frame buffer (or render target) to the color in the *Color* parameter.

**D3DCLEAR\_ZBUFFER:** Clear the depth buffer to the value in the *Z* parameter.

### ***D3DCOLOR Color***

If the D3DCLEAR\_TARGET flag is used then this should contain the 32 ARGB color used to clear each pixel in the frame buffer or current render target. Our application uses a white color setting (0xFFFFFFFF) which is the ARGB color (255,255,255,255). The frame buffer does not use the alpha component of a color but colors must still be specified in 32 bit ARGB format.

### ***float Z***



If the D3DCLEAR\_ZBUFFER flag is set then this value should contain the normalized distance value that each pixel in the depth buffer should be initialized to before rendering. This value is typically set to 1.0. This maps the maximum distance to the far frustum plane in view space.

#### **DWORD Stencil**

If the D3DCLEAR\_STENCIL flag is used then this flag should contain an integer value to store in each stencil buffer entry. Stencil buffers will be covered at a later time and will not be used by our current application.

If this function is unsuccessful then it will return `D3DERR_INVALIDCALL`. This indicates that one or more of the parameters may have been invalid.

### **Beginning and Ending Scenes**

Before calling any primitive rendering functions for a given frame, the application must call the `IDirect3DDevice9::BeginScene` function. When rendering is completed it calls the `IDirect3DDevice9::EndScene` function. The call to EndScene informs the device that the application has finished rendering the current scene. All DrawPrimitive calls will take place between BeginScene and EndScene function calls.

### **Presenting the Frame**

The final step in frame rendering is instructing the device to present the frame buffer to the front buffer. This makes the newly rendered scene visible to the user on the monitor screen. We do this using the `IDirect3DDevice9::Present` function. This function is called outside the BeginScene/EndScene pair.

```
HRESULT IDirect3DDevice9::Present(CONST RECT *pSourceRect, CONST RECT *pDestRect,  
                                  HWND hDestWindowOverride,  
                                  CONST RGNDATA *pDirtyRegion);
```

#### ***RECT \*pSourceRect***

Instead of the entire frame buffer being copied to the front buffer, the application can specify a rectangular frame buffer region to be copied. This parameter holds the address of a `RECT` structure containing the dimensions of the desired region. When this parameter is NULL, the entire frame buffer is copied. This parameter must be NULL if you did not use the `D3DSWAPEFFECT_COPY` swap effect for the swap chain when you created the device.

#### ***CONST RECT \*pDestRect***

A pointer to a `RECT` structure containing the front buffer destination rectangle in window client coordinates. If NULL, the entire client area is filled. If the rectangle is larger than the destination client



area, it is clipped to the destination client area. This parameter must be NULL if the swap chain was not created with `D3DSWAPEFFECT_COPY`.

### ***HWND hDestWindowOverride***

This parameter allows you to specify another window to which your frame buffer output will be displayed. It overrides the device window specified in the `D3DPRESENT_PARAMETERS` structure during device creation. The common value is NULL. This informs the device to carry out its default behavior of copying the frame buffer to the front buffer when performing a presentation with a windowed device. Note that this only works with a windowed device and that it does not remove the association with the device window. For example, key press messages will still be sent to the device window and not to the override window.

### ***CONST RGNDATA \*pDirtyRegion***

This allows you to specify a region (an area of the screen constructed from non-overlapping rectangles) to be copied to the screen. The rectangles are specified in frame buffer coordinates. This value is typically set to NULL.

Passing NULL for all of these parameters is the most common application behavior. This will copy or flip (depending on the swap effect being used by the device) the entire frame buffer (the top of the swap chain if multiple frame buffers have been created) to the front buffer. In windowed mode the front buffer is the client area of the device window specified in the `D3DPRESENT_PARAMETERS` structure used to create the device. In fullscreen mode rendering is always done to the overlay window covering the display.

## **Conclusion**

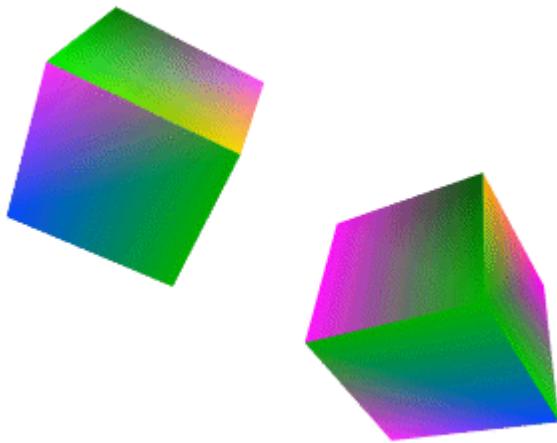
And with that, we now have a good understanding of core DirectX Graphics functionality. We have looked at environment setup, device states, and even shaded primitive rendering. When you have finished studying your workbook projects you will be able to quickly set up rendering environments for future applications and you will have a fully reusable and highly functional set of classes to handle these rather mundane (but essential) setup tasks. You will also have a good feel for the different steps involved in setting up and running your game rendering loop for every frame.

In our next lesson, we will continue our study of primitive rendering. Our focus will be on more efficient, hardware-friendly approaches.

# **Workbook Chapter Two:**

## **DirectX Graphics Foundation**

---



© 2003, eInstitute, Inc.

**You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.**

## 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	<code>m_pD3D;</code>	// Direct3D Object
LPDIRECT3DDEVICE9	<code>m_pD3DDevice;</code>	// 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_PARAMETERS` 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 (D3DADAPTER\_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 REF 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() * 0xFFFFFFF) / 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 );

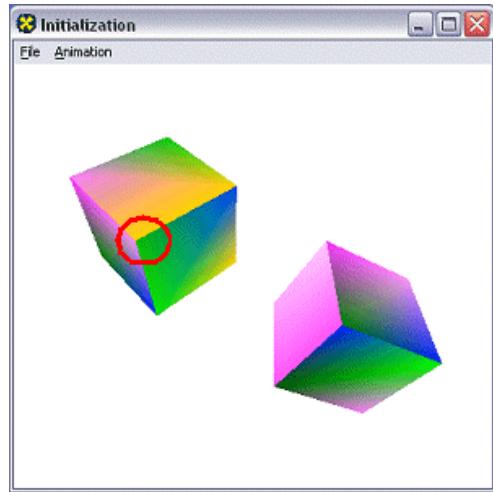
// Our two objects should reference this mesh
m_pObject[ 0 ].m_pMesh = &m_Mesh;
m_pObject[ 1 ].m_pMesh = &m_Mesh;

// 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:

$$V(x,y,z,1) * \text{ProjectionMatrix} = H(X, Z, Z=z, W=z)$$

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.

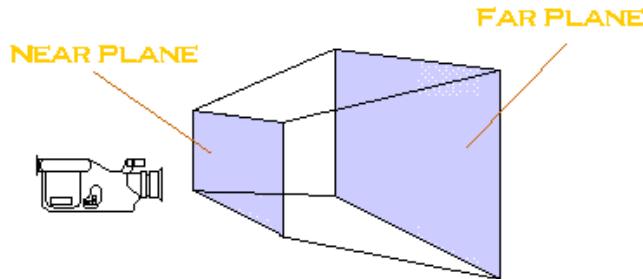
This means that the view space Z values input into the projection matrix multiplication must be mapped to some other range. The solution is to define a minimum and maximum range of acceptable Z values.

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 D3DXMatrixPerspectiveLH 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 Direct3D9 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 Direct3D9 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 IParam )
```

### **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.

### **IParam**

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 & D3DP SHADECAPS_COLORGOURAUDRGB) ) return false;
    if ( !(Caps.PrimitiveMiscCaps & D3DPMISCCAPS_CULLCCW) ) return false;
    if ( !(Caps.ZCmpCaps & D3DPCMPCAPS_LESSEQUAL) ) return false;

    // Supported
    return true;
}
```

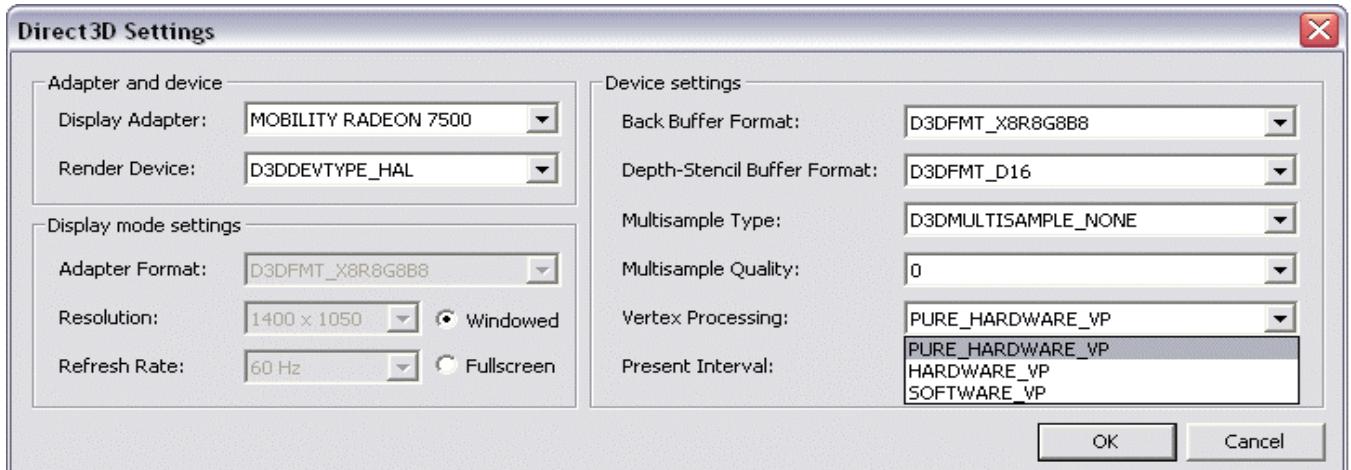
Our application checks the D3DCAPS9.RasterCaps member to ensure that dithering is supported. It checks the D3DCAPS9.ShadeCaps member to confirm Gouraud shading. The PrimitiveMiscCaps member is evaluated to make sure that the cull mode we are using (counter clockwise) is supported. Finally, the default 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 CD3DInitialization::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 CD3DEnumAdapter:

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

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

CD3DEnumAdapter 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_IDENTIFIER9` 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<sup>TM</sup>, nVidia geForce 3<sup>TM</sup>, 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 (CD3DEnumAdapter 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 CD3DEnumAdapter class to the CD3DInitialize adapter array. It enumerates all of the available display modes and stores them in a mode array inside the CD3DEnumAdapter 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.

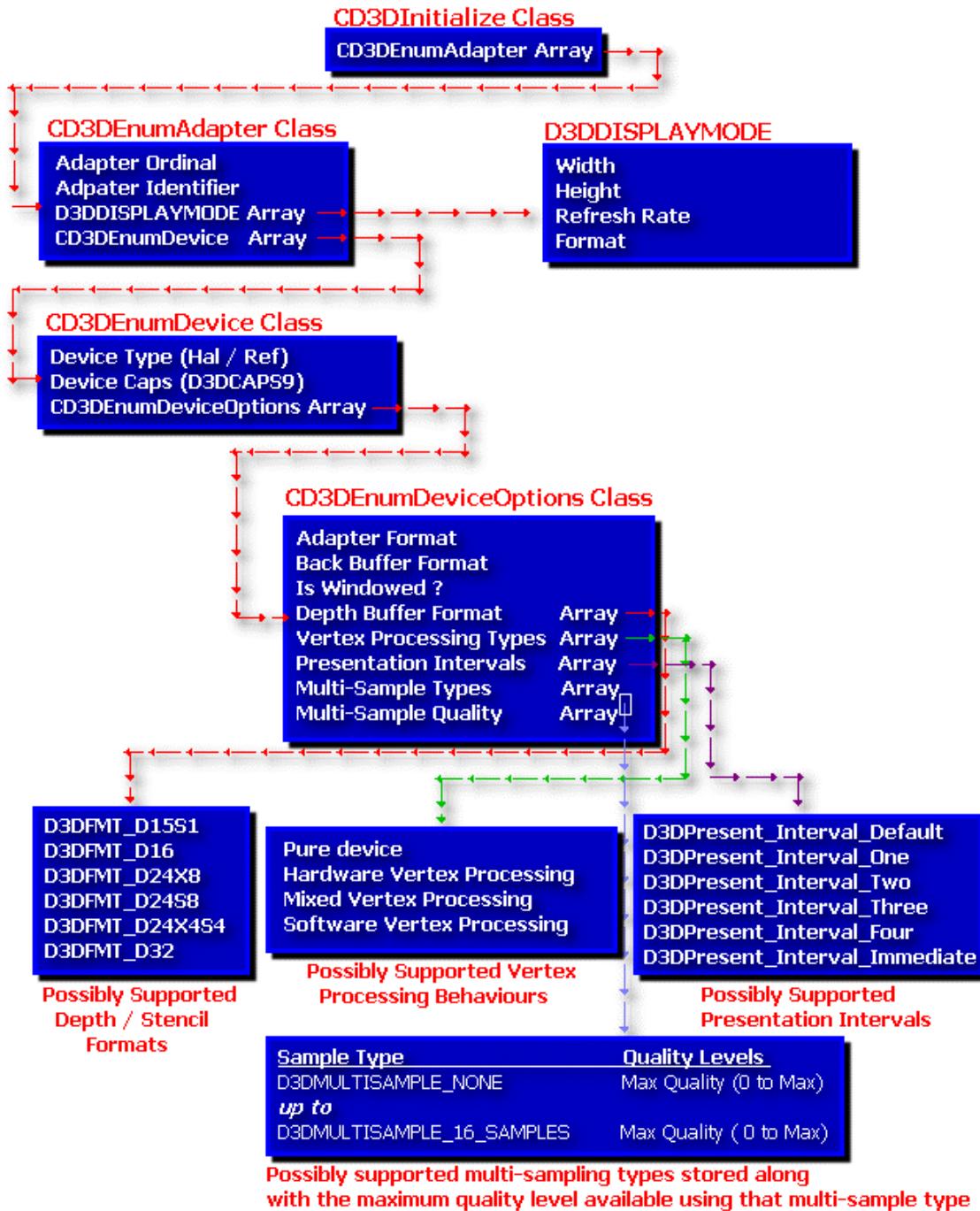


Figure 2.3

## **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 CD3DEnumAdapter 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 CD3DEnumAdapter 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 CD3DEnumAdapter::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 D3DDEVTYPEN DeviceTypes[3] = { D3DDEVTYPEN_HAL, D3DDEVTYPEN_SW, D3DDEVTYPEN_REF };
```

Generally only a HAL or REF device will be available since third party software devices (D3DDEVTYPEN\_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 CD3DEnumAdapter), 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==D3DDEVTYPE_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
pSettings
pSettings->AdapterOrdinal
pSettings->DisplayMode
pSettings->DeviceType
pSettings->BackBufferFormat
pSettings->DepthStencilFormat
pSettings->MultisampleType
pSettings->MultisampleQuality
pSettings->VertexProcessingType
pSettings->PresentInterval
                = true;
                = D3DSettings.GetSettings();
                = pBestOptions->AdapterOrdinal;
                = DisplayMode;
                = pBestOptions->DeviceType;
                = pBestOptions->BackBufferFormat;
                = pBestOptions->DepthFormats[ 0 ];
                = pBestOptions->MultiSampleTypes[ 0 ];
                = 0;
                = pBestOptions->VertexProcessingTypes[ 0 ];
                = 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 D3DDISPLAYMODE 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 == D3DDEVTYPEN_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 CD3DEnumAdapter 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 D3DDISPLAYMODE 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_RefeshRateInHz = pSettings->DisplayMode.RefreshRate;
        d3dpp.BackBufferWidth           = pSettings->DisplayMode.Width;
        d3dpp.BackBufferHeight          = pSettings->DisplayMode.Height;
    } // End if fullscreen

    // Success
    return d3dpp;
}
```

## A Small Bonus

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.

```

// Display the settings dialog
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:

```

// Retrieve users options
m_D3DSettings = SettingsDlg.GetD3DSettings();

// Create the direct 3d device etc.

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 label 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**

$$V = [x \ y \ z \ 1] \times M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & 1 \\ 0 & 0 & b & 0 \end{bmatrix} = P[X = x \ Y = y \ Z = \text{ZBufferInfo} \ W = z]$$

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  $Z/z = 0.0 \text{ to } 1.0$  when the vector is inside the view frustum. ( $Z = \text{output } Z \parallel z = \text{input } Z$ )

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:  $\frac{Z}{Z} \text{ or } \frac{W}{W}$  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:

$$Z = (8.0 - 10.0 = -2) \times 1.11111111 = -2.22222222$$

$$ZB = \frac{Z}{W} = \frac{-2.22222222}{8} = -0.27777777$$

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.

Projection Matrix

$$V = [x \ y \ z \ 1] \times M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & 1 \\ 0 & 0 & b & 0 \end{bmatrix} = P[X = x \ Y = y \ Z = \text{ZBufferInfo} \ W = z]$$

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 to 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  $10.0 * 1.11111111$ :

$$\text{Ratio} = \frac{\text{FarPlane}}{(\text{FarPlane} - \text{NearPlane})} = \frac{100.0}{90} = 1.11111111$$

### Projection Matrix

$$V = [x \ y \ z \ 1] \times M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \text{Ratio} & 1 \\ 0 & 0 & -\text{NearPlane} * \text{Ratio} & 0 \end{bmatrix} = P[X = x \ Y = y \ Z = \text{ZBufferInfo} \ W = z]$$

Our third column is now complete. Let us test it with a z value of 100 again just to make sure:

$$\begin{aligned} Z = Vz * \text{ratio} &= (100 * 1.11111111) (= 111.1111111) \\ &\quad + \\ w * (-\text{NearPlane} * \text{Ratio}) &= 1 * -10 * 1.11111111 (= -11.1111111) \\ &= 111.1111111 - 11.1111111 \\ &= 99.9999999 \end{aligned}$$

When we divide by  $w$  (100 in our example) we get a Z-Buffer value:

$$ZB = \frac{Z}{W} = \frac{99.9999999}{100.00} = 0.99999999$$

### 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*1.1111111 * 1 * (-10      * 1.1111111)
Z=61.1111105      - 11.1111111
Z= 49.9999999 (Output Z from projection matrix)

```

An input z of 55.0 produces the output Z value of 49.9999999. When we divide by  $w$ :

$$ZB = \frac{Z}{W} = \frac{49.9999999}{55.0} = 0.909090908$$

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 1.1111111, 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 = 1.1111111*

Camera Space Z Values:

20.0	40.0	60.0	80.0	100
------	------	------	------	-----

Z Values returned from projection matrix:

11.1111111	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	0.83333333	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

$$\text{Ratio} = \frac{\text{FarPlane}}{\text{FarPlane} - \text{NearPlane}}$$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \text{Ratio} & 1 \\ 0 & 0 & -\text{NearPlane} * \text{Ratio} & 0 \end{bmatrix}$$

## 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 <b>world</b> 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 <b>view</b> 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 );

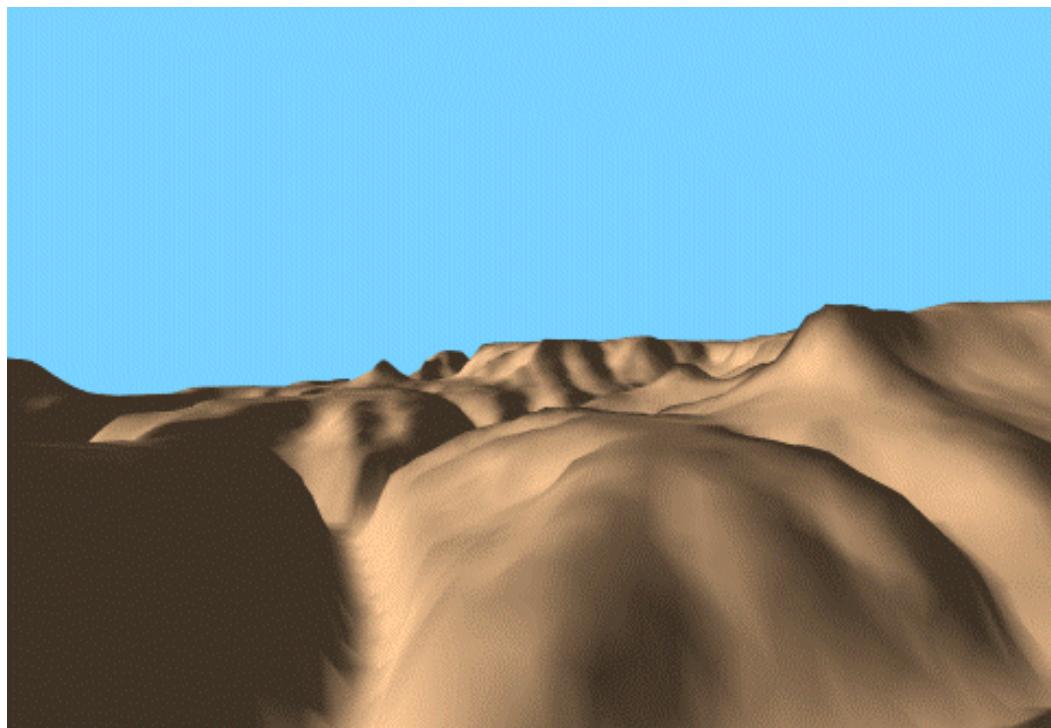
// Lets read it back out just for fun
IntVar = m_IntVector[ 0 ];

// Finally we'll adjust the value and assign it again :)
m_IntVector[0] = IntVar;
```

# Chapter Three:

## Vertex and Index Buffers

---



© 2003, eInstitute, Inc.

You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.



## Table of Contents

Introduction.....	3
Memory Types .....	4
Video Memory .....	5
AGP Memory .....	5
System Memory .....	6
Memory Pool Selection.....	6
Device Resources.....	7
CreateVertexBuffer.....	8
D3DPOOL_DEFAULT .....	11
D3DPOOL_MANAGED .....	13
D3DPOOL_SYSTEMMEM .....	16
D3DPOOL_SCRATCH .....	16
Vertex Buffer Performance.....	17
Vertex Buffer Read Statistics.....	20
Locking Vertex Buffers .....	21
Filling Vertex Buffers .....	23
Vertex Stream Sources.....	24
DrawPrimitive.....	25
Indexed Primitives .....	27
Creating Index Buffers.....	32
DrawIndexedPrimitive.....	34
DrawIndexedPrimitiveUP .....	37
Indexed Triangle Strips.....	37
Degenerate Triangles .....	42
Conclusion .....	45
Further Reading: .....	45



## Introduction

Vertex and index buffers are important device resources used to achieve the best possible application performance during primitive rendering. These resources will replace the vertex arrays we used with the DrawPrimitiveUP call used in Chapter 2. During this lesson we will discuss how indexed primitives remove the need for redundant vertices in our geometry. We will also look at how to take advantage of the vertex cache available on most 3D graphics adapters to minimize pipeline data transfer when possible. Finally, this lesson will provide you with valuable information on how to create and use vertex buffers in an optimal way on both hardware and software vertex-processing devices.

In the last lesson, we used the IDirect3DDevice9::DrawPrimitiveUP call to send vertices through the transformation and rendering pipeline. Recall that the ‘UP’ stands for ‘User Pointer’ because the application passes its own vertex data pointer into the function. This was a pointer to a vertex array located in system memory that the application could freely modify at will. The main problem with this approach is that the vertex data is contained in system memory while the hardware **geometry processing unit** or **GPU** (assuming one exists on the current hardware) requires that this data be accessible in on-board video memory (i.e. **local video memory**) or in **AGP memory** (i.e. **non-local video memory**) in order to work with it at maximum speed.

When vertex data is not in video memory the CPU must copy the system memory vertices over the bus into local video or AGP memory. The GPU does have direct memory access to system memory but it is much slower than accessing data in video memory on a hardware vertex-processing device. Because we are passing in an application created pointer to system memory and because the application can change this data at any time, the driver cannot safely cache the vertex data in video memory because it has no way of knowing whether or not the application has changed the memory contents. Therefore, each time the vertex array is rendered, DirectX Graphics will copy it into another area of memory first so that the GPU can be sure it is accessing the most current data. The new data area where these vertices reside is called a **vertex buffer** and will typically be located in AGP memory or local video memory, if a GPU is available. Once the vertex data is in the vertex buffer, the GPU can access it directly for fast processing. After the vertex data has been used, the temporary vertex buffer that was created is discarded. It will have to be recreated and destroyed each time vertices are rendered. This creates stalls in the rendering pipeline and results in significantly degraded application performance.

When the graphics adapter does not have hardware vertex processing capability, the situation is different. In such a case, the transformation and lighting of our vertices is done by the DirectX Graphics device in software. In this situation, using DrawPrimitiveUP would not degrade performance quite as much as it would in the HW T&L scenario. Nevertheless, the vertex data will still be copied into temporary system memory vertex buffers. So even on non-HW T&L devices we face the cost of creation, copying, and discarding memory each time we render vertex data. While the DrawPrimitiveUP function is indeed convenient, it should never be used in performance critical commercial code.



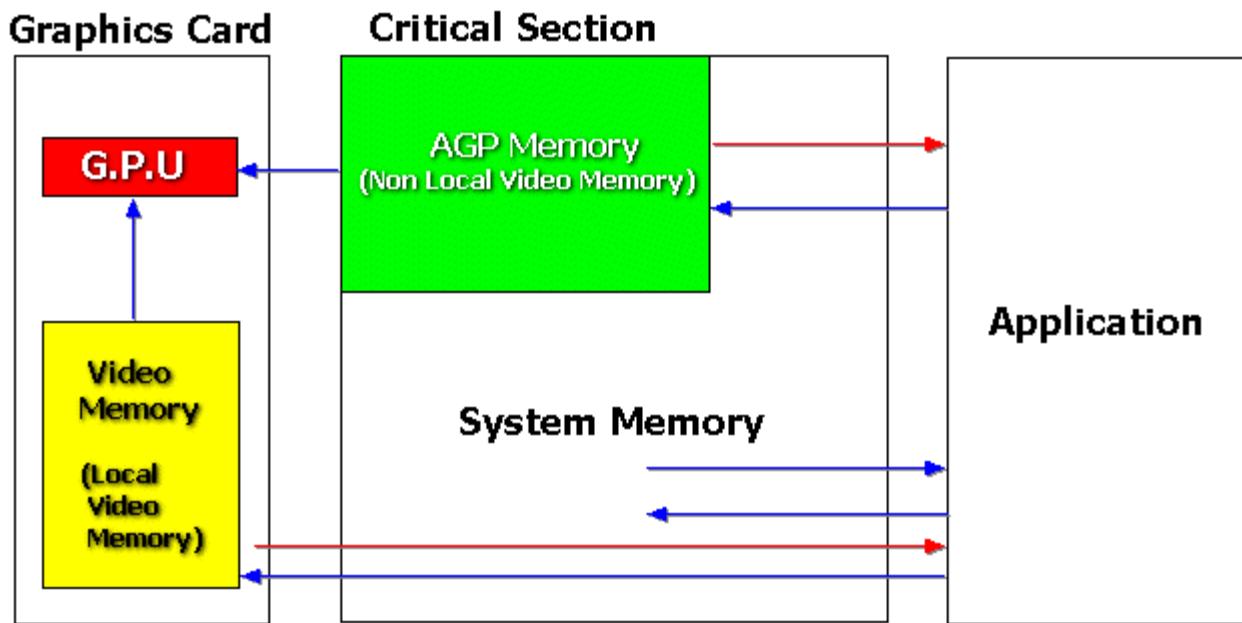
It stands to reason that one way to avoid the vertex-copying overhead of DrawPrimitiveUP is to store our data in a vertex buffer to begin with. This way the driver already has the data available in the correct format. That is exactly what we will do in this lesson.

Vertex buffers have a strict set of rules that, when followed, allow the driver to make optimization decisions about vertex data. For example, you cannot just read or write from your vertex buffer any time you please. You must first explicitly request a lock on the buffer. If the request was successful, you will be returned a pointer to the data (or a copy of the data) to work with. When you are done processing, you must unlock the buffer. This means that the driver can place or cache your vertex buffer in optimal memory without having to worry about the application changing the contents of the buffer without its knowledge. In a system that has a GPU, the vertex buffer will typically be stored either in AGP memory or local video memory. These memory pools can be quickly read by the GPU since it has direct memory access to them. The GPU can extract vertices from the buffer and transform them without having to tie up the CPU with data transfers between memory pools.

### Memory Types

Vertex buffer behaviour is dependant upon parameters defined at creation time. One of the most important performance related factors involves which memory pool the vertex buffer resides in. In most circumstances we will want a vertex buffer to be placed in local video memory or AGP memory. However, when the 3D hardware does not support T&L, then the vertex buffer will need to be created in system memory. This is quite logical; if no T&L facility is available on the graphics hardware, then software vertex processing will occur. Vertex data in system memory is within easy reach of the CPU based software transformation pipeline.

The following diagram shows an application running on a 3D graphics card with hardware vertex processing capability (a graphics card that has a GPU).



**Key:-**

- **Extremely Slow Read Access**
- **Fast Read Access**
- ← **Fast Write Access**

Let us begin by talking about the different memory types available to your application and the performance implications of using each type.

### Video Memory

Modern graphics cards typically have their own on-board memory. The GPU can access content stored in video memory very quickly for both read and write operations. Applications can write to video memory at reasonable speeds, but reading operations are terribly slow and should be avoided at all costs. If we know that a certain resource used for rendering will not change (i.e. it is static), then ideally we will want that resource to be placed in video memory. Again, in the case of vertex buffers, if the graphics hardware does not support T&L (in other words, the card does not have a GPU) then we do not want our vertex buffers placed in video memory. System memory is the preferred choice because it provides fast CPU access.

### AGP Memory

AGP enabled videocards are capable of interfacing at high speeds with reserved portions of system memory. The GPU has direct memory access to AGP memory much like its own local video memory. This means that data can be extracted from AGP memory directly by the GPU without having to



burden the CPU with the request. There is usually a BIOS setting that can be changed to control how much system memory is set aside to be used as AGP memory.

When system memory is reserved as AGP memory, it behaves very differently from standard system memory. AGP memory is flagged as a critical section and it cannot be cached by the CPU. This makes CPU data reads from AGP memory slow – much like reads from local video memory. Writing to AGP memory however is typically very fast. In addition, AGP memory is not allowed to be paged out to disk. This is very different from normal memory that can be written to the hard disk using the operating system's virtual memory manager. So care should be taken if you change BIOS settings where too much AGP memory is reserved.

Vertex buffers will often be placed in AGP memory. AGP memory is fast for the CPU to write to, and it is fast for the GPU to read from. However it is very slow for the CPU to read from due to the fact that the L1/L2 caches are disabled.

As we saw with local video memory, if the current system has no hardware T&L support, AGP memory is a poor location for storing vertex buffers. System memory is once again the best choice in this case.

## System Memory

System memory (heap memory) is the memory pool in which your applications run and in which memory allocations are made with operators like **new** and **delete**. This is where vertex buffers should be placed when there is no GPU available on the current system or if there is a GPU available but the application frequently needs to read back data from the vertex buffer. The latter is not a recommended scenario if a GPU is available. GPU access to vertices in a system memory vertex buffer is typically ten times slower than GPU access to local or non-local video memory vertex buffers.

## Memory Pool Selection

Ideally we want to structure our application so that vertex buffers will be placed in video memory when a GPU is available and system memory when it is not. We want the data stored in these buffers to be static, or if this is not possible, write-only. A video memory vertex buffer will be quick to render but will hurt performance if you have to read from it frequently. Creating the vertex buffer in system memory will be fast for CPU reads but will be significantly slower for the GPU to render since it will have to fetch the vertices over the system bus. Also note that when the GPU accesses a system memory vertex buffer, the CPU must play a role in the communication of that data and this can impact application performance.

If your application requires read access to a vertex buffer then your best solution is to create it in system memory (even if a GPU is available). Of course, there are often solutions to help you work around the performance penalties associated with reading operations. One of the most obvious is keeping a separate copy of the vertex data in system memory to use for CPU reading, and then writing



results out to a separate video memory copy when needed. Memory footprint is the clear downside here, but often it is worth it.

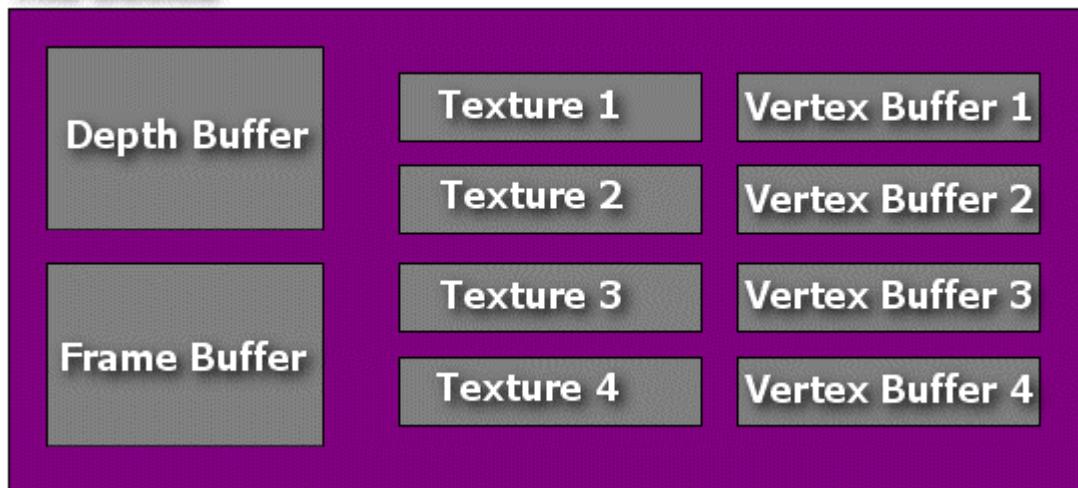
When we create a vertex buffer, we will specify various flags that will be used by the device object and the driver to determine where (in which memory pool) the vertex buffer will be placed.

While there are numerous rules and semantics listed for vertex buffers in the SDK documentation, very often the driver has some degree of autonomy to make its own choices. The more information we provide at vertex buffer creation time as to how we intend to use the buffer, the better chance that the driver will place it in the optimal memory for our needs. We will examine these rules as we progress through the chapter.

## Device Resources

**Resources** are data types that are created and owned by `IDirect3DDevice9` object. They include vertex buffers, textures, index buffers, frame buffers, depth buffers, and more.

### The Device



All resources have interfaces that are derived directly or indirectly from the `IDirect3DResource9` interface. This interface contains a set of common methods that apply to all resource types.

You will create a resource object by calling one of the device interface methods. In the case of a vertex buffer you call the `IDirect3DDevice9::CreateVertexBuffer` method. If creation is successful it will return a new `IDirect3DVertexBuffer9` interface. This process is similar for all resource types. For example, when you call the `IDirect3DDevice9::CreateTexture` method or the `IDirect3DDevice9::CreateIndexBuffer` method, you will get returned `IDirect3DTexture9` and



IDirect3DIndexBuffer9 interfaces respectively. We use the returned interface to manipulate the resource data.

Because the application does not own the resource data area, it cannot simply write data to the resource at will. In the case of a vertex buffer for example, although we have an interface, we have no means of filling it with data or reading data contained within, until we call the IDirect3DVertexBuffer9::Lock method. If the call is successful the method will return a pointer to the resource data area and the application can use it as it would any other memory pointer. It could read or write to the memory pointed to by it or use the pointer in memcpy function calls. We will discuss this in greater detail later in the lesson.

### CreateVertexBuffer

To create a vertex buffer we call the IDirect3DDevice9::CreateVertexBuffer function:

```
HRESULT CreateVertexBuffer
(
    UINT Length,
    DWORD Usage,
    DWORD FVF,
    D3DPOOL Pool,
    IDirect3DVertexBuffer9** ppVertexBuffer,
    HANDLE* pHandle <-(Reserved : Should always be set to NULL)
);
```

#### UINT Length

This parameter is used to tell the device how many bytes the vertex buffer will need. The value must be large enough to store at least one vertex. When using flexible vertex formats (FVF) the size will be equivalent to the size of our vertex structure (in bytes) multiplied by the number of vertices we intend to store in the vertex buffer. For example, if CMyVertex was our vertex structure and you wanted to store 100 vertices in the buffer, you could calculate the length as  $100 * \text{sizeof(CMyVertex)}$ . This would allocate enough memory for 100 CMyVertex structures within the vertex buffer.

#### DWORD Usage

The Usage flag is critical to vertex buffer performance as it can ultimately control which memory pool the vertex buffer will reside in. The D3DUSAGE constants are used by many device resource creation functions. Here we will discuss the constants as they apply to vertex buffers. This parameter can be 0 if no usage flags are required.

#### D3DUSAGE\_DYNAMIC

This flag informs the device object that we intend to modify the contents of the vertex buffer on a frequent basis. If hardware vertex processing is used then the vertex buffer will typically be placed in AGP memory for dynamic buffers and in local video memory for static buffers (although this varies across cards and drivers). There is no D3DUSAGE\_STATIC flag to indicate that we will not need to alter the contents of the vertex buffer throughout the life of the



application. Instead, the lack of a D3DUSAGE\_DYNAMIC flag is interpreted as a static vertex buffer request.

Where the vertex buffer gets placed is ultimately up to the driver. Most nVidia® drivers place all vertex buffers in AGP memory (both static and dynamic) when using a hardware vertex processing device. If the device is using software vertex processing then the vertex buffer (whether static or dynamic) will usually be placed in system memory. So this flag is simply a hint to the driver that we anticipate needing to frequently lock the vertex buffer for updates.

**Note:** Applications should not make IDirect3DVertexBuffer9::Lock calls in time critical code unless the D3DUSAGE\_DYNAMIC flag was specified at creation time.

When you lock a vertex buffer for access, there are several flags that you can pass to the device to minimize pipeline stalls and performance hits.

For example, if we call the Lock() method with the D3DLOCK\_NOOVERWRITE flag, we are promising the device that we will not alter any of the contents already in the vertex buffer although we may add additional vertices to it. This allows the driver to issue the lock, return a pointer, and then carry on immediately rendering from the same buffer our application is adding data to. Without this flag, it would have to wait until the application had finished altering the vertex buffer and unlocked it, before it could continue rendering. These flags (covered later when we look at the Lock method) are **not** available for static vertex buffers. Locking a static vertex buffer puts the GPU into a hard stall.

Dynamic vertex buffers can also be locked using another flag called D3DLOCK\_DISCARD. It is used if you intend to overwrite the contents and do not wish to stall the pipeline. Typically, this is handled by the device issuing a pointer to a new vertex buffer, which can be written while the hardware continues using the previous vertex buffer for transformation and rendering. This is another flag that cannot be used to lock static vertex buffers.

### **D3DUSAGE\_WRITEONLY**

This is a very important flag for maximum performance on a device with hardware vertex processing. It specifies that we do not intend to read data from the vertex buffer at any time. Because reading from video memory vertex buffers is so slow, the driver may decide to place the buffer in system memory if this flag is not specified. So, you will almost always want to include this flag in your creation parameters (assuming buffer reading is not required).

The worst-case scenario is if the driver was to ignore this flag and place the vertex buffer in video memory regardless of our intentions to read from it. This would seriously hurt performance whenever we locked the buffer and read from it. This of course would never happen in a responsibly written device driver, but the point here is that, it is the driver which ultimately decides where the vertex buffer should be placed. This decision is based in whole or in part on the hint flags that we send it during vertex buffer creation. As it turns out, the driver



development kit documentation explicitly states that any vertex buffer that is created without the D3DUSAGE\_WRITEONLY flag set must be placed in system memory. But as mentioned, driver implementations may vary across hardware.

The Game Institute ran some tests on our development machines using static vertex buffers without specifying the D3DUSAGE\_WRITEONLY flag. Benchmark results proved quite conclusively that the vertex buffer was being placed in video memory (either AGP or local). Significant performance hits were recorded for data reads.

### **D3DUSAGE\_SOFTWAREPROCESSING**

This flag indicates that we would like the transformation and lighting of the vertex buffer data to be performed in software using the DirectX Graphics software T&L pipeline. This usage flag must not be used on a device that has been created with D3DCREATE\_HARDWARE\_VERTEXPROCESSING behaviour, although it can be used on devices created with the D3DCREATE\_MIXED\_VERTEXPROCESSING behavior flag. It does not have to be explicitly specified when using a device created with D3DCREATE\_SOFTWARE\_VERTEXPROCESSING since processing is done in software anyway. If this flag is not specified on a hardware vertex processing device or a mixed vertex-processing device then vertex processing is done in hardware.

There are other D3DUSAGE flags that are applicable to vertex buffers but the ones listed above are the ones we are currently interested in. We will return to some of the other usage flags later in the lesson.

### **DWORD FVF**

This parameter tells the device the format of the vertices destined for the vertex buffer. For example, if we used a vertex structure which had an x, y, and z component and also a diffuse color component, we would pass the following flags:

Flags = **D3DFVF\_XYZ | D3DFVF\_DIFFUSE**

Flexible vertex format flags were covered in Chapter 2.

### **D3DPOOL\_POOL**

This flag allows our application to specify which memory pool it would like the resource to be placed into. When combined with the D3DUSAGE flags, it directly governs the performance and behaviour of our vertex buffers. It is worth noting that certain resource types are treated differently even when they share the same D3DPOOL. For now though our focus will be on its application to vertex buffer creation.



```
typedef enum _D3DPOOL
{
    D3DPOOL_DEFAULT = 0,
    D3DPOOL_MANAGED = 1,
    D3DPOOL_SYSTEMMEM = 2,
    D3DPOOL_SCRATCH = 3,
    D3DPOOL_FORCE_DWORD = 0x7fffffff
} D3DPOOL;
```

There are four possible pool types that we can choose for any resource. We will discuss each type along with its relationship to the vertex processing capabilities of the device.

### **D3DPOOL\_DEFAULT**

When we use the default pool, the driver will typically store the vertex buffer in the most optimal memory by taking into account the D3DUSAGE flag. If we specify this pool on a device that is only capable of software vertex processing, or if we specify this pool on a mixed mode device when we have specified the **D3DUSAGE\_SOFTWAREPROCESSING** flag, the vertex buffer will be created in system memory. If we specify this pool type on a mixed mode vertex-processing device without the **D3DUSAGE\_SOFTWAREPROCESSING** flag or if the device is a hardware vertex processing only device, then the vertex buffer will typically be placed in local or non-local video memory for maximum rendering performance. If we have not specified the **D3DUSAGE\_WRITEONLY** flag (even on a hardware vertex processing device) then the situation is more ambiguous. The driver may choose to place the vertex buffer in system memory because it assumes you might want to read from it. Alternatively, the driver may choose to ignore this flag and place the vertex buffer in video memory (local or non-local), which would carry a serious performance penalty if the vertex buffer were to be read from by your application.

Below we list the typical video card driver reaction to specifying the D3DPOOL\_DEFAULT enumerated type in combination with some of the D3DUSAGE flags covered previously.

#### D3DPOOL\_DEFAULT with a Hardware Vertex Processing Device

##### **Usage = D3DUSAGE\_DYNAMIC and D3DUSAGE\_WRITEONLY**

With this combination the driver will usually place the vertex buffer in AGP video memory. Writing to the vertex buffer is typically very fast but reading is extremely slow.

##### **Usage = D3DUSAGE\_DYNAMIC**

The driver may interpret the absence of the D3DUSAGE\_WRITEONLY flag as an indication that you will want to read from the vertex buffer at some point. Taking this into account the



driver *might* place the vertex buffer in system memory to increase reading speed at the cost of compromising rendering performance.

#### **Usage = D3DUSAGE\_WRITEONLY**

The lack of the D3DUSAGE\_DYNAMIC flag and the use of the D3DUSAGE\_WRITEONLY flag generally result in optimal creation. Often this will mean the driver will place the vertex buffer in local video memory or at the very least, in AGP memory. The driver expects that the vertex buffer will not be locked or updated and places it in the memory that provides maximum read performance for the GPU.

#### **Usage = D3DSOFTWARE\_PROCESSING**

This is not a valid flag for a hardware vertex-processing device because the GPU will still transform and light the vertices even if the vertex buffer is in system memory.

#### **Usage = 0**

When we specify no flag, we are indicating that we want to create a static vertex buffer and that we may want to read from it. A driver may decide to place the vertex buffer in video memory where a lock and read would be extremely expensive or it may decide that the lack of the D3DUSAGE\_WRITEONLY flag means the application wants to read from the vertex buffer and place it in system memory to aid read access (when in fact we probably had no such intention).

### D3DPOOL\_DEFAULT with a Software Vertex Processing Device

Using the D3DPOOL\_DEFAULT pool to create a vertex buffer on a software vertex-processing device will always create the vertex buffer in system memory. If this were not the case, performance would be severely degraded since the software module would have to extract the vertices from a buffer located in video memory.

### D3DPOOL\_DEFAULT with a Mixed Vertex Processing Device

Where the vertex buffer is placed on a mixed mode device is based on whether we created the vertex buffer with the **D3DUSAGE\_SOFTWAREPROCESSING** flag. If we did, then the vertex buffer is always created in system memory and behaves in exactly the same way as the software vertex-processing device described previously. The GPU will not be used to transform and light vertices.

If the **D3DUSAGE\_SOFTWAREPROCESSING** flag is not specified then the vertex buffer is treated like the hardware vertex-processing device scenario described above. This is also true of all D3DUSAGE flags specified in the hardware vertex processing case. The **D3DUSAGE\_SOFTWAREPROCESSING** flag allows you to switch functionality on a mixed mode



device between hardware vertex processing (using the GPU) and software vertex processing (using the CPU).

The **D3DPOOL\_DEFAULT** pool is often the preferred pool when you want to maximize performance on systems with a GPU. With the **D3DUSAGE\_WRITEONLY** flag specified, we ensure that the vertex buffer is placed in video memory for optimal rendering performance. Additionally, you should always use **D3DPOOL\_DEFAULT** for dynamic vertex buffers.

**Note:** If a device is lost, all vertex buffers that were created with the D3DPOOL\_DEFAULT type become invalid and must be destroyed and rebuilt again after the device has been reset. This is true of all resources created with the D3DPOOL\_DEFAULT type and not just vertex buffers. This is not true with D3DPOOL\_MANAGED and D3DPOOL\_SYSTEMMEM pool types.

## **D3DPOOL\_MANAGED**

Creating a vertex buffer using the D3DPOOL\_MANAGED type asks the device to manage the vertex buffer for us. The device will not only choose the optimal memory pool for the vertex buffer, it will also maintain a system memory copy of the buffer so that when a device is lost and later reset, it can restore the buffer back to video memory without application intervention.

The additional overhead of maintaining a system memory copy of a video memory vertex buffer on a hardware vertex processing device actually has some advantages. If our application should ever need to read data from the vertex buffer for example then the performance loss is typically not as severe because we will be locking and reading the system memory copy.

Unfortunately, we cannot create dynamic vertex buffers in the **D3DPOOL\_MANAGED** pool. Only static vertex buffers are viable candidates for this pool.

Let us examine the behaviour and creation processes for vertex buffers created in this pool type with the various D3DUSAGE flags.

### **D3DPOOL\_MANAGED with a Hardware Vertex Processing Device**

#### **Usage = D3DUSAGE\_DYNAMIC and D3DUSAGE\_WRITEONLY**

The D3DUSAGE\_DYNAMIC usage flag is not compatible with the D3DPOOL\_MANAGED flag and they should not be used together.

#### **Usage = D3DUSAGE\_DYNAMIC**

The D3DUSAGE\_DYNAMIC usage flag is not compatible with the D3DPOOL\_MANAGED flag and they should not be used together.



### **Usage = D3DUSAGE\_WRITEONLY**

A driver will typically place this static vertex buffer in the optimal memory; usually local video memory or at the very least, non-local video memory. When we lock a managed vertex buffer, we are returned a pointer to the system memory copy that is managed by the device object. Changes made to that copy are committed up to the video memory copy once the vertex buffer has been unlocked.

Results are undefined if you read data back from a managed vertex buffer when you have specified D3DUSAGE\_WRITEONLY. On a local test machine we were able to successfully read back data from a managed vertex buffer on a hardware vertex processing device and it was much faster than reading back from the same vertex buffer created using the D3DPOOL\_DEFAULT type. This is because we were reading from the system memory copy of the video memory vertex buffer managed by the device.

Note however that this is risky. We explicitly told the driver that we do not intend to read and the driver is under no obligation to make sure that the data in the system memory copy of the vertex buffer is correct or current. It will only guarantee that changes you make to the vertex buffer will be synchronized with the video memory copy once the lock has been released.

### **Usage = D3DSOFTWARE\_PROCESSING**

This is not a valid flag for a hardware vertex processing only device even in the case of managed vertex buffers. With a hardware vertex-processing device, the GPU will always transform and light the vertices even if the vertex buffer is in system memory.

### **Usage = 0**

When we do not specify any flags with a managed vertex buffer we are essentially telling the driver that we wish to create a static vertex buffer, which we may want to read from. Usually, this will still result in the vertex buffer being placed in video memory. The device will keep a system memory copy available that can be locked to provide decent CPU read/write performance.

**Note:** Because D3DPOOL\_MANAGED cannot be used to create dynamic vertex buffers, you should never use the D3DPOOL\_MANAGED memory pool for any vertex buffer that your application intends to lock in a time critical situation. Even when the vertex buffer has been placed in system memory by the driver, the GPU must still read from it. Locking it will place the GPU into a wait state and stall the pipeline.



## D3DPOOL\_MANAGED with a Software Vertex Processing Device

### **Usage = D3DUSAGE\_DYNAMIC and D3DUSAGE\_WRITEONLY**

You cannot use the **D3DUSAGE\_DYNAMIC** usage flag with the D3DPOOL\_MANAGED pool. Only static vertex buffers can be created with this pool type.

### **Usage = D3DUSAGE\_DYNAMIC**

You cannot use the **D3DUSAGE\_DYNAMIC** usage flag with the D3DPOOL\_MANAGED pool. Only static vertex buffers can be created with this pool type.

### **Usage = D3DUSAGE\_WRITEONLY**

The vertex buffer will be created in system memory because this is a software vertex-processing device. No system memory copy will need to be maintained as the vertex buffer is already in system memory.

It is still wise to specify **D3DUSAGE\_WRITEONLY** even when using a software vertex-processing device. The software transformation and lighting module may make optimizations based on the fact that the information in the vertex buffer does not have to be available for the application to read. Using this flag will always help you to get maximum vertex buffer performance.

Once again, if you specify **D3DUSAGE\_WRITEONLY** and then read back from the vertex buffer you may get undefined behaviour.

### **Usage = D3DSOFTWARE\_PROCESSING**

This flag is ignored with a software vertex-processing only device because its behavior is automatically implied by the type of device it is. You should still prefer to use this flag so that you can clearly see how your vertex buffers are being created when examine your code.

### **Usage = 0**

The vertex buffer will be created in system memory and can be safely written to and read from. This read-access guarantee may impede rendering performance when compared to vertex buffers created using the **D3DUSAGE\_WRITEONLY** flag.



## D3DPOOL\_MANAGED with a Mixed Vertex Processing Device

Where the managed vertex buffer is placed on a mixed mode device is based on whether we created the vertex buffer with the **D3DUSAGE\_SOFTWAREPROCESSING** flag. If we did, then the vertex buffer is always created in system memory and behaves in exactly the same way as the software vertex-processing device described above. If the **D3DUSAGE\_SOFTWAREPROCESSING** flag is not specified then the vertex buffer is treated the same way as one on a hardware vertex-processing device. This is also true of all the D3DUSAGE flags specified in the hardware vertex processing case.

### **D3DPOOL\_SYSTEMMEM**

A vertex buffer using this pool type is always created in system memory. It will not need to be recreated if the device is lost and reset. This pool is the clear choice for vertex buffers created for use with a software vertex-processing device. In fairness, even if we did specify D3DPOOL\_MANAGED or D3DPOOL\_DEFAULT, a system memory vertex buffer would be chosen in that case. On a hardware vertex-processing device, D3DPOOL\_MANAGED and D3DPOOL\_DEFAULT will usually place the vertex buffer in some form of video memory (assuming proper usage flags). So, if you wish to create a system memory vertex buffer with a hardware vertex-processing device, you must explicitly state this memory pool.

While you normally would not want to do this, perhaps your application requires a dynamic vertex buffer and it needs to read those vertices fairly often. This is a particularly nasty situation. Your best bet would probably be to create the vertex buffer in the system memory pool using the **D3DUSAGE\_WRITEONLY** flag. Locking the vertex buffer would be cheaper because it is a dynamic vertex buffer and CPU access would be decent because we are reading the vertex data back from memory that can be cached. Since managed vertex buffers cannot be dynamic, this is probably the best you can do.

System memory dynamic vertex buffers are generally slow on a hardware vertex-processing device. The penalty associated with the GPU having to fetch the vertices, coupled with the device management overhead for dynamic vertex buffers, degrades performance considerably -- about 10% of the speed of reading vertices from a video memory vertex buffer.

### **D3DPOOL\_SCRATCH**

This pool places the vertex buffer in system memory and it does not need to be recreated when the device is lost and reset. Unlike the D3DPOOL\_SYSTEMMEM pool type, vertex buffers created in this pool are completely inaccessible to the Direct3D device. This means they cannot be used for rendering.



You can think of vertex buffers in this pool type as simply being vertex containers. You can use these vertex buffers to store data, which you will later copy to another vertex buffer that is accessible from the device.

The **D3DPOOL\_SCRATCH** pool type vertex buffer can be created, locked, and copied. It is not a pool type you will use very often with vertex buffers, but it can be useful for other resource types.

## Vertex Buffer Performance

Let us explore some different vertex buffer creation possibilities and discuss the outcomes. The code assumes that **CVertex** is a defined vertex structure and that **pDevice** is a pointer to a valid IDirect3DDevice9 interface.

### I. Managed Static Vertex Buffer -- optimal render performance

```
IDirect3DVertexBuffer9 *pVertexBuffer;
DWORD fvf = D3DFVF_XYZ | D3DFVF_DIFFUSE;
int num_verts = 36;
pDevice->CreateVertexBuffer (sizeof(CVertex) * num_verts , D3DUSAGE_WRITEONLY , fvf ,
                             D3DPOOL_MANAGED, &pVertexBuffer , NULL)
```

#### Outcome A: Hardware Vertex Processing Device

The vertex buffer would be created in video memory with a system memory backup. Optimal render performance is the result, with the cost of increased memory footprint. Locking this vertex buffer will stall the software pipeline because it is a static vertex buffer and reading back from the buffer could result in undefined behaviour.

#### Outcome B: Software Vertex Processing Device

The vertex buffer would be created in system memory. Locking this vertex buffer will stall the software pipeline because it is a static vertex buffer and reading back from the buffer could result in undefined behaviour.

### II. Managed Static Vertex Buffer -- non-optimal render performance

```
IDirect3DVertexBuffer9 *pVertexBuffer;
DWORD fvf = D3DFVF_XYZ | D3DFVF_DIFFUSE;
int num_verts = 36;
pDevice->CreateVertexBuffer ( sizeof(CVertex) * num_verts , 0 , fvf , D3DPOOL_MANAGED,
                             &pVertexBuffer , NULL)
```

#### Outcome A: Hardware Vertex Processing Device

The driver may interpret the lack of a D3DUSAGE\_WRITEONLY flag as an indication of your desire for read access. It is likely that because this is a managed mode vertex buffer and therefore has a system memory copy for read access, the actual vertex buffer will still be placed in some form of video memory. You can safely lock this vertex buffer and read and write its contents. When it is locked, the system memory copy of the vertex buffer (if it has been placed in video memory) will have its pointer



returned. Changes made to the system memory copy will be committed to the video memory vertex buffer once the lock has been released. This is a static vertex buffer, so locking is still very expensive. It is also likely that by guaranteeing read access to the CPU, the rendering performance will be compromised.

### **Outcome B: Software Vertex Processing Device**

The vertex buffer will be created in system memory and can be locked, read from, and written to with confidence. Again, locking a static vertex buffer is expensive since it can cause a stall in the pipeline.

#### **III. Static Vertex Buffer -- optimal render performance**

```
IDirect3DVertexBuffer9 *pVertexBuffer;
DWORD fvf = D3DFVF_XYZ | D3DFVF_DIFFUSE;
int num_verts = 36;
pDevice->CreateVertexBuffer ( sizeof(CVertex) * num_verts , D3DUSAGE_WRITEONLY , fvf ,
                               D3DPOOL_DEFAULT, &pVertexBuffer , NULL);
```

### **Outcome A: Hardware Vertex Processing Device**

The driver will place the vertex buffer in video memory (AGP or local) for optimal GPU read access. Unlike the D3DPOOL\_MANAGED type, a system memory copy of the vertex buffer will not be maintained by the device object. This minimizes system memory overhead but requires your application to manually recreate the vertex buffer if the device is lost and reset.

Speed is optimal for the GPU when transforming and rendering vertices from this vertex buffer. Unlike the managed vertex buffer where locking returns a pointer to the system memory copy, the pointer returned from locking this buffer will typically be an aliased pointer directly into video memory. Therefore writing to this vertex buffer can be marginally faster than writing to a managed vertex buffer because the copy synchronization process of the system memory vertex buffer and the video memory vertex buffer is not necessary when the lock is released. Although you should not try to read back from this buffer because it was created with D3DUSAGE\_WRITEONLY, we were successfully able to do so during some tests. The performance results were (as one might imagine) simply terrible because the reads were being done directly from video memory.

### **Outcome B: Software Vertex Processing Device**

The vertex buffer will be placed in system memory so that the software pipeline can access the data as quickly as possible. The device makes optimization assumptions based on the fact that you are not going to be reading the data back from the vertex buffer when your application locks it. Obviously, locking the vertex buffer and reading back from it could result in undefined behaviour.

#### **IV. Managed Dynamic Vertex Buffer -- optimal render performance**

```
IDirect3DVertexBuffer9 *pVertexBuffer;
DWORD fvf = D3DFVF_XYZ | D3DFVF_DIFFUSE;
int num_verts = 36;
pDevice->CreateVertexBuffer ( sizeof(CVertex) * num_verts ,
                               D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY ,
                               fvf , D3DPOOL_DEFAULT, &pVertexBuffer , NULL)
```



### Outcome A: Hardware Vertex Processing Device

Typically the driver will place the vertex buffer into some form of video memory. With nVidia® hardware for example, AGP seems to be the default choice. Write accesses are typically quick and locking the vertex buffer can be extremely efficient when the correct locking flags are used. No pipeline stalls will occur when locking a dynamic vertex buffer. The pointer returned from the lock is typically an aliased pointer directly into video memory. Reading from this buffer would result in terrible performance at best and undefined behaviour at worst due to the fact that the driver may have swizzled the data into a proprietary format (not expecting your application to read it back).

### Outcome B: Software Vertex Processing Device

The vertex buffer will be placed into system memory. Locking this vertex buffer is much more efficient than locking a static vertex buffer as certain mechanisms are in place to prevent stalls in the pipeline. This buffer should not be read.

## V. Static Vertex Buffer – inefficient

```
IDirect3DVertexBuffer9 *pVertexBuffer;
DWORD fvf = D3DFVF_XYZ | D3DFVF_DIFFUSE;
int num_verts = 36;
pDevice->CreateVertexBuffer ( sizeof(CVertex) * num_verts , 0 , fvf , D3DPOOL_DEFAULT,
&pVertexBuffer , NULL);
```

### Outcome A: Hardware Vertex Processing Device

Results are undefined here because we created a vertex buffer in the default pool but we have not specified D3DUSAGE\_WRITEONLY. What happens from this point on is up to the driver and incorrect assumptions may be made.

### Outcome B: Software Vertex Processing Device

The vertex buffer is placed in system memory and can be reliably read from and written to, although locking can cause the performance penalty seen with all static vertex buffers.

## VI. Dynamic Vertex Buffer in System Memory

```
IDirect3DVertexBuffer9 *pVertexBuffer;
DWORD fvf = D3DFVF_XYZ | D3DFVF_DIFFUSE;
int num_verts = 36;
pDevice->CreateVertexBuffer ( sizeof(CVertex) * num_verts ,
D3DUSAGE_WRITEONLY | D3DUSAGE_DYNAMIC ,
fvf , D3DPOOL_SYSTEMMEM, &pVertexBuffer , NULL);
```

### Outcome A: Hardware Vertex Processing Device

The vertex buffer is created in system memory and can be efficiently locked and written to. It should not be read from. This vertex buffer can still be transformed and lit in hardware by the GPU although a performance hit will result from the fact that the GPU has to retrieve the vertices from system memory.



## Outcome B: Software Vertex Processing Device

The vertex buffer is created in system memory and can be efficiently locked and written to. It should not be read from. This vertex buffer can still be transformed and lit in hardware by the GPU although a performance hit will result from the fact that the GPU has to retrieve the vertices from system memory.

### Vertex Buffer Read Statistics

We discussed earlier that we should avoid reading from a vertex buffer -- especially a static vertex buffer. Below you will see some test results that indicate the performance of the different resource pools for vertex buffers. This helps us to identify where the driver was placing the test (static) vertex buffers. During our test we locked a vertex buffer containing 1089 vertices and read them back 10,000 times in succession. We then unlocked the buffer. The test machine was an Athlon® 1.4 GHz with a GeForce 3™ graphics card. Results were averaged over three tests:

<b>D3DPOOL_DEFAULT</b>	(Hardware Vertex Processing)	23 s 80.1601 ms
<b>D3DPOOL_MANAGED</b>	(Hardware Vertex Processing)	42.7314 ms
<b>D3DPOOL_SYSTEMMEM</b>	(Hardware Vertex Processing)	42.7683 ms

#### **D3DPOOL\_DEFAULT**

When we chose the default pool the vertex buffer was placed in video memory. The pointer returned from the lock was aliased directly into video memory. Note the significant drop in performance; the tests took over 23 seconds when the other two took much less than a second. This is because we are reading directly from some type of video memory. On our test machine, the driver placed the vertex buffer in video memory even if we did not specify **D3DUSAGE\_WRITEONLY** and we were still able to still read back from the buffer. The read times were unaltered by this. This was also true when we used a dynamic vertex buffer in the same pool. Each time, the vertex buffer was placed in video memory, which resulted in a huge performance hit when reading. Note that other drivers may decide to place the vertex buffer in system memory if the **D3DUSAGE\_WRITEONLY** flag is not specified. This would speed up read access but impair rendering performance.

#### **D3DPOOL\_MANAGED**

In this case the vertex buffer was still placed in video memory by the driver, but the device object has maintained a separate system memory copy of it. The meaning of this sentence is obscureLock calls returned pointers to this system memory copy and reading was much faster. We see quite clearly that reading from a managed vertex buffer is much faster than reading from a video memory vertex buffer. Writing to a managed vertex buffer is typically slightly slower due to the fact that an update to the video memory version must eventually take place. However, because the main vertex buffer is in video memory rendering speed is not significantly compromised.

#### **D3DPOOL\_SYSTEMMEM**

As expected, reading from a system memory vertex buffer is relatively fast. The results were the same as reading from a managed vertex buffer because we are reading from system memory in both cases.



Unlike the managed pool however, rendering would take a performance hit since the GPU will have to fetch the vertices over the bus from system memory.

We carried out the same read tests on a software vertex-processing device. Read times were also comparably fast since the vertex buffer was always in system memory. They are not shown above since the results are basically the same as reading from a **D3DPOOL\_SYSTEMMEM** pool on a hardware vertex-processing device.

## Locking Vertex Buffers

Once the vertex buffer is created, we need to fill it with vertex data. This is typically done at application startup for static vertex buffers. We call the `IDirect3DVertexBuffer9::Lock` method to retrieve a pointer to the vertex buffer data area. After we have finished filling the vertex buffer we must remember to call the `IDirect3DVertexBuffer9::Unlock` method to relinquish control of the vertex buffer back to the driver. Every call to Lock must be matched with a call to Unlock. This is very important.

**Note:** As you will see in later lessons, all resources follow the same rules for locking and unlocking to gain temporary access to the resource data area. You should never store the pointer returned from a Lock method since it will be invalid once the resource is unlocked. Further, there is no guarantee that a second call to the lock function on the same resource will return the same pointer. In fact, this is often not the case.

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

### **UINT OffsetToLock**

`OffsetToLock` specifies an offset into the vertex buffer in bytes. Locking can be optimised in some situations (especially with managed resources) if we specify only the region of the vertex buffer that we wish to modify. For example, if you did not need access to the first ten vertices in the buffer but did need access to the rest, you would want to pass in the value of `10 * sizeof(CVertex)`. This will return a pointer to the 11<sup>th</sup> vertex in the vertex buffer. If you pass in zero, the pointer returned will point to the start of the vertex data.

### **UNIT SizeToLock**

`SizeToLock` defines how many vertices you need access to, starting from `OffsetToLock`. If you pass in zero to both the `OffsetToLock` and the `SizeToLock` parameters the entire buffer will be locked and the pointer returned will point to the start of the data area. Otherwise, this value is used to lock only a section of the vertex buffer. If you only needed access to the 11<sup>th</sup>, 12<sup>th</sup>, 13<sup>th</sup>, 14<sup>th</sup>, and 15<sup>th</sup> vertices in the buffer you would use:



OffsetToLock = 10 \* sizeof(CVertex) and SizeToLock = 5 \* sizeof(CVertex).

### **VOID \*\*ppbData**

This is the address of a pointer that will point to the vertex data when the call returns. It is a temporary pointer that should be discarded once the resource has been unlocked. Usually you will pass a pointer to your own vertex structure type and cast it to void for the call.

### **DWORD Flags**

This will be a combination of one or more flags to aid the device in selecting an efficient locking strategy. The possible values are:

- D3DLOCK\_DISCARD
- D3DLOCK\_NO\_DIRTY\_UPDATE
- D3DLOCK\_NO\_SYSLOCK
- D3DLOCK\_READONLY
- D3DLOCK\_NOOVERWRITE

#### **D3DLOCK\_DISCARD**

This flag states that the application will write to the entire locked region. This allows the runtime to discard the current vertex buffer and a pointer to a new buffer is returned immediately. The discarded vertex buffer can continue to be used by the GPU while the new buffer is being filled.

This flag is only valid when locking a dynamic vertex buffer (or any other dynamic resource). It cannot be specified during the lock call if your vertex buffer was not created with the D3DUSAGE\_DYNAMIC flag. Additionally, it is recommended that this flag only be used for buffers created with D3DUSAGE\_WRITEONLY.

#### **D3DLOCK\_NOOVERWRITE**

This flag promises the device that the application will not alter any of the vertex data currently in the buffer. It could be used if you wanted to append vertex data to the end, or if you wanted to read from the vertex buffer. Because you are promising that your application will not alter the contents, the driver can lock the resource, return the pointer, and then continue to render from this buffer knowing that the vertex data is still current. The driver does not have to wait for the lock to return to continue processing any cached data. It is the most efficient locking flag.

This flag is only valid when locking a dynamic vertex buffer (or any other dynamic resource). If you specify both D3DLOCK\_DISCARD and D3DLOCK\_NOOVERWRITE then D3DLOCK\_DISCARD is ignored and only D3DLOCK\_NOOVERWRITE is used.



### D3DLOCK\_READONLY

This flag promises the driver that your application will not alter the data in the buffer or attempt to add data to it. It will only read from it. This can be beneficial if a driver was to store the vertex buffer in a non-native format internally for performance reasons. If this were the case then the data would have to be uncompresssed into a readable format for the application and then recompressed after the lock has been released to update the vertex buffer. If this flag is specified then the recompression is not necessary as the data has not changed.

The lock function will fail if this flag is specified when locking a vertex buffer created with the D3DUSAGE\_WRITEONLY flag.

### D3DLOCK\_NODIRTYUPDATE

By default, a lock on a resource adds a dirty region to that resource. This flag prevents any changes to the dirty state of the resource. Applications should use this option when they have additional information about the set of regions changed during the lock operation. You will probably not use this lock flag very often with vertex buffers.

### D3DLOCK\_NOSYSLOCK

The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will happen whilst the resource is locked. This flag causes the system-wide critical section not to be held for the duration of the lock.

A lock operation of this type is typically pretty slow, but it does enable the system to perform other duties, such as moving the mouse cursor. This option is useful for long-duration locks that would otherwise adversely affect system responsiveness, such as the lock of the back buffer for software rendering.

## Filling Vertex Buffers

The following code example shows how to create a static vertex buffer to hold six vertices. In this code **CVertex** is assumed to be an already defined vertex structure such as the one used in our demo applications. **pDevice** is assumed to be a valid IDirect3DVertexBuffer9 interface. Error checking is removed for readability.

```
// We will need a pointer to the vertex buffer interface
// In our example this is assumed to be a CGameApp class member variable
IDirect3DVertexBuffer9 * m_pVertexBuffer;
.....
// Declare a pointer to use for the lock.
CVertex *pVertex = NULL;
ULONG     ulUsage = D3DUSAGE_WRITEONLY;

// Create our vertex buffer ( 36 vertices (6 verts * 6 faces) )
m_pD3DDevice->CreateVertexBuffer( sizeof(CVertex) * 36, ulUsage,
                                    D3DFVF_XYZ | D3DFVF_DIFFUSE,
                                    D3DPOOL_MANAGED, &m_pVertexBuffer, NULL );
```



```
// Lock the vertex buffer and get ready to fill data
m_pVertexBuffer->Lock( 0, sizeof(CVertex) * 36, (void**)&pVertex, 0 );

// 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 );

// Unlock the buffer
m_pVertexBuffer->Unlock( );
```

This is a pretty straightforward example. Notice that we must call the Unlock function once we are finished filling the buffer. You are allowed to nest calls to Lock/Unlock pairs but any calls to render the buffer will fail if there are any outstanding locks on it. As you can see, the IDirect3DVertexBuffer9::Unlock method takes no parameters and should be paired with a prior call to IDirect3DVertexBuffer9::Lock.

## Vertex Stream Sources

In order to render a vertex buffer with the fixed function pipeline, we must set it as the currently active vertex buffer and make sure that the device knows the vertex format. As we did in our previous applications when we were not using vertex buffers, we must call the SetFVF function and specify the components in the vertices in our buffer using the flexible vertex format flags.

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

We tell the device to get the vertices from our vertex buffer using the IDirect3DDevice9::SetStreamSource function:

```
// Set the vertex stream source
m_pD3DDevice->SetStreamSource( 0, m_pVertexBuffer, 0, sizeof(CVertex) );
```

Several streams can be used to pass data from multiple vertex buffers. This can be useful if you wish to store position components in one vertex buffer (in stream zero) and have the texture coordinates stored in another vertex buffer (in stream two). In all of our applications, we will be using a single vertex stream (stream zero).

Let us have a look at the definition of the IDirect3DDevice::SetStreamSource function:

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



### **UINT StreamNumber**

This is the number of the stream you wish to bind the vertex buffer to. We will be using stream 0 for our applications.

### **IDirect3DVertexBuffer9 \*pStreamData**

The pointer to the interface of the vertex buffer you wish to bind to the stream.

### **UINT OffsetInBytes**

Offset from the beginning of the stream to the beginning of the vertex data measured in bytes. To find out if the device supports stream offsets, see the D3DDEVCAPS2\_STREAMOFFSET constant in D3DDEVCAPS2. You will usually set this value to zero (indicating no offset). Stream offsets are not supported by all devices.

### **UINT Stride**

The stride of our vertex format is the amount of bytes from the start of one vertex to the start of the next vertex in the vertex buffer. Basically, this is the size of a single vertex.

## **DrawPrimitive**

At this point we have created and filled a vertex buffer, set the FVF, and attached the vertex buffer to stream zero. All that remains is to send the vertex data to the rendering pipeline. In the last chapter, we did this by calling the DrawPrimitiveUP function. This time we will call the IDirect3DDevice9::DrawPrimitive function instead. This tells the device to extract the vertices from the vertex buffer currently bound to the vertex stream(s).

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

The DrawPrimitive function fires the vertices from the currently set vertex buffer into the transformation and lighting pipeline (assuming they are untransformed vertices). We will set our world, view, and projection matrices prior to calling the function just as we did in Chapter 2.

### **D3DPRIMITIVETYPE PrimitiveType**

Describes how the vertices in the vertex buffer are to be rendered as primitives. Valid values are D3DPT\_POINTLIST, D3DPT\_LINELIST, D3DPT\_LINESTRIP, D3DPT\_TRIANGLELIST, D3DPT\_TRIANGLESTRIP, or D3DPT\_TRIANGLEFAN.



## UINT StartVertex

Although our application can use many vertex buffers (one for each object in our scene if we wish), it is often beneficial to store multiple objects within a single buffer. One of the reasons is that changing vertex buffers (by calling `IDirect3DDevice9::SetStreamSource`) can be a moderately expensive operation. If we store many objects within a single buffer we can minimize the number of vertex buffer changes that our application needs to make. This parameter allows us to store the meshes in a single vertex buffer sequentially and render one section at a time.

For example, we could have mesh 1 stored in the vertex buffer using the first 100 vertices and mesh 2 following mesh 1 in the vertex buffer with another 100 vertices. To render mesh 1, we would set its world matrix and call `DrawPrimitive` with a `StartVertex` parameter of 0 and a `PrimitiveCount` with a number such that its faces are rendered using the first 100 vertices. Then we could set the second mesh world matrix and call `DrawPrimitive` with a `StartVertex` of 100 and a primitive count value such that it renders the appropriate number of triangles for that mesh taking into account the **D3DPRIMITIVETYPE** being used.

## UINT PrimitiveCount

The number of primitives to render in this call. The value will be based on the primitive type:

- PointList (`PrimitiveCount = NumberOfVertices`)
- LineList (`PrimitiveCount = NumberOfVertices / 2`)
- LineStrip (`PrimitiveCount = NumberOfVertices - 1`)
- TriList (`PrimitiveCount = NumberOfVertices / 3`)
- TriStrip (`PrimitiveCount = NumberOfVertices - 2`)
- TriFan (`PrimitiveCount = NumberOfVertices - 2`)

The next code snippet demonstrates rendering multiple objects where each mesh is stored in its own vertex buffer. The vertex buffers are assumed to hold untransformed vertices, and each object in the world is assumed to have a correctly initialized world matrix and a pointer to its own vertex buffer containing the vertex data. All vertices share the same FVF code set at application startup. The vertex data is arranged to be rendered as a triangle list.

```
// Clear the buffers
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 < NumberOfObjectsInWorld; 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 primitives as a triangle list
m_pD3DDevice->DrawPrimitive( D3DPT_TRIANGLELIST,
                               0,
                               m_pObject[I].NumberOfVertices/3 );

} // Next Object

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

// Present the buffer
m_pD3DDevice->Present( NULL, NULL, NULL, NULL );
```

Notice that we call the SetStreamSource function during each iteration of the loop because in this example each object has its own vertex buffer.

A more efficient approach might be to store all of the objects that share the same flexible vertex format in the same vertex buffer. In this case, each object would need to store an index into the vertex buffer where its vertices begin. We would then render that section by calling DrawPrimitive and specifying this index as the **StartVertex** parameter. We would also be able to move the call to SetStreamSource outside of the loop and set it once for those objects.

Before moving on to the next section, please open your workbook to Lab Project 3.1 and spend some time examining the source code. This project addresses creating, filling, and rendering vertex buffers.

### Indexed Primitives

In Lab Project 3.1, the GPU had to transform 36 vertices per cube when only 8 unique points existed. It is certainly good that we were able to render the entire cube with one function call (versus our previous applications), but this still seems extraordinarily wasteful. When a mesh has hundreds or thousands of triangles (as will the next mesh we examine) the performance implications of all of this redundant processing are clearly not good.

In this section, we will address the concern about data redundancy while preserving the preference for rendering with as few function calls as possible. This solution will apply to all of our primitive types (strips, fans, or lists), so our ability to store data in formats that suit our needs will also be preserved. The technique we will use is called **indexed primitive rendering**. Beyond simply resolving the redundancy issue, there is another important benefit we will see. Indexed rendering allows hardware to utilize a small local memory cache for temporary vertex storage. This vertex cache can, under the right circumstances, significantly improve application performance.

Rendering with indices is a straightforward concept. In addition to our vertex buffer, we will send the device a second buffer filled with indices into that vertex buffer. This buffer is called an **index buffer**. Each element in the index buffer is the index of a vertex in the vertex buffer. We essentially treat this concept as two parallel arrays. One array (the vertex buffer) holds the building-block vertices. The



second array (the index buffer) holds references into the first array that are used select out the vertices needed to construct triangles.

If we were rendering a triangle list using indices, then the first three indices in the index buffer would describe the vertices in the vertex buffer used to form the first triangle. The next three indices would describe the vertices in the vertex buffer that comprise the second triangle. And so on. This allows us to reuse the same vertex in multiple triangles simply by specifying its index in each triangle that requires it. This approach can completely eliminate the need for duplicated vertices when all vertex properties are shared, and in turn eliminate redundant vertex processing.

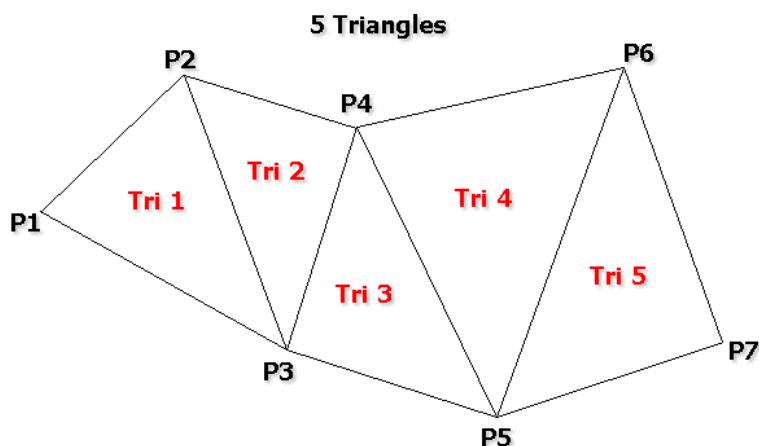


Figure 3.1

Fig 3.1 depicts seven unique position vertices used to build five triangles. If we wanted to render this list of triangles as a triangle list, we would need to duplicate vertices in the vertex buffer because the device expects three vertices for each triangle when using the **D3DPT\_TRIANGLELIST** primitive type.

The vertex buffer would look like the list of vertices shown below:

Tri 1	Tri 2	Tri3	Tri 4	Tri 5
P1 , P2 , P3 ,	P2 , P3 , P4 ,	P3 , P4 , P5 ,	P4 , P5 , P6 ,	P5 , P6 , P7

Positions P3, P4, and P5 are all duplicated three times because they belong to three separate triangles.

*Note that indeed we could render this example mesh as a triangle strip and eliminate the redundant vertices without the need for indices, but since this is an indexing example, please ignore strips for now.*

Under an indexed based scenario, the situation shifts to become:



## Vertex Pool

VertexBuffer = P1, P2, P3, P4, P5, P6, P7 = 7 vertices

This vertex buffer now serves as a pool rather than a triangle list. The triangle list is moved to the index buffer:

Tri 1	Tri 2	Tri 3	Tri 4	Tri 5
-----	-----	-----	-----	-----
IndexBuffer = 0,1,2 , 1,2,3 , 2,3,4 , 3,4,5 , 4,5,6				

Each element in the index buffer describes the offset (zero-based) into the vertex buffer of the vertex to be used in the correct location. For example, Triangle 4 references vertices P4, P5, P6 using index values 3, 4, and 5.

Note that although we have to allocate a new resource (the index buffer) we generally wind up saving a considerable amount of memory. Indices are typically 16 bit values (although 32 bit values are possible as well). In the case above we managed to eliminate 9 vertices from the vertex buffer. Consider even a simple vertex structure that stored a position (3 floats) and a diffuse color (1 DWORD). That is 16 bytes. For the 9 vertices we eliminated we reduced the buffer size by 144 bytes. Our 18 indices at 16 bits each take 36 bytes of storage for an overall savings of 108 bytes. If the vertex format was more complex (as will usually be the case) memory savings can start to add up.

More important is the point that using indices enables the GPU (if hardware vertex processing is enabled) to cache vertices so that they do not have to be processed multiple times. This can improve performance by an order of magnitude. In the above arrangement, vertices P1, P2 and P3 would be transformed and lit first. When we render the second triangle, vertices P2 and P3 are already in the vertex cache and do not have to be transformed and lit again. And so on for the other triangles. The vertex cache is a pretty scarce resource that is implemented on nVidia® based cards as FIFO buffers. The next table lists the cache size on the GeForce™ series of cards.

NVidia Model	Vertex Cache Size
geForce	10 Vertices
geForce 2	10 Vertices
geForce 3	18 Vertices
geForce 4	18 Vertices

The vertex cache is a valuable resource so it is important to order your indices so that triangles that use the shared vertex are located close together in the vertex buffer. This ensures that when a vertex enters the cache, other triangles can be rendered that use that vertex before it is flushed from the cache. If you do not do this then there is a good chance that the vertex will have been removed from the cache by the time the next triangle using it is transformed and lit. When this happens, that same vertex will have to be pumped through the transformation pipeline again.



**Important:** The vertex cache is only available when using indexed primitives.

Let us look at one more example, just to make sure we have the concept fully nailed down. We return once more to our favorite 3D shape:

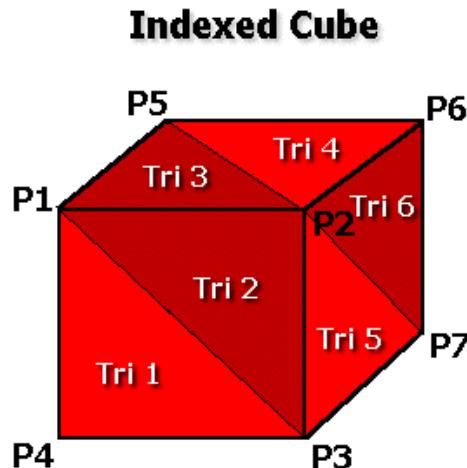


Figure 3.2

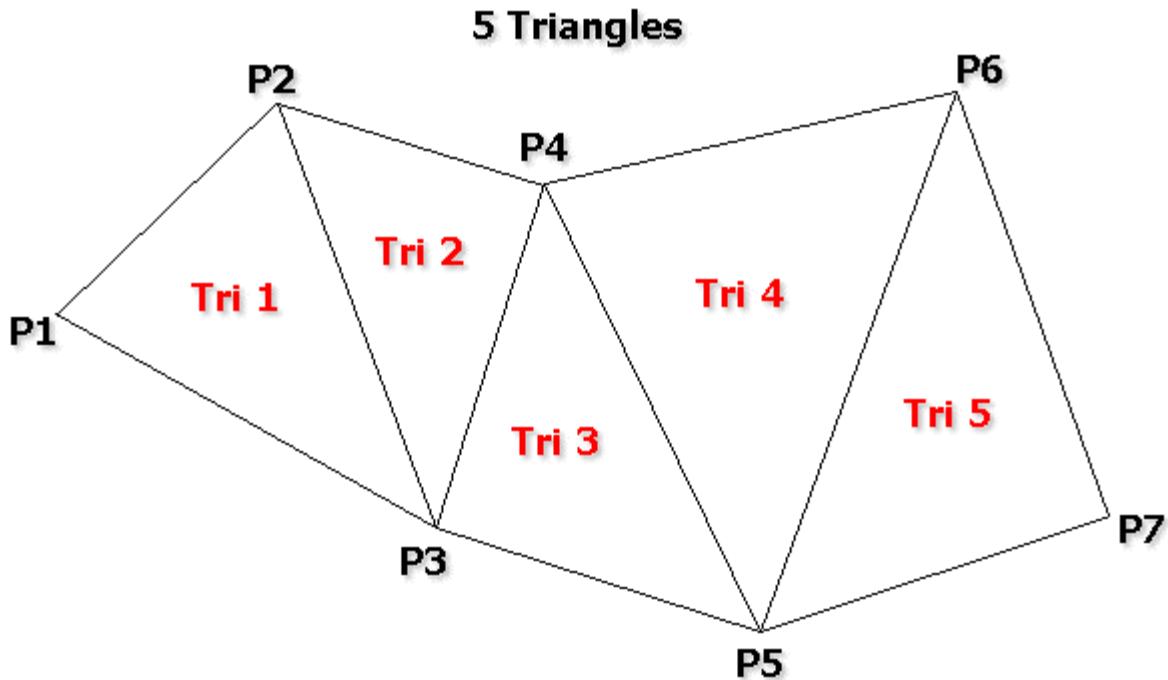
In Fig 3.2 we have labeled only the seven vertices that are used by the six visible triangles making up the three cube faces. Ignoring the back faces for now, we could create a vertex buffer for the six visible triangles as:

Vertex Buffer = P1 , P2 , P3 , P4 , P5 , P6 , P7

The corresponding index list for the six triangles would be:

Tri 1	Tri 2	Tri 3	Tri 4	Tri 5	Tri 6
-----	-----	-----	-----	-----	-----
Index Buffer = 0,2,3	, 0,1,2	, 4,1,0	, 4,5,1	, 1,6,2	, 1,5,6 = 18

We have represented six triangles as a triangle list using only seven vertices. Notice that the index count is now what the old vertex count used to be for each primitive. For a triangle list, the number of indices needed is NumTriangles \* 3. But we are not limited to indexed triangle lists. We can use indices with any D3DPRIMITIVETYPE. For example, take another look at the image we saw earlier:



If we wanted to render the above as an indexed triangle strip we would create the following vertex and index buffers:

Vertex Buffer = P1 , P2 , P3 , P4 , P5 , P6 , P7

There is no change here, since the vertex buffer is just a vertex pool that will be referenced by the index buffer to create triangles:

Tri 1 Tri 3  
----- ----- ...  
IndexBuffer = 0 , 1 , 2 , 3 , 4 , 5 , 6  
-----  
Tri 2

Note the consistency with the strips we saw in the last chapter. The first three vertices in the vertex buffer describe the first triangle, and then every additional vertex created a new triangle by using the last two vertices of the previous triangle. This same behaviour carries over when using indexed triangle strips.

We can calculate the number of indices needed to render an indexed triangle strip as  $\text{NumTriangles} * 2$ . This is identical to the way we calculated the vertices needed for a non-indexed triangle strip in Chapter 2.



## Creating Index Buffers

Like vertex buffers, index buffers are device resources that are derived from `IDirect3DResource9`. We can create static and dynamic index buffers (using the same `D3DUSAGE` flags), lock and unlock for read/write access, and we can set them as the active index buffer so that the device will use the index buffer to fetch the indices when rendering. To create an index buffer we call the `IDirect3DDevice9::CreateIndexBuffer` method.

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

### **UINT Length**

This specifies the length (in bytes) that you wish your index buffer to be. There are two different index formats you can use (16 or 32 bit). This format is specified in the `D3DFORMAT` parameter. To create an index buffer to hold ten 16-bit indices, the length of the buffer would need to be  $10 * 2 = 20$  bytes.

### **D3DUSAGE Usage**

Identical to the vertex buffer usage options discussed earlier and the same rules apply: if you need to lock the index buffer in time critical situations then make sure it is created with both the `D3DUSAGE_WRITEONLY` and `D3DUSAGE_DYNAMIC` flags.

### **D3DFORMAT Format**

There is a choice of two format types that are applicable to index buffers: `D3DFMT_INDEX16` or `D3DFMT_INDEX32` (describing 16- or 32-bit indices respectively). Normally you will use 16-bit indices. If you have more than 65,535 vertices within a single vertex buffer then you could use 32-bit indices -- although even then it is not strictly necessary. As we will see later, we can use a special offset parameter during the rendering call to address just this sort of situation. This is preferable to using what is essentially twice as much memory and bus bandwidth during rendering.

### **D3DPOOL Pool**

The implications for index buffers with regards to pool type are essentially the same as for vertex buffers. If you use a managed index buffer, then you will not have to recreate the index buffer should the device become lost. The driver will also try to place the index buffer in optimal memory. If you choose the default pool, then the index buffer will also be put into optimal memory but will have to be recreated by your application when the device is lost and restored. If you intend to read from the index buffer often then you should place the index buffer in either the managed pool or system memory pool. If you wish to create a dynamic index buffer then it must go in the default pool. Finally, you will want to place the index buffer into system memory on a software vertex-processing device.



## IDirect3DIndexBuffer9\*\* ppIndexBuffer

This is where we pass the address of a pointer to an IDirect3DIndexBuffer9 interface. If the device was able to create the index buffer successfully then this pointer will point to a valid IDirect3DIndexBuffer9 interface when the function returns.

## HANDLE \*pHandle

Reserved. This parameter should be set to NULL.

The next code snippet creates a static managed index buffer that would hold 36 16-bit index values.

```
IDirect3DIndexBuffer9 * pIndexBuffer;
DWORD ulUsage = D3DUSAGE_WRITEONLY;
pDevice->CreateIndexBuffer(sizeof(USHORT) * m_nIndexCount, ulUsage, D3DFMT_INDEX16,
                           D3DPOOL_MANAGED, &pIndexBuffer, NULL );
```

Providing the function was successful, we can now lock the buffer using the returned interface. The IDirect3DIndexBuffer9::Lock method should look familiar:

```
HRESULT Lock
(
    UINT OffsetToLock,
    UINT SizeToLock,
    VOID **ppbData,
    DWORD Flags
);
```

This lock method is exactly the same as the IDirect3DVertexBuffer9::Lock method. The first two parameters allow us to lock only a region of the index buffer. If we pass zero for both of these parameters then we will get back a pointer to the start of the index buffer data area. The third parameter is where we pass the address of a pointer that will point to the data area should the lock be successful. The final parameter can be any of the D3DLOCK flags that we discussed earlier when we discussed vertex buffers:

- D3DLOCK\_DISCARD
- D3DLOCK\_NO\_DIRTY\_UPDATE
- D3DLOCK\_NO\_SYSLOCK
- D3DLOCK\_READONLY
- D3DLOCK\_NOOVERWRITE

Refer back to the section on vertex buffer if you have forgotten the benefits these flags can provide. Once we have our index buffer, we can lock it as follows.

```
USHORT *pIndex;
pIndexBuffer->Lock (0, sizeof(USHORT) * m_nIndexCount, (void**)&pIndex, 0);
```

Provided the lock is successful, we can use the returned pointer to place values into our index buffer.

```
*pIndex++ = 0;
```



```
*pIndex++ = 1;
```

Once we have finished placing the values into the index buffer, we unlock it.

```
pIndex->Unlock();
```

### DrawIndexedPrimitive

Rendering indexed primitives is a simple affair. First, we attach the vertex buffer to stream 0 as we did before. Then we need to inform the device about the index buffer we wish to use. The IDirect3DDevice9 interface has a method called SetIndices that allows you to pass in an interface to an index buffer:

```
HRESULT SetIndices( IDirect3DIndexBuffer9 *pIndexData );
```

As with all device state changes, these buffers will remain active until they are changed. This means we can set the vertex buffer, and set the index buffer and they will remain the current index and vertex buffers used for rendering until they are unset.

Finally, we call the rendering function IDirect3DDevice9::DrawIndexedPrimitive:

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

We will examine the parameters slightly out of order as it should make the concepts easier to understand.

#### D3DPRIMITIVETYPE Type

This tells the device how the indexed primitives are arranged. In this case, indices define the triangles, so the vertices can be stored in an arbitrary order so long as the indices reference them correctly given the specified primitive type. The possible parameters here can be D3DPT\_POINTS, D3DPT\_LINELIST, D3DPT\_LINESTRIP, D3DPT\_TRIANGLELIST, D3DPT\_TRIANGLESTRIP, or D3DPT\_TRIANGLEFAN.

#### UINT StartIndex

This value describes the first index in the currently set index buffer that we want to start rendering with. For example, if we had 100 indices and a StartIndex of 30, only the last 70 indices in the index buffer would be used.



### **UINT PrimitiveCount**

This informs the driver how many primitives you wish to render. There must be enough indices in the index buffer to fulfill this request. For example, if we were rendering using a D3DPT\_TRIANGLESTRIP primitive type and we wanted to render 100 triangles, there would need to be 102 indices in the index buffer. If we were using a D3DPT\_TRIANGLELIST primitive type, there would need to be 300 indices in the index buffer. If you have specified a StartIndex value that is non-zero, then there must be enough indices in the array from the specified offset in the index buffer to the end of the buffer to fulfill the primitive count request.

### **INT BaseVertexIndex**

This allows you to specify a value that will be added to all index values before they are used to index into the vertex buffer. If we specified a base vertex index value of 1000, and our index buffer has three indices in it with the values (6, 7, and 8), the driver would add 1000 to each index and fetch vertices 1006, 1007, and 1008. This allows you to use the same index values and map them to different areas of a vertex buffer. It also solves the problem discussed earlier regarding 16- and 32-bit index values since you can now use this value to offset beyond the 65,535 limitation imposed by 16-bit indices.

### **UINT MinIndex**

This is the index of the first vertex used in the call. The BaseVertexIndex value will be added to this value when rendering so this does not need to be taken into account at this time. If we have a three-element index buffer consisting of indices (20, 21, and 22) and we had a BaseVertexIndex value of 200, we would specify a MinIndex of 20. When transforming these vertices, the device will add the BaseVertexIndex value to the MinIndex value such that it knows the minimum vertex used in the call is actually 220.

### **UINT NumVertices**

This is the number of vertices in the vertex buffer used in this call. Let us say for example that we have an index buffer (10,11,12,13,14,15,16,17,18). Let us also say that we have set a BaseVertexIndex value of 100 and we have a StartIndex value of 3. This is how it looks:

Index Buffer	= 10,11,12, 13,14,15, 16,17,18
StartIndex	= 3
BaseVertexIndex	= 100
MinIndex	= 13
NumVertices	= 6

The StartIndex value 3 means that the first three indices are skipped over and are not used in this call. The section of our index that will be used is:

IndexBuffer = 13,14,15, 16,17,18

BaseVertexIndex will be added to the indices so the device will use vertices:

VerticesUsed = 113,114,115, 116,117,118



Because we are skipping the first 3 indices, the minimum vertex index is 13 because this is the lowest vertex index used in our index buffer. When rendering, the device will add the BaseVertexIndex value to the MinIndex value and it knows that vertex 113 is the first vertex used.

Finally, we render two triangles from the remaining six vertices in the buffer. This creates triangle 1 from vertices 113,114,115 and triangle 2 from vertices 116,117, 118.

It is sometimes initially difficult to understand the need for MinIndex and NumVertices when it would seem that the primitive count should ultimately describe how many vertices we are using. But this is not strictly true because in our examples we have used vertices stored consecutively in our vertex buffer. However you might have an index buffer with one triangle using the indices (0, 9, 350). In this instance, we would have to set the MinIndex to 0 and the number of vertices used to 350.

When using a hardware vertex-processing device, the MinIndex and NumVertices parameters are ignored. This is because the GPU has its own vertex cache, allowing it to very efficiently grab vertices when they are needed and store them in this memory. When we are using a software vertex-processing device however, the code can transform the vertices much quicker if it transforms a block of vertices in advance. This is why we need to pass in the MinIndex and NumVertices parameters. It processes the block of vertices in this range in one pass before it uses them for rendering.

This brings up an important optimization point. If, as in the above example, we had an index buffer with indices (0, 9, 350) the software transformation engine would have to transform all 350 vertices in advance even though we are only using three of them. This is why it is crucial to store vertices in the vertex buffer in an ordered fashion. They should be grouped such that a single mesh's vertices are all in one section, another mesh in another section and so on. Although this is not as critical on a hardware vertex-processing device, it is still important to store vertices in the vertex buffer in a localized manner so that the GPU vertex cache is used to its maximum potential. The vertex cache has very limited storage space so you should try to organize your indices in such a way that all triangles that share a vertex are stored together in the index buffer.

In our lab projects for this lesson, most of these parameters are simplified by the fact we are rendering using the entire contents of the index buffer and are not using a BaseVertexIndex value. Both BaseVertexIndex and StartIndex can be set to zero.



## DrawIndexedPrimitiveUP

It is worth mentioning that we do not have to use index buffers to use indexed rendering. In Chapter 2 we used the DrawPrimitiveUP function to render vertices using application managed arrays rather than vertex buffers. Similarly (although we will not use it in the course), there is a function in the IDirect3DDevice9 interface called DrawIndexedPrimitiveUP. It allows you to pass user defined pointers to vertices along with an array of indices stored in normal application memory. It works just like the DrawIndexedPrimitive we studied in the last section so you should have no trouble understanding it should you choose to experiment with it.

```
HRESULT DrawIndexedPrimitiveUP
(
    D3DPRIMITIVETYPE PrimitiveType,
    UINT MinVertexIndex,
    UINT NumVertexIndices,
    UINT PrimitiveCount,
    const void *pIndexData,
    D3DFORMAT IndexDataFormat,
    CONST void *pVertexStreamZeroData,
    UINT VertexStreamZeroStride
);
```

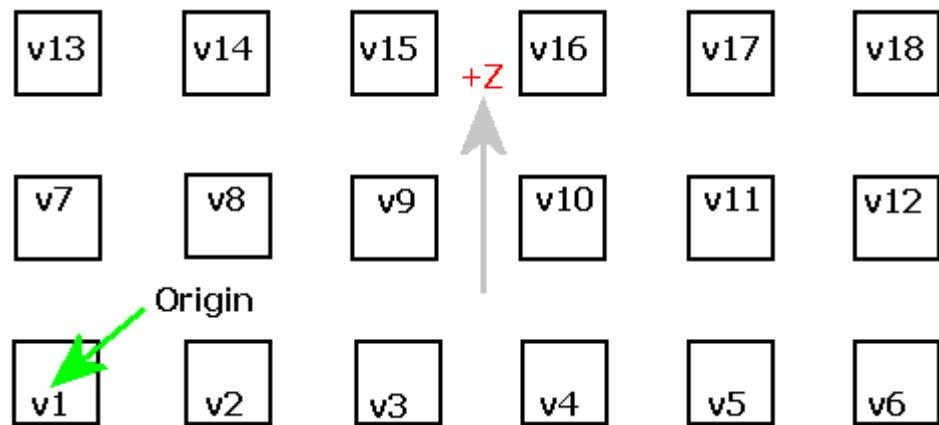
## Indexed Triangle Strips

It is now time to examine some geometry that is a little more challenging than the cubes we have been using to date. In Lab Project 3.2 we are going to build and render a terrain (an outdoor landscape) using vertex and index buffers. The terrain will be rendered using indexed triangle strips. If you are unfamiliar with terrains, quad grids, and height maps, this would be a good time to open your workbook and read the first few pages of discussion for Lab Project 3.2. This will give you some foundation as to how we will create a terrain and some important performance issues to consider. Once you have finished reading these pages, please continue with the next section in this text before beginning to examine any code.

3D graphics cards often have a penchant for triangle strips (and especially for indexed triangles strips). They can typically process and render indexed triangle strips faster than any other primitive type. Certainly it would be preferable if we could store and render each terrain submesh as a single triangle strip using one call to DrawIndexedPrimitive. This will be much faster than rendering one row of quads at a time. Although, if you remember how strips work, you might be wondering how you could render an entire mesh with multiple rows of quads as a single strip. For example, rendering the first row of a strip would seem easy enough, but once we get to the end of the first row, how could we tell the device not to draw a connecting triangle between the first and second row, and then continue rendering the second row as normal? The answer is that you cannot; at least not quite in that way. But you can use something known as a degenerate triangle and this will help you accomplish that goal.



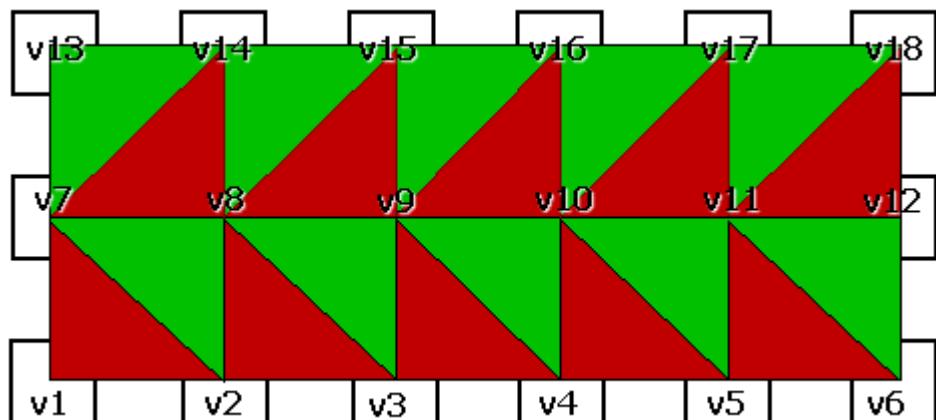
Let us first look at how the vertices will be arranged in the vertex buffer. The following diagram shows the vertices in world space, with the origin of the coordinate system at the bottom left vertex. We are looking down on the vertices from above with the increasing Z-axis going up the screen. In the diagram the vertices are arranged in 3 rows of 6 vertices (a 6x3 mesh if you like). In our application each mesh will be similar to this but will be arranged as 17 rows of 17 vertices.



As we read in each row of the image, the vertices are arranged in rows stretching out from the origin along both the positive X- and Z-axes. The vertex buffer really is as simple as that. Each mesh vertex buffer will be a 17x17 pool of vertices arranged in rows.

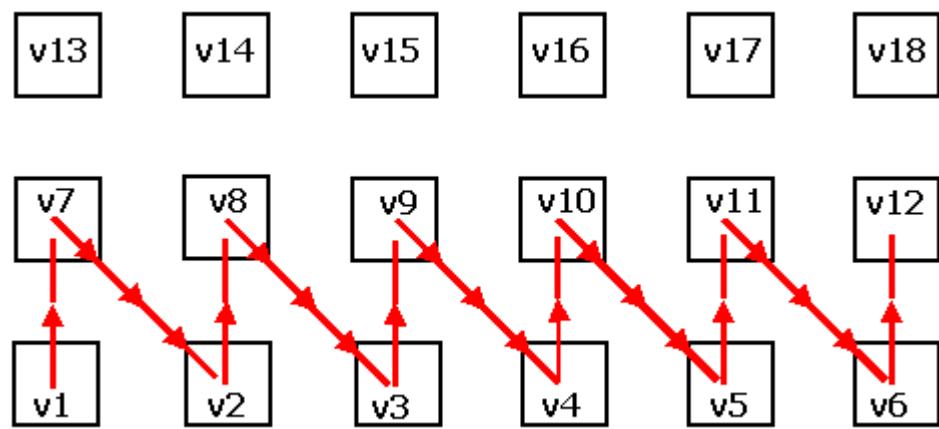
The index buffer is going to be a little more complicated. We want each group of four vertices to form a quad (2 triangles). In the above example, the middle row of vertices will be reused in both the first and second row of quads. This is where indices pay off. Without them, each row of quads would need its own duplicate vertices and that would significantly increase the terrain vertex count.

The following image shows one way to connect the vertices into quads to create a piece of terrain. In the following diagram, each quad has its two triangles colored differently so that we can better see the triangle count and arrangement:



You can see that vertex v8 is used in four separate quads (and specifically in six triangles). Because we are using indices we do not have to duplicate this vertex six times; we simply have to make sure that each triangle in the index buffer that uses it has its index.

We will now need to order the indices in the index buffer so that we create an indexed strip for rendering. Given what we already know about strips, it is easy to see how the first row of indices could be ordered. This is shown in the following image. The run of indices will start at the vertex v1 and move right along the bottom row.



Recall that in the case of a triangle strip primitive type, the first three indices define the first triangle and every additional index will define a new triangle. The last two indices from the previous triangle are used along with the new index to define the next triangle.

**Vertex Buffer = v1 , v2 , v3 , v4 , v5 , v6 , v7 , v8 , v9 , v10 , v11 , v12**

Since indices are zero based, v1 = index[0] and v7 = index[6] and so on.

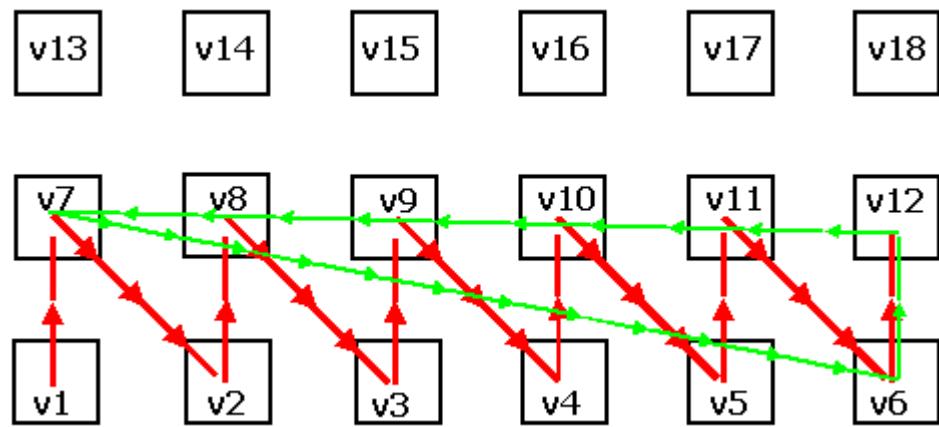
**Index Buffer = 0 , 6 , 1 , 7 , 2 , 8 , 3 , 9 , 4 , 10 , 5 , 12**

So the first three indices will define the triangle (v1, v7, v2) and the fourth index will create triangle (v7, v2, v8). The fifth index will define triangle (v2, v8, v3) and the sixth index will define triangle (v8, v3, v9). These six indices have defined two quads (four triangles) of our terrain. The pattern continues for the rest of the row. The pattern here is that for each pair of rows, we add horizontally matched pairs (a vertex from each row). The bottom row of quads in the above diagram could have the indices built in code like so:

```
for ( int a=0; a< NumberOfVerticesInRow; a++ )
{
    AddIndexToIndexBuffer ( a );
    AddIndexToIndexBuffer ( a + NumberOfVerticesInRow );
}
```

At the end of this loop the first row of quads would be complete. In the above code, **a** is the index of the vertex in the bottom row in the image and **a+NumberOfVerticesInRow** is the index of the vertex in the next row.

Of course we know that when we pass the array of indices as a strip, each triangle is supposed to be connected. This means that we cannot just get to the end of the current row and start the next row or the result would be a large triangle stretching right across the terrain:



The final three indices (10, 5, and 11) in the first row describe triangle (v11, v6, v12). Remember that each new index added generates a new triangle using the last two indices from the previous triangle. So adding the index of vertex v7 (index 6) would do the following:

Indices before vertex v7 has its index is added

**10 , 5 , 11**

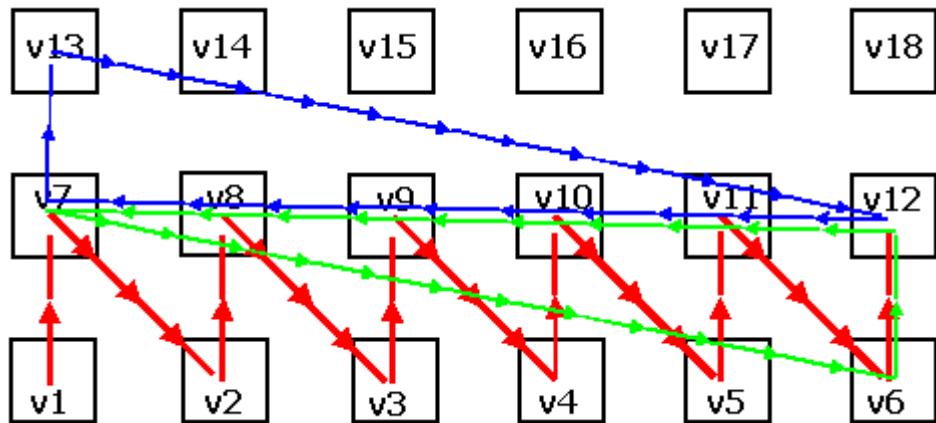
Indices after v7 has its index is added

**5 , 11 , 6**

This result is the unwanted triangle stretching across the terrain. Things get worse when you note that we add two vertices from two rows at a time (the quad top and quad bottom vertices). So, if we then added the index to vertex v13, another unwanted triangle would be formed (the blue triangle in the image below).

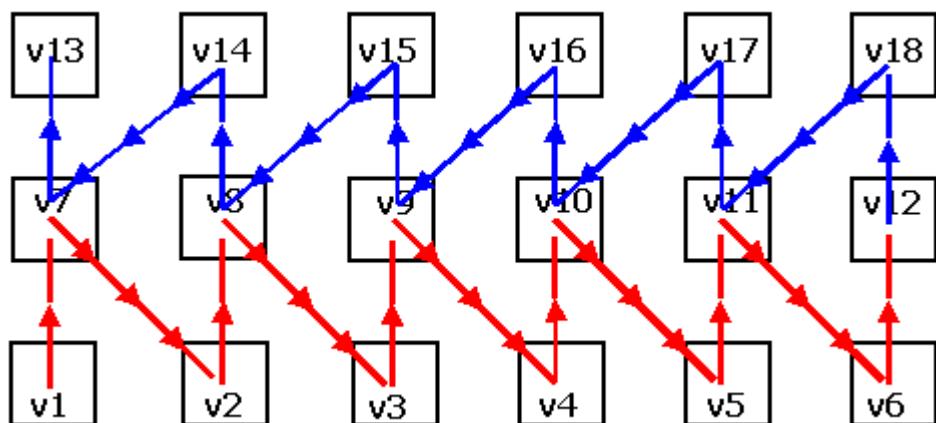
Indices after vertex v13 has its index added:

**11 , 6 , 12**



To solve these problems we will start the second row from the opposite side so that the terrain strip is indexed using a snaking pattern. The first row has its triangles indexed from left to right, then the next row has its triangles index from right to left, then left to right, and so on.

There are a few items to consider. First, we still need a way to move up to the next row without a triangle being rendered. Second, we recall that when we use strips, the device expects every odd triangle in the strip to have a counter clockwise winding order and every even triangle to have a clockwise winding order. If at any time an even triangle has a counter clockwise winding or if an odd triangle has a clockwise winding order, the device interprets this as a back facing triangle and culls it. So let us say that the last triangle in the first row was represented by indices to v11, v6 and v12. This is an odd triangle and is counter clockwise and therefore it is interpreted as a valid triangle facing the camera. We will want to start the next row where the first triangle would be constructed from indices to vertices v12, v18, and v11.



While this looks like it should solve the problem, it actually does not. Let us examine why:



That last triangle in the first row has indices to v11, v6, and v12. The next triangle we need is at the end of the second row made from indices to vertices v12, 18, and v11. But this is impossible because we add one index at a time and each new index creates an entirely new triangle.

So at the end of the first row, we have indices to vertices:

(v11, v6, v12)

Since v12 is the ideal starting index of our first row two triangle, we might try to add an index to v18 next so that the index buffer looks like this:

(v6, v12, v18)

If you look at the diagram you will see that we have just created another triangle that we certainly do not want to render. So we need a way to move from triangle (v11, v6, v12) to (v12, v18, v11) without drawing intermediate triangles.

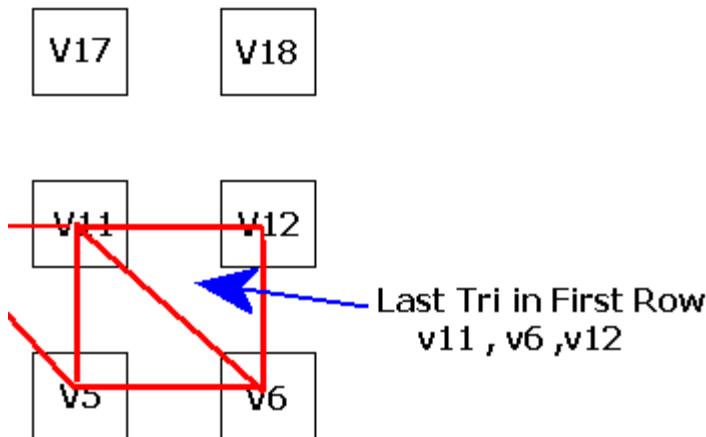
### Degenerate Triangles

A degenerate triangle is a triangle that has no volume. It is invalid for rendering and is quickly rejected by the device. They represent the solution to the problems discussed in the last section. We will use degenerate triangles to move from one row to the next without having to draw inappropriate triangles. We do this by inserting one index in such a way that it creates three degenerate triangles. After that our index buffer will be in the right order to start ordering the next row.

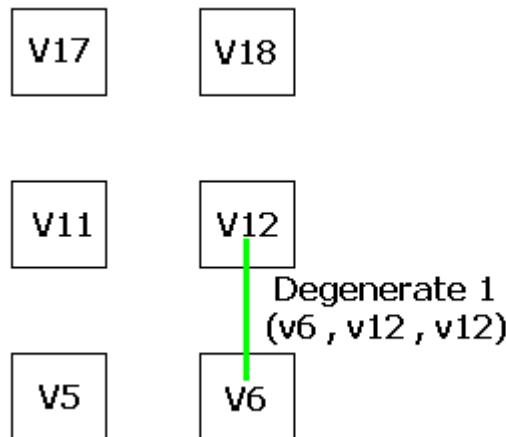
A classic example of a degenerate triangle is one where all of the indices reference the same vertex (or a vertex in the same position). The triangle in this case would be infinitely small and would be rejected by the pipeline. Another example would be when a triangle has two vertices that are the same. This means that there are only two unique vertex positions forming the triangle (essentially describing an infinitely thin line). Although we have primitive types that we can use to render lines, when we describe a triangle as a line in this way, it is rejected because it has no volume.

We will use both of these types of degenerate triangles to aid us in moving from one row to the next. Without degenerate triangles we would not be able to render the submesh as a single strip using a single call to DrawIndexedPrimitive. Degenerate triangles like this are quickly rejected so they carry very little performance penalty, if any. We will insert an extra index at the start of each new row (except the first row), which will actually cause three degenerate triangles to be created. Let us have a look how this works.

The following diagram shows the final quad of the first row. The row of quads is indexed left to right by adding a vertex from each row. For example, we add indices for v4 and v10, then v5 and v11, followed by indices for v6 and v12. The final valid triangle on that row is made from indices using vertices v11, v6, and v12.

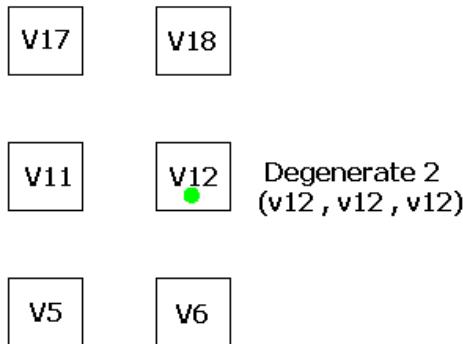


At this point we have reached the end of the row and can move up and start adding the pairs of vertices for the next row of quads (consisting of vertex pairs in rows 2 and 3). For example, v12 and v18 followed by v11 and v17, and so on. However, before we start the second row, we add the first index twice. In our example, this means we add an extra index to vertex v12 before we start adding the indices for the second row. Remember that the index list almost works like a FIFO buffer. The last three indices added are used to render the current triangle. This changes the index buffer so that a new triangle is constructed as follows.

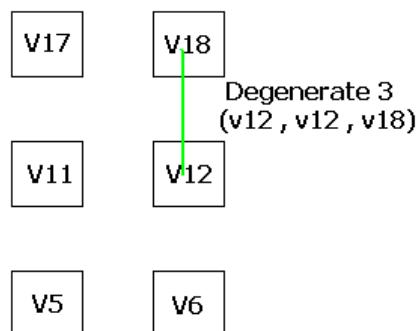


Because the last triangle was (v11, v6, v12), adding an extra v12 at the start of the next row creates a triangle where two of its indices reference the same vertex. This is rejected by the pipeline and it is not rendered.

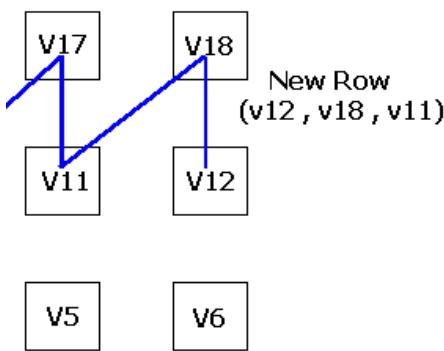
Now we can start constructing the second row. First we add the first index of the first vertex (v12). We now have a situation where the last three indices in the index buffer are now all the same. This creates another degenerate triangle as shown below.



It is now time to add the index to the second vertex from quad row two, which will be the top of the quad in row 3 (v18). This creates another degenerate triangle, since the last three indices now look like this:



At this point the whole thing has sorted itself out and we can carry on adding the vertices for the row as normal. Next we add an index to vertex v11 and we now have our first proper triangle for row 2. The last three vertices in the index buffer are now v12, v18, and v11. Adding an index to v17 creates the second triangle (v18, v11, v17) and so on until we reach the end of the row.



Take some time to reread this section before continuing if you are still not sure how this works. Try getting some paper and a pencil and sketch it out for yourself.



In terms of writing code, this is all very simple. For every row but the first, we add its first vertex index twice instead of just once. This will generate the three degenerate triangles and allow us to shuffle the indices in such a way that we can move to the next row without error.

### Conclusion

We have just studied some of the most important aspects of 3D graphics programming with DirectX. We learned how to render using vertex buffers as well as indexed primitives. We learned about efficiently creating device resources and proper use of memory pools. We also learned how to index and render rows of quads as a single triangle strip using degenerate triangles. And we learned how to efficiently lock and fill dynamic buffers.

There were a few functions that we did not cover in our second demo relating to the way the camera works. These functions will be examined in great detail in the next chapter.

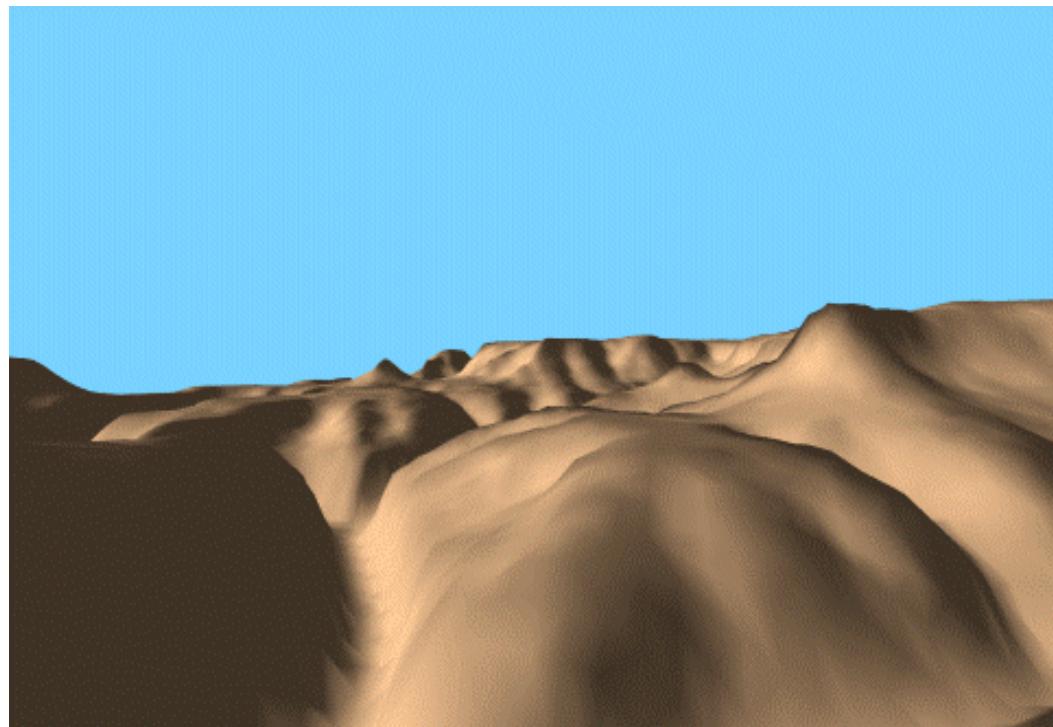
### Further Reading:

You can find a wealth of information on vertex buffers and index buffers and how to use them efficiently at the nVidia website [www.nvidia.com](http://www.nvidia.com).

# **Workbook Chapter Three:**

## **Vertex and Index Buffers**

---



© 2003, eInstitute, Inc.

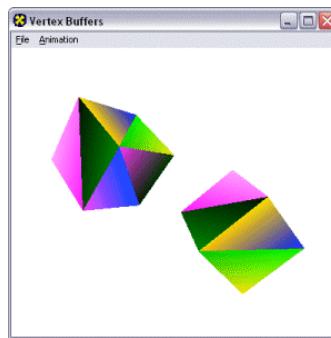
**You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.**

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 our next lesson (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,
                                              D3DPPOOL_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 --  $6 \times 2 \times 3 = 36$ ). 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

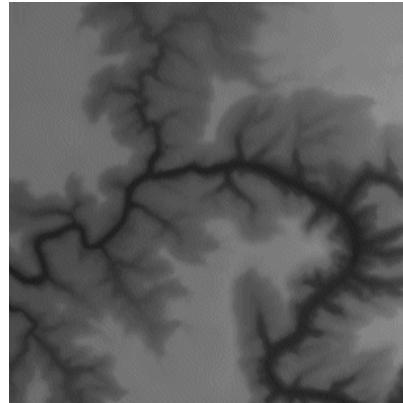
In this project we will take a step beyond our low polygon cubes and create a series of meshes totaling 66,049 vertices in all. Do not fear, we are not going to code all of these vertices by hand. Instead we will be using a 2D image to help generate the mesh. This image will be 257x257 pixels and each 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

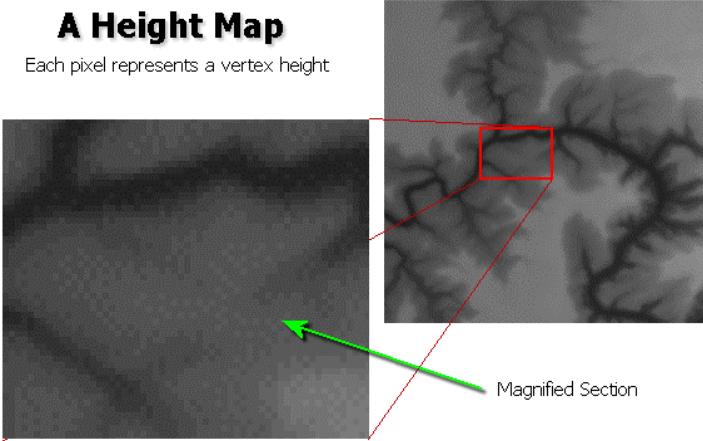
Lab Project 3.1 includes a folder called ‘*Data*’ which contains the image that will be used for the height map. The image is in .RAW format. This format is supported by most popular paint and image editing packages. It is a grayscale image, using 8 bits per pixel. In this demo we will use the value 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

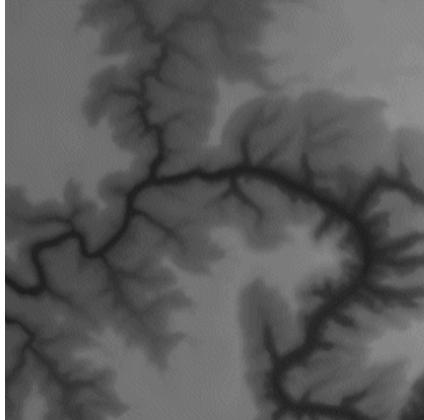
D3DXVECTOR3 ScaleVector (2.0f, 4.0f, 2.0);

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.

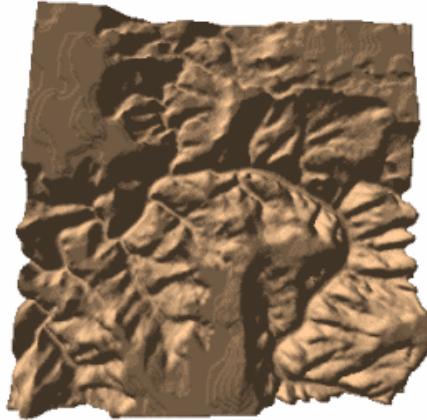
Image Pixel Position	Pixel Color (GreyScale)	Generated World Space Vertex
X=5 : Y=15	100 = RGB ( 100 , 100 , 100 )	X:10 Y:400 Z:30
X=0 : Y=100	64 = RGB ( 64 , 64 , 64 )	X:0 Y:256 Z:200
X=134 : Y=200	127 = RGB ( 127 , 127 , 127 )	X:268 Y:508 Z:400

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:

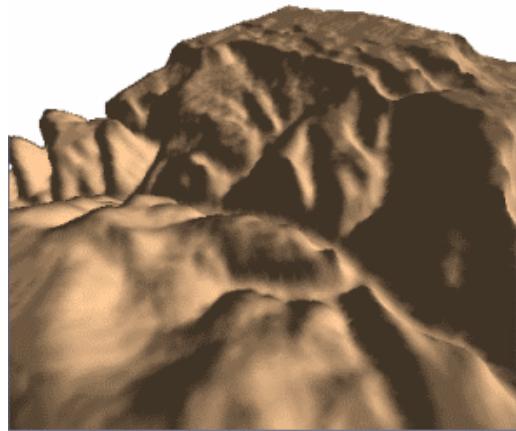
**The 2D Image Map**



**The 3D Terrain**



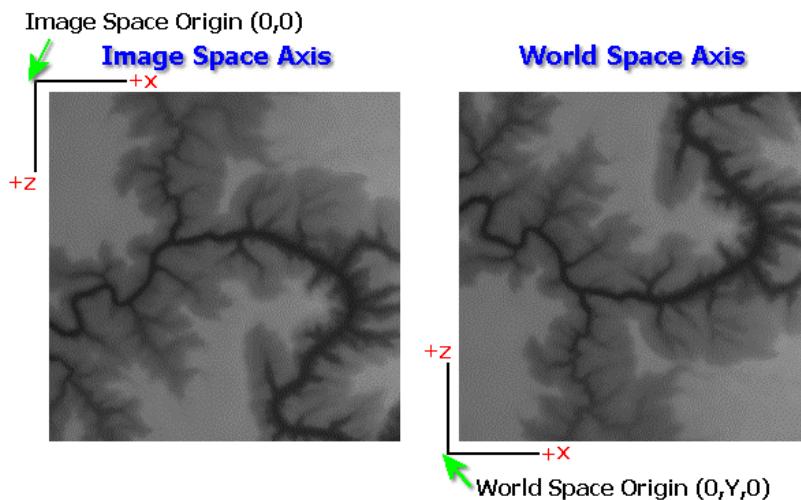
When the camera is placed onto the terrain you can see that even an un-textured terrain can look quite impressive (certainly better than two cubes).



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: 1<sup>st</sup> person, 3<sup>rd</sup> 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 tempo      rary 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;
    }

    // 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 CMeshe s 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 3<sup>rd</sup> 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 3<sup>rd</sup> 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 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 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.

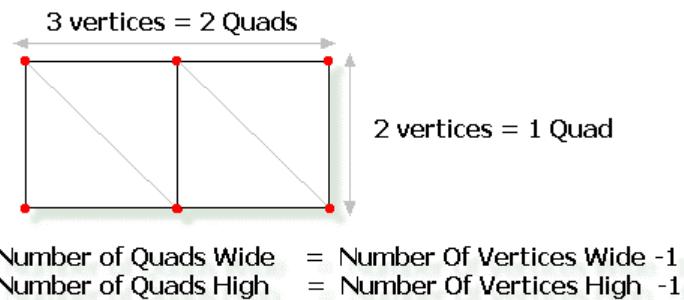
For the IDirect3DDevice9::Clear method we have passed a light blue color for the frame buffer. This provides a simple colored background for our sky.

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 3<sup>rd</sup> 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(LPCSTR 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 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, ..... }**

```

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:

$$\text{AddMesh}((256/16) * (256/16)) = \text{AddMesh}(16 * 16) = 256 \text{ meshes needed}$$

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:

$$\begin{aligned} \text{BlocksWide} &= (256)/16 = 16 \\ \text{BlocksHigh} &= (256)/16 = 16 \end{aligned}$$

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
```

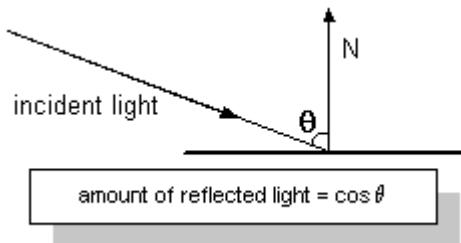
We loop through the 17 rows and 17 columns of the current mesh and use the StartX and StartY variables to create the vertex values relative to the mesh to which they belong. For example, if we are

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.

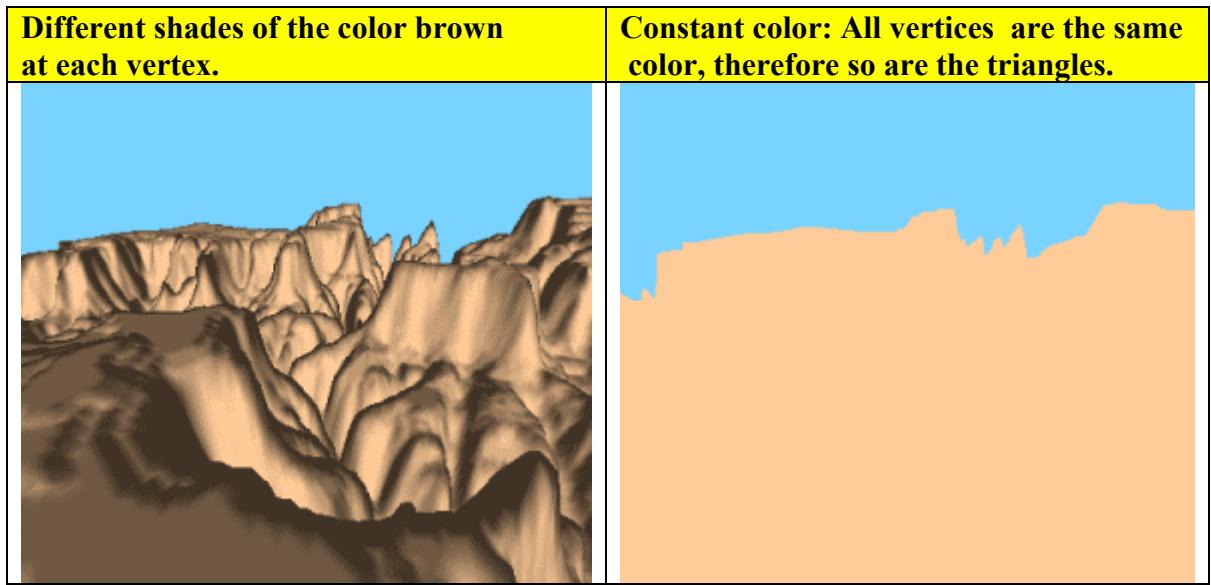
$$P_{color} = M_{diff\_refl} \ I_{point}$$

Combining the equations we get:

$$P_{color} = M_{diff\_refl} \ I_{light} (\mathbf{N} \cdot \mathbf{L})$$

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  $\mathbf{N}$  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.



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 neighbouring 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 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:

$$3 + (7 \times 10) = 74$$

So array element 73 in the height map array would be the height for pixel (3, 10) in the image.

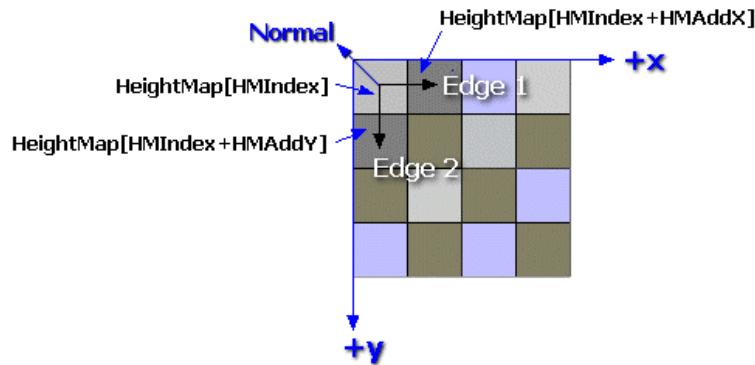
When we discussed face normals in Chapter 1, we realized that when we have three vertices that define a triangle, we can calculate the direction that triangle is facing by creating two vectors using two of its edges and then performing a cross product operation on them. The result is a vector that is 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+HMAAddX`) and the pixel below (`HMIndex+HMAAddY`).

### Image Space Normal Calculation



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 `HMAAddX` and `HMAAddY` variables as shown below.

```
// Calculate the number of pixels to add in either direction to
// obtain the best neighbouring heightmap pixel.
if ( x < (m_nHeightMapWidth - 1) )
    HMAAddX = 1;
else
    HMAAddX = -1;

if ( z < (m_nHeightMapHeight - 1) )
    HMAAddZ = m_nHeightMapWidth;
else
    HMAAddZ = -(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 + HMAAddX] * m_vecScale.y;
y3 = (float)m_pHeightMap[HMIndex + HMAAddZ] * m_vecScale.y;
```

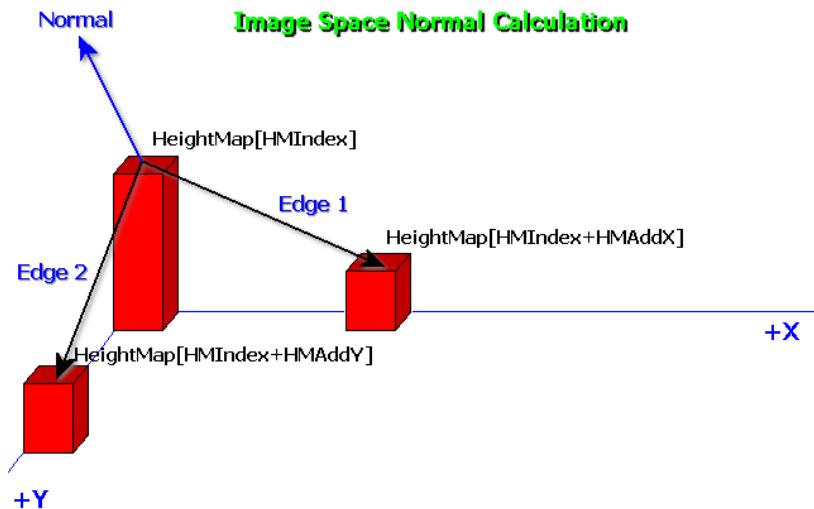
The above code samples the three L-Shaped height values from the height map and scales them by the Y component of our scale vector so that we can create two edge vectors with the correct scale of our terrain.

```
// 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:

$$(m\_vecScale.x, 40-20 * m\_vecScale.y, m\_vecScale.z) = \text{Edge 1} = (8, 40, 8)$$

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.

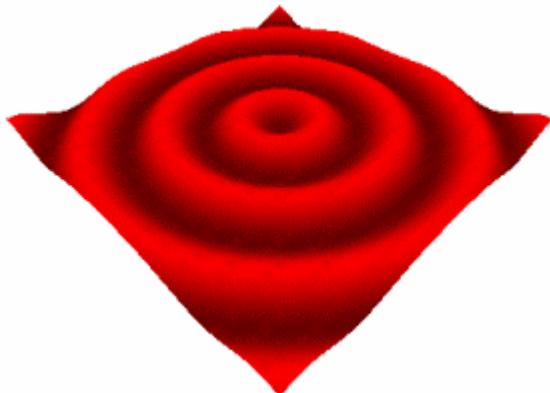
Imagine for example if we had a function that returns only the visible polygons from the current position of the camera. The call might look something like this:

```
VisibilitySystem->GetVisibleTriangles(&mtxView, pVertexBuffer);
```

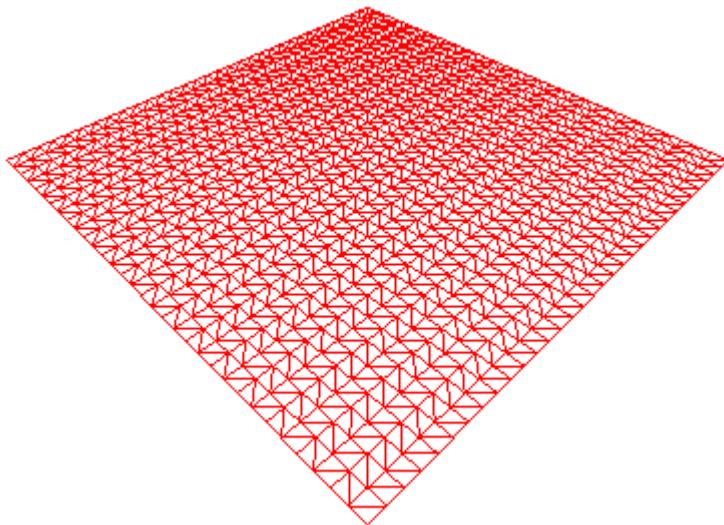
If the game world was made up of hundreds of thousands of polygons, we might not want to store them all in a video memory vertex buffer. This would take up a good deal of space that may be better reserved for textures and other important resources. We could store the level in a system memory vertex buffer 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  $33 \times 33$  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  $33 \times 33$  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.

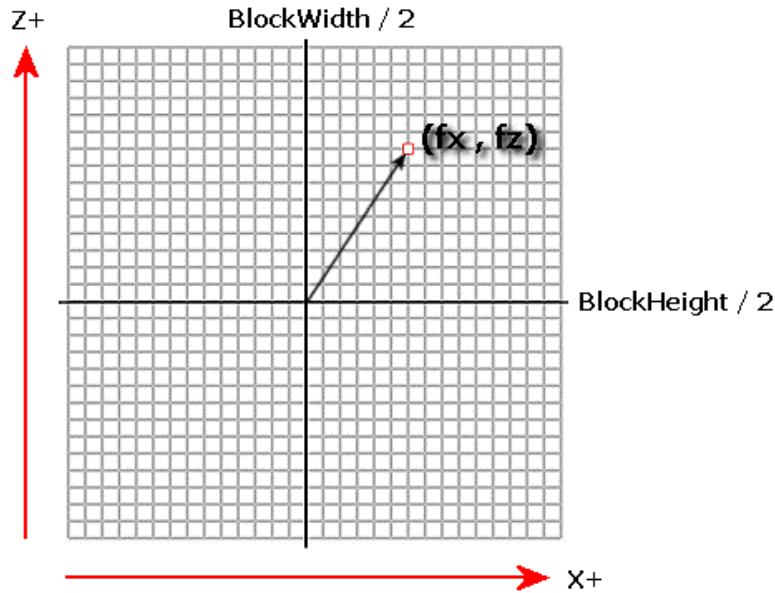
```
// Lock the vertex buffer
hRet = m_pVertexBuffer->Lock( 0, sizeof(CVertex) * (BlockWidth * BlockHeight),
                               (void**)&pVertex, D3DLOCK_DISCARD );
```

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).

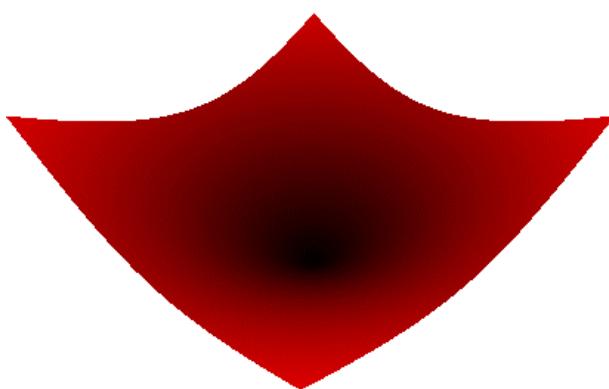


The next thing we do is calculate the length of this vector:

```
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):

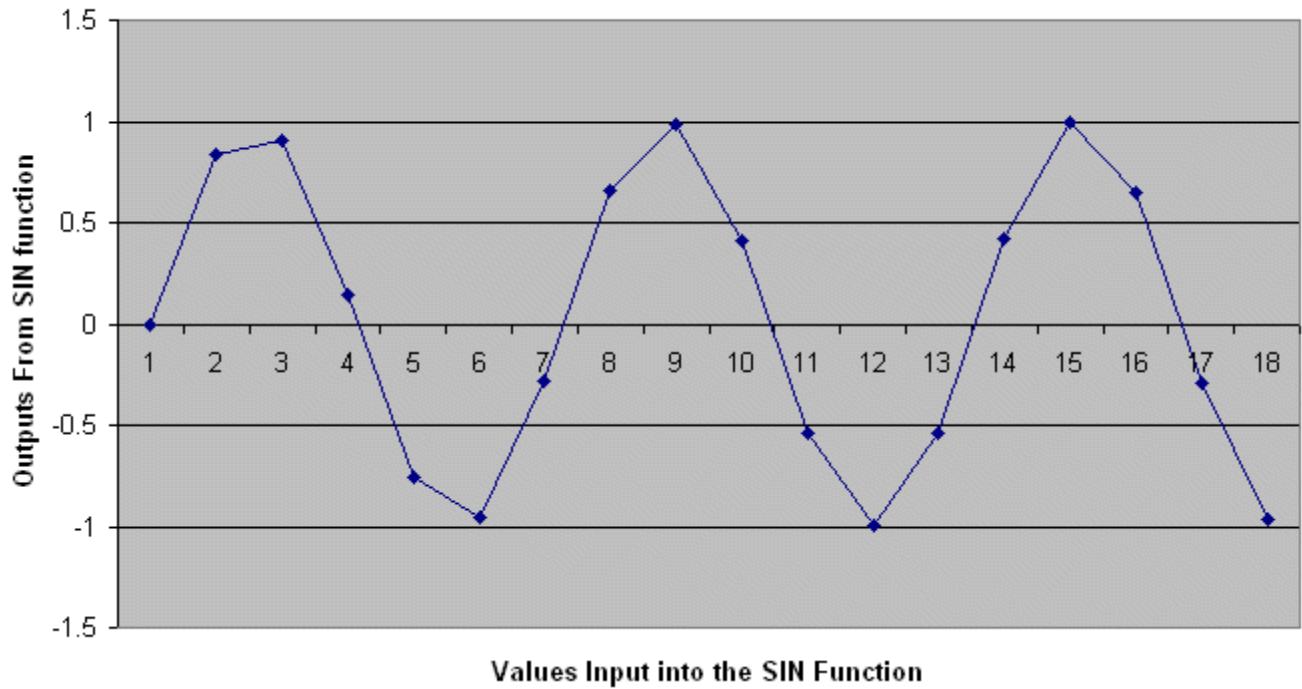


**Vertex Height = Distance From Center**

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  $0 - 18$ .

## The SIN Function



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( );
}

```

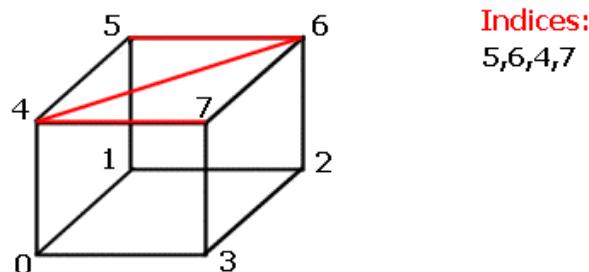
## Questions and Exercises

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

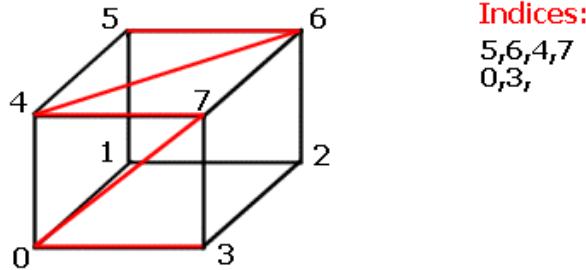
In this lesson, we learned how degenerate triangles can be used to create a continuous triangle strip even when the triangles that need to be rendered do not form a consecutive line of primitives. We used degenerate triangles to move from the end of one row up to the beginning of another row in such 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 3<sup>rd</sup> 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.

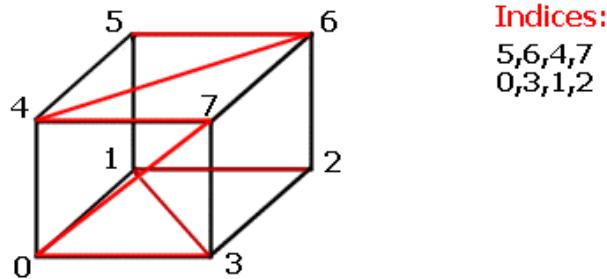


Remember that for indexed tri-strip, after the first two indices in the index array, every additional index from that point on creates a new primitive using the new index and the last two indices from the previous triangle. Also remember that unlike other primitive types, the driver expects every even 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.

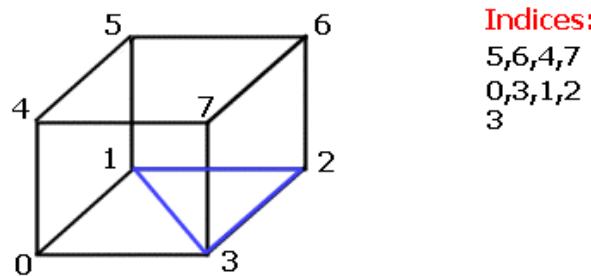
Next, we added two more indices and created the next two triangles. These triangles created the front face of the cube. The first triangle has a clockwise winding order when viewed from the front and the second face is counter-clockwise as expected. If the camera were positioned in front of this cube 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).



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.

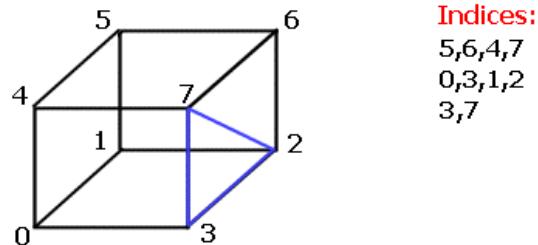


At this point we have the top, front, and bottom faces of the cube stored as a continuous strip of triangles. This is where things get tricky. We could carry on up the back side of the cube in the same way but then there would be no way to render the left and right face without breaking our strip or 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.

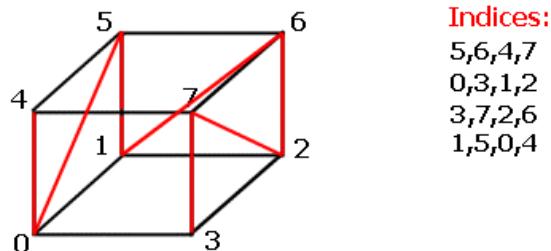


Now when we add the next index (index 7) this creates a triangle on the right side of the cube (2,3,7). You might think at first that it should be visible since it is obviously clockwise, but remember that we

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.



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:



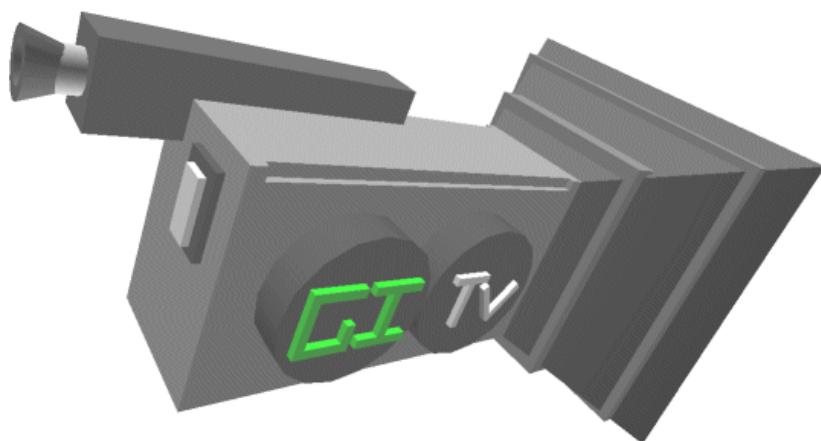
To sum this method up, we represent the top, front, and bottom faces as a normal strip. We then insert an additional index to vertex 3, which creates two incorrectly ordered triangles. This gets us into the position where we can then represent the left, back, and right faces in the same way.

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`.



# Chapter Four: Camera Systems

---



© 2003, eInstitute, Inc.

You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.



## Table of Contents

Matrix Inversion and Local Space .....	3
The View Matrix.....	4
Vectors Revisited.....	7
Matrices and Planes .....	8
The View Space Planes.....	13
The View Space Transformation (Under the Microscope).....	16
The Inverse Translation Vector .....	18
Viewports.....	22
The Viewport Matrix .....	22
Viewport Aspect Ratios .....	24
Camera Manipulation I .....	25
Camera Manipulation II .....	28
Vector Regeneration .....	33
First Person Cameras .....	36
Third Person Cameras.....	37
Conclusion .....	64



## Introduction

In Lab Project 3.2 we concentrated on the rendering code for a terrain demo. But that application also allowed the player to maneuver around the terrain in one of three camera modes: first person, third person, or spacecraft. This allowed us to pitch, roll, and yaw the camera as well as strafe and lean it from side to side. We included a limited gravity system that forced the camera fall to the ground when it found there was no ground underneath it, and a simple friction model that allowed for smooth movement and direction changes over the terrain. In this chapter we will discuss that camera system, as well as how to create your own camera management system. By the time we are finished you should have a thorough understanding of how to work with the view matrix at a low level and you will be able to create almost any camera system you need for your games.

### Matrix Inversion and Local Space

In Chapter 1 we learned to think of a camera in terms of an inverse transformation that repositions scene geometry in such a way that the relationship with the origin of the world coordinate system reflects the relationship the geometry would have with the local position and orientation of the camera. Repositioning the geometry in this way means that when we render the scene, we are essentially rendering it from the world origin as if we were looking through the lens of a virtual camera positioned there. To accomplish this, we need to apply the opposite rotations and translations we applied to our camera to every vertex in our world.

This is intuitive when we think of what is occurring. We know that we can take any vertex  $\mathbf{P}$  from model space and produce a new vertex  $\mathbf{P}'$  in world space by applying a series of transformations using matrix multiplication. The relationship between these two vertices is represented as:

$$\mathbf{P}' = \mathbf{P} * \mathbf{M}_{\text{world}}$$

Note that the algebraic inverse of this equation describes the reverse relationship. To solve for  $\mathbf{P}$ :

$$(\mathbf{1} / \mathbf{M}_{\text{world}}) \mathbf{P}' = \mathbf{P} * (\mathbf{1} / \mathbf{M}_{\text{world}}) \mathbf{M}_{\text{world}}$$

$$\mathbf{P} = \mathbf{M}_{\text{world}}^{-1} \mathbf{P}'$$

$\mathbf{M}_{\text{world}}^{-1}$  is the inverse of matrix  $\mathbf{M}_{\text{world}}$ . When we multiply the world space point  $\mathbf{P}'$  by this matrix, we get back the original local space point  $\mathbf{P}$  as expected. So we can say that  $\mathbf{M}_{\text{world}}^{-1}$  undoes the effect that  $\mathbf{M}_{\text{world}}$  had on  $\mathbf{P}$ . Again this makes sense since we used one to cancel out the other in the equation above so that  $\mathbf{P}$  was left alone on one side of the equation.

More generally, if matrix  $M$  holds a series of equations that transform points from coordinate space A into coordinate space B, then its inverse  $M^{-1}$  will hold equations that reverse the relationship -- taking points from space B into space A. If space A is the local coordinate space of entity X, then any points that exist in space B can enter into local space A simply by multiplying them by X's inverse matrix.



This is the fundamental idea behind any camera system. When we render a scene, we wish to do it with respect to the camera through which the scene is viewed. The goal then is to transform every vertex in the world into the local space of the camera. If we build a world matrix for the camera based on user input, that matrix tells us where the camera is in the world and how it is oriented with respect to the world axes. To get some other object in the world into the local space of the camera for the purposes of rendering, all we need to do is multiply its world space vertices by the inverse of that camera's world matrix (which we call the view matrix).

An alternative way of thinking about it is that we are actually undoing the effect of moving the camera around and bringing it back to the world origin such that it looks down +Z (just as it does in its own local space given a left-handed coordinate system). As expected then, any matrix multiplied by its inverse returns the identity matrix:

$$\mathbf{I} = \mathbf{M} * \mathbf{M}^{-1}$$

The rows and columns of an identity matrix perfectly describe the primary 3D coordinate system. Thus, it is as though we never moved or rotated the camera at all.

Creating a virtual camera is usually done by writing a class that exposes methods such as Camera::MoveForward and Camera::PitchUp and Camera::Strafe, etc. The camera class has the job of maintaining the view matrix (the camera local space matrix), and rebuilding it to comply with calls to its methods. This class need not only be a view matrix manager. It is often useful to let it manage the projection matrix as well. This way we can expose functions to change the field of view and set the near and far clip planes.

Before we start writing any code, let us first examine in more detail some of the view matrix properties introduced in Chapter 1. We want to understand exactly why inverse matrices look and work the way they do. In particular, we want to see why storing the right, up, and look vectors of the virtual camera in the columns of the view matrix -- rather than the rows as we do in a world matrix -- transforms vertices from world space to view space. We will also examine why the fourth row of the view matrix has to be calculated using three dot products instead of simply negating the world space position vectors of the camera.

### The View Matrix

We recall from Chapter 1 that a standard world matrix contains the orientations of an object's local coordinate system as well as the current position of that system origin in the world coordinate space:

#### World Matrix

Right Vector.x	Right Vector.y	Right Vector.z	0
Up Vector.x	Up Vector.y	Up Vector.z	0
Look Vector.x	Look Vector.y	Look Vector.z	0
Position.x	Position.y	Position.z	1

The following table shows that the view matrix contains three vectors describing the inverted local coordinate system of the camera and a vector in the fourth row which contains an inverse translation based on the camera position. This translation will move vertices in such a way that their resulting positions will share a relationship with the world origin that previously reflected their relationship with the camera (world) position.

### View Matrix

Right Vector.x	Up Vector.x	Look Vector.x	0
Right Vector.y	Up Vector.y	Look Vector.y	0
Right Vector.z	Up Vector.z	Look Vector.z	0
- (Position • RightVector)	- (Position • UpVector)	- (Position • LookVector)	1

The Right vector is stored in the first column of the matrix and describes the orientation of the camera local space X axis. The second column contains the camera Up vector which describes the orientation of the camera local space Y axis. Finally, the Look vector describes the orientation of the camera local space Z axis (Fig 4.1).

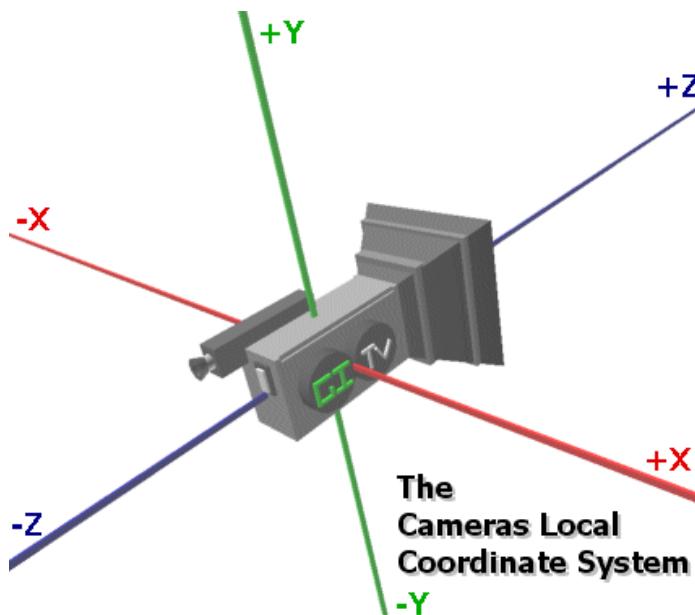
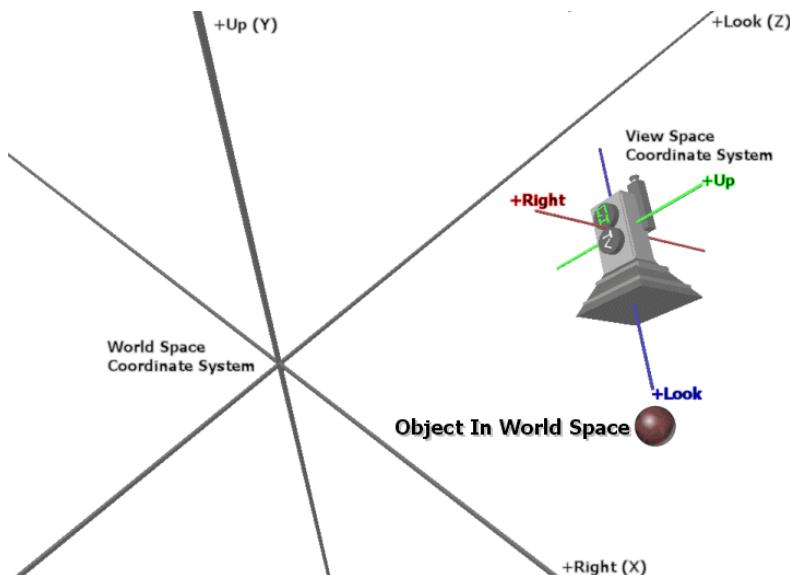


Figure 4.1

The vectors in the view matrix describe the camera local coordinate system axes along with relative positional information that we will discuss momentarily. If we take a view matrix and invert it, we would get back a world matrix describing the cameras location and orientation in the world (Fig 4.2).



**Figure 4.2**

Fig 4.2 shows how the camera object might be drawn as a mesh using the inverse of the view matrix. Remember that the view matrix is already inverted, so inverting again it results in a standard world matrix. If we wanted to draw the camera as a mesh object, this is the matrix we would use.

We know that to draw the sphere, we want its coordinates to be relative to the camera local system. As discussed, to get an object A into the local space of another object B, we need only multiply all of A's vertices by the inverse of B's world matrix. Since B in this case is our camera, we need only invert its world matrix and we are all set. This inversion produces what we commonly refer to as the view matrix. Fig 4.3 shows the sphere object after it has been transformed into view space. Notice that the camera is at the system origin and that the sphere is still directly in front of the camera, as it was in world space. The relationship is perfectly maintained when the sphere moves into camera local space.

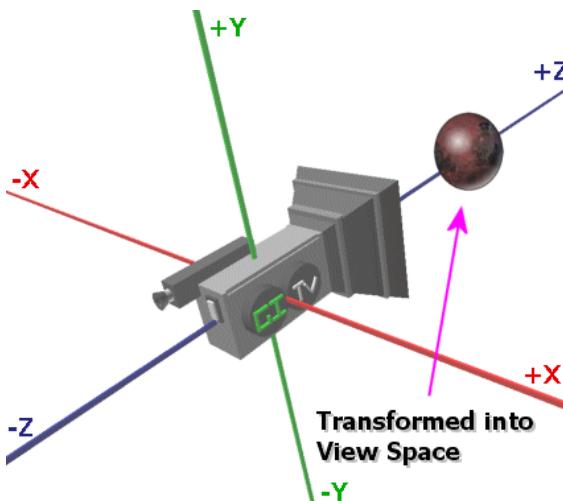


Figure 4.3

## Vectors Revisited

In order to understand why multiplying a vector with a matrix transforms that vector from one virtual space to another, we revisit the subject of vectors and matrices and discuss another way of thinking about them -- which you may or may not already be doing at this point. For the purposes of this discussion when we refer to a vector, we are talking about a position vector, although this concept applies more generally.

**Note:** To be clear up front, we are going to take a very informal approach to the mathematics in this chapter -- as we have tried to do all along. This will make the concepts as reader-friendly as possible for those who are not so mathematically inclined. We hope that those of you who are schooled mathematicians forgive the liberties we take with some of the subject matter. If you require a more precise and formal understanding of vectors, vector spaces, subspaces, etc. a linear algebra course would be required.

Although we discuss many different spaces (model space, world space, and view space) we are, in a sense, ultimately dealing with a single mathematical space. In this space we can define locations using a coordinate system (left handed in our case) where the X axis runs from left to right, the Y axis runs from bottom to top and the Z axis runs from back to front. This is the same coordinate system used to characterize our data mathematically whether we are said to be in model space, view space, or world space. All of these spaces are essentially subsets of the single coordinate space, and in each, 3D vectors are used represent a location. For example, a vector of (10, 20, 30) represents a position that is offset from the system origin along the X axis a distance of 10, offset along the Y axis a distance of 20 and offset along the Z axis at a distance of 30.

A vector belongs to a particular subspace based on our selection of system origin and orientation. When we are talking about a model space vector for example, we are using the vector as a position



relative to the center of the mesh. When we transform the mesh vertices into world space, all we have really done is simply moved the vertex to a new position in the same mathematical coordinate system. Now the vertices of the mesh are not centered about the origin of the local coordinate system anymore (although they still could be) and are instead centered about some other location in the world that is assumed to be the object's world space position. In world space, the origin of the coordinate system is now assumed to be the origin of the entire world and all vectors are now defined relative to it. In a sense, the vectors have simply had their positions altered. When we apply the view space transformation to the vertices of an object to take it from world space to view space, all we have done is once again reposition the vertices in the same mathematical space such that the cameras position is assumed to be at the origin of the system. All vertex positions have been recharacterized relative to this new origin. Therefore, all these transformations are doing, however complex they may seem at first, is moving around some collection of vertices within the same mathematical 3D representation. With each transformation, the origin is assigned a new meaning and the positions reflect new distance values relative to that origin. This is a very important point.

Up until now, we have thought of vectors in one of two ways. We have thought of a vector as a set of offsets describing a position that is some distance away from the origin of the mathematical space along the X, Y, and Z axes by the amounts described in each vector component. We have also thought of a vector as describing a direction and magnitude from the origin of that mathematical space. That is, travelling in the direction of the vector from the origin of the coordinate system for the length of the vector will bring us to that same location in the 3D world. Whether we think of a vector as a collection of offsets or as a direction and a magnitude they both still describe the same location in 3D space.

There is yet another way that we can think of vectors which is especially useful when trying to understand transformations. Hopefully this will allow us to perceive transformation matrices in a much more intuitive way.

## Matrices and Planes

For the time being, we will forget all about the concept of world, view, and model space and simply think of a more general mathematical space. We will call it *3-space* because it has three dimensions. We will use a coordinate system in 3-space to find our way around. In this system, the x axis will run from right to left, the Y axis from bottom to top and the Z axis from back to front. These system properties have been shared by all of our relative spaces so far: model, view and world.

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{matrix}$$

We can see that the columns of the upper 3x3 portion of the matrix on the right contain unit vectors. The first column contains a vector that describes the orientation of the 3-space X axis. The second column contains a unit length vector describing the orientation of the 3-space Y axis. Finally, the third column contains a vector describing the 3-space Z axis. When we multiply a vector with a matrix, we know that we perform a dot product between the input vector and each column of the matrix. The resulting vector's X component is the result of the dot product between the input vector and the X column of the matrix. The Y component of the resulting vector is the result of performing a dot



product between the input vector and the second column of the matrix. And of course, the resulting vector's Z component is calculated by performing a dot product between the input vector and the third column of the matrix. Now, it may not be obvious at this point why multiplying the input vector with these three columns would transform it from one space to another. But let us start thinking about the unit length vectors stored in the columns of a transformation matrix in another way.

In the lecture for Chapter 1 we discussed the plane equation and how it could be used to classify a point with respect to a plane. The common form of the plane equation is:

$$Ax + By + Cz + D = 0$$

If the result is zero, the point is said to lie on the plane. Otherwise the result is some distance from the point to the plane where a positive value means the point is in front of the plane and a negative value means the point is behind the plane.

x, y, and z in this equation are the components of the point P that we are classifying. A, B, and C are the 3-space components of the plane normal. Finally, D describes the plane's distance from the origin. That is, this is the distance you would have to travel from the origin of the coordinate system, following the direction of the plane normal until you intersected the plane. When a plane passes through the system origin then D = 0. As such, only the normal will be needed to represent a plane of this type and the equation can be simplified:

$$Ax + By + Cz = 0$$

This calculation should look familiar since it is just a dot product between the vector and the plane normal.

We noted that the upper 3x3 section of the matrix above contained unit vectors. We can now think of each of these three vectors as being normals for three planes that pass through the system origin. Remembering that plane normals are always perpendicular to the plane, we can see that in the case of an identity matrix, the X column of the matrix represents a plane normal of (1, 0, 0). This describes a plane that passes through the origin of the coordinate space -- the YZ plane. The second column represents the normal of the 3-space XZ plane that passes through the origin. Finally, the third column represents a normal that describes the 3-space XY plane, once again that passes through the origin.

In Fig 4.4 we clearly see the three planes described by the 3x3 identity matrix. As these planes have zero distances (D = 0), they pass through the origin of the coordinate system.

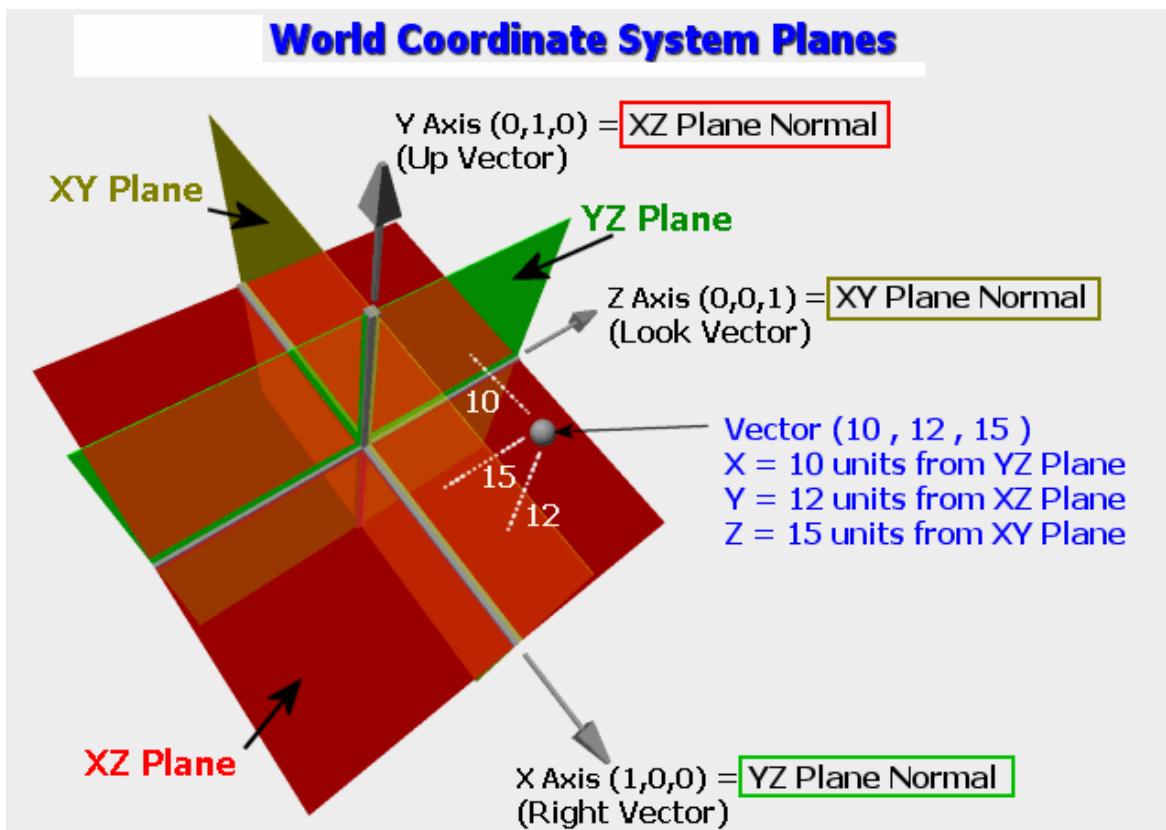


Figure 4.4

When we multiply a vector by a matrix, we can see that each dot product is simply classifying the input vector against each of these three planes. When we perform the dot product between the input vector and the X column of the matrix, we are calculating the distance from the input vector to the YZ plane. This distance becomes the X component of the resulting vector. When we multiply the input vector with the second column of the matrix, we are classifying the point against the XZ plane and the resulting distance becomes the Y component of the output vector. Finally, the dot product between the input vector and the third column calculates the distance from the input vector to the XY plane, which becomes the resulting vector's Z component. So we can think of the transformed 3D vector as being a collection of three distances that describe a location relative to the YZ, XZ and XY planes respectively.

The vector  $(10, 12, 15)$  in Fig 4.4 describes a location that is a distance of 10 units from the YZ plane along the YZ plane normal, 12 units from the XZ plane along the XZ plane normal and 15 units from the XY plane along the XY plane normal. Now we see why an identity matrix creates an output vector identical to the input vector. If we define a vector as a set of distances relative to the world aligned planes and the identity matrix contains these same world planes, we get back these same distances in our vector components.

To better understand this, let us break down the vector/matrix multiplication process so that we can see the vector being multiplied with each column individually. First, we will see an example of multiplying the input vector with the X column of the matrix. This produces a distance that is used as the X component in the resulting vector. Note that Fig 4.5 labels the first column of the matrix as being the Right vector. You are probably thinking that this is only true for an inverse matrix but that is not quite so. For example, a world matrix stores the object's right vector in the first row rather than the column of the matrix as we have seen already, but the right vector of the coordinate system is always contained in the first column. This will become clear in a moment.

Fig 4.5 shows how the X component of the resulting vector from a vector/matrix multiply is the result of classifying the input vector against the YZ plane. This distance is calculated by performing a dot product between the input vector and the YZ plane normal. Always remember that when we perform a dot product between a unit vector and a non-unit vector, we can think of the non-unit vector as the point in space and the unit vector as a normal describing a plane that passes through the origin. The dot product can be understood in terms of the plane equation in the instances when the plane distances are zero. You will find this quite a useful way of thinking about the dot product.

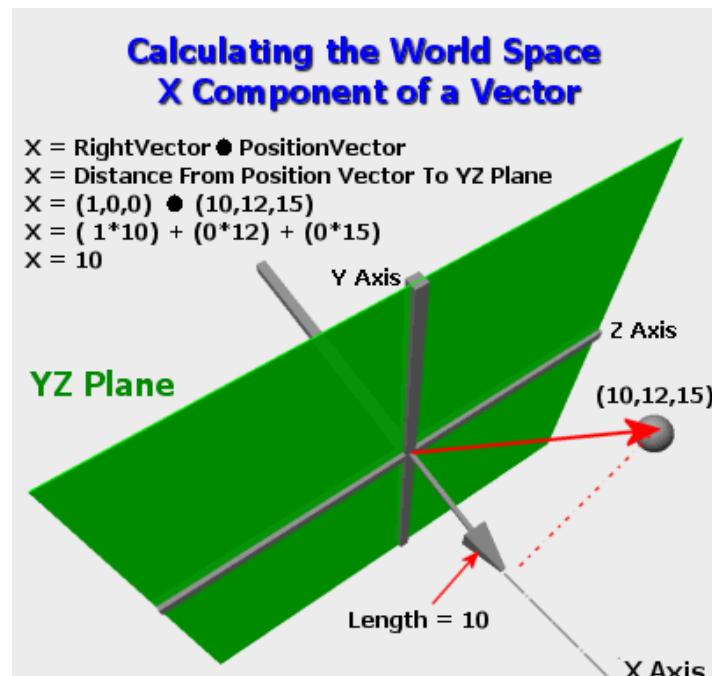


Figure 4.5

Fig 4.6 shows how the Y component of the resulting vector is calculated as the distance between the input vector and the XZ plane normal stored in the second column of the transformation matrix.

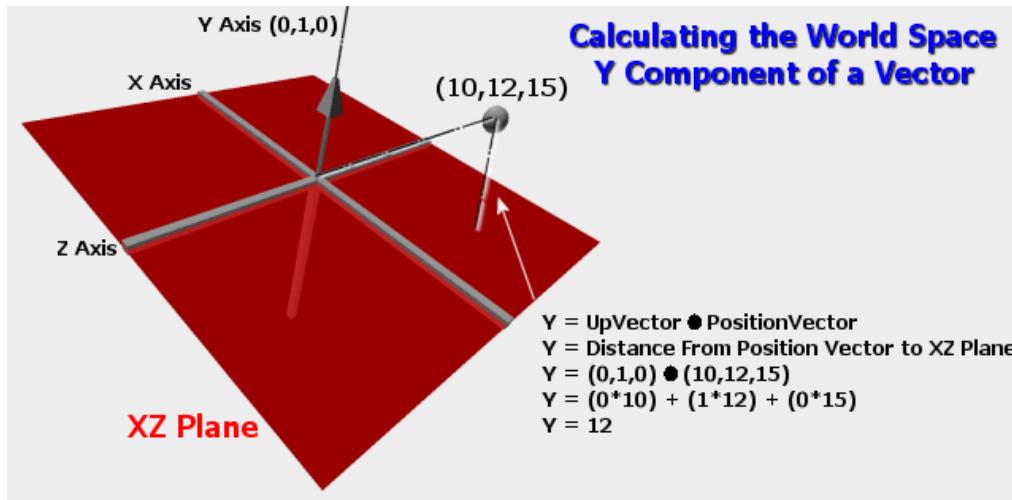


Figure 4.6

Once again, we refer to the second column of the transformation matrix as the Up vector of the coordinate system for which the input vector is going to be redefined. This does not change the fact that the second *row* of a world matrix contains an *object's* Up vector.

Finally, Fig 4.7 shows the result of calculating the Z component of the transformed vector. It is the distance from the input vector to the XY plane along the XY plane normal. Note again that the matrix in this case is an identity matrix.

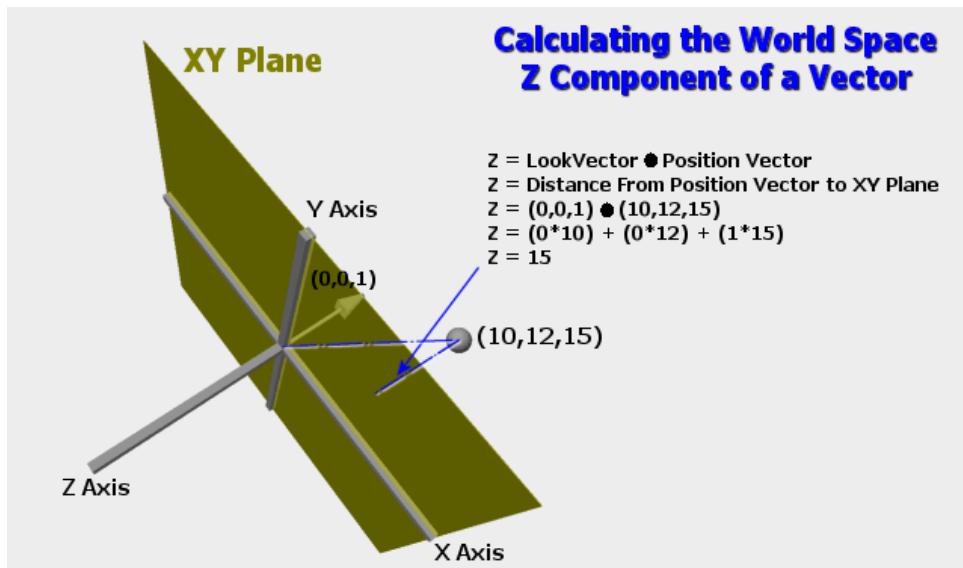


Figure 4.7



Thinking of vectors as being a set of distances and matrices as containing a set of planes that pass through the origin of a coordinate system really does allow us to visualize transformations from one space to another in a more robust way. We now know that when we multiply a vector with a matrix we are in fact classifying the vector against three planes to create a new vector. When we apply a rotation to the matrix, we are in fact rotating the Right, Up, and Look vectors in the columns of the matrix, which means, we are in fact rotating the planes themselves. When we have a rotated matrix such as this, multiplying it with the input vector -- which has its distances defined relative to the three world-aligned planes -- redefines the vector such that its distances are now relative to the rotated planes stored in the matrix. We will now go on to see exactly what this means by looking at the view space transformation. This will hopefully put all of the pieces into place.

### The View Space Planes

We now know that an identity matrix contains three planes aligned with the 3-space X, Y, and Z axes. If our camera has been rotated, then its Right, Up, and Look vectors will no longer be aligned with the world coordinate axes and the planes stored in the view matrix must have rotated also.

#### View Matrix

Right Vector.x	Up Vector.x	Look Vector.x	0
Right Vector.y	Up Vector.y	Look Vector.y	0
Right Vector.z	UpVector.z	Look Vector.z	0
- (Position • RightVector)	- (Position • UpVector)	- (Position • LookVector)	1

Looking at the upper 3x3 section of the view matrix then, we can say that the first column of the view matrix contains the normal of the camera's *local* YZ plane, the second column contains the normal of the camera's *local* XZ plane and the third column represents the normal of the camera's *local* XY plane.

Let us consider a quick example. We start with a camera that is perfectly aligned with the world X, Y, and Z axes. We now want to rotate it left around the world Y axis by an angle of 45 degrees. A positive angle will always rotate an object clockwise about the rotation axis from the perspective of looking from the positive end of the axis towards the negative end of the axis -- referred to as 'looking down the axis'. We can build a matrix that yaws the camera left by 45 degrees by creating a standard rotation matrix:

```
D3DXMatrixRotationY(&mtxViewMatrix, D3DXToRadian(-45));
```

This call produces the following matrix:



## Rotation

0.707107	0	0.707107	0
0	1	0	0
-0.707107	0	0.707107	0
0	0	0	1

Because we will wish to use this matrix as a view matrix, we do not need a matrix that will rotate an object left 45 degrees. Instead we need an inverted matrix that will rotate all of the vertices in our world right 45 degrees; this will create the appearance that our camera has rotated left. The above matrix is the camera's world matrix. If we were rendering the camera as a mesh, this matrix would rotate the camera mesh left 45 degrees – as it would any other mesh (a matrix is a matrix – it has no particular affiliation with a specific object). Inverting a matrix consists of transposing (swapping the rows and the columns) the upper 3x3 portion of the matrix and adding an equation (discussed momentarily) to calculate the fourth row of the matrix. We note that the relationship between the inverse and the transpose does not hold true in all cases and in fact, some matrices are not invertible at all. But when we are dealing with orthogonal unit vectors as we are with our linear transformation matrices, this will always work. Again, we refer you to a more serious study of linear algebra for the precise rules and properties.

For now our camera is assumed to have a position of (0,0,0) because the fourth row of our matrix is zeroed out. The inverted matrix is:

```
D3DXMatrixInverse(&mtxViewMatrix, NULL, &mtxViewMatrix);
```

## Inverse Rotation

0.707107	0	-0.707107	0
0	1	0	0
0.707107	0	0.707107	0
0	0	0	1

Inverting a matrix is not a cheap operation. Since all we are doing is negating the rotation angle, we could generate the same view matrix simply by flipping the sign of the angle passed into the function:

```
D3DXMatrixRotationY(&mtxViewMatrix, D3DXToRadian(45));
```

We now pass a positive angle instead. The resulting rotation matrix will be the same as if we had passed in the negative angle above and then inverted the result. This is actually quite an important point, because we can apply rotations to the view matrix simply by building a rotation matrix and multiplying it with the current view matrix to achieve an additive rotation. The previous discussion had just taught us, that if we wanted to pitch a mesh 45 degrees upwards we would need to create an X axis rotation matrix that rotated the mesh -45 degrees about its X axis. You are reminded again that positive rotation angle values perform a clockwise rotation from the perspective of looking towards the negative end of the rotation axis from the positive end.



```
D3DXMATRIX mtxViewMatrix , mtxRotationMatrix;
pDevice->GetTransform(D3DTS_VIEW, &mtxViewMatrix);

D3DXMatrixRotationX(&mtxRotationMatrix, D3DXToRadian(-45));
D3DXMatrixMultiply(&mtxViewMatrix, &mtxViewMatrix, &mtxRotationMatrix);

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

The above code will actually rotate the camera about its own local X axis (its right vector). Changing the order of the multiplication to:

```
D3DXMatrixMultiply(&mtxViewMatrix, &mtxRotationMatrix, &mtxViewMatrix);
```

would rotate the camera about the world's X axis and not its own. We will see why this is the case in a moment.

The above code is erroneous in that we wanted to pitch the camera up 45 degrees. Although we know that a negative angle should pitch the camera upwards, the rotation matrix has not been inverted but is being multiplied by the view matrix -- which *is* an inverse matrix. Therefore, we would actually achieve a rotation in the opposite direction and the above code would pitch the camera down 45 degrees. We can fix this by inverting the rotation matrix before we multiply it with the view matrix; this would rotate the camera in the direction we would expect and is consistent with the way a world matrix would be rotated. However, an inverse is an expensive operation and we know that if we invert a rotation matrix we get the same matrix as if we had created that matrix with a negated angle to begin with. It would seem then that in this case it would be much cheaper to build a rotation matrix with an opposite angle of rotation rather than generate it normally and then flip it. Below, we show two ways that we could rotate the camera upwards about its own X axis.

### Example 1: Rotate Camera upwards 45 degrees

```
D3DXMatrixRotationX (&mtxRotationMatrix, D3DXToRadian (-45));
D3DXMatrixInverse (&mtxRotationMatrix, NULL, &mtxRotationMatrix);
D3DXMatrixMultiply (&mtxViewMatrix, &mtxViewMatrix, &mtxRotationMatrix);
```

This example rotates the camera as we would expect in keeping with our rotation rules. A negative X axis rotation should rotate an object upwards about its X axis, but as we are dealing with the view matrix -- which is inverted -- this means we actually want to build a matrix that instead rotates the world down. Because the view matrix is inverted, we also invert the rotation matrix. This will change the rotation matrix such that it now contains a positive rotation and not a negative rotation. In other words, it will rotate vertices downwards, which is what we want. Finally we multiply it with the view matrix and we have a new view matrix that now rotates vertices down 45 degrees. This gives the illusion that the camera has been rotated up 45 degrees. Therefore, the inversion of the rotation matrix allows us to rotate the camera using the same (sign) angle of rotation as we would use for normal world objects.

### Example 2 : Rotate Camera upwards 45 degrees

```
D3DXMatrixRotationX (&mtxRotationMatrix, D3DXToRadian(45));
```



```
D3DXMatrixMultiply( &mtxViewMatrix, &mtxViewMatrix, &mtxRotationMatrix )
```

In this second example we avoid the overhead of the inversion at the cost of inconsistency when specifying rotation angles to rotate the camera. Our code now has to know that angles must be negated. A slight hack, but faster certainly.

So assuming that we have a view matrix that has been rotated to some degree, the Look, Up and Right vectors have been rotated as well. They are pointing in new directions whilst still remaining orthogonal to each other. As these vectors can also be perceived as plane normals, the planes have also been rotated in the same way. Whichever method we use to rotate the camera left 45 degrees about its Y axis, would result in the following view matrix:

### Inverse Rotation

Right Vector	Up Vector	LookVector	
0.707107	0	-0.707107	0
0	1	0	0
0.707107	0	0.707107	0
0	0	0	1

This matrix describes the camera as being at the origin of world space looking halfway between the negative X axis and the positive Z axis. Let us now examine the transformation of a vector into view space using this view matrix.

#### The View Space Transformation (Under the Microscope)

Fig 4.8 depicts a two-dimensional scene viewed top-down. It contains a virtual camera described by the example view matrix above and a world space vector (-2, 0, 10). This specific top-down view was chosen because the camera has an Up vector that is perfectly aligned with the world Y axis (0,1,0) and our world space vector Y component will not be altered by the transformation. Therefore, we can simplify this transformation in the diagram and think of it in terms of only the Right vector and the Look vector of the view matrix.

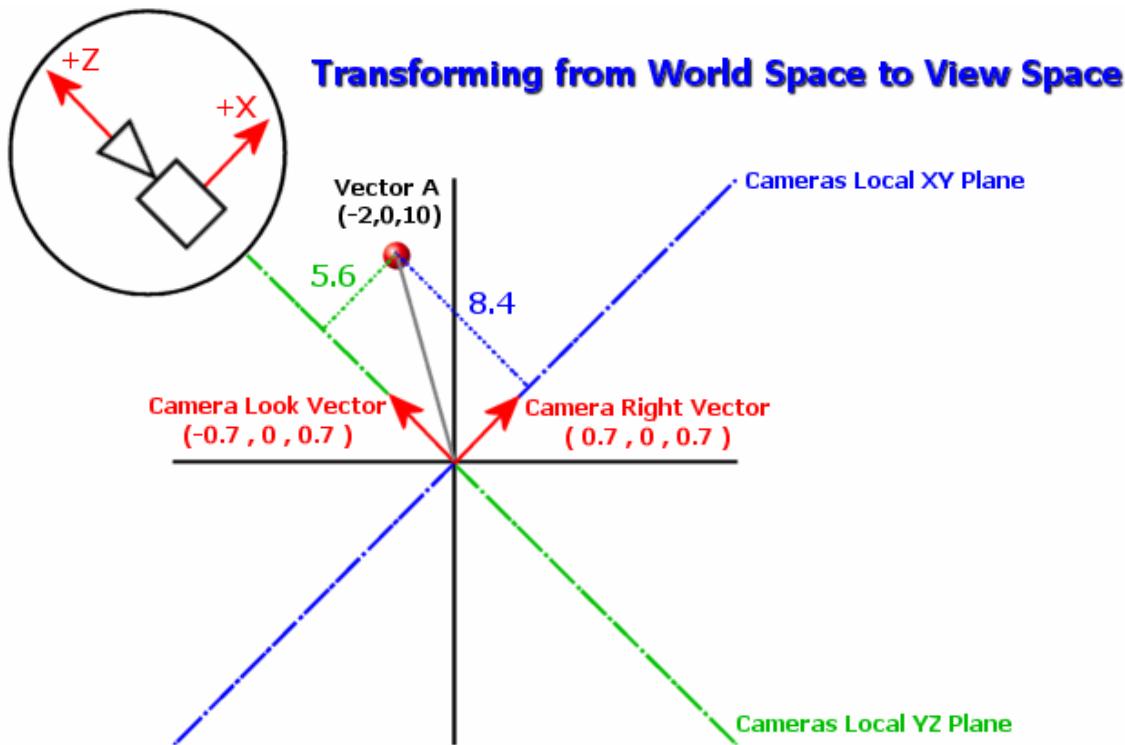


Figure 4.8

The black horizontal line is the world space X axis (the world space XY plane) and the vertical black line is the world space Z axis (the world space YZ plane). Because we are looking down on the world, we cannot see the Y axis. In the circular inset at the top left of the diagram you can see that the orientation of the virtual camera is a 45 degree rotation to the left about the Y axis. The position of the camera is assumed to be at the origin of the world space coordinate system in this example. The two red arrows show the orientation of the Look and Right vectors stored in the matrix and the blue and green lines show the planes that these two vectors describe. For example, the Look vector, when treated as a plane normal, describes the blue plane (the camera local XY plane). The Right vector describes the plane shown as the green line (the camera local YZ plane). As we can see, the camera space XY and YZ planes are misaligned from the world space XY and YZ planes by 45 degrees.

Now we get to the really important part. We know that when we have a vector such as the one shown in the diagram, it is defined as a collection of distances from the world space planes. The world space position seen above (-2, 0, 10) simply means that this vector is -2 units from the world YZ plane, a distance of 0 units from the world XZ plane and a distance of 10 units from the world XY plane. When we multiply our world space vector with the view matrix, we are actually recalculating the three distances such that they are now relative to the planes stored in the view matrix instead of the world space planes. When we multiply our vector with the first column of the matrix we are classifying the point against the camera local YZ plane. You can see that this returns a distance 5.6. The dot product between the input vector and the Y column of the matrix simply leaves the input value (0) unchanged because the second column of our view matrix in this example is a Y identity column. The Z column

result -- the distance from the vector to the camera local XY plane -- is 8.6. Therefore, the vector has been transformed from world space vector  $(-2, 0, 10)$  to a new view space vector  $(5.6, 0.0, 8.6)$ .

Now look at Fig 4.8 and rotate your head left 45 degrees so that the camera XY plane looks like the world Z axis. You will see what relationship this vector will have with the origin -- which will be assumed to be the camera position. Now, slowly rotate your head back upright and imagine that the camera planes and the world space point are rotating with you. The camera planes should now be aligned with the world planes. At this point we see that the new view space point describes the location of the world space point having been rotated 45 degrees to the right. Remember, the camera is imagined to be rotated left so we move the vector right. This is an intuitive way to think of transformations. We are simply classifying the input point against the planes. This has the same effect as taking those planes and rotating them until they align with the world planes and rotating the point along with them. As such, we treat it like any other point in that the distances are assumed to be offset from the world space planes. This creates the rotation shown in Fig 4.9.

### The View Space Vector

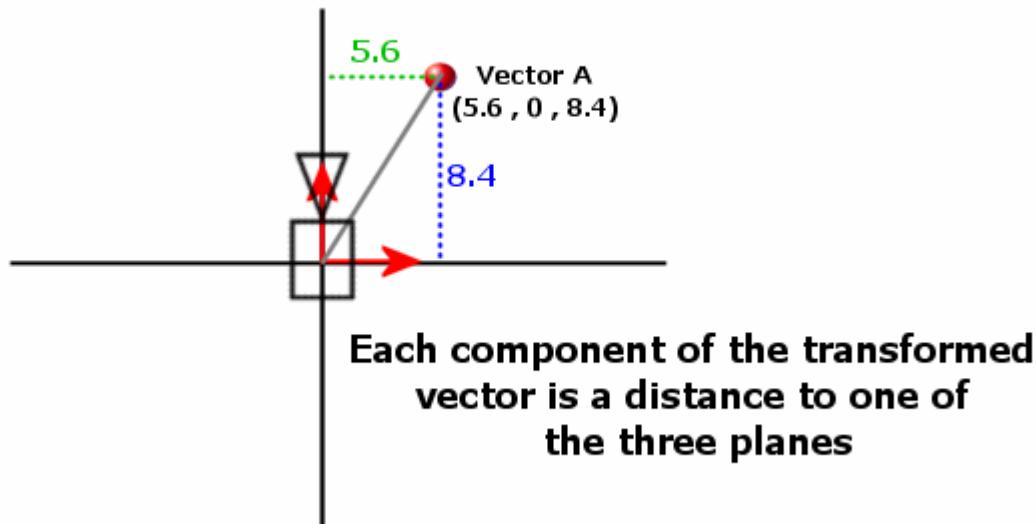


Figure 4.9

We classify the input vector against the camera local planes and then use the returned vector to describe a position relative to the world aligned planes. The result is the perceived rotation.

#### The Inverse Translation Vector

If the translation vector in the view matrix is zeroed out as in the above example then this completes the transformation from world space to view space. Otherwise, we noted that the fourth row of the inverse matrix was calculated by performing three dot products between the world space camera position and the respective vector axis stored in that column. We have highlighted the inverse



translation section of the view matrix below. This will only be zeroed out when the camera is positioned at the origin of world space.

## View Matrix

Right Vector.x	Up Vector.x	Look Vector.x	0
Right Vector.y	Up Vector.y	Look Vector.y	0
Right Vector.z	UpVector.z	Look Vector.z	0
- (Position • RightVector)	- (Position • UpVector)	- (Position • LookVector)	1

We learned in Chapter 1 (before we started using matrices) that all we had to do was subtract the camera position from the input vector and then perform the rotation. This rotated the input vector about the camera position instead of the world space origin. However, when using concatenated matrices we do not have the luxury of choosing the order in which our rotation and translation is done. The rotation is performed first (per vector component that is). If we were to simply store the negated camera position in the translation vector of the matrix, the results would be incorrect. Instead we need to know how much to subtract from the input vector *after* it has been rotated about the world space origin such that it sits in the corrected position in view space.

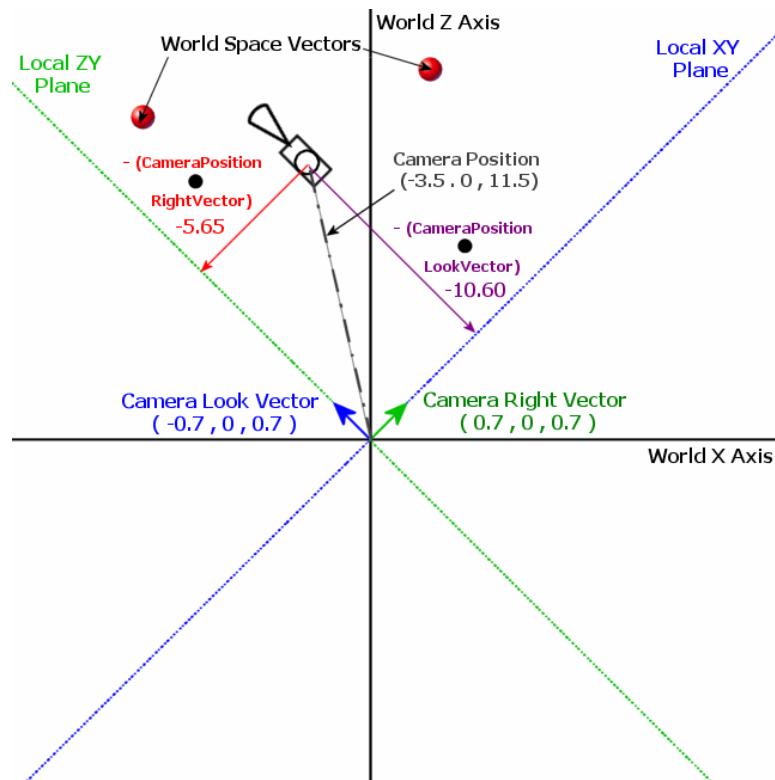


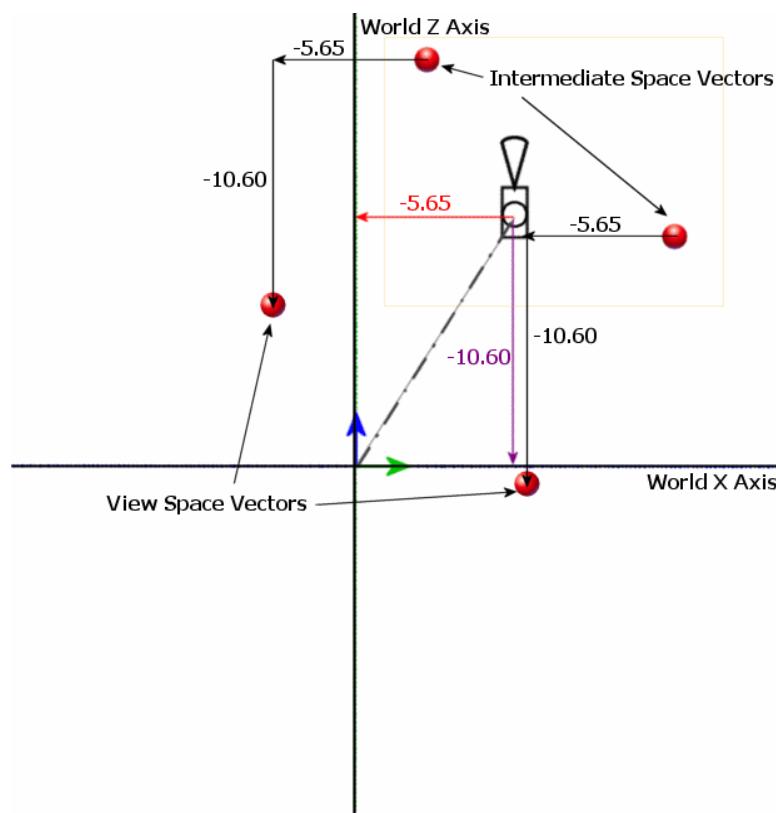
Figure 4.10



Fig 4.10 depicts a top down view of world space with a camera rotated 45 degrees left about the Y axis. The Y axis in this diagram cannot be seen because we are looking directly down it onto the world. This time the camera position is not at the origin but is at world position (-3.5, 0, 11.5). We see two example world space vectors shown as the red spheres in the image. If we were not using matrices, we would simply subtract the camera position from each vector such that the virtual camera is moved back to the origin. We would then perform the three dot products with the world space position vectors and each of the camera local planes as we did above, which effectively rotates the camera local planes right 45 degrees (along with the world space vectors) such that they are aligned with the world space planes. This means that the vertices of a mesh would be rotated 45 degrees right about the camera position. The problem we face now is that the rotation happens *before* the translation when using matrices. So we need to know how much to subtract from the input vector after it has been rotated into its new position.

Imagine that we subtracted the camera position from each of the two world vectors. You should be able to see how the position of these vectors would indeed share a relationship with the origin that they had previously shared with the camera position. We know that before the translation vector gets added to the input vector, the input vector is classified against the camera local planes. In this example it would create a new vector that has been rotated 45 degrees right about the origin from its previous position. Now imagine that the two world space vectors shown above had been rotated about the origin 45 degrees right but had not yet had the translation vector applied. You could say at this point that the

vectors are not in world space or view space but rather some intermediate space. The position of these intermediate space vectors is seen in Fig 4.11.



**Figure 4.11**

We can see that if we subtracted the camera world space position (-3.5, 0, 11.5) from these intermediate space vectors, the resulting vectors would not have a relationship with the origin that they had previously had with the camera. This is because we can think of the intermediate space vectors and the camera world space position vector as being in two different spaces at this point. No immediately meaningful relationship exists yet between them. Therefore, we need to know how much to subtract the vectors in intermediate space, not in world space.

The answer is surprisingly simple. If we look at Fig 4.10 and Fig 4.11 and imagine that instead of just rotating the vectors 45 degrees by classifying them against the camera local planes, we also rotated the actual camera position itself, once again by classifying the camera position against the local planes stored in the view matrix, this rotates not only the input vector into the new intermediate rotated position, but also rotates the camera position itself 45 degrees about the origin into intermediate space. Fig 4.11 shows that in this example, the intermediate camera position vector would be (5.65, 0, 10.60). Now that the camera position and the vector we are transforming are in intermediate space and have had their relationships maintained, we can simply subtract the distances stored in this new camera position from each of the intermediate space vectors and get them into true view space.

If we look at the fourth row of the view matrix, we see that it contains three dot products. When we think about how the input vector is multiplied against the first three columns of the matrix to classify the input vector against the three matrix planes, we can see that exactly the same thing is happening here. Only now we are classifying the camera position against the camera local planes instead of the input vector. Therefore, the fourth row is like an inline vector/matrix multiply -- a vector/matrix multiply within a matrix. So then all that is happening here is that we are classifying the camera position itself against the camera local planes. Thus in our example, the position is rotating 45 degrees to the right. We can see in the first of the two previous diagrams that the fourth row of the first column of the matrix calculated the distance from the camera position to the cameras local ZY plane ( $x = 5.65$ ). The camera is not pitched at all so the Y component of the transformed vector will remain zero ( $y = 0$ ). Finally, when we multiply the camera position against the Look vector in the third column of the fourth row, we are calculating the distance from the camera position to the camera local XY plane ( $z = 10.60$ ). The image above shows what the new rotated camera position would look like if it actually existed as a physical object. At this point we can simply subtract this new camera position vector from the input vector. We note that the fourth row of the matrix is added and not subtracted from the resulting vector component currently being calculated. So we negate the value and it will now subtract the adjusted camera position. Now the input vector is successfully transformed into view space. We can see in Fig 4.10 that when we add the intermediate camera position (-5.65, 0, -10.60) to the intermediate space vectors, we effectively move the intermediate space vectors into their final view space positions and the camera is then situated at the origin. Of course, all of this is actually done per-component of the resulting vector, so adding the fourth row as a separate stage is not what happens. Instead we rotate and inverse translate each component one at a time to get the same results.



## Viewports

In DirectX Graphics, we have the ability to limit scene rendering to only a portion of the frame buffer. We use the D3DVIEWPORT9 structure to inform the device of the rectangular region of the frame buffer to which rendering should be limited. The D3DVIEWPORT9 structure is shown below.

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

### **DWORD X**

### **DWORD Y**

These members define the coordinate of the top left corner of the viewport rectangle in frame buffer pixel coordinates. If you set both of these to zero, then the top left corner of the viewport will match the top left corner of the frame buffer.

### **DWORD WIDTH**

### **DWORD HEIGHT**

These values define the width and height of the viewport in frame buffer pixel coordinates. The viewport will be a rectangular region on the frame buffer with the coordinates (X, Y, X+Width, Y+Height).

### **Float MinZ**

### **Float MaxZ**

These values can be used to remap the Z-Buffer depth values calculated by the projection matrix and the divide by W into another range of values. Usually you will set these to 0.0 and 1.0 so that the depth values calculated from the projection matrix and the divide by W are passed straight through to the rasterizer.

## The Viewport Matrix

In Chapter 1 we learned that after the divide by W, our vertex coordinates are in 2D projection space coordinates in the range  $[-1, 1]$  in the X and Y dimensions. In our software pipeline demo we used the following formula to map these coordinates into screen space coordinates:

```
ScreenX = projVertex.x * ScreenWidth / 2 + ScreenWidth / 2  
ScreenY = -projVertex.y * ScreenHeight / 2 + ScreenHeight / 2
```



This works correctly when we are assuming that the frame buffer is taking up the entire screen (or the entire window). When rendering to a viewport however, DirectX Graphics also has to take into account the viewport origin and its width and height so that all of the projection space coordinates in the -1 to +1 range get mapped to coordinates that fall only within the viewport rectangle.

```
ScreenX = projVertex->x * ViewportWidth / 2 + ViewportX + ViewportWidth / 2;  
ScreenY = -projVertex->y * ViewportHeight / 2 + ViewportY + ViewportHeight / 2;
```

When we discussed the DirectX transformation pipeline in earlier lessons we examined the core matrices that are used in the process: World, View and Projection. There is actually a fourth matrix which the vertices are multiplied by which contains the above formula to map projection space coordinates into screen space coordinates. This matrix is shown below. The third column maps the vertex depth value into the [minZ, maxZ] range of the view port.

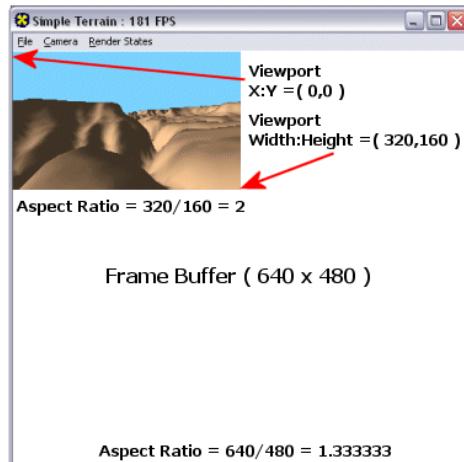
### The Viewport Matrix

$ViewportWidth / 2$	0	0	0
0	$-ViewportHeight / 2$	0	0
0	0	$ViewportMaxZ - ViewportMinZ$	0
$ViewportX + ViewportWidth / 2$	$ViewportHeight / 2 + ViewportY$	$ViewportMinZ$	1

All of this is managed behind the scenes by the device object. We simply fill in the details of the D3DVIEWPORT9 structure and send it to the device with a call to:

```
IDirect3DDevice9::SetViewport (CONST D3DVIEWPORT9 *pViewport);
```

The function will force the device to rebuild its viewport matrix based on the settings that we have passed in with the D3DVIEWPORT9 structure. When the device is first created, the default state of the viewport matrix is to map projection space coordinates to the entire area of the frame buffer. If we created a 640x480 frame buffer, the default viewport will be 640x480 also, with its top left corner at (0, 0). Just to be clear, when we set a viewport, this does not simply truncate the portions of the scene that are outside the viewport. The entire scene is rendered into the view port as shown in Fig 4.12:



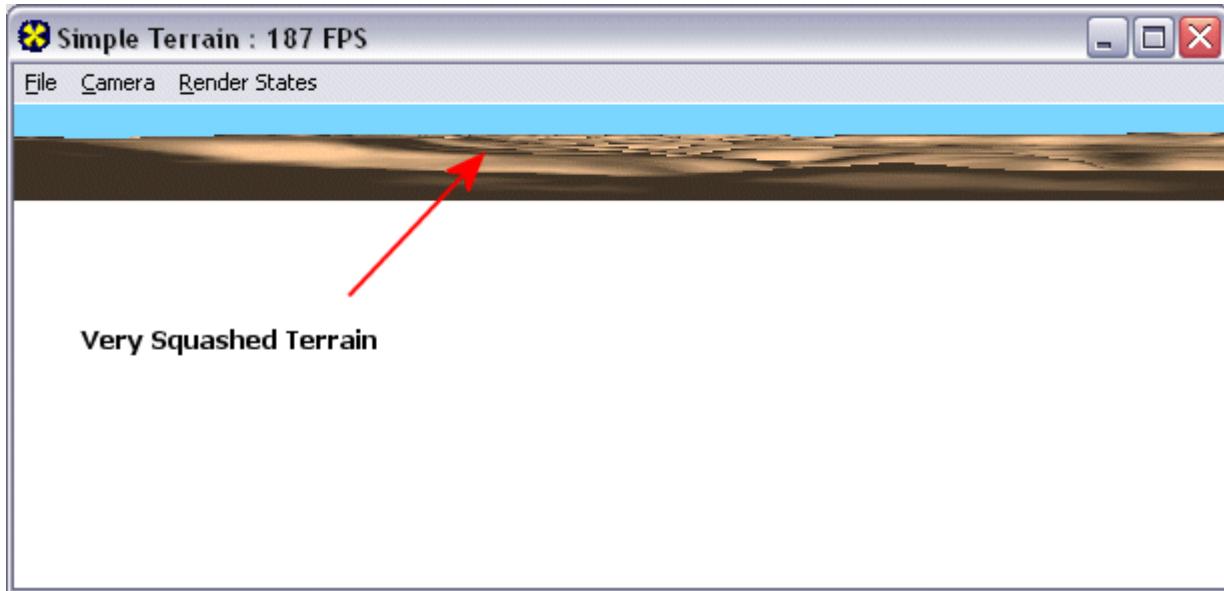
**Figure 4.12**

In the above example, we have a 640x480 frame buffer and a viewport rectangle of (0, 0, 320, 160). Note that the entire scene is rendered into the viewport rectangle within the frame buffer. When we present the frame buffer (assuming we do not provide a presentation rectangle) the entire frame buffer is still displayed.

Viewports can be very useful. For example, you may use them when programming a split screen two player game. You could set the view port so that it takes up the top half of the frame buffer and then render the scene in that viewport from player one's position. Then you could set the view port such that it takes up the bottom half of the frame buffer and render the scene again, this time from the second player's position.

### Viewport Aspect Ratios

We must ensure that if we use a viewport that does not span the entire frame buffer, that we use the aspect ratio calculated using **ViewportWidth/ViewportHeight** rather than **FrameBufferWidth/FrameBufferHeight** when we build the projection matrix. The previous image showed us that the frame buffer had an aspect ratio of 1.333333, but when we set the viewport to 320x160 the aspect ratio of the viewport was 2.0. It is important that we use the aspect ratio of the viewport since this is where the scene will be rendered. Using the 320x160 viewport shown above, the image would look squashed if we did not adjust the aspect ratio to reflect the new settings. Just because we may have an elongated viewport, does not mean we wish to see the geometry in our scene elongated. Fig 4.13 shows how the same image of the terrain would look using a wide but shallow viewport without recalculating the aspect ratio of the viewport.



**Figure 4.13**

**Note:** As with all device states, when the device is lost, the viewport information is also reset. Therefore you must remember to reset your viewport settings when resetting a device.

### Camera Manipulation I

In Chapter 1 we discovered that we can multiply one matrix with another matrix to generate a resulting matrix that will transform vectors in the same way that the two source matrices would have done individually. Therefore, it is safe to assume that if we were to build a rotation matrix, let us say a matrix that rotates vectors 45 degrees around the X axis, and then multiply our view matrix by that rotation matrix, we would have created a resulting matrix that not only transforms the vertices from world space into view space, but one that also rotates them around the world X axis. The following code snippet retrieves the currently set view matrix from the device and rotates it 45 degrees about the X axis (pitching it down).

```
D3DXMATRIX matView, matRotx;

// Get View matrix from device (will not work on a pure device)
pDevice->GetTransform(D3DTS_VIEW, &matView);

// Built Rotation matrix about X axis
D3DXMatrixRotationX(&matRotx, D3DXToRadian(-45));

// Multiply the view matrix with rotation matrix
D3DXMatrixMultiply(&matView, &matView, &matRotx);

//Set the new modified view matrix
pDevice->SetTransform(D3DTS_VIEW, &matView);
```

Because all of the same transformations can be applied to the view matrix as to any world matrix, we can think of the view matrix as a physical camera object even if this is not technically correct. Let us assume that the view matrix was set to an identity matrix before the above code was executed. We already know that if the view matrix is an identity matrix then the Look, Up, and Right vectors in the view matrix exactly match the axes of the world coordinate system.

Because of the order of the above matrix multiplication, we are in fact performing what is known as a *camera local rotation*. Instead of rotating the camera about the world X axis, we are in fact rotating the camera about its own Right vector. The red arrow in Fig 4.14 shows the direction of rotation the code would generate.

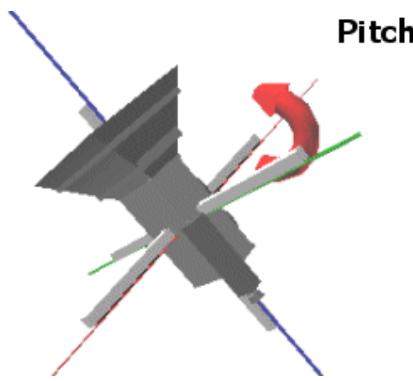


Figure 4.14

Because the code performed a camera relative rotation, we see now that we can perform accumulative rotations. Regardless of the orientation of the camera in world space, the above code will always pitch the camera down (or up if we negate the rotation angle) relative to itself and not the world. For example, if you stand on your head and look up, you are looking at the floor. But it is still ‘up’ with respect to your current situation. If somebody was observing you however, they might describe you as looking down at the floor, given their perspective.

If we were to change the matrix multiplication order from `ViewMatrix*RotationMatrix` to `RotationMatrix*ViewMatrix` this would rotate the camera about the world X axis. This would not perform a localized rotation but would instead perform a world rotation. So take care to use the matrix multiplication order that produces the results you desire. When rotating a non-inverted matrix (an object world matrix for example) the opposite is true: `WorldMatrix*RotationMatrix` would perform a non-localized rotation and `RotationMatrix*WorldMatrix` would apply localized rotation.

D3DX includes helper functions that allow us to build rotation matrices for the Y and Z axes also.

```
D3DXMATRIX matView, matRoty;

// Get View matrix from device (will not work on a pure device)
pDevice->GetTransform(D3DTS_VIEW, &matView);

// Built Rotation matrix about Y axis
```

```

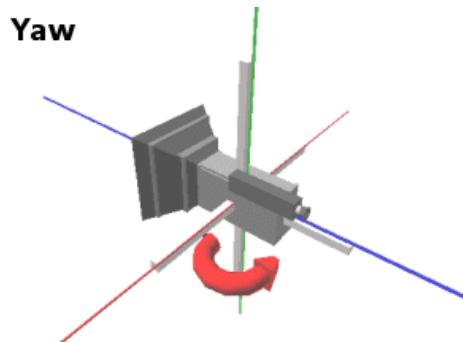
D3DXMatrixRotationY(&matRoty, D3DXToRadian(45));

// Multiply the view matrix with rotation matrix
D3DXMatrixMultiply(&matView, &matView, &matRoty);

//Set the new modified view matrix
pDevice->SetTransform(D3DTS_VIEW, &matView);

```

Notice the matrix multiplication order we are using to rotate the camera about its own Up vector (view space Y axis). This allows us to yaw the camera left or right relative to itself. Changing the multiplication order would change this so that the camera was always rotated about the world Up vector rather than the camera Up vector



**Figure 4.15**

This system provides us with a convenient way to handle rotating left and right in a game. Notice that the red arrow in the Fig 4.15 shows the direction of rotation about the Up vector that the camera will have applied to it. As we now know, a positive angle would create a matrix that would rotate vectors right, but because we are not inverting the rotation matrix before multiplying it with the view matrix (which is already inversed) the rotation direction is switched. Therefore, a positive rotation angle would rotate the camera left.

Finally the above code could also be changed to rotate the camera about the Z axis to create a Roll effect. Rolling is the effect you get in a flight simulation where pushing left and right on the joystick banks the plane.

```

D3DXMATRIX matView, matRotz;

// Get View matrix from device (will not work on a pure device)
pDevice->GetTransform(D3DTS_VIEW, &matView);

// Built Rotation matrix about z axis
D3DXMatrixRotationZ(&matRotz, D3DXToRadian(-45));

// Multiply the view matrix with rotation matrix
D3DXMatrixMultiply (&matView, &matView, &matRotz);

//Set the new modified view matrix
pDevice->SetTransform(D3DTS_VIEW, &matView);

```

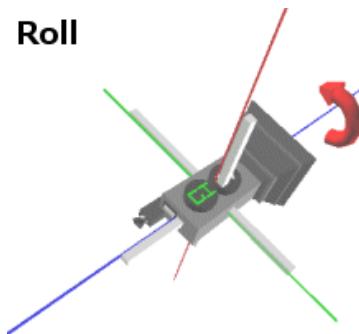


Figure 4.16

The red arrow in Fig 4.16 shows the direction of rotation that this code would apply to the cameras Up, Right and Look vectors stored in the view matrix. Again, we would typically associate a negative rotation angle as applying a clockwise rotation (roll right) when rotating a world matrix, but since we are applying the rotation matrix (without inverting it) to the view matrix, the rotational direction is flipped.

So we now have the ability to easily rotate a virtual camera about all three of its axes. As we know, matrix multiplication is **not** commutative and the order in which the matrices are passed to the multiplication function is critically important.

## Camera Manipulation II

In this section we are going to look at an easier way to ensure proper local camera rotations. We are going to abandon the D3DXMatrixRotate functions as a means of applying rotations to our view matrix. Instead, we will manually rotate the Look, Up, and Right vectors in the view matrix ourselves so that the rotations are always relative to any desired arbitrary axis. We can maintain and rotate these vectors separately and simply rebuild the view matrix each time they change. Not only will this allow us to perform the relative rotations that matrix multiplication provided, but it will allow us to rotate our vectors around any axis we choose. This might not sound so easy until you realize that D3DX has a function for building a matrix that rotates vectors about any arbitrary axis. We simply send the function a unit vector and an angle:

```
D3DXMatrixRotationAxis(D3DXMATRIX *pOut, CONST D3DXVECTOR3 *pV, FLOAT Angle);
```

### **D3DXMATRIX \*pOut**

This is the address of a D3DXMATRIX structure that will contain the newly generated matrix.

### **D3DXVECTOR3 \*pV**

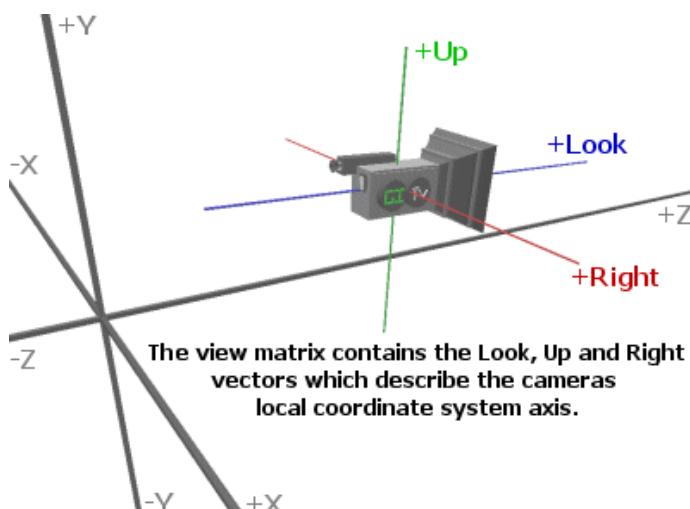
This is an arbitrary unit length vector that is treated as the axis of rotation. For example, if you passed in a vector of (1,0,0) then this would produce the same rotation matrix as D3DXMatrixRotationX. Because we can pass in vectors that are not limited to the world space axes, it

means that we can generate a rotation matrix that will rotate the camera about any arbitrary world space axis, as well as the camera Look, Up and Right vectors when camera-relative rotations need to be applied.

### FLOAT Angle

The angle in radians to rotate about the passed axis.

Now let us imagine that we are trying to create a spacecraft camera system. Assume that we want the left and right actions on the joystick to produce local yaw, the forward/back actions on the joystick to produce local pitch and a left/right action on the joystick with the fire button down to produce local roll. Fig 4.17 shows the where the virtual camera might be in the world:



**Figure 4.17**

Now let us see what the code might look like that reacts to the user pulling the joystick backwards. Your input routine may call a function like the following to rotate the camera about its local X axis by the specified angle. Note that we are extracting the vectors from the view matrix but you would probably store the four view matrix vectors (look, up, right, and position) as variables for easier access and to run this with a pure device.

```
void Pitch(IDirect3DDevice9* pDevice, float Angle)
{
    D3DXMATRIX matView, matRotx;
    D3DXVECTOR3 RightVector, UpVector, LookVector;

    pDevice->GetTransform(D3DTS_VIEW, &matView);

    RightVector.x = matView._11;
    RightVector.y = matView._21;
    RightVector.z = matView._31;

    UpVector.x = matView._12;
    UpVector.y = matView._22;
```

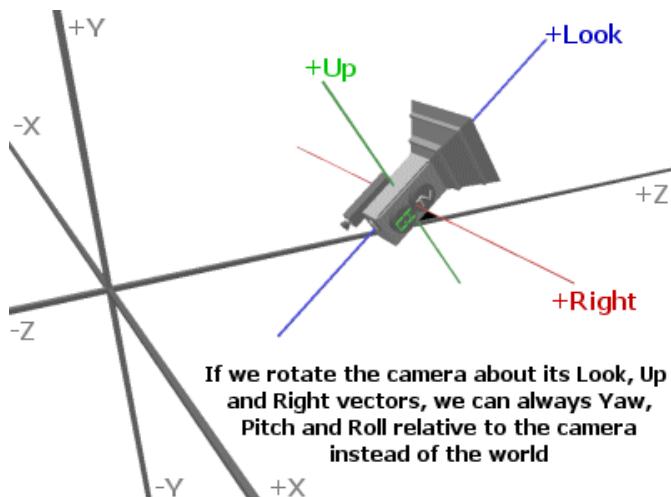


```
UpVector.z = matView._32;  
  
LookVector.x = matView._13;  
LookVector.y = matView._23;  
LookVector.z = matView._33;  
  
D3DXMatrixRotationAxis (&matRotx , &RightVector , Angle );  
  
D3DXVec3TransformNormal (&UpVector , &UpVector , &matRotx);  
D3DXVec3TransformNormal(&LookVector , &LookVector , &matRotx);  
  
matView._12=UpVector.y;    matView._13=LookVector.z;  
matView._22=UpVector.y;    matView._23=LookVector.z;  
matView._32=UpVector.y;    matView._33=LookVector.z;  
  
pDevice->SetTransform(D3DTS_VIEW , &MatView);  
}
```

This example assumes we are not using a PURE device since it uses the GetTransform function to retrieve the current view matrix from the device. Our final code will manage its own copy of the view matrix making this call unnecessary but we have used that method here to better show the process. The code does the following:

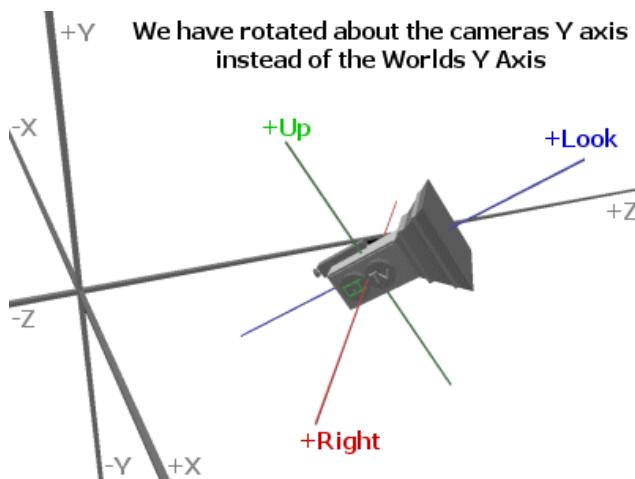
- It retrieves the current view matrix
- It manually extracts the cameras local axes from the view matrix and stores them in RightVector, UpVector and LookVector for the local X,Y and Z axes respectively.
- Because we are pitching up, we wish to rotate the camera about the RightVector (local X axis). We build a rotation matrix that will rotate vectors about that axis (whatever orientation it may be). Because the rotation is about the Right vector, the vector itself will be unchanged. All we have to do is rotate the Look and Up vectors about the Right vector.
- Once we have multiplied the Look and Up vectors with the rotation matrix, we place them back into the view matrix so that the view matrix now contains the new orientation.
- Notice that we do not have to place the Right Vector into the view matrix because it has not been changed by this function.

Fig 4.18 shows what the view matrix and its vectors would look like if the above function was called to pitch the camera up 45 degrees (a negative angle would rotate downwards). Notice how the right vector is unchanged, but the Look vector and the Up vector have been rotated such that they are no longer aligned with the world Y and Z axes:



**Figure 4.18**

So in order to rotate the camera about its local X axis, all we have to is rotate the Up and Look vectors about the Right vector. Regardless of the orientation of the Right vector in the world, this will always pitch the camera up and down relative to itself. Hopefully, the above code snippet has given you everything you need to write a function that Yaws. Looking at the diagram, you should be able to see that in order to perform local Yaw we have to rotate the Right and Look vectors about the Up vector. Fig 4.19 shows what the camera should look like if we were to apply a 45 degree Yaw.



**Figure 4.19**

The next piece of code is a function that allows the camera to rotate left and right about its own Y axis. This function is very similar to the Pitch function with the exception that we now wish to rotate the Right and Look vectors about the Up vector. This means the Up vector will be unchanged.

```
void Yaw (IDirect3DDevice9* pDevice , float Angle)
```



```
{  
    D3DXMATRIX matView, matRoty;  
    D3DXVECTOR3 RightVector, UpVector, LookVector;  
  
    // Retrieve device view matrix  
    pDevice->GetTransform(D3DTS_VIEW, &matView);  
  
    // extract right, up and look vectors  
    RightVector.x = matView._11;  
    RightVector.y = matView._21;  
    RightVector.z = matView._31;  
  
    UpVector.x = matView._12;  
    UpVector.y = matView._22;  
    UpVector.z = matView._32;  
  
    LookVector.x = matView._13;  
    LookVector.y = matView._23;  
    LookVector.z = matView._33;  
  
    // build matrix to rotate vectors about the Up vector  
    D3DXMatrixRotationAxis(&matRoty, &UpVector, Angle);  
  
    // rotate right and look vectors about the up vector  
    D3DXVec3TransformNormal(&RightVector, &RightVector, &matRoty);  
    D3DXVec3TransformNormal(&LookVector, &LookVector, &matRoty);  
  
    // place modified vectors back into the view matrix  
    matView._11=RightVector.y;      matView._13=LookVector.z;  
    matView._21=RightVector.y;      matView._23=LookVector.z;  
    matView._31=RightVector.y;      matView._33=LookVector.z;  
  
    // send modified view matrix to the device  
    pDevice->SetTransform(D3DTS_VIEW, &matView);  
}
```

You should now have little trouble writing your own Roll function that rotates the camera about its local Z axis. It would be a good idea if you opened up Notepad right now and had a go at this to make sure that you understand what is happening. Remember to refer back to the table for the ViewMatrix to remind yourself which vectors are stored in which columns. Once you have tried implementing this function yourself, check it against the code listed below:

```
void Roll(IDirect3DDevice9* pDevice, float Angle)  
{  
    D3DXMATRIX matView, matRotz;  
    D3DXVECTOR3 RightVector, UpVector, LookVector;  
  
    // Get Current View Matrix  
    pDevice->GetTransform(D3DTS_VIEW, &matView);  
  
    // Extract the right, up and look vectors  
    RightVector.x = matView._11;  
    RightVector.y = matView._21;  
    RightVector.z = matView._31;  
  
    UpVector.x = matView._12;  
    UpVector.y = matView._22;  
    UpVector.z = matView._32;
```



```
LookVector.x = matView._13;
LookVector.y = matView._23;
LookVector.z = matView._33;

// Build matrix to rotate vector about the LookVector
D3DXMatrixRotationAxis ( &matRotz , &LookVector , Angle );

// Rotate Up and Right vectors about the Look vector
D3DXVec3TransformNormal ( &UpVector , &UpVector , &matRotz );
D3DXVec3TransformNormal( &RightVector , &RightVector , &matRotz );

// Place modified vectors back into view matrix
matView._11=RightVector.x;    matView._12=UpVector.y;
matView._21=RightVector.y;    matView._22=UpVector.y;
matView._31=RightVector.z;    matView._32=UpVector.y;

// Send the modified view matrix back to the device
pDevice->SetTransform(D3DTS_VIEW , &matView);
}
```

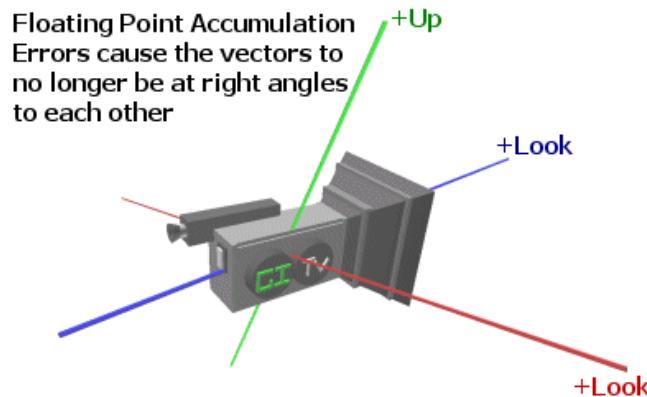
Another thing to bear in mind is that we can store the world space position of the camera and allow our application to work with that position vector just like any other object position in the world. When the position or orientation of the camera changes, we can place the Look, Up, and Right vectors into a view matrix and calculate the inverse translation vector using the camera world space position. It is much more intuitive for our application to move the camera using a world space position and calculate the inverse translation vector when inserting it into the fourth row of the view matrix rather than have to store the position as an inverse translation vector.

## Vector Regeneration

The finite resolution of floating point numbers on the PC leads to some trouble as we continually rotate our vectors. The vector/matrix multiplications we are performing involve many floating point multiplications and over time, errors can start to accumulate. The problem is that a float can only store a finite number of digits. Let us imagine that we want to store the value of PI (defined as 3.14159265358979323846...) within a single precision float. This value will be truncated before it is stored, so that perhaps our float variable holds 3.141593.

If we multiply this float by 36.0 we should see a return value of 113.097384. However, because of the floating-point limitation, the result is rounded to 113.0974 before storage. If we divide by the same value again (36.0), we find that we end up with a value of 3.14159444444444, which is again rounded to 3.141594.

So simply multiplying and then dividing by the same value produces a float which is 0.000001 adrift from the original value. This may not seem like much, but over time this type of error accumulates. When these errors creep into our vectors, we can end up with a situation where the camera coordinate system axes are no longer perpendicular to each other:



**Figure 4.20**

As you can imagine, the problem gets worse as we perform more consecutive operations. You can see in Fig 4.20 that rotating the camera around the corrupted Up vector would no longer perform a proper Yaw. Additionally, the accumulations can also cause the vectors to lose their unit length status which also has adverse effects. To combat this, we must perform vector regeneration on these vectors at regular intervals. This means rebuilding the vectors to ensure that they remain orthogonal and unit length. In our application, we will do this every time the camera is rotated in some way. The technique for regenerating the vectors is shown below.

- Normalize the Look vector so that it is always unit length
- Regenerate the Up vector by performing a cross product with the Look and Right vectors to return a new Up vector that is perpendicular to them
- Normalize the new Up vector to make sure it is unit length
- Regenerate the Right vector by performing a cross product with the Look and Up vectors to return a Right vector that is perpendicular to them
- Normalize this new Right vector to make sure it is unit length

We now have a regenerated set of unit length vectors that are mutually perpendicular. Notice that we only normalize the Look vector and do not actually regenerate it by performing a cross product between the Up and Right vectors. This would cause the Look vector to be snapped suddenly to another vector and would cause a noticeable shift to the player. For this reason, we leave the orientation of the Look vector alone and simply normalize it.

The following code handles rotation about all three axes and regenerates the vectors before placing them back into the view matrix. This function expects rotations to be specified in radians.

```
void Rotate (IDirect3DDevice9* pDevice , float x , float y , float z )
{
    D3DXMATRIX      matRotate , matView;
    D3DXVECTOR3     RightVector , UpVector , LookVector;

    // Extract Local Camera axes from view matrix
    pDevice->GetTransform ( D3DTS_VIEW , &matView);
```



```
RightVector.x = matView._11; UpVector.x = matView._12 ; LookVector.x = matView._13;
RightVector.y = matView._21; UpVector.y = matView._22 ; LookVector.y = matView._23;
RightVector.z = matView._31; UpVector.z = matView._32 ; LookVector.z = matView._33;

if ( x != 0 )
{
    // Build rotation matrix
    D3DXMatrixRotationAxis( &matRotate, &RightVector, x );
    D3DXVec3TransformNormal( &LookVector, &LookVector , &matRotate );
    D3DXVec3TransformNormal( &UpVector , &UpVector , &matRotate );
} // End if Pitch

if ( y != 0 )
{
    // Build rotation matrix
    D3DXMatrixRotationAxis( &matRotate, &UpVector, y );
    D3DXVec3TransformNormal( &LookVector, &LookVector, &matRotate );
    D3DXVec3TransformNormal( &RightVector, &RightVector, &matRotate );
} // End if Yaw

if ( z != 0 )
{
    // Build rotation matrix
    D3DXMatrixRotationAxis( &matRotate, &LookVector, z );
    D3DXVec3TransformNormal( &UpVector , &UpVector , &matRotate );
    D3DXVec3TransformNormal( &RightVector, &RightVector, &matRotate );
} // End if Roll

// Perform 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 );

// Place the new rotated vectors back into the view matrix
matView._11 = RightVector.x; mtxView._12 = UpVector.x; mtxView._13 = LookVector.x;
mtxView._21 = RightVector.y; mtxView._22 = UpVector.y; mtxView._23 = LookVector.y;
mtxView._31 = RightVector.z; mtxView._32 = UpVector.z; mtxView._33 = LookVector.z;

// Send the new view matrix back to the device
pDevice->SetTransform ( D3DTS_VIEW , &matView );
}
```

**Note:** Whether you are rotating your camera by rotating its Look, Up and Right vectors manually as shown above, or whether you are rotating the camera by performing matrix multiplication as shown earlier, vector regeneration still needs to be done when performing cumulative rotations. Even when performing cumulative matrix multiplication, we are really just rotating the Look, Up and Right vectors inside the view matrix. So you will need to periodically extract the vectors, regenerate them and insert them back into the matrix.



## First Person Cameras

The previous function rotates the camera about its own Up, Right, and Look vectors. This is ideal for a space ship camera system. For a first person camera system however, we need to make some changes. We will discuss those changes briefly since we will cover the complete camera code in the workbook.

- Pitch must be limited so that we cannot loop completely upside down. In a first person game, the camera acts as the head of the player with regards to up/down rotation. In reality, our head's rotational capacity is limited by our neck. Generally we clamp pitch to +89 degrees up and -89 degrees down.
- Yaw (Y Axis rotation) has to be handled differently for a first person camera. In the previous camera code above, we always rotated about the camera local Y axis (the Up vector). This does not work for a first person camera. If you load up any first person perspective game and rotate your head so that you are looking at the roof (or floor) and then move the mouse left or right, you will notice that the camera will no longer move left or right relative to itself, but will actually spin around in a tight circle. This is because in a first person camera situation, we want to always yaw around the *world* Y axis instead of the camera Y axis. Fig 4.21 demonstrates this concept.

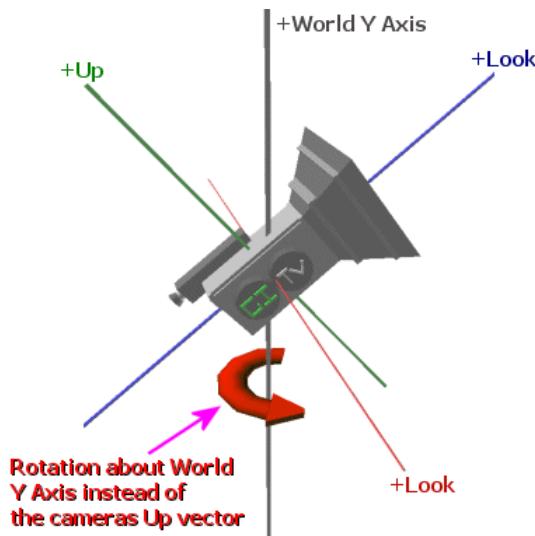


Figure 4.21

This actually makes perfect sense since pitching up and down is equivalent to rotating your head up and down. Yawing left and right is equivalent to actually spinning your whole body around in a circle. Therefore, if you yaw whilst the camera is pitched up, it is equivalent to standing in a room, looking up at the ceiling, and spinning around.

## Third Person Cameras

Third person camera systems are used in games like Tomb Raider™, Splinter Cell™, Mario64™ and many others. A third person camera is quite different than the other types we discussed because we have to limit the player's ability to move the camera such that it does not lose focus on what it is supposed to be looking at. The idea is that the user no longer directly controls the camera, but instead controls the player avatar. The camera stays focused on that avatar as it moves around the environment. In our third person camera system, although the camera will always be looking at the player, we will have the ability to specify an offset vector that allows us to control the distance and angle from which the player should be viewed. You can think of this as attaching the camera to the player avatar with a large stick. When the player moves, the camera will move also. When the player turns left or right, so will the camera.

If this was the limit of our control, then the camera system would seem far too rigid. Let us imagine that we specify an offset vector that puts the camera directly behind the player object. This would mean that we would always be looking at the back of the avatar no matter how quickly they rotate left or right. To make things more interesting and fluid we can introduce a time lag when the player rotates. In this case the camera will still be behind the player but it may take  $\frac{1}{4}$  of a second to catch up to a new rotation. This allows us to see our player turn before the camera drifts into the correct position again.

Fig 4.22 is a crude representation of this concept. The camera is attached to the player and offset by the vector shown as the white line. In our demo, the camera is always positioned behind and above the player as specified by the offset vector that we specify. In third person mode, the player object can be yawed left or right, and the camera has its position rotated about the player in a large circle using a radius specified by the offset vector.

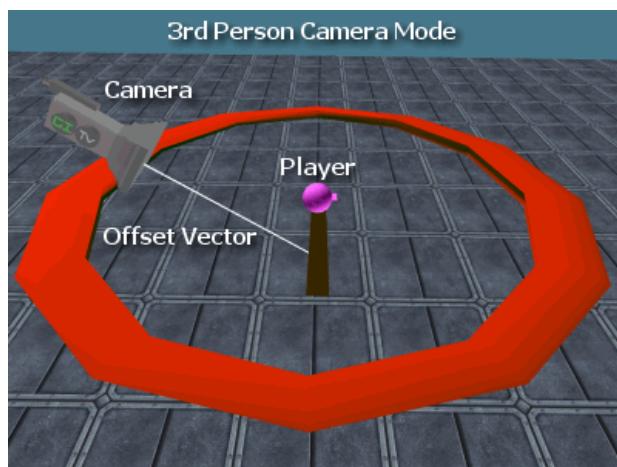


Figure 4.22

## The View Frustum

As discussed earlier in the course, the projection matrix defines a camera view volume. Recall that the 1<sup>st</sup> and 2<sup>nd</sup> columns of the projection matrix store values that define the angle of the horizontal and vertical field of view respectively. We also use the 3<sup>rd</sup> column of the projection matrix to map view space vertex Z components to the finite [0, 1] range used for depth buffering. To achieve this mapping we used the projection matrix to set minimum and maximum view distances, referred to as the near and far planes. Any view space vertices with Z components greater than the far plane distance stored in the projection matrix are rejected by the pipeline prior to the homogeneous W divide (i.e. the projection). Likewise, any view space vertices that have Z components smaller than the near plane distance will also be rejected in the same place by the pipeline. When we consider the shape of the space defined by the viewing angles as well as the minimum and maximum depths, we note that the result resembles a truncated pyramid. This pyramidal volume is called a **frustum**. It is similar to a typical pyramid shape except that the tip of the pyramid is sliced off by the near plane and the base is sliced off by the far plane (Fig 4.23).

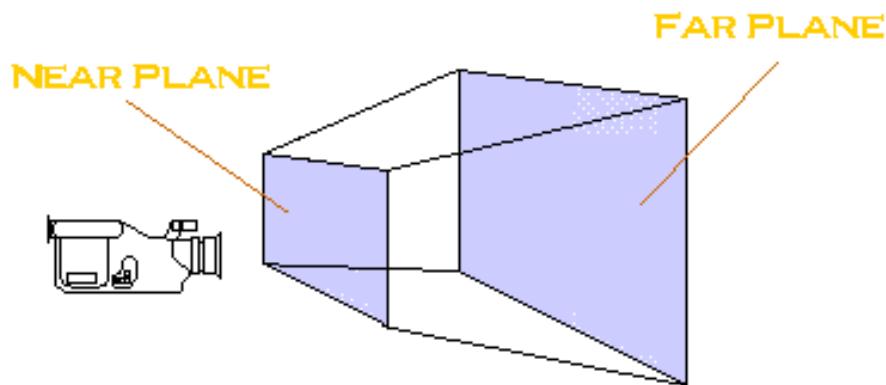


Figure 4.23

It could be said that the 1<sup>st</sup> column of the projection matrix describes the normal for two planes: the left and right planes of the view frustum (where one is the negation of the other). Polygons that fall between these two planes are thus within the horizontal field of view of the camera. The 2<sup>nd</sup> column of the projection matrix could likewise be said to describe a normal for two planes, the top plane and the bottom plane of the frustum. If a polygon exists between these two planes, then it is within the camera's vertical field of view. Finally, the 3<sup>rd</sup> column of the projection matrix could be said to describe two additional frustum planes: the near plane and the far plane. If a polygon is positioned such that it lies completely or even partially within the area between all six of these planes, then all or part of it is considered *inside* the viewing frustum and would therefore be visible to the viewer.

In a moment we will see how and why the columns of the projection matrix describe these six frustum planes, but for now just take it on faith that they do. Bear in mind that the orientations of these planes are controlled by the values we store in the projection matrix. As a result, we have the ability to change the shape of the view frustum. Nevertheless, whilst we can change the depth, width, and height of the frustum, it will always be constructed from six planes and will look like the shape shown in Fig 4.23.

## Frustum Culling

While the pipeline will test each triangle against the view frustum prior to projection, and reject the polygon from further processing if it is outside the frustum (or clip it when it is half inside and half outside), this test is run *after* the vertices have been transformed from model space to homogenous clip space. So while the frustum rejection mechanism in the DirectX pipeline does enable the system to avoid rendering polygons that will never be seen, it is only after the expensive transformation and lighting calculations have been performed for each vertex that this rejection becomes possible.

Ideally what we would like to do is perform this test prior to our polygons entering the pipeline and thus avoid these calculations when possible. It would also be nice if we could do this without having to test every single polygon in a level. Considering the size of modern game levels, a higher level frustum test at the object (mesh) level would be much more efficient.

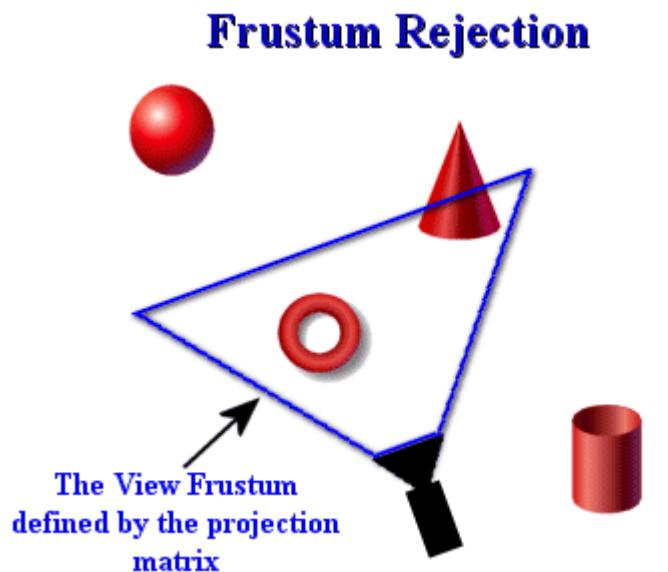


Figure 4.24

In Fig 4.24, we see a camera and its view frustum in world space along with a handful of geometric objects in the scene. (Remember that the frustum has six planes -- we just cannot see the top and bottom planes in this diagram given the top-down view). The sphere is well beyond the far plane of the



frustum and therefore all of its polygons will be rejected by the pipeline. The cylinder is partially behind and to the right of the camera, so it too will be rejected by the pipeline. The torus has all of its polygons inside the six planes of the frustum and therefore will not be rejected by the pipeline -- it will be rendered in its entirety. Finally, the cone is partially inside the view frustum and in this case, all of its triangles that are beyond the far plane will be rejected and any cone triangles that span the far plane (half inside/half outside) will be clipped by the pipeline so that only the section of the triangle that is inside the far plane would be rendered.

If the cylinder and sphere were 20,000 polygon objects, and we rejected them in such a way that we would not even bother calling `DrawPrimitive`, consider how many tens of thousands of potential calculations we would save in that case. And keep in mind that they would ultimately be discarded by the pipeline anyway, but only after all of their vertices had been transformed and lit.

All we need is a way to get access to the frustum planes in world space (or even model space) and we could perform the frustum test ourselves, rendering only the polygons that we find are visible. That is, we would like to **cull** the visible polygons in our scene and make sure they are rendered, while the rest are ignored. This process is called **frustum culling** and it is exactly what we are going to study in the remaining portions of this lesson. Since we generally do not want to do this test at the per-polygon level or take on the responsibility of clipping triangles, we will also need to figure out some way to run this test at the object/mesh level. If we did the per-polygon testing and clipping ourselves (even if we did not have to transform the vertices) on the CPU, it would still be slower than just running the entire process on the GPU and letting it deal with these issues.

We should think of our frustum culling code as less of an exact frustum culler and clipper than a first line of defence for quickly rejecting the vast bulk of scene geometry before it enters the rendering pipeline. If we leave the borderline cases (i.e. partial intersections) for the pipeline to handle, we can come up with an extremely efficient way of rejecting large batches of polygons with a few simple tests.

### Axis-Aligned Bounding Boxes (AABB)

Given the infinite number of complex shapes our meshes can assume, testing a mesh for intersection with the frustum would be difficult indeed. What sort of algorithm could we design that could handle any mesh we throw at it that did not require per-polygon testing at the end of the day? Forget it. We will need an alternative polygon aggregate that can be created easily and tested quickly for intersection against the frustum (i.e. the planes of the frustum) for real-time work. The solution is to bound our complex objects with simple volumetric shapes and then test these **bounding volumes** for collision with the frustum.

Depending on the volume we choose, our intersection tests will have different levels of accuracy and efficiency, generally trading off one for the other. The most common bounding volumes are simple shapes like boxes, spheres, cylinders, cones, and so on (boxes and spheres are generally the most popular). For now, we will keep things simple and focus on boxes. In the next course in this series, we

will discuss other bounding volumes and even more efficient ways to do frustum intersection at the scene level.

For this lesson we will use a box shaped bounding volume whose orientation is such that its sides are always aligned with the standard axes  $(1,0,0)$ ,  $(0,1,0)$ , and  $(0,0,1)$ . This is called an axis-aligned bounding box (AABB). In Fig 4.25 we see an example of our models and their AABBs. Our culling system now resolves to a set of frustum/AABB intersection tests for each object in the scene.

## Frustum Rejection with AABB's

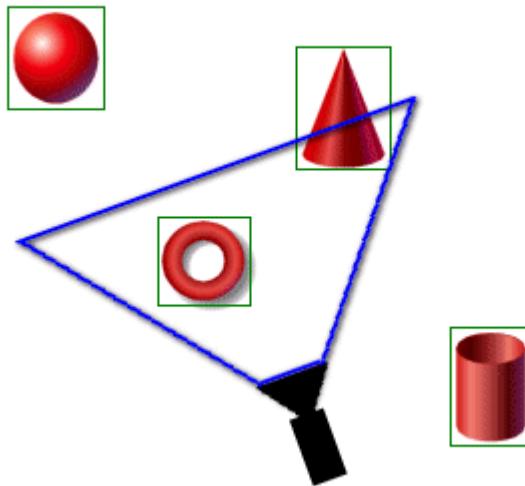


Figure 4.25

We can see in Fig 4.25 that both the sphere and cylinder objects do not require any rendering since they are outside the frustum. This can be determined using a simple box/frustum test for each object rather than  $N$  polygon/frustum tests. If these two objects had 20,000 polygons each, we have just avoided transforming, lighting, and ultimately rejecting 40,000 polygons. Remember that T+L takes place at the vertex level, so this is a substantial savings indeed.

As the cone bounding box partially intersects the frustum, we will just render the object and let the pipeline cull and clip the object as usual. Likewise for the torus as its bounding box is completely inside the frustum. In fact, we could theoretically speed things up in the torus case by telling the pipeline not to bother trying to test any polygons for clipping, but for the time being we will not worry about this concept.

For now our approach will be to render any mesh whose AABB is not fully outside of the frustum. Any objects with intersecting AABBs will be rendered as we normally do. In the next course in this series we will explore ways to optimize this system using spatial partitioning and scene graphs. In that case the scene itself will be divided into a hierarchical arrangement of bounding volumes so that even

objects themselves can be aggregated and tested against the frustum (sort of like bounding volumes around groups of bounding volumes). Again, for now we will stick to the basics.

We know at this point that we will need our meshes and/or objects to store a bounding box. The coordinate space we choose for the intersection tests requires some consideration. If we perform the test in world space, then we will need to convert the frustum planes to world space. We will also have to recalculate the extents of the AABB if the object moves or rotates (because the AABB will be described in world space units). We can see in Fig 4.26 how a single cone mesh would have different sized and shaped AABBs depending on its position and orientation in the world. Certain objects can rotate without altering the shape of the AABB (a sphere for example). In that case we could get away with not recalculating the bounding box, but simply updating its position when the object is moved.

### Axis Aligned Bounding Boxes (Top Down View)

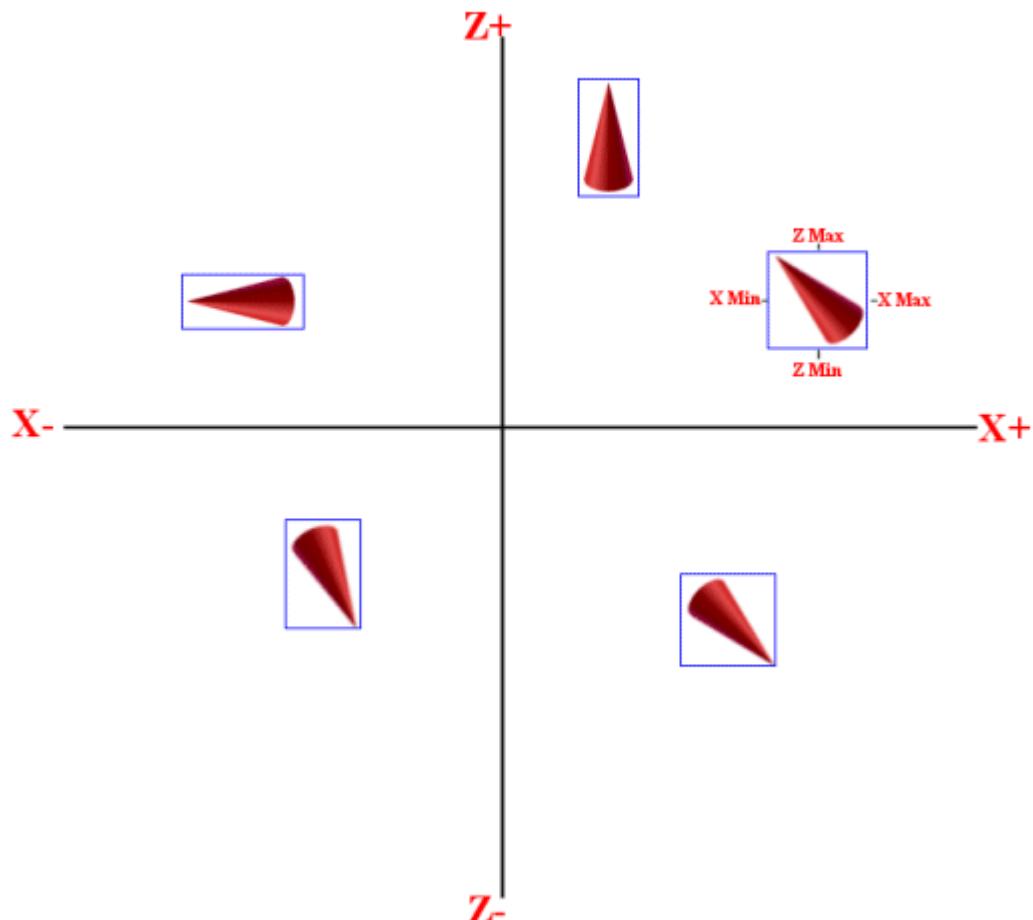


Figure 4.26



Recalculating an AABB can be costly if the mesh consists of thousands of vertices. This is because calculating an AABB requires looping through each vertex and recording the highest and lowest X, Y, and Z vertex extents. Of course, this is only an issue if the object is dynamic -- and many objects in a scene are not. In the case of our terrain for example, we divided the entire terrain into a grid of smaller meshes. Each one of these meshes could have its AABB calculated at application start-up and never require re-calculation because the terrain meshes never move. Since the terrain meshes are defined in world space to begin with, storing their AABBs in world space and performing the frustum test in world space is probably the best choice. We will simply need to extract the frustum planes from the projection matrix and convert them into world space just once at the beginning of each frame. We can then reuse them for testing all world space AABBs. Note that there are faster solutions for recalculating an AABB after a rotation, but the result will generally be a less tightly fit bounding volume which can result in false positives on the intersection test. (Everything is a tradeoff.)

For high polygon dynamic objects, recalculating a world space AABB can become expensive, so a model space AABB (which will never change) might be a better choice. That is, you would build the bounding box using the model space (un-transformed) vertex positions. Of course, for this to be useful we would need to transform the frustum planes into model space using the inverted mesh world matrix to perform the intersection test. Depending on the number of such objects in your scene, this can also become expensive. The more common solution for dynamic objects is to use a bounding sphere since it will not need to be recalculated if the object rotates. Spheres are very fast to test, but do not generally provide as tight a fit as a box. The result is an increase in false positives for intersection tests and thus rendering objects that are, in reality, fully outside the frustum. A common solution to this problem is to perform multiple tests. For example, one might do a sphere/frustum test first for rough culling. Since a sphere test is faster than a box test, this is a good first choice. If the sphere test indicates an intersection, a second test can be done using a tighter fitting bounding volume (like a box) to see if the intersection result was indeed accurate.

Even view space frustum testing is possible if desired, although it is not very commonly used. By and large the preferred choice is to use world space intersection testing whenever possible since it requires no transformations to take place prior to the test. This is the method we will discuss here in this lesson and the one we will use in our lab projects.

## Calculating an AABB

Calculating an AABB is a straightforward process regardless of whether we are calculating it in model space, world space, or even view space. An AABB can be stored using just two 3D vectors (6 floats) which keep track of the minimum and maximum X, Y, and Z components of the mesh respectively. The following snippet of code shows how we calculate the bounding box for each of our terrain meshes in our demonstration project. Because the terrain vertices are already stored as world space vertices, there is no need for us to multiply the vertex position by the mesh world matrix. However, if each mesh was defined in model space and we wanted to calculate a world space bounding box for it, we would first need to transform the vertex by the world matrix. Then the code would be identical to that shown.



```
// Calculate the mesh bounding box extents
m_BoundsMin = D3DXVECTOR3( 999999.0f, 999999.0f, 999999.0f );
m_BoundsMax = D3DXVECTOR3(-999999.0f, -999999.0f, -999999.0f);

for ( ULONG i = 0; i < m_nVertexCount; ++i )
{
    D3DXVECTOR3 * Pos = (D3DXVECTOR3*)&m_pVertex[i];
    if ( Pos->x < m_BoundsMin.x ) m_BoundsMin.x = Pos->x;
    if ( Pos->y < m_BoundsMin.y ) m_BoundsMin.y = Pos->y;
    if ( Pos->z < m_BoundsMin.z ) m_BoundsMin.z = Pos->z;
    if ( Pos->x > m_BoundsMax.x ) m_BoundsMax.x = Pos->x;
    if ( Pos->y > m_BoundsMax.y ) m_BoundsMax.y = Pos->y;
    if ( Pos->z > m_BoundsMax.z ) m_BoundsMax.z = Pos->z;
}
```

As you can see in the above example code, we represent the bounding box using two 3D vectors: one to store the minimum extents of the bounding box and one for the maximum extents. We could subsequently select components from each vector to describe the eight corner points of the box.

Notice how we set the minimum vector components initially to very high values and set the maximum vectors components to very small values. We then loop through every (world space) vertex in the mesh and test each X, Y, and Z component against the corresponding component in both the minimum and maximum vectors. If for example we find that the vertex X component is larger than the current maximum vector X component, then the vertex X component will become the new maximum X component. Likewise, if the vertex X component is smaller than the currently stored X component in the minimum vector, then this becomes the new minimum X component. We do this for all vertices and all components so that when the loop ends, we have stored the minimum and maximum extents of the mesh along all three coordinate system axes. These two vectors now represent a world space axis aligned bounding box and all the mesh vertices will be contained inside. Again, remember that if you want a world space bounding volume and your vertices are defined in model space, then you will need to multiply each vertex by the object's world matrix before performing the component tests above.

### Camera Space Frustum Plane Extraction

Now we will need to learn how to extract the frustum planes for use in intersection testing. Before continuing, it is important that you fully understand the projection matrix topics discussed in Lessons 1 and 2, so refer back to those discussions if you are feeling a little rusty.

This discussion also assumes that you know what a plane is, so we will not cover that topic in great detail. However, we should note that there are two popular methods for storing plane representations. The first form stores the plane normal and a point known to be on that plane. The second form stores the plane normal and a distance to that plane from the origin of the coordinate system. For example, if we have a plane in world space, the distance tells us how close that plane passes by the world origin



(0,0,0). If a plane passes through the origin of the coordinate system then the distance to the plane is 0. This second form is the one we will be using for frustum extraction.

The great thing about the projection matrix is that it holds all of the plane information for the view frustum. This means that instead of having to pump every vertex in our scene through the projection matrix to see if it is visible or not, we can extract the frustum planes from the projection matrix so that they are in camera space (or world space, but more on that later) and then use those planes for fast intersection testing on the CPU.

Let us look at an example projection matrix that has a near plane of 10.0 and a far plane of 100.0 with an FOV of 60 degrees along the Y axis. As discussed in Lesson 1, the X axis FOV will be slightly different to compensate for screen distortion caused by the aspect ratio of the current screen or viewport dimensions.

To create this projection matrix using D3DX:

```
D3DXMatrixPerspectiveFovLH(&proj_m, D3DXToRad(60.0f), 1.333333f, 10.0f, 100.0f);
```

Just to refresh your memory, the following shows us how the values of the projection matrix are calculated by the function. The actual projection matrix follows.

$$\text{Ratio} = \frac{\text{FarPlane}}{(\text{FarPlane} - \text{NearPlane})} = \frac{100.0}{90} = 1.11111111$$

$$-\text{Ratio} * \text{NearPlane} = -(10 * 1.11111111) = -11.11111111$$

$$\text{AspectRatio} = \frac{\text{ScreenWidth}}{\text{ScreenHeight}} = \frac{800}{600} = 1.33333333$$

$$m_{11} = \frac{1}{\frac{\tan(60/2)}{\text{AspectRatio}}} = 1.299038109$$

$$m_{22} = \frac{1}{\tan(60/2)} = 1.732050808$$



$$M = \begin{bmatrix} X & Y & Z & W \\ 1.299038109 & 0 & 0 & 0 \\ 0 & 1.732050808 & 0 & 0 \\ 0 & 0 & 1.111111111 & 1 \\ 0 & 0 & -11.11111111 & 0 \end{bmatrix}$$

In the projection matrix, the first three rows of each column represent a vector of varying magnitude that is aligned with the camera space X, Y, and Z axes respectively. You can see for example, that the first three rows of column 1 is a vector that points down the positive X axis in camera space.

**X Vector = (1.299038109, 0, 0)**

This vector is fully aligned with the camera space x axis (1, 0, 0) and the only difference is that it is not a unit length vector. If we were to normalize this vector so that it had a length of 1.0, it would be exactly the same as the X axis in camera space:

**Normalized X Vector = camera space Z Axis = (1, 0, 0)**

This is also true for the first three rows of the Y column. It is a non-unit length vector exactly aligned with the Y axis in camera space.

The first three rows of the W column represent a unit length vector aligned with the camera space Z axis.

**W Vector = camera space Z Axis = (0, 0, 1)**

It is important to us that the **X** and **Y** vectors are not unit length, because these vectors hold vital information about the relationship they have with each other. The **W** vector describes a movement of 1.0 unit down the cameras space **Z** axis, and the **Y** vector (for example) describes the ratio of movement down the **Y** axis for each unit of **W**. In other words, the direction of the plane normals are described as ratios of movement along the **X** or **Y** axes, in relation to one unit of movement along the **Z** axis. If you do not recall why this would be the case, just remember that **W = Z axis**.

Furthermore, the fourth row of each column can be used to extract the plane distance so that we will then have our complete set of plane information. For the moment however, forget about the fourth row of each vector since the Left, Right, Top, and Bottom clip planes all have distances of 0 in camera space. We will be using only the first three rows (which represent the plane normals) for the time being.

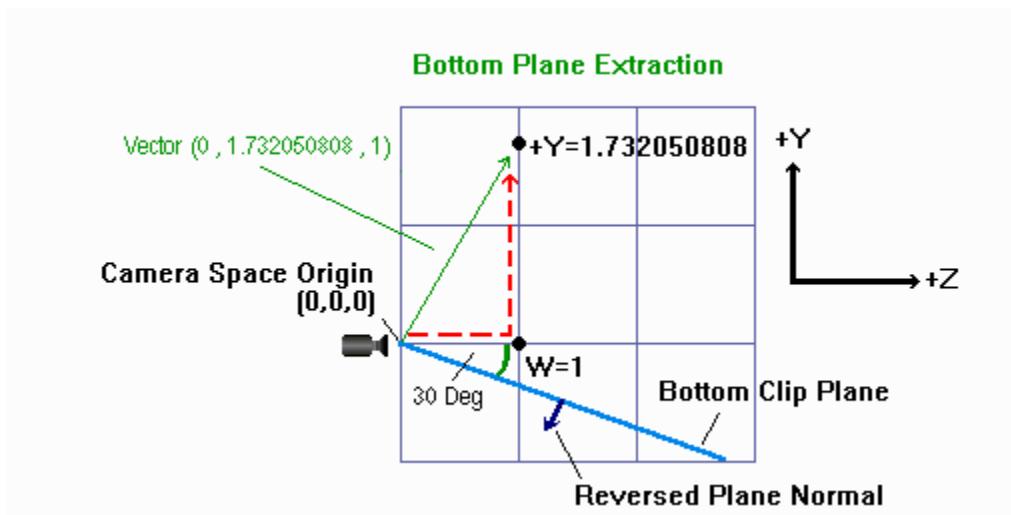
Let us first see how we could extract the Bottom frustum plane normal of our 60 degree FOV frustum so that the normal is facing outwards. The following line of pseudo-code creates an un-normalized



plane normal for the bottom frustum plane by adding the W column of the projection matrix to the Y column of the projection matrix. We will discuss why we negate the result in a moment.

### **BottomPlane->Normal = - (Column\_4 + Column\_2)**

Fig 4.27 should make everything clear. In the diagram we are looking down the negative X axis in camera space. The camera space Y axis is assumed to run bottom to top and the camera space Z axis from left to right.



**Figure 4.27**

First look at the camera space origin, and notice how the W vector of the projection matrix  $(0,0,1)$  represents a point at  $(0,0,1)$  along the camera space Z axis. Also note how the Y vector of our projection matrix represents a point at  $(0, 1.732050808, 0)$  along the camera space Y axis. If we combine these two vectors using the calculation shown above, we end up with the vector:

$$\begin{matrix} X & Y & W \end{matrix} \\ \text{Column}_4 + \text{Column}_2 = (0, 0, 1) + (0, 1.732050808, 0) = (0, 1.732050808, 1)$$

We have now created a 3D coordinate in camera space. If we forget about the X coordinate for now because it is 0, and plot this point on some graph paper with a side-on view of camera space (Fig 4.27), we can see that this coordinate is plotted at  $Y=1.732050808$  and  $Z=W=1$ . Remember that a coordinate is really just a direction vector that describes the direction and distance to a point from the origin of the coordinate system (camera space in this example). The *green arrow* in the above diagram shows the direction this vector is facing and we can see that it is in fact the *un-normalized plane normal* for the bottom frustum plane. A plane is always perpendicular to its normal, so to test this, if we rotate the normal around by 90 degrees, we should have a line representing the bottom frustum plane in the above diagram. We can see that this is the case; the blue line in the above diagram is at 90 degrees to the green direction vector. The angle between this plane and the camera space Z axis is exactly half of our FOV. This is correct because when we extract the Top plane, which will also be at an angle of 30



degrees from the camera Z axis, together they will form an angle of 60 degrees (30 degrees top and bottom) off the camera space Z axis. So indeed we can see that the vector that we have just created (the green arrow) describes the orientation of the Bottom frustum plane. It is not yet unit length of course, but if we were to normalize it, we would have the plane normal such that the ‘front’ of the Bottom frustum plane would be facing inwards (pointing towards the Top plane actually).

Our preference in this lesson will be that our frustum planes have their normals face outwards instead of inwards. To do this, we simply negate the resulting vector, which has the effect of flipping the green line in the above diagram (see the direction of the blue arrow labelled ‘*Reversed Plane Normal*’). That is why we used the minus sign in the initial formula. After normalizing this inverted vector, we would have our outward facing plane normal (blue arrow) for the Bottom frustum plane. Note that the choice to flip the direction of the plane normal vectors during extraction is a matter of preference only. We did this because when the frustum planes all point outwards and we test a point against the plane, the results of the dot products are more intuitive. If any point is found to be outside a frustum plane, it will have a positive distance returned.

This technique can be repeated to extract all six clip planes. It should be clear from the diagram, that if we *subtracted* column 2 from column 4, rather than adding them like we did above, we will end up with a coordinate:

$$\text{Column}_4 - \text{Column}_2 = (0, 0, 1) - (0, 1.732050808, 0) = (0, -1.732050808, 1)$$

This new vector is the green arrow in Fig 4.28. We can see that it works exactly the same way as the previous diagram. This time, subtracting the Y vector instead of adding it to the W vector, returned the un-normalized, inward facing plane normal for the Top clip plane.

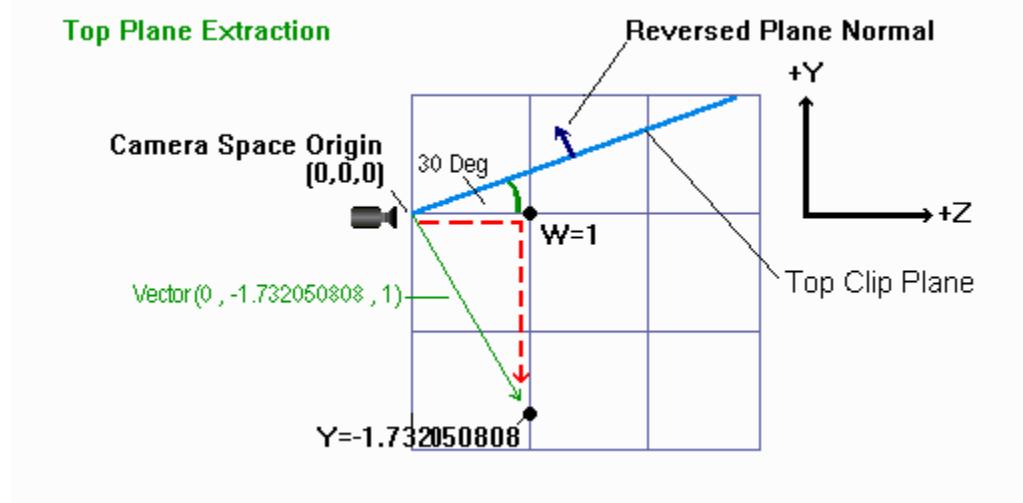


Figure 4.28



Since we prefer our plane normals to face outward, we flip the direction 180 degrees by negating the result so that the green arrow in the above diagram would face in the same direction as the blue arrow. So the extraction of the top frustum plane becomes:

#### **TopPlane->Normal =-(Column\_4 – Column\_2)**

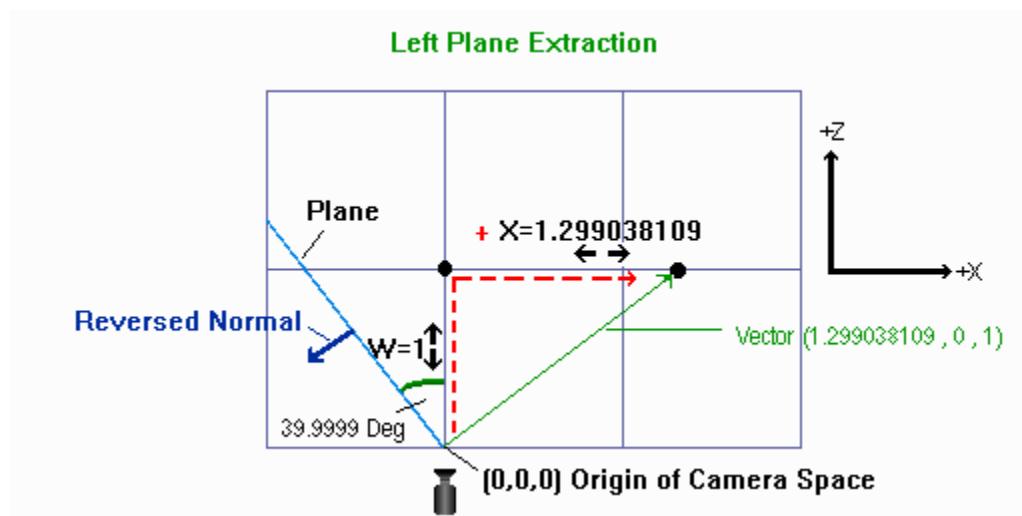
Keep in mind that this vector is not normalized, so if we need normalized planes we will have to do this next.

Take some time to study how the above examples worked. If you have fully understood everything discussed, you should be able to figure out how the Left and Right frustum plane normals could be extracted. In this case we add or subtract the X column of the projection matrix from the W column instead (and invert the result for outward facing planes).

**LeftPlane->Normal =-(Column\_4 + Column\_1)**

**RightPlane->Normal =-(Column\_4 – Column\_1)**

Fig 4.29 shows how the Left plane extraction works. The diagram looks at camera space from a top-down view such that the camera space Z axis runs from bottom to top and the camera space X axis runs from left to right.



**Figure 4.29**

By adding column 1 of the matrix to column 4 of the matrix, we create a vector that is perpendicular to the left clip plane (the green arrow). Because we have created the vector using column 1 instead of column 2, we end up with a left clip plane that is 39.99 degrees from the camera space Z axis. Remember that this describes only *half* the FOV since the Right clip plane will also be at an angle of 39.99 degrees from the camera space Z axis.



So far we have looked only at extracting the plane normal from the matrix. However, the distance to the plane is also calculated as part of the process.

#### **Left Plane->Normal=-(Column\_4 + Column\_1)**

In this case we are in fact extracting four pieces of information about the plane. If we look at the addition at the component level, we would see this more clearly:

Left Plane->Normal.X= - (m14 + m11)  
Left Plane->Normal.Y= - (m24 + m21)  
Left Plane->Normal.Z= - (m34 + m31)  
**Left Plane->Distance = - (m44 + m41)**

We have not discussed how it is exactly that the fourth row contains the distance parameter, and for good reason. Because plane extraction from the projection matrix extracts the planes in camera space, the camera is the center of the origin. As the center of camera space is the origin for the Left, Top, Right, and Bottom planes (our view cone starts at camera (0,0,0)), it means the distance for these planes will always be 0.

The Near and Far planes however are perpendicular to the camera, and are also some distance away. In our example so far, we know that our near plane is at a distance of 10 units and our far plane is at a distance of 100 units (these were our settings when we created the matrix). Let us look at the how we extract the far plane first:

#### **Far Plane->Normal = -(Column\_4 – Column\_3)**

Do you see the recurring pattern here?

Just as with the other planes, we use column 4 again, but this time use column 3 as the vector to subtract from it, because this column contains the Z information for the near and far planes. If we write out the far plane extraction formula above, we can more clearly see what is happening:

**Far Plane->Normal.x = - (m14 - m13)**  
**Far Plane->Normal.y = - (m24 - m23)**  
**Far Plane->Normal.z = - (m34 - m33)**  
**Far Plane->Distance = - (m44 - m43)**

Have another look at the projection matrix we are working with:



$$M = \begin{bmatrix} 1.299038109 & 0 & 0 & 0 \\ 0 & 1.732050808 & 0 & 0 \\ 0 & 0 & 1.111111111 & 1 \\ 0 & 0 & -11.11111111 & 0 \end{bmatrix}$$

First let us see what subtracting the first three rows of W and Z produces as a direction vector:

$$-(\text{Column}_4 - \text{Column}_3) = -( (0,0,1) - (0,0,1.111111111) ) = (0, 0, 0.11111111)$$

This result certainly seems correct, for if the vector were normalized it would become:

$$\text{Normalize}( (0, 0, 0.111111) ) = (0, 0, 1)$$

This is just what the far plane normal should be -- facing down the camera space Z axis.

Now let us see what happens when we try to extract the distance to this plane (which we know to be 100 units away from the camera):

$$\text{Distance} = -(m_{44} - m_{43}) = -(0 - -11.11111111) = -11.11111111$$

That is not correct at all now is it?

Remember that the plane normals must be normalized in order to make them unit length. What we have to do then is, in addition to normalizing the plane normals, normalize the distance as well. Therefore, we could say that whilst the projection matrix does indeed contain the frustum plane information, this plane information on the whole is not normalized. When we normalize the plane, the plane normals we have extracted will become unit length and the plane distances we have extracted will also be scaled by the same amount. This way they will accurately describe the plane distance. If we do this for all six planes, we will finally have our six normalized frustum planes.



## Normalizing a Plane

Recall from Lesson 1 that normalizing a vector scales the vector so that it has a length of 1 while still keeping the individual components of the vector in proportion with each other. To calculate the length of a vector we used the following formula:

$$\text{VectorLength} = \sqrt{X^2 + Y^2 + Z^2}$$

If for example, we had a vector of (3,0,9), the length of that vector would be calculated as:

$$\text{VectorLength} = \sqrt{(3 * 3) + (0 * 0) + (9 * 9)} = 9.4868329$$

This vector is quite clearly not a unit vector because it has a length greater than 1.0. In order to make this vector unit length, we divide each component of the vector by its length:

$$\text{UnitVector} = \left( \frac{3}{9.4868329}, \frac{0}{9.4868329}, \frac{9}{9.4868329} \right) = (0.316227768, 0, 0.9486833)$$

Just to verify that this worked and that we indeed have a unit normal, let us calculate its magnitude:

$$\text{VectorLength} = \sqrt{(0.316227768 * 0.316227768) + (0 * 0) + (0.9486833 * 0.9486833)} = 1.0$$

We now know how to normalize a vector, but to normalize a plane we must also normalize the distance value. This is not a problem. All we have to do is divide the distance value by the length of the direction vector also, because the direction vector and distance value are proportional to each other in the matrix. So, we will get the length of the plane vector that we extract from the projection matrix and divide this vector by its own length to normalize it. This creates a unit length frustum plane normal. Then we divide the plane distance by the vector length and we are done. Let us try that now with our Far plane information:

<b>Far Plane-&gt;Normal.x = - (m14 - m13) = - (0 - 0)</b>	<b>=0</b>
<b>Far Plane-&gt;Normal.y = - (m24 - m23) = - (0 - 0)</b>	<b>=0</b>
<b>Far Plane-&gt;Normal.z = - (m34 - m33) = - (1 - 1.11111111)</b>	<b>=0.11111111</b>
<b>Far Plane-&gt;Distance = - (m44 - m43) = - (0 - -11.11111111)</b>	<b>= -11.11111111</b>

First we normalize the plane direction vector (0, 0, 0.11111111):

$$\text{Vector Length} = \sqrt{(0 * 0) + (0 * 0) + (0.11111111 * 0.11111111)} = 0.11111111$$

Now we need to divide each component of the vector by this length to normalize it to a proper plane normal:



$$\text{Plane Normal} = \left( \frac{0}{0.11111111}, \frac{0}{0.11111111}, \frac{0.11111111}{0.11111111} \right) = (0,0,1)$$

Finally, we divide the distance by the original vector length (0.11111111) and see what happens:

$$\text{Distance} = \frac{-11.11111111}{0.11111111} = -100$$

Now there is our correct plane distance. In case you are wondering why it is a negative number, the distance is always negative if we are behind the plane. Because we have flipped our far plane so that it faces away from the camera, the origin of camera space is indeed 100 units behind the plane.

So frustum plane extraction is really just a case of adding/subtracting the  $X$ ,  $Y$  and  $Z$  vectors from the projection matrix to/from the  $W$  vector of that same matrix. Then we normalize the planes by dividing the vector by its length, and divide the plane distance by the vector length as well.

Before we list all of the extraction formulas for each plane, for completeness, let us look at how the final plane extraction works (the Near plane). We know that it should be 10 units away from the camera in camera space and, unlike the far plane, have its normal facing *towards* the camera.

The near plane is actually an exception to the approach we have been using, in that we do not have to use the  $W$  column vector for addition/subtraction. In fact it is the easiest case, because the full set of plane information is already contained inside the  $Z$  vector of the projection matrix. All we have to do is extract and normalize the 3<sup>rd</sup> column of the matrix and we have our near plane.

Our 3<sup>rd</sup> column looked as follows:

$$\begin{bmatrix} 0 \\ 0 \\ 1.11111111 \\ -11.11111111 \end{bmatrix}$$

If you have forgotten what these numbers represent, then refer back to Lesson 2 where we talked about how to set up this column in detail. The plane can be extracted simply by doing the following:

**Near Plane = -(Column\_3)**

Therefore:



**Near Plane->Normal.x=-m13**  
**Near Plane->Normal.y=-m23**  
**Near Plane->Normal.z=-m33**  
**Near Plane->Distance =-m43**

All we have to do is normalize the above information, and we will have a plane normal of (0, 0, -1) and a plane distance of 10.0.

First we extract the normal, which will of course be **(0, 0, -1.1111111)**. Remember that the value **1.1111111** was our scaling ratio to map the input z value to a Z-Buffer value. Also recall that the 4<sup>th</sup> row in the column is the actual distance to the near plane, multiplied by the ratio. Therefore, both of these values need to be divided by the scaling ratio to reduce the normal to a unit vector and reduce the distance back into a camera space distance (which should result in a distance of 10.0).

$$\text{Vector Length} = \sqrt{0.0^2 + 0.0^2 + -1.1111111^2} = 1.1111111$$

$$\text{Near Plane->Normal} = \left( \frac{0}{1.1111111}, \frac{0}{1.1111111}, \frac{-1.1111111}{1.1111111} \right) = \mathbf{(0, 0, -1)}$$

$$\text{Near plane->Distance} = \frac{11.1111111}{1.1111111} = \mathbf{10.0}$$

### Frustum Extraction Code

The following code snippet assumes that we have a plane structure defined as follows:

```
struct PLANE
{
    D3DXVECTOR3 Normal;
    FLOAT Distance;
};
```

It also assumes that we have allocated an array of six PLANE structures to hold the six frustum planes.

```
PLANE Planes[6];
```

Finally, our code assumes that the projection matrix has already been set up correctly.

```
D3DXMATRIX M; //our projection matrix
```



The following code extracts and normalizes the planes. If you prefer inward facing planes then simply remove the minus sign from the beginning of each line.

```
// Left clipping plane
Planes[0].Normal.x = -(M._14 + M._11);
Planes[0].Normal.y = -(M._24 + M._21);
Planes[0].Normal.z = -(M._34 + M._31);
Planes[0].Distance = -(M._44 + M._41);

// Right clipping plane
Planes[1].Normal.x = -(M._14 - M._11);
Planes[1].Normal.y = -(M._24 - M._21);
Planes[1].Normal.z = -(M._34 - M._31);
Planes[1].Distance = -(M._44 - M._41);

// Top clipping plane
Planes[2].Normal.x = -(M._14 - M._12);
Planes[2].Normal.y = -(M._24 - M._22);
Planes[2].Normal.z = -(M._34 - M._32);
Planes[2].Distance = -(M._44 - M._42);

// Bottom clipping plane
Planes[3].Normal.x = -(M._14 + M._12);
Planes[3].Normal.y = -(M._24 + M._22);
Planes[3].Normal.z = -(M._34 + M._32);
Planes[3].Distance = -(M._44 + M._42);

// Near clipping plane
Planes[4].Normal.x = -(M._13);
Planes[4].Normal.y = -(M._23);
Planes[4].Normal.z = -(M._33);
Planes[4].Distance = -(M._43);

// Far clipping plane
Planes[5].Normal.x = -(M._14 - M._13);
Planes[5].Normal.y = -(M._24 - M._23);
Planes[5].Normal.z = -(M._34 - M._33);
Planes[5].Distance = -(M._44 - M._43);

// Normalize the planes
for ( int i = 0; i < 6; i++ )
{
    // Get magnitude of Vector
    float denom = 1.0f / D3DXVec3Length(&Planes[i].Normal);
    Planes[i].Normal.x *= denom;
    Planes[i].Normal.y *= denom;
    Planes[i].Normal.z *= denom;
    Planes[i].Distance *= denom;
}
```



It is worth mentioning that you are not required to normalize the planes if you all you want to do is classify a point against a plane to see if it is in front or behind. The sign of the classification will be correct even if the planes are not normalized -- which is all you need for simple front/back tests. If you need to know the correct distance to the plane, then they will need to be normalized.

### World Space Frustum Plane Extraction

As mentioned, extracting the planes from the projection matrix results in planes defined in camera space. This means that the distance value of the Top, Bottom, Left and Right planes will always be zero. If this is the case, then why extract it at all? Why not just set it to 0 in the plane structure automatically? We do this is because we can combine the **view matrix** with the **projection matrix**, and without any alteration to our extraction code, extract the frustum planes in world space.

The problem with our frustum planes being in camera space is that testing geometry or AABBs against these planes requires that we transform all objects from world space into camera space. With even a moderately sized scene, this can mean pushing a lot of vertices through the view matrix. With the frustum planes in world space, we do not have to do this since our objects (and their bounding volumes) are generally defined in world space to begin with. This is far more optimal.

**NOTE:** The code to extract the frustum planes in world space is exactly the same as the code we have already created. All we have to do is combine the View Matrix and the Projection Matrix (multiply them together) prior to extracting the planes. This will automatically extract planes in world space. This is also why, in the previous code, we extracted the distance for the Top, Bottom, Left, and Right planes. In world space, the Left, Top, Right, and Bottom frustum planes (which always pass through the camera origin) may not be anywhere near the origin of world space. They might also be rotated at an angle so that the camera space Z axis is not aligned to the world space Z axis. In this situation the distance of each plane will be the distance from the plane to the origin of world space. Thus, the distances of all planes will most often not be zero anymore.

To extract the frustum planes in world space, all we have to do is concatenate the view and projection matrices before running the plane extraction code:

**D3DXMatrixMultiply(&M , &ViewMatrix, &ProjMatrix);**

**M** is the matrix that stores the result of the matrix multiplication.

What you must remember is that if the camera moves, we have to re-extract the frustum planes, because they move and rotate with the camera when the view matrix is updated. Usually this means extracting the frustum planes once each frame, or at least every time the view matrix is changed. This is not necessary if you are extracting planes in camera space since the projection matrix usually remains unchanged throughout the life of the application.

Combining the view matrix with the projection matrix has the effect of rotating and translating the plane information using the camera's position and orientation. Before we finish up here, let us see this



working using some real numbers. We will use the same 60 degree projection matrix from our previous example, and we will create a new view matrix to test it out.

For this example, let us set up a simple view matrix that will make things easy to follow. We will position the camera at (0,0,0) in world space (camera space origin=world space origin), but will rotate the camera 45 degrees to the right. The Up vector will be aligned with the world Up vector (0,1,0).

**Note:** Remember that all vectors MUST be unit length vectors in the view matrix

$$\begin{array}{cccc} \text{RIGHT} & \text{UP} & \text{LOOK} \\ \left[ \begin{array}{cccc} 0.707107 & 0 & 0.707107 & 0 \\ 0 & 1 & 0 & 0 \\ -0.707107 & 0 & 0.707107 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array}$$

In our example view matrix, the Look vector is pointing in equal proportion along the X and Z axes. Since the Look vector is actually the camera space Z axis, you should be able to see that the angle between the world Z Axis (0,0,1) and the camera space Z Axis (0.707107, 0, 0.707107) is 45 degrees. The Up vector (camera space Y) is the same as the world Y axis, since we have not pitched or rolled the camera at all. The angle between the Right vector (camera space X) and the world space X axis is also 45 degrees -- as it should be.

When we multiply the above matrix with our projection matrix, the frustum plane normals should be rotated 45 degrees. We will test this out by extracting the Left plane to make sure that this definitely works out.

$$\begin{array}{c} \begin{array}{cccc} \text{RIGHT} & \text{UP} & \text{LOOK} & \\ \left[ \begin{array}{cccc} 0.707107 & 0 & 0.707107 & 0 \\ 0 & 1 & 0 & 0 \\ -0.707107 & 0 & 0.707107 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \times \begin{array}{c} X \\ 1.299038109 \\ 0 \\ 0 \end{array} \end{array} \\ \text{View Matrix} \quad \quad \quad \text{Projection Matrix} \end{array}$$

The resulting combined matrix is shown next:



$$\begin{bmatrix} X & Y & Z & W \\ \begin{bmatrix} 0.918559 & 0 & 0.785674 & 0.707107 \\ 0 & 1.732051 & 0 & 0 \\ -0.918559 & 0 & 0.785674 & 0.707107 \\ 0 & 0 & -11.111111 & 0 \end{bmatrix} \end{bmatrix}$$

While the numbers in the above matrix may not immediately leap out at you as explaining what has happened, it is easy to verify our theory if we plot the X, Y, and W values on some graph paper. However, if you look at the new W vector, it should make things clear. Remember that this vector in the projection matrix points straight down the camera Z axis ( $W=0,0,1$ ). In camera space this is always the camera Look vector. In the resulting matrix however, it has now been rotated to match the Look vector in the view matrix, which makes clear that it has been rotated 45 degrees. The same is true for the X, Y, and Z vectors – they have all been rotated 45 degrees as well.

Fig 4.30 shows an example of a Left frustum plane extraction. The image on the left is the camera space version and it is identical to the one we saw earlier. On the right, we have plotted the vectors from the new combined matrix to see how the Left plane has been rotated. Recall that the normal vector and the distance for the left clip plane are still extracted as:

$$\text{Left Plane-} > \text{Normal.X} = -(\text{m14} + \text{m11}) = \mathbf{1.62566}$$

$$\text{Left Plane-} > \text{Normal.Y} = -(\text{m24} + \text{m21}) = 0$$

$$\text{Left Plane-} > \text{Normal.Z} = -(\text{m34} + \text{m31}) = \mathbf{-0.211452}$$

$$\text{Left Plane-} > \text{Distance} = -(\text{m44} + \text{m41}) = 0$$

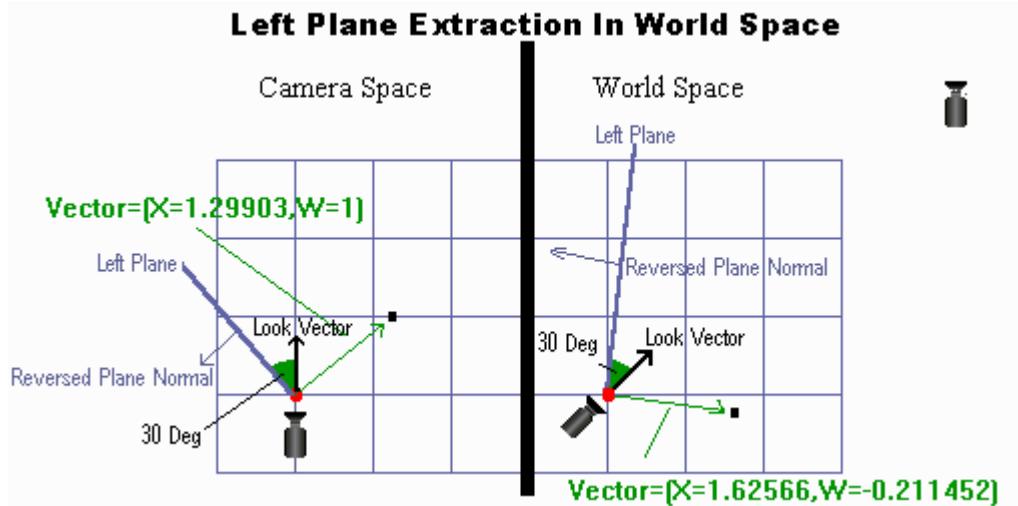


Figure 4.30



The green arrows in Fig 4.30 show the plane normal stored in the matrix while the plane itself is shown by the thick blue lines. Notice in the image on the right (where we see the rotated camera) that the green vector has also been rotated – indicating the left plane has been rotated as well.

Before finishing up here, it is worth mentioning that you can extract the frustum planes for model space testing using the following set of combined matrices:

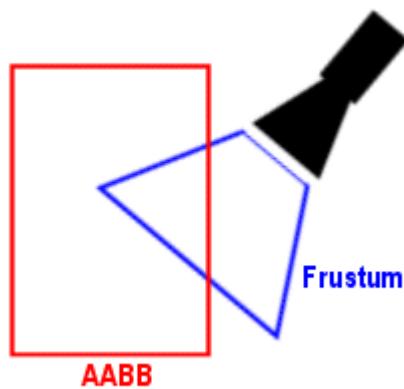
***WorldMatrix\*ViewMatrix\*ProjectionMatrix***

The world matrix would belong to the object whose local space you wish to run the test in.

### Frustum Culling an AABB

In order to test whether or not an AABB is within the frustum, we have to check each of the six frustum planes against the bounding box. You might be thinking that in order to test whether or not an AABB is within the frustum, all we have to do is check the eight corner points of the box to see if any of them are behind all the frustum planes and therefore at least partially contained within the frustum. In fact, this is not sufficient, as Fig 4.31 demonstrates:

**The AABB intersects the frustum but all  
of the AABB's corner points are outside**



**Figure 4.31**

In Fig 4.31, although the corner points of the bounding box are outside the view frustum, the bounding box would still be considered partially visible, as one of its edges intersects the frustum.

The solution is to test each plane against a single point on the bounding box. This point will be one of the corners of our bounding box, but which corner we use depends on the orientation of the current frustum plane being tested. If we imagine that the AABB is completely outside the frustum and just



about to intersect the current plane being tested, the point we wish to test would be the first corner point on the box that would intersect it. This point is called the *negative* or *near* point.

What we will do is examine each component of the plane normal, and select an appropriate AABB corner as the near point. We can then test this near point against the plane and if it is outside the plane then we know for a fact that the entire bounding box is also outside the frustum. This near point selection process will happen once for each plane, because each plane will have a different orientation. The basic concept is as follows:

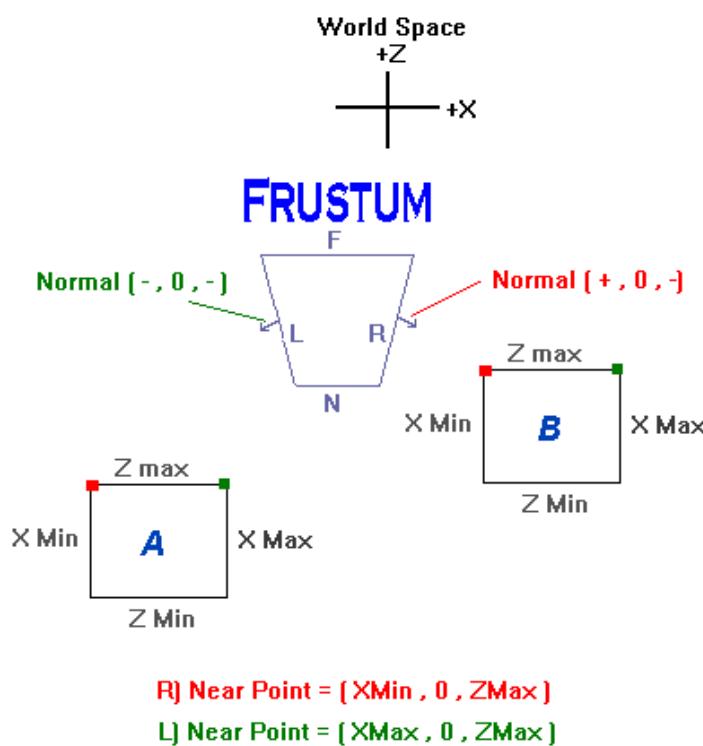
For each frustum plane, we examine the components of the plane normal:

- If the 'x' component of the plane normal is negative, then we use the bounding box's x maximum point as our near point's x component. Otherwise, we use the bounding box's x minimum component.
- If the 'y' component of the plane normal is negative, then we use the bounding box's y maximum point as our near point's y component. Otherwise, we use the bounding box's y minimum component.
- If the 'z' component of the plane normal is negative, then we use the bounding box's z maximum point as our near point's z component. Otherwise, we use the bounding box's z minimum component.

At this point, we will have constructed a near point (3D vector) to test against the current frustum plane. If this point is in front of the current plane (remember that frustum planes point outwards), then we can exit the test immediately. We do not have to check the other planes in this case, because the near point being outside (in front) of a plane tells us that the entire AABB must be outside the frustum.

If the near point is behind the plane, then we must continue to test the other planes. For every plane that we test, we have to build a new near point using the logic above. Again, if at any point the current near point is in front of the current plane being tested, then we can exit the test and know that the AABB is completely outside the frustum. If we test all six planes and do not find a near point that is in front of one of the planes, then this means that the AABB is at least partially inside the frustum. In our workbook example code, when this happens, the function returns a value of 'true' and the object will be rendered.

If you are having trouble visualizing this concept, the following diagram should help. It depicts two AABBs (A and B) and a set of six frustum planes.



Let us imagine that we first want to check the left (L) clip plane of the frustum and the camera is facing due north. In this case, the left clip plane's normal would be facing in the negative x direction, which means that we will use the AABB x maximum point as our near point x component. The same is true for the z component of the plane normal. This too would be a negative component, meaning we would use the AABB maximum z component as the near point's z component. (Forget about y for now, because we are looking top-down and the plane has no y orientation.) In this case then, the near point (corner point of the AABB) used will be in the top right of the box, as indicated by the green square (the left box in the diagram). Looking at this corner point on Box A and remembering that planes are infinite, you should be able to see that if this near point is in front of the left plane, the whole AABB must be as well -- and therefore there is no way that this AABB can be inside or even partially inside the frustum.

When testing the right plane, a different near point is used because the plane normal's x component now faces in a positive direction, which means we use the AABB's x minimum component as the near point's x component. Z is still facing the same way (negative), so we still use the AABB's max z component. This gives us the corner indicated by the red boxes in the diagram

Look at Box B and you should also be able to see that if the red corner is in front of the right plane, then the entire AABB must be also.



Let us do a quick run through for the bounding boxes depicted in the previous diagram. We will start with Box A. First we test the left frustum plane, which creates the green near point seen above. For Box A, the near point is in front of the left plane, so the box is not within the frustum and the function returns false. Next we test Box B. Once again, we test the left plane against the near point (green corner) and discover that this point is behind the left plane. This means it *might* be within the frustum. In this case, instead of returning from the function, we move on to test the next plane, which in this case is the right plane. Because we are testing the right plane now, a different near point is created (the red corner) and tested against the right plane. In this case, however, the near point of Box B is in front of the right plane, so the entire AABB must be in front also. This causes the function to return false. We will do this for all six planes, unless we find that the current near point is in front of the current plane being tested. In summary, if an AABB is within the frustum, then the near points generated for it during the test will be behind all six of the planes.

Let us now take a look at a function called IsBoxInFrustum, which will be passed a bounding box as two vectors (minimum and maximum extents) along with an array of six planes containing the frustum. The frustum planes should have already been extracted at this point and stored in this array using code similar to the extraction code discussed earlier. IsBoxInFrustum can be called while we are rendering to see whether an object about to be drawn can actually be seen given the current position and orientation of the camera represented by the input planes. If the function returns false, then we do not want to draw the object. The code is basically a bunch of conditional statements that build the near point for the current plane being tested. Once the near point is found, a simple dot product between the near point and the plane determines whether or not the near point is in front of or behind the plane:

```
bool IsBoxInFrustum(D3DXVECTOR3 *bMin, D3DXVECTOR3 *bMax, PLANE *FrustumPlanes)
{
    D3DXVECTOR3 NearPoint;
    PLANE *Plane=FrustumPlanes;

    for (int i=0;i<6;i++)
    {
        if (Plane->Normal. x > 0.0f)
        {
            if (Plane->Normal. y > 0.0f)
            {
                if (Plane->Normal. z > 0.0f)
                {
                    NearPoint. x = bMin. x; NearPoint. y = bMin. y; NearPoint. z = bMin. z;
                }
                else
                {
                    NearPoint. x = bMin. x; NearPoint. y = bMin. y; NearPoint. z = bMax. z;
                }
            }
            else
            {
                if (Plane->Normal. z > 0.0f)
                {
                    NearPoint. x = bMin. x; NearPoint. y = bMax. y; NearPoint. z = bMin. z;
                }
            }
        }
    }
}
```



```
        else
        {
            NearPoint. x = bMin. x; NearPoint. y = bMax. y; NearPoint. z = bMax. z;
        }
    }
else
{
    if (Plane->Normal. y > 0.0f)
    {
        if (Plane->Normal. z > 0.0f)
        {
            NearPoint. x = bMax. x; NearPoint. y = bMin. y; NearPoint. z = bMin. z;
        }
        else
        {
            NearPoint. x = bMax. x; NearPoint. y = bMin. y; NearPoint. z = bMax. z;
        }
    }
    else
    {
        if (Plane->Normal. z > 0.0f)
        {
            NearPoint. x = bMax. x; NearPoint. y = bMax. y; NearPoint. z = bMin. z;
        }
        else
        {
            NearPoint. x = bMax. x; NearPoint. y = bMax. y; NearPoint. z = bMax. z;
        }
    }
}

// near extreme point is outside, and thus
// the AABB is totally outside the polyhedron
if(D3DXVec3Dot(&Plane->Normal, &NearPoint) + Plane->Distance > 0)
    return false;

Plane++;
}

return true;
}
```

The camera class in our lab project contains very similar code to what we have explored here in these last few sections. A function called CCamera::CalcFrustumPlanes takes no parameters and extracts the planes from the camera view matrix and stores them in the CCamera::m\_Frustum array (an array of six plane structures). Although this function is called each frame to keep the world space frustum planes up to date, the camera maintains a Boolean variable called ‘m\_FrustumDirty’ which is set to true only when the projection matrix or the view matrix has been updated. The function will test this Boolean and re-extract the frustum planes only if one of these matrices has been altered. During the CTerrain::Render function, the bounding box of each terrain mesh is passed into the



CCamera::BoundsInFrustum function. The call will return true if the bounding box is inside or partially inside the frustum and we will know that the terrain submesh needs to be rendered. Be sure to look at the source code to the CCamera class for more details.

This concludes our coverage of frustum culling for this course. However, in the next course in this series, our frustum culling code will take a significant leap forward. Not only will it allow for more bounding volume types to be tested, but it will also add code that distinguishes between ‘fully inside’ and ‘partially inside’. This will be important for the types of hierarchical bounding volume structures we will be working with in our engine design. Finally, we will add some optimization code that minimizes redundant plane testing in scene hierarchies and also provides frame-to-frame coherence (i.e. the intersection status that resulted in the last frame is often going to be the same in the next frame – we can speed things up considerably with this in mind). Do not concern yourself with these issues for the moment. For now, just be sure that you understand the basic concepts discussed here in this lesson. We will be getting to more complex visibility determination strategies in due time.

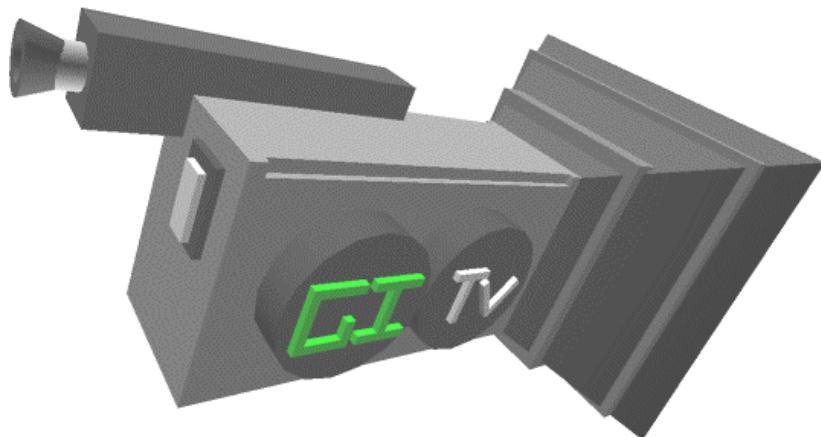
## Conclusion

We now have a good understanding of transformations, the view matrix, and the differences between various popular camera systems. We have also had another look at the projection matrix and now better understand the relationship it has with what things are considered visible and what items in the scene do not need to be drawn. Please turn to your workbook and examine the source code and additional discussion for this lesson. The camera and player classes we create in this lesson will provide a nice framework on which to build and refine your own camera systems down the road.

# **Workbook Chapter Four:**

## **Camera Systems**

---



© 2003, eInstitute, Inc.

**You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.**

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 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:

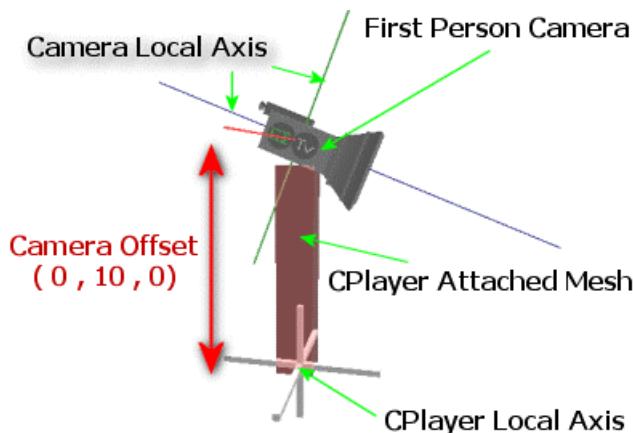
- Our application will not directly move the camera. It will attach the camera to a CPlayer object. Our application will call functions such as CPlayer::Move. This will move the player and the camera that is attached to the player depending on whether the Player has been put into first person mode, third 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  $\frac{1}{4}$  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)

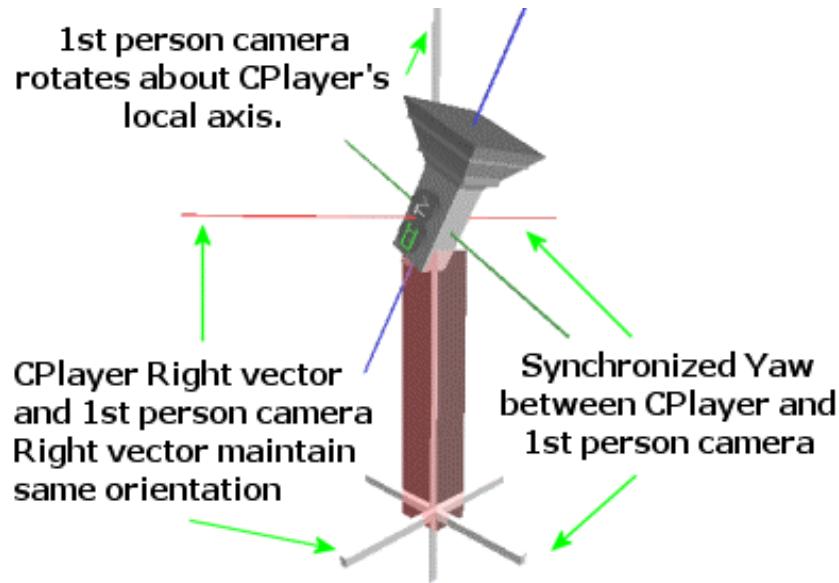
When we specify an X axis rotation, we wish to pitch the player upwards or downwards. It makes little sense for the player object to rotate about its own X axis. In real life if we wanted to look up or down we would rotate our heads back or forward, not our whole body. So, when the player is in first 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.

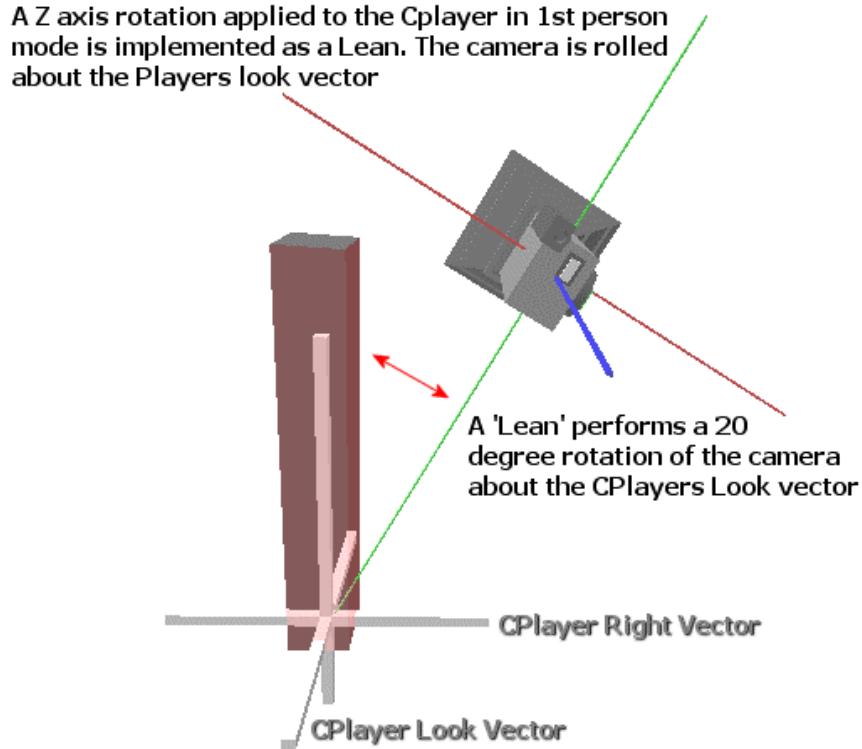
The following image shows how a Yaw works. The camera has already been pitched up so that it is rotated about its own Right vector. This is like the user rotating his head up to look at something in the sky. Now if we were to perform a Y axis rotation on the player, he would rotate about his own Up vector 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.

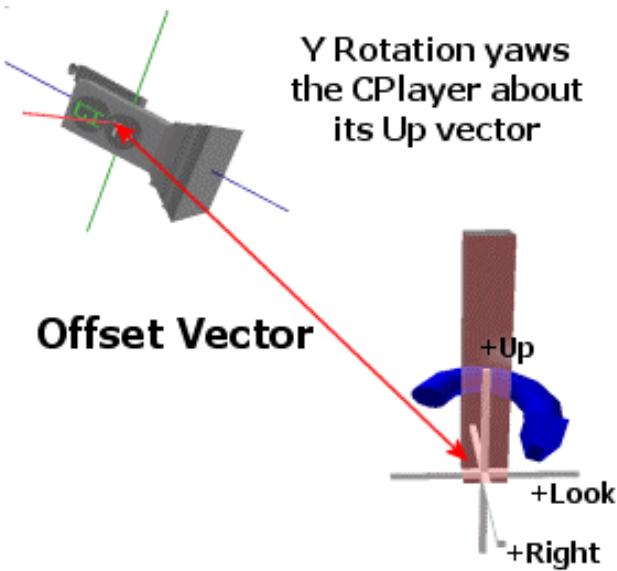
### CPlayer / CCamthirdPerson Overview

The nice thing about driving 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.

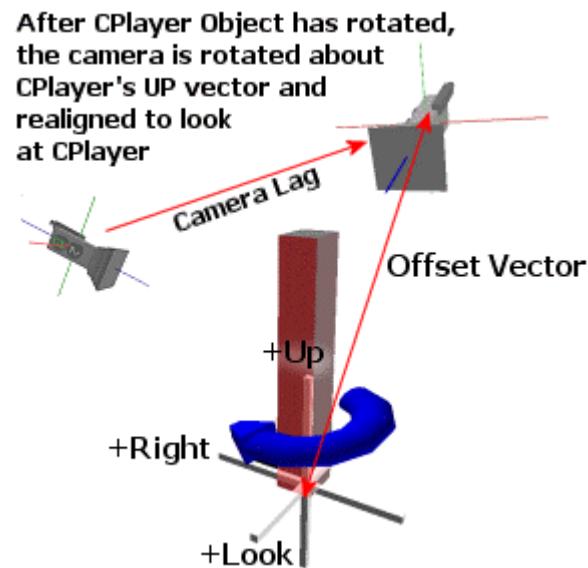
In third person mode, CPlayer ignores X and Z rotations passed into the CPlayer::Rotate(X,Y,Z) function. This is because the player can only rotate left and right. This was true in first person mode also, but we had the ability to pitch or roll the camera as the head of the player. In response to a Y 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  $\frac{1}{4}$  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  $\frac{1}{4}$  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.



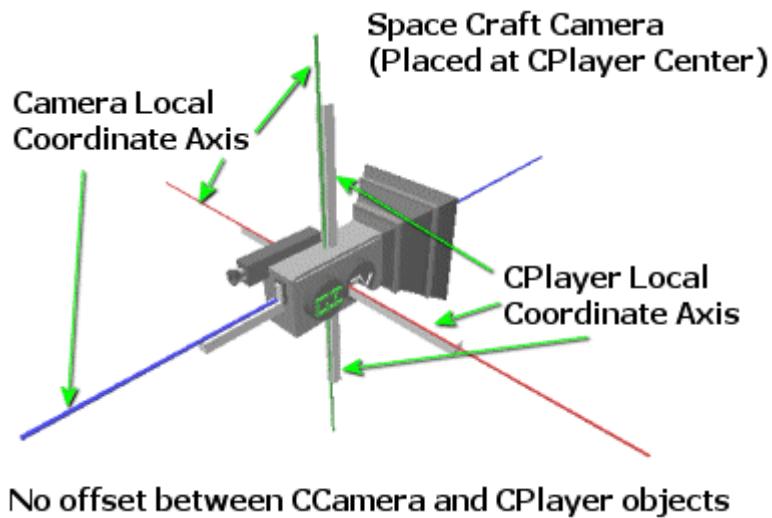
We will take a close look at the code to all the camera classes in just a bit. For now we just need to understand what we will need each camera class to accomplish.

### CPlayer / CCamSpaceCraft Overview

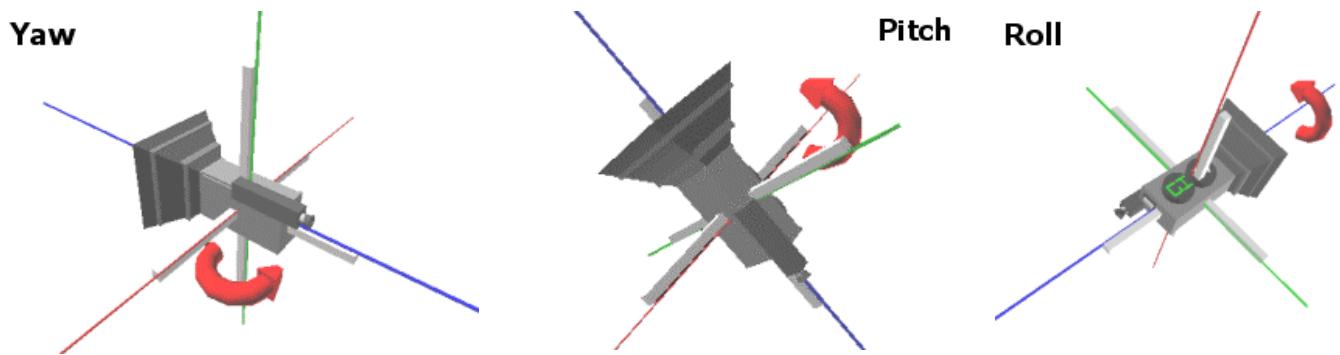
When the player is in space craft mode, the camera used will be a CCamSpaceCraft object. This is actually the easiest camera mode to understand since it works exactly like the code we studied in the textbook. Unlike the other two modes where the CPlayer could only be rotated about the Y axis, in this 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:



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  
    ::CheckMenuItem( 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  $\frac{1}{4}$  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  
    ::CheckMenuItem( 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, there is a course teaching DirectInput techniques right here at the Game Institute. Check the website for more details.

```

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( );

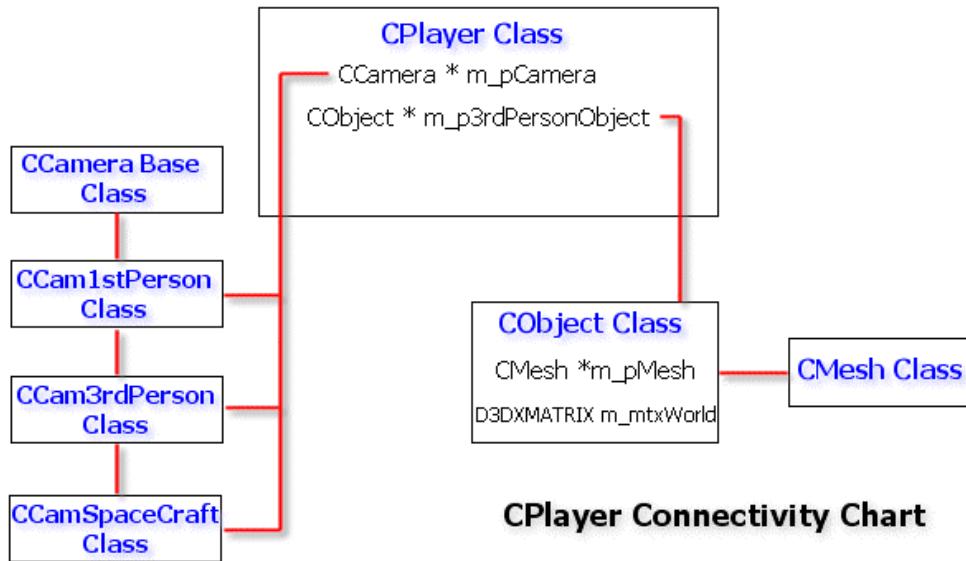
// Request our player render itself
m_Player.Render( m_pD3DDevice );

// 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.



## Coding the Camera System

```

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 initialised 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 CCamer. 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.

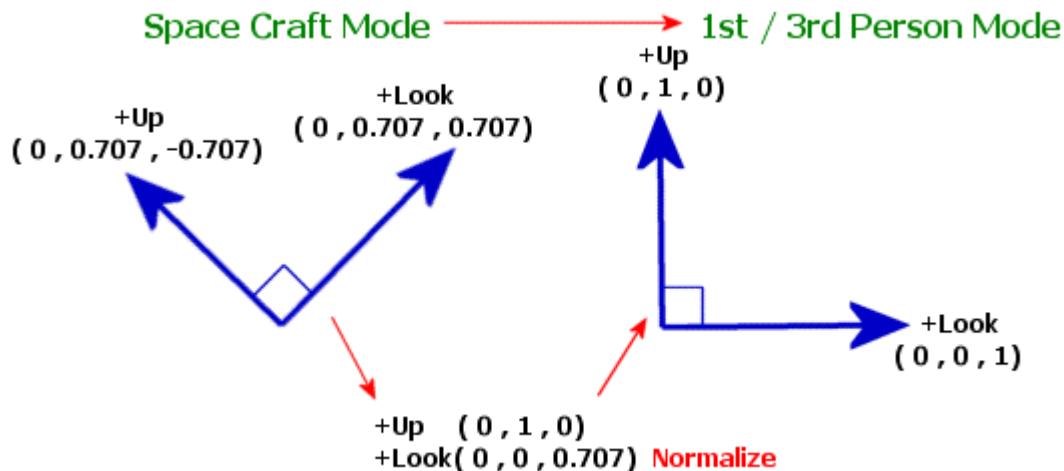
```

// 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:

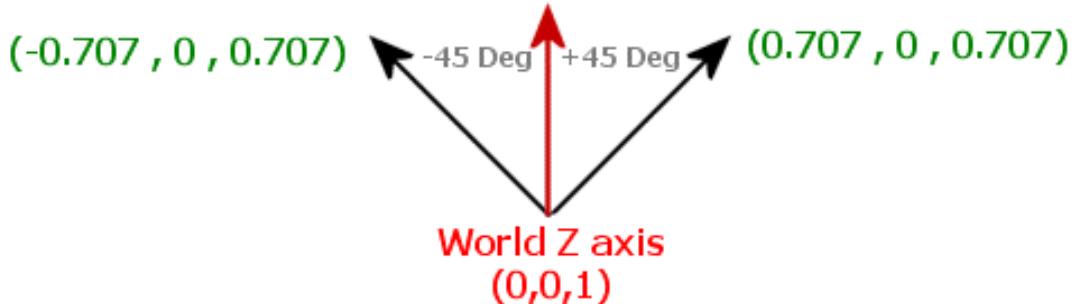
```

// 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.

## Calculating the sign of Yaw



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 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 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 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 );
    }
}
```

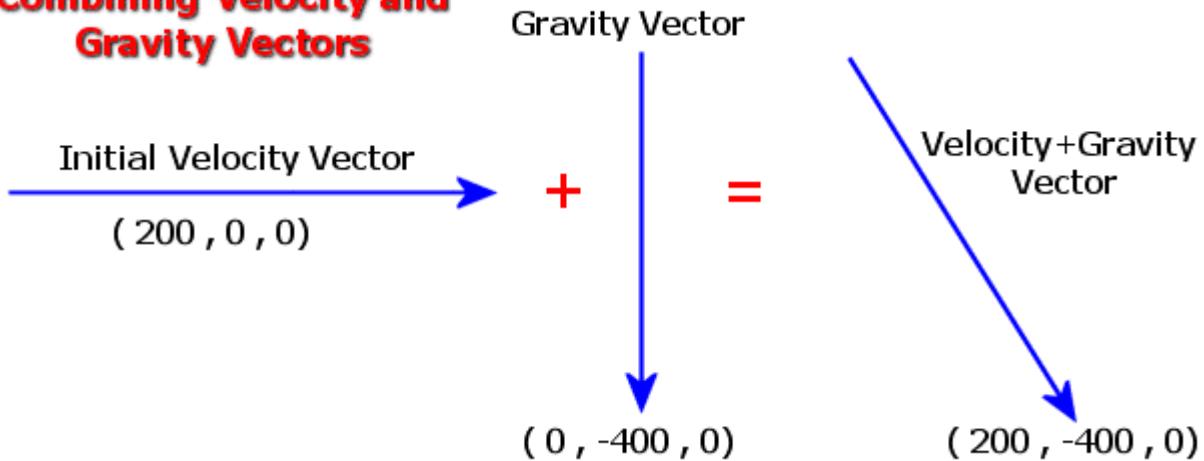
## CPlayer::Update

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.

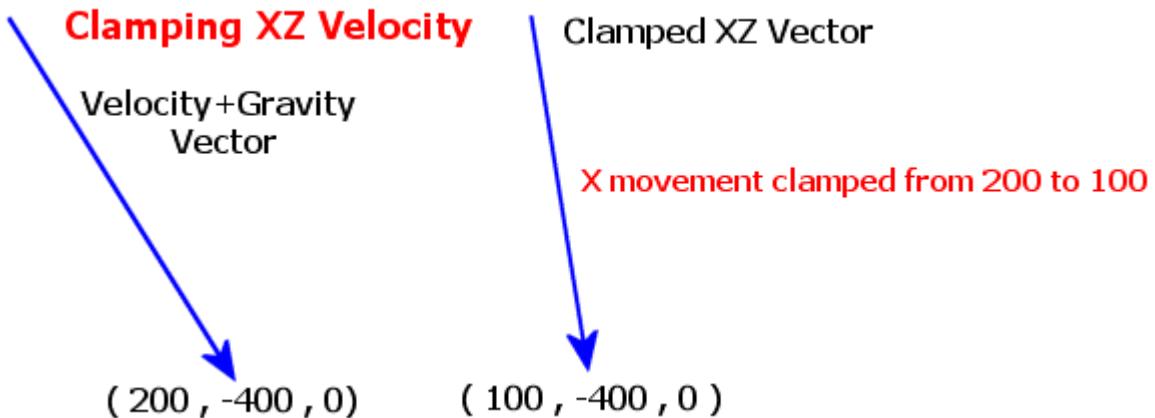
## Combining Velocity and Gravity Vectors



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;
```

## Calculating Deceleration Vector Direction

**Velocity Vector  
(Length = 400)**



**Inverted Velocity Vector  
(Length = 400)**



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
```

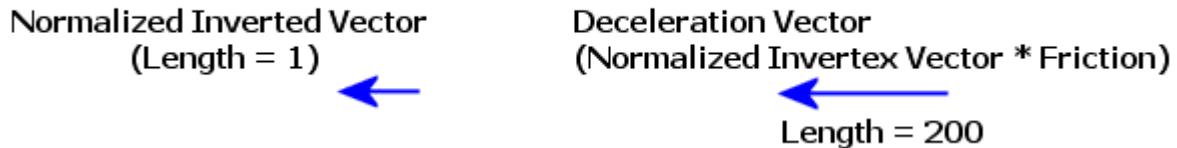
```

        m_vecVelocity += vecDec * Dec;
    }

```

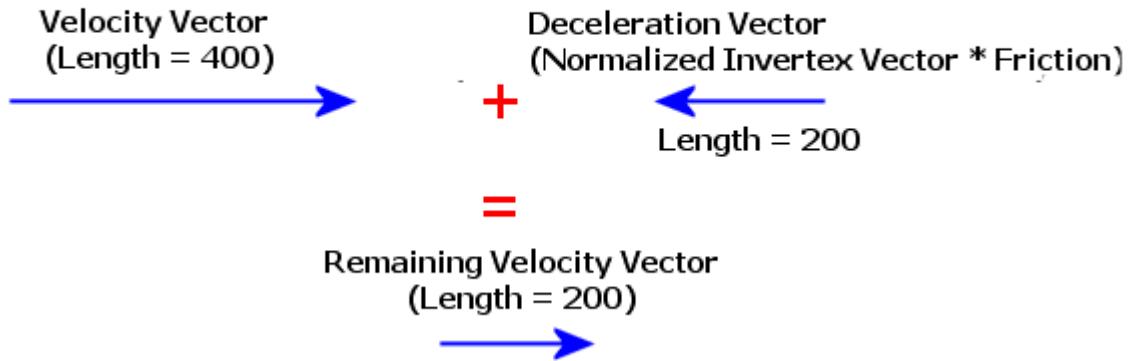
The next time this function is called, the velocity vector will have decreased (assuming the application has not requested additional acceleration). Remember that acceleration is always applied when the user is holding down one of the movement keys. The following image depicts scaling the unit deceleration vector by a drag coefficient of 200 units.

### Calculating Deceleration Vector Part II



Finally, this deceleration is added to the velocity vector so that its length is diminished.

### Applying Friction to the Velocity Vector



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\_bViewMatrix** 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 camea. 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

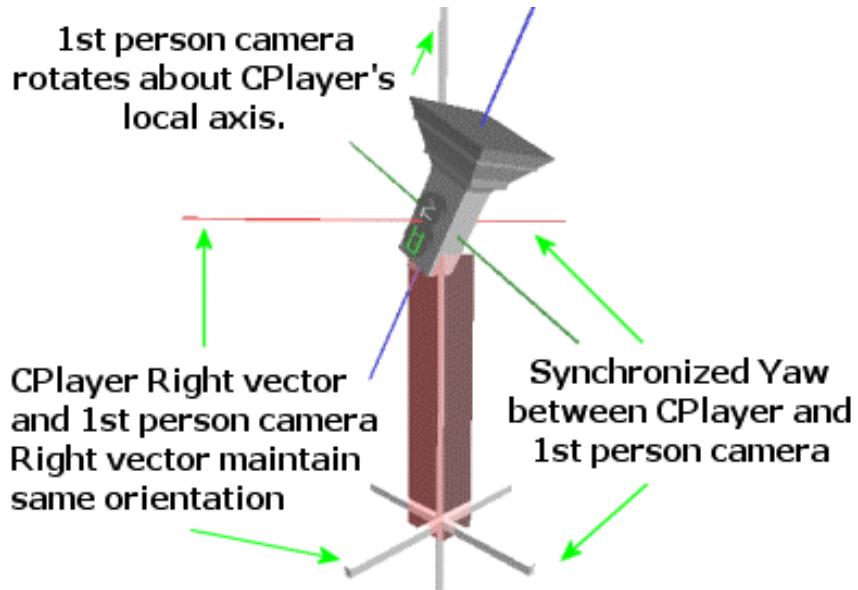
The Rotate function is an overridden virtual function. Recall that in CGameApp::ProcessInput we call CPlayer::Rotate in response the mouse being dragged with one or more buttons down. The CPlayer::Rotate function rotates the CPlayer Up, Look and Right vectors and then calls the attached camera's Rotate 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.

```

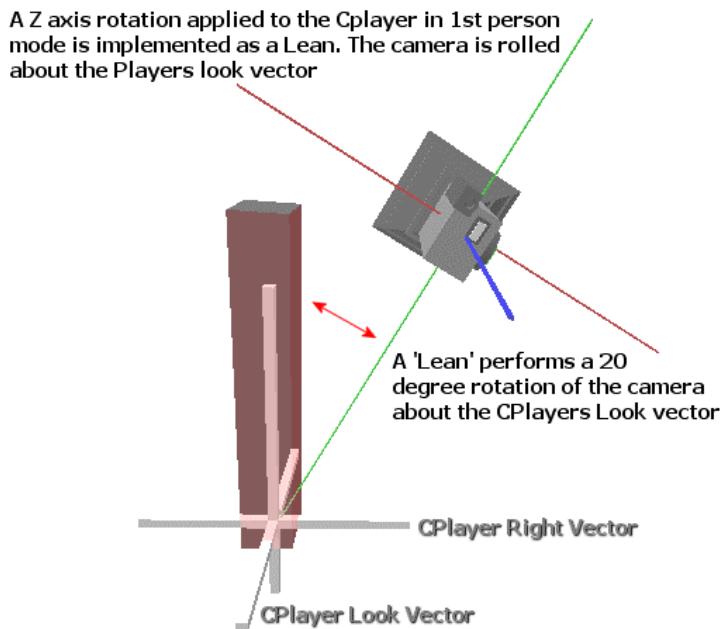
if ( y != 0 )
{
    // Build rotation matrix
    D3DXMatrixRotationAxis( &mtxRotate, &m_pPlayer->GetUp(), D3DXToRadian( y ) );

    // 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;
}

```

Below 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();

    } // End if Pitch

    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
```

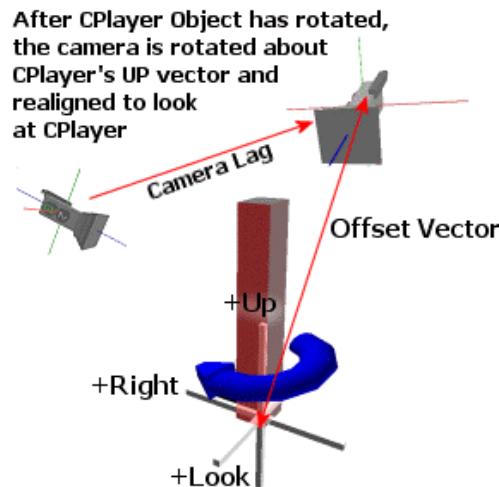
```

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 updated
    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):

**fImageSpaceX = 72/10 = 7.2**

**fImageSpaceZ = 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 7<sup>th</sup> and 8<sup>th</sup> pixel horizontally in the image map and between the 2<sup>nd</sup> and 3<sup>rd</sup> pixels vertically down the image map.

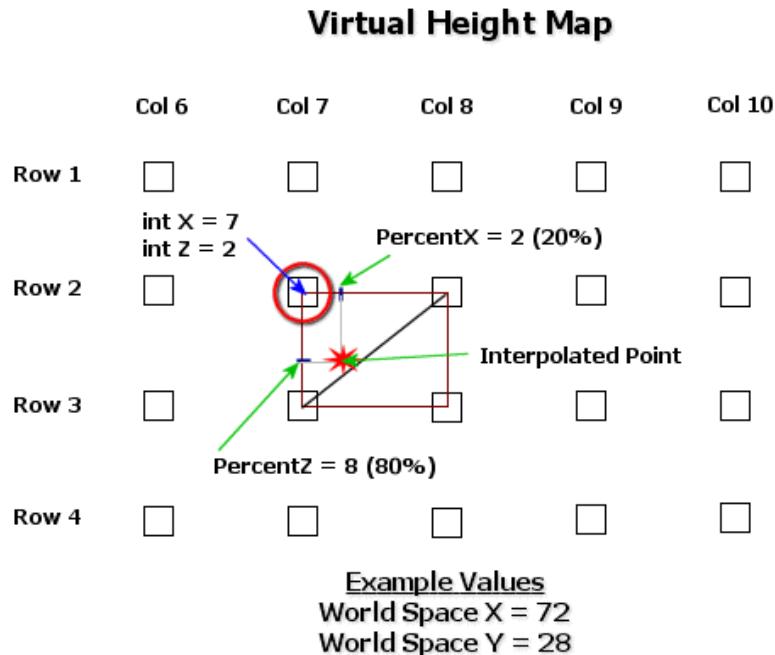
**iImageSpaceX = 7**  
**iImageSpaceY = 2**

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.

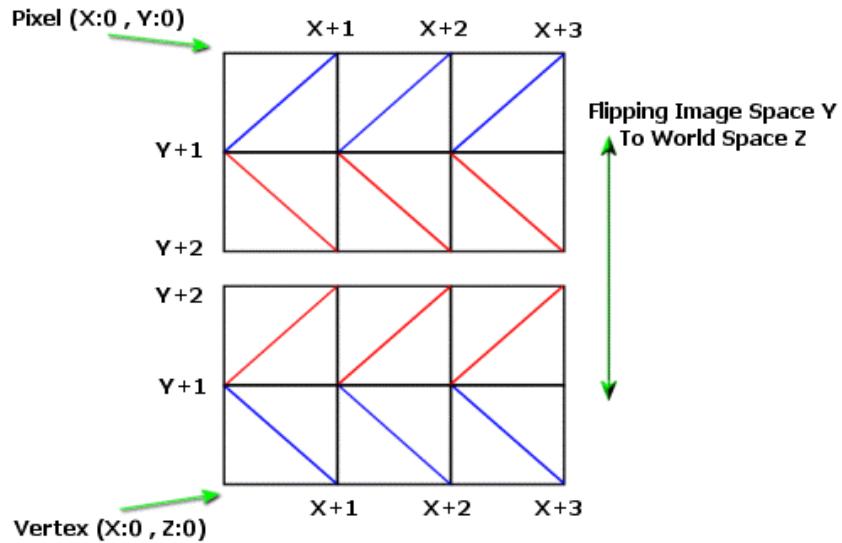


We know that the image map cannot use fractional coordinates because its pixels are at discrete locations. For example, there is no way for us to access pixel (7.2, 2.8) in an image. But the above image shows that if we imagine a virtual height map such that this is the case, we find that the pixels

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).

```

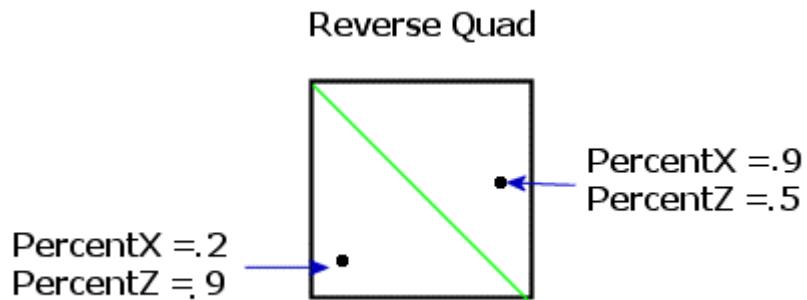
if ( ReverseQuad )
{
    // First retrieve the height of each point in the dividing edge
    fTopLeft      = (float)m_pHeightMap[ix + iz * m_nHeightMapWidth] * m_vecScale.y;
    fBottomRight  = (float)m_pHeightMap[(ix + 1) + (iz + 1) * \
                                         m_nHeightMapWidth] * m_vecScale.y;
}

```

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
```

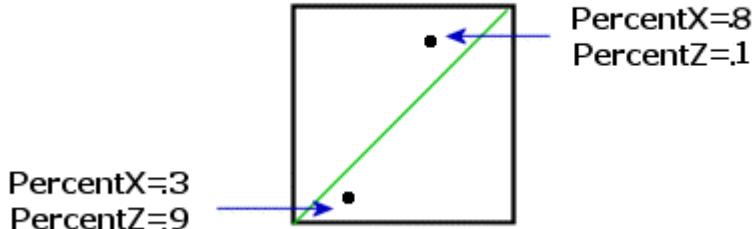
The fBottomLeft variable contains the height of the bottom left vertex in the quad. We use the three triangle points to create a top right vertex height which is co-planar with the other three triangles points. This top right vertex may not be the height of that vertex in the terrain, but it does not 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:

### Non Reversed Quad



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

```

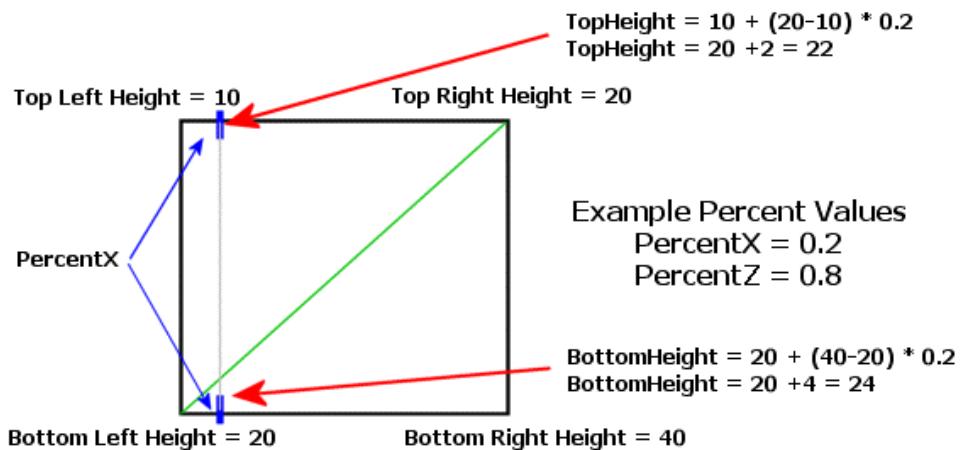
At this point we have a planar quad of height values. We multiply the top right height value by the fPercentX fraction and add this to the top left height value. This creates an edge between the top left and right points in the quad. We interpolate along the edge to get the height of that edge at the 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:

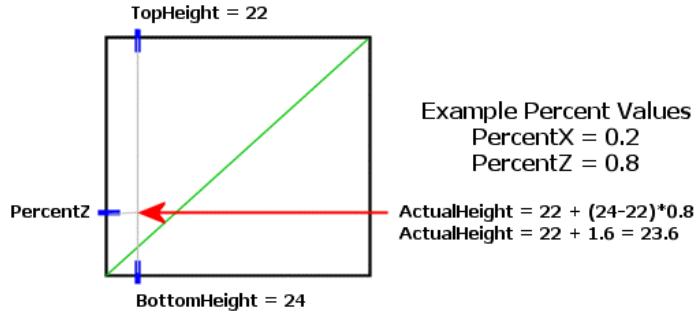


As you can see, the PercentX value tells us how far we have to interpolate the height value along the top and bottom edges. In the above example, the final height values for the top and bottom edges are 22 and 24 for the top and bottom edges respectively.

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

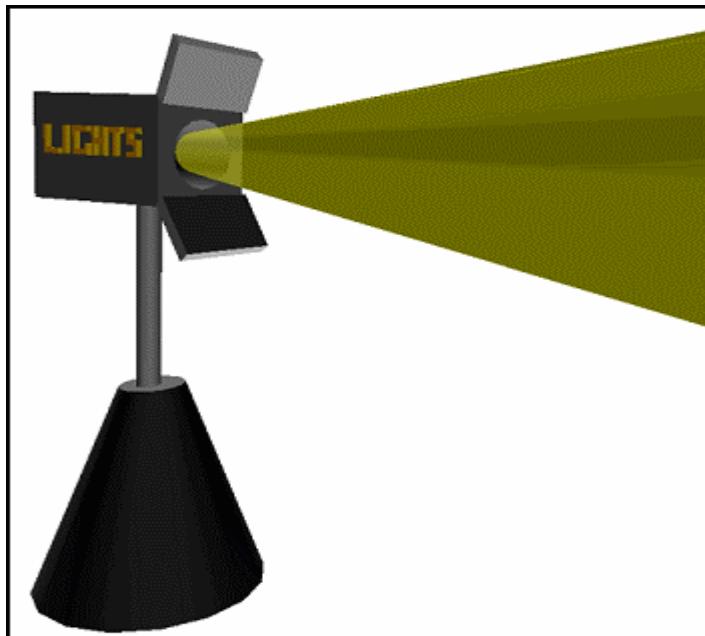
1. What is vector regeneration and why is it necessary?
2. If you resize your viewport such that it changes the aspect ratio, do you need to rebuild the projection matrix?
3. This final exercise will demonstrate whether you have a thorough understanding of this new material. We would like you to add 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 CCamthirdPersom 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.

# Chapter Five:

# Lighting

---



© 2002, eInstitute, Inc.

You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.



## Table of Contents

Introduction.....	3
Emissive Illumination .....	4
Global Illumination.....	4
Direct Lighting.....	6
Diffuse Light.....	6
Specular Light.....	9
The Basic Lighting Equation .....	10
DirectX Graphics – The Lighting Pipeline .....	11
Enabling DirectX Graphics Lighting .....	12
Unlit Vertices .....	13
Vertex Normals.....	14
Setting Lights .....	18
Light Limits .....	19
Light Types .....	21
Point Lights.....	21
Position .....	23
Direction .....	24
Range .....	24
FallOff.....	24
Attenuation.....	24
Theta and Phi .....	27
Spot Lights.....	28
Diffuse, Ambient, and Specular.....	29
Position .....	29
Direction .....	29
Range .....	29
Attenuation.....	29
Phi and Theta .....	29
FallOff.....	30
Directional Lights .....	33
Materials .....	36
Specular and Power.....	40
Material Sources .....	42
DirectX Vertex Lighting Advantages .....	47
DirectX Vertex Lighting Disadvantages.....	47
Conclusion .....	51



## Introduction

The interaction between light and the surfaces that reflect it is responsible for everything we see. Consider what happens when we enter a completely dark room. Until we turn on a light, we would be unable to see any of the objects within the room. Now imagine turning on a small overhead lightbulb. As the light comes on, its energized photons are emitted outwards in all directions striking surfaces, being absorbed and reflected, each time losing a little more of their energy. Objects are now visible to us as the photons reflected off of their surfaces reach our eyes. The color of those objects depends on the frequency of the light as well as the various properties of the surface. If the surface properties were such that the surface would only reflect red light, then the blue and green color components of the light would be absorbed and only the red component would be reflected back to the viewer. This object would appear red to us. If we shone a green light at such a surface, the surface would appear black because it absorbs the green and blue components and only reflects red. Some surfaces, such as metal, are shiny and when light is reflected off of them we notice highlights on their surface. Again, this is a result of the properties of the surface, not the light itself.

Given that light is ultimately responsible for what we see in the real world, it is fair to say that the more realistically we can model it in the virtual world, the more realistic our games will look. Light and shadow play a critical role in creating mood and establishing atmosphere in a game. A brightly lit dungeon is not likely to be very eerie or frightening from the player's perspective. Lighting techniques are one of the most researched topics in the computer game industry and new and more sophisticated approaches are constantly evolving. The recent advent of programmable hardware shader programs really opened up new doors to game developers. You can see this reflected in many of the titles hitting shelves today. Creating realistic lighting effects that run in real time is a challenging task to say the least. This chapter starts us down that path.

In this lesson we will assemble a mathematical model for scene lighting suitable for use in real-time applications. This model will necessarily be only an approximation (at best) since producing lighting that is even close to being physically correct is simply not possible on modern hardware. In the next course in this series we will examine techniques that generate more realistic looking lighting than the model we are about to study.



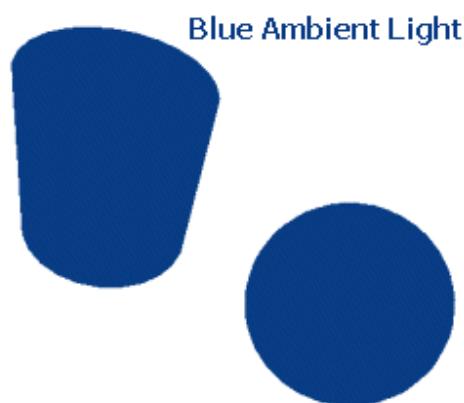
## Emissive Illumination

We begin our lighting model at the most basic level -- one where the surface itself emits its own illumination based on some inherent property of its material. We refer to this as the **emissive** property. One might think of a neon sign or even a light source itself as examples. The material produces its own illumination even in the absence of a specific external light source.

## Global Illumination

When a surface reflects incoming light, those photons will strike other surfaces in the scene and contribute to their coloring. This process continues in all directions for every surface in the scene until eventually a global level of lighting is established and all surfaces are lit equally. This is referred to as indirect global illumination because all surfaces are lit indirectly as a result of light scatter from other surfaces rather than directly by a scene light source. A global **ambient** lighting term can be used to very roughly approximate this general effect of light scattering in the environment. This approach can be used to provide a constant global illumination level for all vertices in the scene. The ambient color will be set to a desired value that is added to each vertex color.

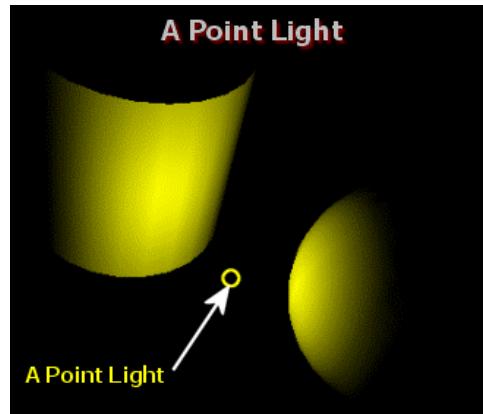
Ambient light is useful when there are areas in the scene not affected by any of the direct light sources. Without ambient lighting (or an alternative technique), such areas would be rendered totally black. Since ambient light is evenly dispersed everywhere in the scene, it is applied to all vertices in equal amounts. As a result, we generally set it to a very low intensity so as not to interfere with the effects of our direct light sources. With only ambient light enabled and no other direct light sources in the scene, the color of each surface will be identical -- assuming that they all use the same material. Fig 5.1 shows a cylinder and a sphere rendered only with a blue global ambient light setting.



**Figure 5.1**

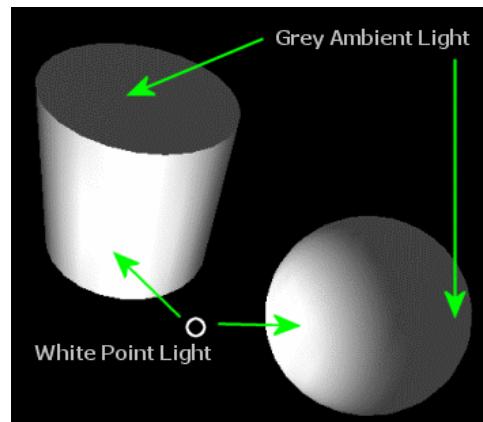
Fig 5.2 shows the same cylinder and sphere with no ambient light and one yellow point light source added to the scene and positioned between the cylinder and the sphere. We will examine point lights in

the next section when we discuss direct lighting, but for now we notice that the vertices (and therefore the faces) that are facing away from the light source are not lit at all. This is because the vertex normals of the top cylinder face and those toward the back of the sphere are facing away from the light source.



**Figure 5.2**

Now look at the same cylinder and sphere lit with a white point light and a light grey global ambient light setting (Fig 5.3). All faces will include the minimum grey color even if they are not influenced by the point light:



**Figure 5.3**

As we will see later on, even direct light sources can include ambient light emitting properties to add to the global ambient light level of the scene. It should also be noted that in both of the diagrams above, it is assumed that the surface material reflects all ambient light. If the material did not reflect ambient light, then the ambient light color would have no effect on the final color of the rendered surface. This would indicate that the material absorbs all ambient light and reflects none back to the viewer.



While a global ambient lighting term is one way to make sure that surface vertices that are not directly influenced by the directional light sources in the scene do not remain completely black, the downside is that all surfaces will receive the same color. There are a number of ways around this limitation; one example would be to attach a directional light source to the player/camera that aligns with the Look vector. This is often called a headlight. Essentially it allows for a direct light source to affect areas of the scene in the player's view where the static light sources may not have reached. It is not a perfect solution, but it can often produce satisfactory results.

## Direct Lighting

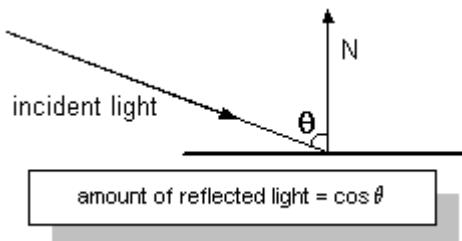
Direct lighting generally makes the largest contribution to the final appearance of a surface. It is broken into two sub-categories: **positional** (where light emanates from a specific identifiable point in space) and **directional** (where light comes from a general direction whose source is infinitely far away).

We can add light sources to a scene that have positions, orientations, ranges, colors, and intensities. The lighting engine can calculate which lights in the scene contribute to the color of every vertex rendered. Each vertex is first checked to see if it is within the range of a given light. If it is, light can be attenuated with respect to distance and the remaining light color (intensity) scaled by the cosine of the angle between the vertex normal and the vector describing the direction of the light source from that vertex. The resulting light color is then used to determine two different types of reflections: diffuse and specular. How much of that diffuse and specular color gets reflected depends on the material we currently have set. We will discuss materials later in the lesson.

## Diffuse Light

The effect of a diffuse light source on a surface is dependant upon the spatial relationship between the two. When a surface is perpendicular to a directional light source then the full intensity of the light strikes the surface. When the surface is oriented at an arbitrary angle with respect to the light source, the intensity of the reflected light is reduced. Lambert's Law describes the amount of diffuse light that strikes some point in space as the full intensity of the light scaled by the cosine of the angle between two vectors. The first of these vectors is a unit length vector describing the direction from the point in space to the light source. The second vector is another unit length vector describing the direction that point is facing.

Fig 5.4 shows a ray of light striking a point on a surface, and the surface normal. If we invert and normalize the incoming light direction vector and perform the dot product between this vector and the face normal, the cosine of the angle returned can be used to scale the contribution of the incoming light source.

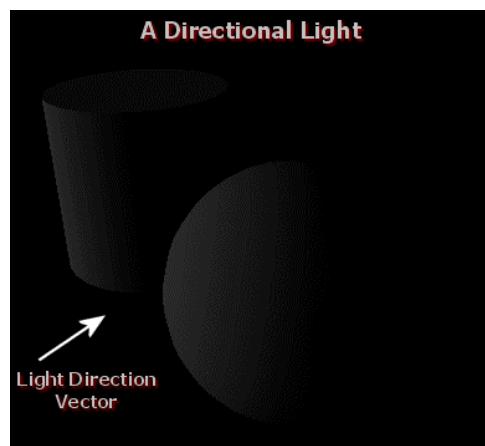


**Figure 5.4**

In Chapter 3 we modeled the effect of incoming light at a particular surface point – a vertex. The assumption was that our terrain surfaces were ideal diffuse Lambertian surfaces that scatter light equally in all directions. We will continue to make that assumption for all diffuse surfaces in this lesson.

The color of the surface at a particular point does not depend on the location or orientation of the viewer because light reflecting off of a diffuse surface travels equally in all directions. Together with the diffuse reflectance property of the currently set material, diffuse light is responsible for contributing to what we would perceive to be the actual color of a vertex. If we have a light that emits white diffuse light and we have a material that reflects only the red component of the diffuse light, then the vertex will appear to be red.

When a vertex has its color calculated, the diffuse colors of all the direct lights that influence that vertex are combined. The position of each light source, the normal of the vertex, and the orientation, range, and attenuation of the light are all factors used to calculate the total diffuse light at a given vertex. This light is then modulated with the diffuse reflectance properties of the material to create the perceived color of the object.



**Figure 5.5**

Fig 5.5 shows the same cylinder and sphere lit by a white directional light source. No ambient light is used (global ambient light color = black). The material describes the diffuse reflection as medium grey where the angle between the incident light vector and the vertex normal is 0 degrees (they are facing each other exactly). As you can see, as the angle between the vertex normal and the incident light

vector increases, the grey color is scaled down. At some point the incident light vector and the vertex normal is greater than 90 degrees. At that point the vertex will no longer be lit since its normal is facing in the opposite direction of the light direction vector. With Gouraud shading enabled, the faces using such vertices fade away into darkness.

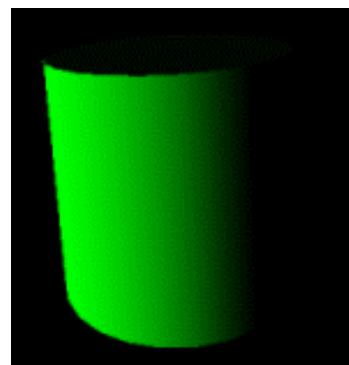
Fig 5.6 shows a cylinder lit by a white directional light shining in from the left side. The material used reflects all incoming diffuse light. This means that if the entire object is lit by white light, the vertices will reflect all the white light that reaches it. However, take a close look at the image.



**Figure 5.6**

Notice that the vertices are only completely white at the exact points where the vertex normal and the incident light vector are the same. The other vertices reflect all diffuse light as well (because they use the same material) but they do not receive the same amount of light due to the angle between their vertex normals and the incident light vector.

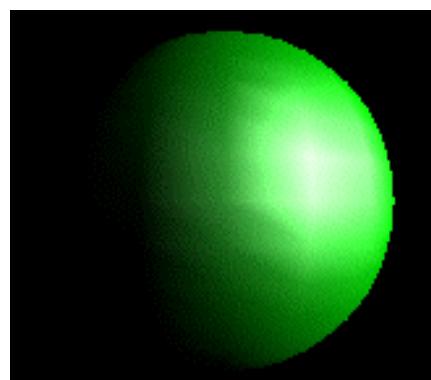
Fig 5.7 shows the same cylinder with the same intense white light shining on it. However, in this example, the material used by the cylinder only reflects green diffuse light. So the red and blue color components of the diffuse light that hit each vertex are totally absorbed and the object appears green.



**Figure 5.7**

## Specular Light

Specular lighting creates surface highlights that make objects appear shiny and smooth. Unlike diffuse lighting, specular lighting is view dependant because light is not scattered equally in all directions. A perfect specular surface (like a polished mirror) would reflect light such that it mirrored the incoming ray. A rougher specular surface like a metallic facade introduces some scattering but nevertheless reflects light in a roughly mirrored fashion (i.e. still primarily along one directional axis). As the angular relationship between the camera look vector, the vector between the vertex and the viewer, and the vector between the vertex and the light source changes, the highlights will appear to move across the surface of the object.

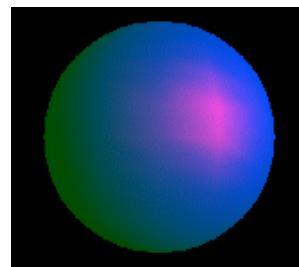


**Figure 5.8**

In Fig 5.8, we see a sphere lit by a bright white diffuse light shining in from the right. The light also includes a white specular component. The material reflects green diffuse light and all specular light. The highlight helps us to gauge the location of the light source more accurately. This is another important visual cue that contributes to the overall realism of the lighting chosen for the scene.

In addition to emitting a diffuse color, each light in our scene can be configured to emit a specular color using a separate property. This color will be modulated with the specular reflectance properties of the currently set material to control the color of the highlight.

To better see that diffuse and specular colors are calculated separately, Fig 5.9 shows the same sphere and the same white light. This time we are using a material that reflects only blue diffuse light and red specular highlights. A dark green global ambient color is used to make the faces completely in shadow on the left side of the sphere visible. All of these colors are added together so that the red specular highlight has been modulated with the blue diffuse color to create a purple/pink highlight.



**Figure 5.9**

### The Basic Lighting Equation

The total illumination (**I**) level of a vertex can now be described by:

$$\mathbf{I} = \mathbf{A} + \mathbf{D} + \mathbf{S} + \mathbf{E}$$

**A (Ambient Light)** is the sum of the global ambient light color and the ambient color emitted from all lights that influence the vertex -- modulated by the material ambient reflectance property.

**D (Diffuse Light)** is the sum of all the diffuse colors from each light source that influences the vertex - - modulated by the material diffuse reflectance property.

**S (Specular Light)** is the sum of all specular colors from each light that influences the vertex -- modulated by the material specular reflectance property.

**E (Emissive Light)** is a color that is emitted by the vertex itself, not by a light source. A material can be used that has an emissive color such that even if the vertex receives no light of any kind, the emissive color will contribute to the vertex color.



## DirectX Graphics – The Lighting Pipeline

The DirectX Graphics fixed function pipeline conducts lighting operations at the per-vertex level. This is the lighting we looked at in Chapter 3 when we added color to our terrain. Vertex lighting is generally fast enough to be done dynamically, but it does have limitations. We will discuss some of the benefits and limitations of including vertex level lighting support in our games later in the lesson.

In the CTerrain::BuildMeshes function in Lab Project 3.2, we calculated the color of each vertex in our terrain by generating a temporary vertex normal using heightmap data. This normal was used to measure the angle between the vertex and the incoming light's direction vector using the dot product. The cosine of the angle between these two vectors scaled the color and then we stored the color in the vertex. As each triangle was rendered, the colors stored at each vertex in the triangle were interpolated across the surface during rendering (Gouraud shading). This was simple but effective vertex lighting and is not much different from what we are going to see throughout this lesson. The calculations will be a bit more complicated at times, but the concepts will be the same.

We did not enable the DirectX lighting pipeline in our demo because we had already lit the vertices when the terrain was assembled. The device was told that the vertices already contained their colors and dynamic calculation was unnecessary. The obvious drawback with this technique is that it is completely static. If we wanted to change the direction a light was shining, we would need to recalculate the lighting for every vertex affected by that light all over again. This is a slow CPU bound process and our preference is not to have to run it in real time. But enabling the DirectX Graphics lighting module allows for hardware – or fast software – lighting of vertices that can be run in real time applications. If the graphics card does not have hardware support for lighting, then we can create a software vertex processing device and the DirectX graphics software lighting module will be used instead. Although it runs on the CPU, the software lighting module that ships with DirectX Graphics is actually very respectable performance-wise and can be used in commercial applications.

When we call one of the DrawPrimitive functions with lighting enabled, the vertices passed in (typically in a vertex buffer) will not only be transformed but will also have their color calculated. We will no longer store a color value in our vertex which describes the final color of the vertex but will instead store a vertex normal at each vertex describing the direction the vertex is facing in model space. When a vertex position is transformed, so is its normal. Once the vertices are fully transformed, the lighting module will take into account all of the active light sources in the scene to determine their influence on a particular vertex. If a vertex is within the range of the light's influence, that light's color, and the angle between the vertex normal and the light direction vector (along with a few other factors to be explored later in the lesson) will be used to determine a final vertex color.

Adding lights to a scene is not sufficient to inform the lighting pipeline how the vertex should have its final color calculated. The way a surface reflects light is a major factor in determining the final color. As we will discover a little later in the lesson, a **material** (a collection of reflectance property settings) will describe how light should be reflected or absorbed by a vertex.



**Note:** there are alternative approaches that do not transform the vertex normals directly but instead back transform light sources into local model space using inverse matrices. This is done for efficiency; the results are the same.

## Enabling DirectX Graphics Lighting

In order to effectively use the DirectX Graphics lighting pipeline we will need to do the following:

- Enable the lighting pipeline
- Ensure that all vertices that need to be lit include vertex normals.
- Add lights to the scene.
- Define and set materials to describe the reflectance properties of the vertices.

Enabling/Disabling the lighting pipeline is done using the IDirect3DDevice9::SetRenderState function:

```
m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, TRUE ); //enables Direct3D lighting module  
m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE ); //disables Direct3D lighting module
```

One thing to bear in mind is that an application can mix pre-lit meshes with unlit meshes. In a space based game you may want to light all of the vertices of your space craft and planets using the light from a local sun or nebula. The color of each vertex will be calculated based on the orientation of the meshes with respect to the light sources. So objects in the scene would use unlit vertices with vertex normals so that they can be updated dynamically.

But we may also want to render a HUD (heads-up display) to provide information about the speed of the craft, laser energy remaining, shield integrity, etc. We would probably want this HUD to be a constant brightness at all times. You would not want the HUD information to become dull or perhaps even unreadable when the pilot positioned his spaceship such that the HUD was facing away from a light source. In the case of the HUD, you will make sure that the vertices store the colors themselves. After all other objects had been rendered you could disable lighting and render the HUD using pre-lit vertices. A render loop that uses both types might look like:

```
m_pDevice->BeginScene()  
  
m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, TRUE );  
  
CSpaceCraft->Render(); // This mesh does not have its final color stored in the vertex.  
// It includes vertex normals defined in model space.  
// The lighting pipeline will use this to calculate the vertices  
// final color by calculating the relationship between light sources  
// in the scene and each vertex  
  
m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );  
  
CHeadsUpDisplay->Render(); // This mesh's vertices have no vertex normals.  
// Color is stored in the diffuse member of the vertex  
// structure. This is the type of mesh we have been using in  
// previous applications.
```



```
m_pDevice->EndScene();
```

## Enabling Specular Highlights

Specular highlights are not calculated by the lighting pipeline by default so you must explicitly turn on the specular calculations with the following render state:

```
IDirect3DDevice9::SetRenderState(D3DRS_SPECULARENABLE, TRUE) // enable specular highlights  
IDirect3DDevice9::SetRenderState(D3DRS_SPECULARENABLE, FALSE) // disable specular highlights
```

Enabling specular highlighting will reduce performance to some degree, but it does dramatically add to the realism of the scene.

## Enabling Global Ambient Lighting

To set the global ambient color in DirectX Graphics, we must enable the lighting pipeline and then set the appropriate render state for the device:

```
// A R G B  
DWORD MyCOLORARGB = 0x00FF0000;  
m_pd3dDevice->SetRenderState( D3DRS_AMBIENT, MyCOLORARGB);
```

We specify the color of the ambient light as an ARGB DWORD. Each byte holds a value in the range [0, 255] for each of the alpha, red, green, and blue components. In lighting calculations, the alpha component has no effect and can be set to zero. The color value that we specify here will be added to each vertex that is rendered while this render state is set. We can change the color of the ambient light at any point throughout the lifetime of our application by calling the above function again and specifying the new ambient light color.

## Unlit Vertices

The vertices we have used in previous applications have been using a pre-lit format, where the color stored in the diffuse component of the vertex is sent directly to the rasterizer for color interpolation across the pixels of the triangle. In order to use the DirectX lighting pipeline, we must store a vertex normal in our vertex structure.

```
struct UnlitVertex  
{  
    float x;  
    float y;  
    float z;  
    D3DXVECTOR3 Normal;  
};
```

We have removed the diffuse color component from our vertex structure and added a new 3D vector member called **Normal** to hold the orientation of the vertex in model space. The following code snippet is from Lab Project 5.1 (in CObject.h) and shows our new CVertex class:



```
class CVertex
{
public:

    // Constructors & Destructors for This Class.
    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 Variables for This Class

    float      x;                      // Vertex Position X Component
    float      y;                      // Vertex Position Y Component
    float      z;                      // Vertex Position Z Component
    D3DXVECTOR3 Normal;    // Vertex Normal
};
```

Most of the time, we will load geometry from a file that was created using a world editor such as GILEST™ or a modeling package like 3D Studio MAX™. Vertex normals are often calculated in the editor and saved into the file so that the application can load the data directly into vertex buffers. Later in the lesson, we will load in an IWF file exported from GILEST™ to see how to extract the appropriate vertex information. There may be times however when you will need to calculate vertex normals yourself, so let us take a look at what vertex normals are used for and how they can be generated.

## Vertex Normals

Your first thought might be to simply supply DirectX Graphics with a surface normal for each triangle we render. Or perhaps even better, the DirectX pipeline could generate the face normal for our triangle automatically and just use it on the spot. But this will not work. First, a vertex may be shared by two or more faces and each face might have completely different surface normals. Second, The DirectX Graphics pipeline calculates the lighting in the transformation and lighting stage. This occurs before the vertices are assembled into primitives to be rendered. When using a software vertex processing device, the device will transform and light an entire range of vertices long before any per-triangle relationship has been established. And of course, the purpose of the lighting calculation module is to generate colors for each vertex, not just for each face.

Since a vertex is a single point in space, it may be a little strange to think of it having its own orientation. But in fact this is actually not so confusing if you consider a vertex to be a single sample point on a surface where we can collect light. We recall that a triangle is planar and that the plane itself has orientation. The vertices themselves share that orientation – as would all points on that plane.

Imagine if you took a single triangle and subdivided it into millions of smaller triangles. Now we can think of the vertices of these sub-triangles as tiny points filling the surface. Our light will strike these points and be reflected based on the orientation of the surface. Since the surface is really the sum of its parts (the tiny triangles) it would make sense that they all assume the same orientation in space. If we were to bend the corner points of the big triangle such that we created a curved bulging surface, we

recognize that the orientations of many of those subtriangles -- and thus their vertices -- changes. Now imagine the little triangles starting to merge together, welding themselves into larger triangles. The curve starts to become less smooth as the process moves along. The orientation of a specific vertex on the surface becomes more of an average orientation based on the triangles meeting at that point. Of course, this was always the case from the beginning, even before we bent the triangle. It is just that the average normal for the subtriangle vertices in that case would always result in the same value: the normal of the surface.

Assigning a normal to each vertex gives us finer control of the color that gets generated. Vertices belonging to the same triangle can each have different vertex normals. Each can describe an average orientation of triangles that meet at that point, rather than just the orientation of a single parent triangle. This provides smoother shading effects. The more vertices (and vertex normals) available to capture light samples, the more accurate a lighting model we achieve.

The cube in Fig 5.10 image depicts vertex normals that are the same as their parent surface normal. The arrows depict the direction of the vertex normals belonging to each face.

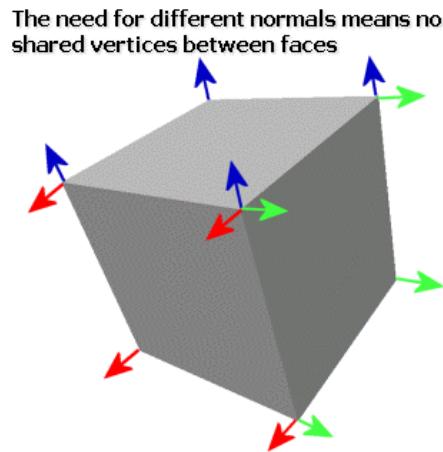


Figure 5.10

This arrangement actually works well for a cube since each face of the cube reacts to the light as a whole. The top face is a lighter shade of gray because its surface normal is parallel to the light source shining down. The front and side faces are not so brightly lit because they point away. Using a single surface normal for each face of the cube provides very sharp and distinct edges. It's easy to see where one face ends and the next face begins. Note that there are three vertex normals at each corner point of the cube. In this particular case, we find ourselves back in a situation where we need to duplicate vertices between faces. At any given corner of the cube, there are three vertices with duplicated positions and different vertex normals.

The cube example is a rare case where we typically do not want to share vertex normals because we want a sharp and defined edge between each face. In Chapter 3 we learned that eliminating this sort of redundancy is very desirable. We did this by generating normals for shared vertices that were an

average of the normals of all of the faces to which a shared vertex belongs. Fig 5.11 shows a cylinder with duplicated vertices.

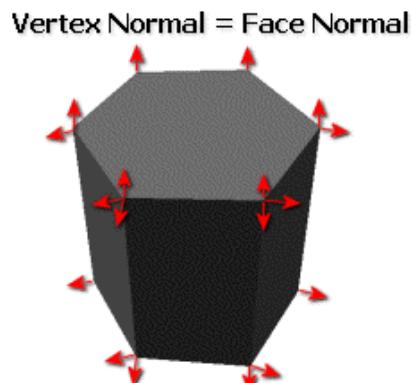
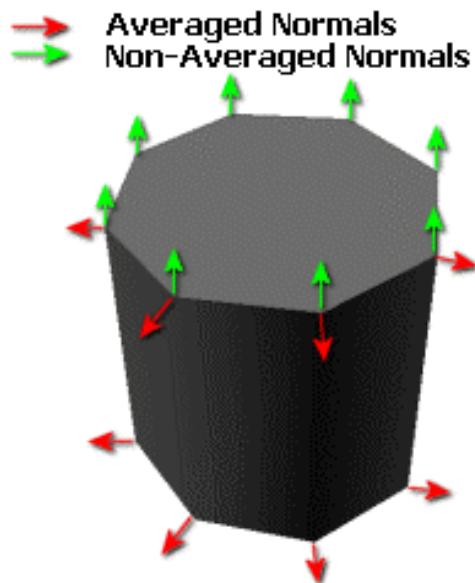


Figure 5.11

Just like the cube, each face in the cylinder above has its own unique set of vertices. Each vertex has a normal that is equal to the face normal to which it belongs. Unlike the cube mesh however, a cylinder mesh should usually be perceived as a more rounded object. Sharp and defined edges between faces are not ideal.

Fig 5.12 shows the same cylinder where the side faces share vertices with neighboring faces. Each vertex is shared by two faces and the color in both faces at that point is the same. We no longer have sharp color changes as we move from one face to the next. This makes for a more rounded appearance. Note that the top face does maintain its own unique set of vertices because we do want a sharp edge between it and the side faces.



**Figure 5.12**

The vertex normals that are shared among faces have been averaged so that they are no longer aligned with any particular face normal. Their orientations describe a vector halfway between the two parent face normals.

It is easy to generate averaged normals. Simply calculate a normal for each face to which the vertex belongs, add them together, and normalize the result to ensure a unit length vector.

```
D3DXVECTOR3 CalculateVertexNormal( int VertexIndex , int *IndexArray )
{
    D3DXVECTOR3 VertexNormal ( 0.0f , 0.0f , 0.0f )
    for (int a=0; a< NumberOfIndices/3;a+=3)
    {
        int index1 = IndexArray[a];
        int index2 = IndexArray[a+1];
        int index3 = IndexArray[a+2];

        if ( index1==VertexIndex || index2==VertexIndex || index3==VertexIndex)
        {
            VertexNormal += CrossProduct ( index1 , index2 , index3);
        }
    }
    Normalize (&VertexNormal);
    return VertexNormal;
}
```

The above code snippet assumes that `IndexArray` is a list of triangle indices. It accepts a vertex index and then checks to see if any triangles reference that vertex. If so, then the cross product is performed on the three vertices and the resulting face normal is added to `VertexNormal`. At the end of the function, we normalize the result to ensure unit length and return the average vector.



Obviously the above code completely depends on how the vertex and index data is stored. But it should give you enough of an idea to use it as a template to write your own function.

With each vertex now containing its own normal, the pipeline line has what it needs to accurately compare the relationship between the direction the vertex is facing and the orientation of any lights in the scene.

There is one important thing to note about the use of vertex normals and the DirectX lighting pipeline. Vertex normals are transformed by the upper 3x3 portion of the currently set world matrix (technically the concatenated world/view matrix) during the render call. This insures proper world space orientation for the normal since it is assumed that the vertex normals passed in were created using model space data. If your world transformation matrix uses a scaling component, this will be part of that upper 3x3 matrix (see Chapter 1). This scaling creates an undesirable outcome since the normals that are scaled will wind up losing their unit length status. When this happens, the results of the dot product used in the lighting equation will be affected and the lighting engine will not produce correct results. To address this problem, there are two solutions.

The first solution is provided by DirectX. There is a render state that can be activated prior to the call to DrawPrimitive that will re-normalize the vertex normal data after the transformation. As you might imagine, this can be a costly operation since normalization involves three multiplications and a square root calculation. To enable and disable this render state, simply call:

```
IDirect3DDevice9::SetRenderState(D3DRS_NORMALIZENORMALS, TRUE); //turn on  
IDirect3DDevice9::SetRenderState(D3DRS_NORMALIZENORMALS, FALSE); //turn off
```

The second solution, which is generally preferable, is not to include scaling data in your world matrices. That is, make sure that your models conform to the appropriate world scale before you export them from your modeling package. While this may sound like a cop out and it is indeed nice to be able to scale models on the fly, we generally prefer to avoid the overhead of the per-vertex re-normalization processing in a real-time situation.

## Setting Lights

DirectX Graphics allows your application to store a set of properties for each light in your scene in an array of memory slots on the device. **IDirect3DDevice9::SetLight** is called to assign light properties to specific memory slots. Calling **IDirect3DDevice9::SetLight** by itself does not make a light active. By default, lights are disabled until explicitly turned on, even after they have been set. The device simply stores these settings until such time as you enable that light with a call to **IDirect3DDevice9::LightEnable**.

```
HRESULT IDirect3DDevice9::SetLight(DWORD Index, CONST D3DLIGHT9 *plight)
```

### DWORD Index



This zero-based offset is used to specify the desired slot for the property set contained in the second parameter.

### **D3DLIGHT9 \*plight**

The second parameter to the **SetLight** function will be the address of a D3DLIGHT9 structure. This structure contains settings which describe how a light should be used by the device to contribute to vertex coloring. These settings include the type of light it is, the position of the light in the world, the direction the light is facing, the range of the light and many other properties which will be discussed in this lesson.

```
typedef struct _D3DLIGHT9
{
    D3DLIGHTTYPE      Type;
    D3DCOLORVALUE     Diffuse;
    D3DCOLORVALUE     Specular;
    D3DCOLORVALUE     Ambient;
    D3DVECTOR         Position;
    D3DVECTOR         Direction;
    float             Range;
    float             Falloff;
    float             Attenuation0;
    float             Attenuation1;
    float             Attenuation2;
    float             Theta;
    float             Phi;
} D3DLIGHT9;
```

The light type you choose to create determines which members need to be filled in. For example, the Theta and Phi values are only used by the device if the light is of type **D3DLIGHT\_SPOT** (a spot light).

Your application can call **IDirect3DDevice9::SetLight** at any time to update light properties. This allows for dynamic effects like pulsing lights or lights that move about the level. In Lab Project 5.1, we will use this technique to move some lights around our terrain in real time.

### Light Limits

There is no maximum limit on the number of lights that you can set (memory permitting), but there is a limit on the number of lights that can be active at any one time. This limit is generally eight active lights even on many of the latest 3D graphics cards. You can find out how many lights can be simultaneously enabled by checking the **MaxActiveLights** member of the **D3DCAPS9** structure. The following snippet of code demonstrates how you could retrieve and store the number of active lights supported by the current device.

```
DWORD      MaxLights;
D3DCAPS9  DeviceCaps;
pD3DDevice->GetDeviceCaps (&DeviceCaps);
MaxLights = DeviceCaps.MaxActiveLights;
```



Although this limit might seem offputting at first, it is quite unlikely that the application would need any more than eight lights to affect a single vertex. Often only two or three will suffice. Too many lights may result in washing out the vertex as the light colors are accumulated.

While the light limit does not pose a problem from the perspective of a single vertex, it is common that a game world will have many lights, perhaps even hundreds. So the active light limit does mean that we will need to implement some form of light management system that enables lights in the immediate vicinity of the object being drawn and disables the rest.

Finally, although vertex lighting is fast when compared to other more complex lighting techniques, each light adds a per-vertex cost. Keeping the number of lights for any particular group of vertices to a minimum is an important performance consideration.

Lab Project 5.2 will implement a basic light management system. It will load in level data created with the GILEST™ level editor that will have many lights in it. We will learn how to implement a system that allows us to enable only the minimum amount of lights for each rendered object and still achieve good visual results.



## Light Types

There are three types of direct light sources that can be added to the scene with the SetLight function. Each behaves differently and requires certain members of the D3DLIGHT9 structure to be filled in correctly. DirectX Graphics supports **point lights**, **spot lights** and **directional lights**.

```
typedef enum _D3DLIGHTTYPE
{
    D3DLIGHT_POINT = 1,
    D3DLIGHT_SPOT = 2,
    D3DLIGHT_DIRECTIONAL = 3,
    D3DLIGHT_FORCE_DWORD = 0x7fffffff
} D3DLIGHTTYPE;
```

### Point Lights

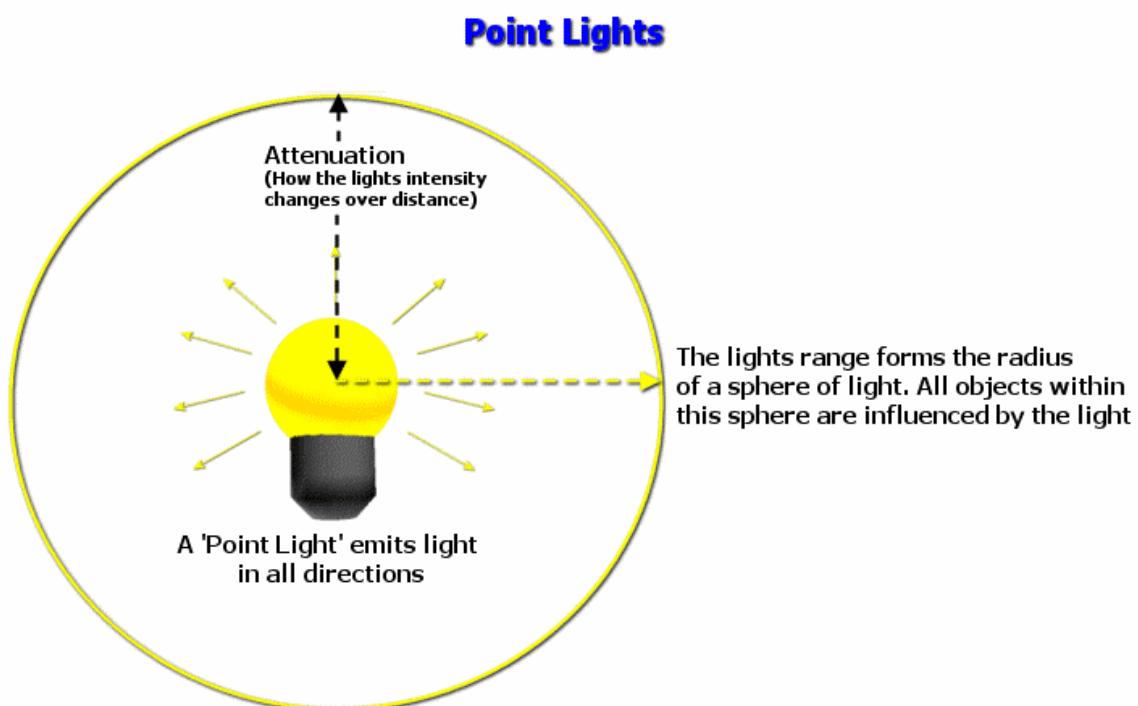


Figure 5.13

Point lights will probably be a light type you will use very often, as it works much like a light bulb in the real world. When we create a point light, we supply a world space position as well as a range value that describes a spherical bounding radius around the light position. This is the range of the light's intensity. Any vertices that are close enough to the light source to fall within its radius will have their color influenced to some degree, provided the vertex is not facing away from the light source. Point lights emit light from the center outward in all directions as Fig 5.13 shows.



A point light can be initialized so that the amount of light a vertex within its range receives is scaled by the distance from the vertex to the light. This property is called **attenuation**. The D3DLIGHT9 structure has three attenuation members that allow us to supply constants to be used in the attenuation equation. We will talk about this equation in more detail later in the lesson. Vertices that are outside the range of the light are not influenced by the light at all and are quickly rejected by the lighting module. At the outer ranges of the point light radius the contribution of that light on the vertex is very slight. Vertices gradually fade out of range rather than transitioning abruptly at the boundary of the sphere. This minimizes any sudden changes from light to dark.

The following code sets up a point light that emits yellow diffuse light, white specular light, and dark blue ambient light.

```
D3DLIGHT9 MyPointLight;
MyPointLight.Type = D3DLIGHT_POINT;
```

First we fill in the desired light type, which in this case is a point light. Next we will setup the three color values that any light source can emit: diffuse, specular, and ambient. Each is represented in the D3DLIGHT9 structure as a D3DCOLORVALUE structure:

```
typedef struct _D3DCOLORVALUE
{
    float r;
    float g;
    float b;
    float a;
} D3DCOLORVALUE;
```

Each member is normally in the range [0.0, 1.0] although numbers outside the range can be used. A value of 1.0 for any color component means the component is at full intensity. If we set the r, g, and b fields to 1.0 then the color will be bright white. If we set all the members to 0.0, the color will be black. The alpha component is ignored when specifying light colors.

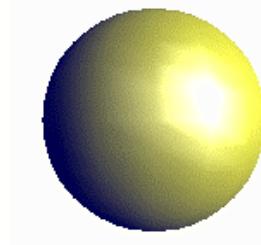
```
MyPointLight.Diffuse.r = 1.0f;
MyPointLight.Diffuse.g = 1.0f;
MyPointLight.Diffuse.b = 0.0f;
```

If the currently set material reflects the red and green components of diffuse light, then the vertex color will have the resulting yellow diffuse color added to its color. Next we can set up the specular color of the light source. Usually you will want specular highlights to be white, but you can set any value you wish:

```
MyPointLight.Specular.r = 1.0f;
MyPointLight.Specular.g = 1.0f;
MyPointLight.Specular.b = 1.0f;
```

Because specular lighting is view dependant, the number of vertices that have their color modified by the specular calculation is typically small – although the calculation itself still takes place for each. The angle between the camera, the vertex and the light direction vector is used to scale the specular color of the light source. Then the color is scaled by the attenuation values so that the distance from the vertex

to the position of the light is taken into account. The remaining color is modulated with the specular reflectance property of the material and added to the diffuse color and ambient color of the vertex.



**Figure 5.14**

The sphere in Fig 5.14 has a material that reflects all diffuse, specular, and ambient light. The specular highlights are white even though the color of the sphere is mostly yellow. The sphere is reflecting all of the yellow light it receives. Notice that the polygons facing away from the light source are colored by dark blue ambient light.

The D3DLIGHT9 structure includes an Ambient member. Earlier we talked about ambient lighting at the global level set through a render state call. Additionally, each light source can also emit its own ambient light color which is used when the ambient light for each vertex is calculated by the pipeline. The ambient light that is added to each vertex is equal to the sum of the global ambient light color and all of the other ambient lights that influence that vertex. The following code sets the ambient light emitted by the light source to dark blue.

```
MyPointLight.Ambient.r = 0.0f;
MyPointLight.Ambient.g = 0.0f;
MyPointLight.Ambient.b = 0.2f;
```

When a vertex is in range of a light source that includes an ambient color, the ambient light color is also added to the vertex color. Vertex normal orientation is not a factor here. However, unlike the global ambient light level that is applied to all vertices equally, the ambient color applied to each vertex from the direct light source is attenuated and is only received by vertices within the range of the light source. In Fig 5.14, no global ambient light level was used so only the vertices within the range of the light source were updated

## Position

Point lights and spot lights are both positional light sources with actual locations in the world. With a point light, light is emitted from this position equally in all directions. This is analogous to real life, where a bed side lamp for example emits light from a position in the real world -- the position of the light bulb.

To set the light position we fill in the **Position** member of the **D3DLIGHT9** structure. For example, to set the world space position of the light source to location (40, 90, 20):



```
MyPointLight.Position = D3DXVECTOR3(40.0f , 90.0f , 20.0f );
```

## Direction

Direction is a unit length 3D vector which specifies the direction that the light source is shining. Point lights emit light equally in all directions so this member will not be used for this light type. Directional lights and spot lights do not emit light equally in all directions and we will specify an orientation vector for those light sources.

## Range

Range determines the outer boundary of the light source. Point lights and spot lights use the Range member to determine if a particular vertex falls within their zone of influence. Directional light sources do not specify a range because they are infinite. For a point light, range describes a spherical radius. DirectX Graphics can quickly calculate whether a vertex is outside this sphere and reject it before more costly calculations are done.

## FallOff

The FallOff member of the **D3DLIGHT9** structure is only used with spot lights. We will discuss the FallOff member when we discuss spot lights later in the lesson.

## Attenuation

Under normal circumstances the illumination from a point light source decreases according to the inverse square of the distance between the light source and the surface (or vertex in this case):

$$\text{Attenuation} = 1/D^2$$

$$D = |(\text{VertexPosition} - \text{LightPosition})|$$

Attenuation scales the intensity of the light as distance increases between the light source and the surface. The problem with using the attenuation formula above is that as distance values get closer to the maximum range of the light source, the difference between values becomes insignificant. As distance gets closer to zero the variations become much larger very quickly. Coefficients and new variables can be added to the equation to address some of these concerns:

$$\text{Color}_{\text{(current)}} = \frac{1}{\text{Attenuation1} + (\text{Attenuation2} \times D) + (\text{Attenuation3} \times D^2)} * \text{Color}_{\text{(orig)}}$$

The Attenuation1 member describes a **constant attenuation** factor, Attenuation2 describes a **linear attenuation** factor, and Attenuation3 describes a **squared attenuation** factor. These values taken together can create a variety of attenuation curves. Notice that the equation does not take account of



the Range value of the light source in any way. The constant value allows us to avoid 0 in the denominator.

Generating the correct attenuation curve requires some degree of experimentation to get the correct falloff over the range of the light. Vertices that are outside the range of the light are not lit at all, so if you do not set the attenuation values properly (so that the intensity of the light degrades to approximately zero as D reaches the range of the light), you will see a sharp cutoff point. Therefore you must think carefully about the values you use. For example, if you set the following attenuation values, you would get no attenuation at all:

$$\text{Attenuation1} = 1.0 \quad \text{Attenuation2} = 0.0 \quad \text{Attenuation3} = 0.0$$

$$\text{Attenuation Value} = \frac{1}{1 + (0 \times D) + (0 \times D^2)} = 1.0$$

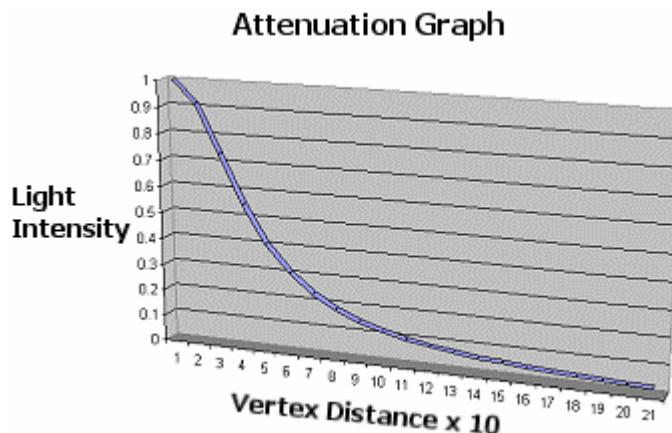
$$\text{Color}_{(\text{current})} = \text{Attenuation Value} * \text{Color}_{(\text{orig})}$$

$$\text{Color}_{(\text{current})} = \text{Color}_{(\text{orig})}$$

In this case vertices within the range of the light are lit with the same intensity irrespective of the distance between the vertex and light source. If you want your light intensity to attenuate, then you will often have to experiment with these values so that they provide attenuation consistent with the range of the light. One easy way to do this is to manipulate the values in a spreadsheet so that the intensity of the light is near zero at the light's range.

In the following attenuation graphs, the point light source has a range of 200 world units. The goal is to find values such that the light intensity is close to full power (1.0) at a distance of zero from the light source and decreased to nearly zero near the outer range (i.e. at a distance of 200 units from the light).

**Example 1:**       $\text{Attenuation1} = 1.0 \quad \text{Attenuation2} = 0.0002 \quad \text{Attenuation3} = 0.0009$

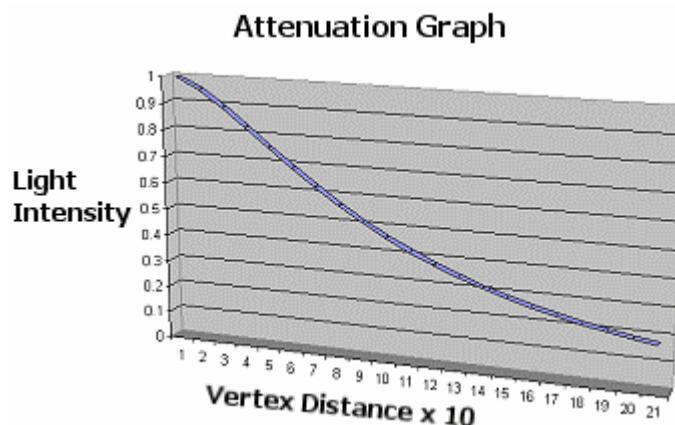


The above graph shows how the color emitted from the light would be scaled by distance. At a distance of 200 units it is close enough to zero not to produce a noticeable illumination discontinuity at the edge of the light's range. Changing the attenuation values allows you to modify the shape of the graph. This is certainly not a very linear attenuation. Look at the way the curve dips within the first 90 units of the light's range. Only 50 units from the light source, the color that would reach the vertex is less than half of its full intensity.

The results are subjective and it can take some trial and error to find an attenuation curve you are happy with. Often you will not be able to manipulate the coefficients of the equation to produce the exact curve that you desire but you can usually find something that looks good in practice.

In this next example, we used some different attenuation values for the same light source. Once again, the goal is to make the attenuation decrease as near to zero as possible at the light's max range. The following values are not quite as successful at doing this as the last example, but it does provide a more linear attenuation.

**Example 2:** Attenuation1 = 1.0    Attenuation2 = 0.004    Attenuation3 = 0.0001

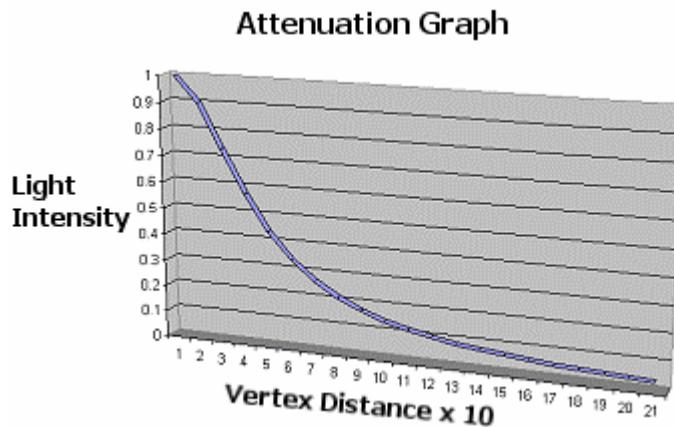


The graph above indicates a much more linear falloff, but at the max range (200) the light intensity is still just under 0.2. This indicates that vertices just inside the range will still be receiving 20% of the light's full power while vertices just outside the lights range will receive none. This may cause a visual glitch in the lighting but it is not likely to be too noticeable.

Adjusting the squared coefficient from 0.0001 to 0.0007 drastically alters the shape of the graph so that it is very close to zero at the light's range but without as linear an attenuation:



**Example 3:** Attenuation1 = 1.0 Attenuation2 = 0.004 Attenuation3 = 0.0007



If you wish your light source to have its intensity attenuate, you will need to create attenuation values that suit the range of your light source. The above values would not be at all suitable for a light that only had a range of 30 units because at the light's outer range, the vertex would still receive approximately 65% of the color. This would cause a steep decline in illumination for vertices just outside the lights range.

For those of you familiar with the Microsoft Excel® spreadsheet application, we have included with this lesson the spreadsheet used to create the attenuation graphs in the previous pages. This will allow you to experiment with the coefficients to produce settings that suit your own light sources.

Point lights and spot lights both use the Attenuation1, Attenuation2, and Attenuation3 members of the **D3DLIGHT9** structure as coefficients in an equation that scales light intensity based on the distance from the vertex to the light source. Directional lights do not use the attenuation values because they have infinite range.

### Theta and Phi

Theta and Phi are only used for spotlights. We will cover spot lights in the next section.

In this next example, we will create a point light at world space position (10, 60, 20) with a range of 200 units. We will use the attenuation settings from our final attenuation graph above.

```
// Setup a point light
D3DLIGHT9 MyLight;
ZeroMemory(&MyLight, sizeof(D3DLIGHT9));

MyLight.Type = D3DLIGHT_POINT; // Point Light

MyLight.Diffuse.a = 1.0f; // Blue Diffuse Light
MyLight.Diffuse.r = 0.0f;
MyLight.Diffuse.g = 0.0f;
```



```
MyLight.Diffuse.b = 1.0f;  
  
MyLight.Specular.a = 1.0f; // White Specular Light  
MyLight.Specular.r = 1.0f;  
MyLight.Specular.g = 1.0f;  
MyLight.Specular.b = 1.0f;  
  
MyLight.Ambient.a = 1.0f; // Dark Grey Ambient Light  
MyLight.Ambient.r = 0.2f;  
MyLight.Ambient.g = 0.2f;  
MyLight.Ambient.b = 0.2f;  
  
MyLight.Position = D3DXVECTOR (10.0f , 60.0f , 20.0f); // Position  
MyLight.Range = 200; // Range  
  
MyLight.Attenuation0 = 1.0f; // Attenuation Coefficients  
MyLight.Attenuation1 = 0.004;  
MyLight.Attenuation2 = 0.0007;  
  
m_pDevice->SetLight (0, &MyLight); // Set this light as light index zero  
m_pDevice->LightEnable(0, TRUE); // Turn the light on
```

## Spot Lights

A spot light is a positional light that emits light in a specific direction. A DirectX spot light is very much like the type of spot light you would see in a theatre: a cone shaped beam of light shining toward the stage. The cone can be narrowed or widened to focus the beam of light on a single actor or an entire cast. When we fill in the **D3DLIGHT9** structure for a spot light, we set the position and direction of the light vectors in the **D3DLIGHT9** Position and Direction members respectively. The Phi and Theta members are angle values used to define the light cone exiting the spotlight and traveling along its direction vector. Just as a point light has a sphere of influence in which all vertices inside it are considered for lighting, the Phi and Theta members define a cone of influence for the spot light. Technically, these values describe two cones of influence: an inner and outer cone as we will see momentarily.

### Spot Lights

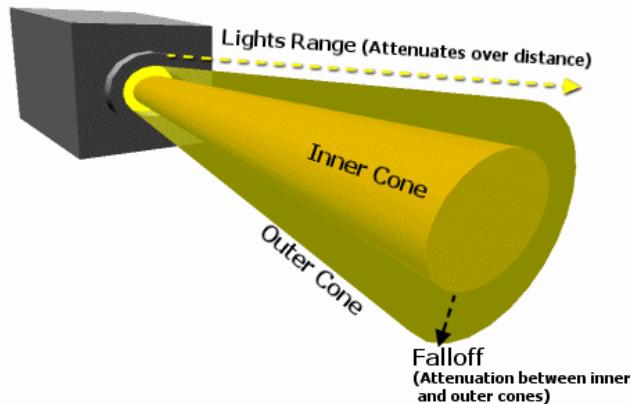


Figure 5.15



Let us begin by discussing which members of the **D3DLIGHT9** structure are used to create a spot light effect and how to fill them in correctly.

### **Diffuse, Ambient, and Specular**

The diffuse, ambient, and specular members of the **D3DLIGHT9** structure are the same for spot lights as they were for point lights. They describe the diffuse, ambient, and specular colors emitted by the spot light. Any vertices that are within the range of the light and within its cone of influence will receive these colors to some extent, provided the vertex normals are not facing away from the light source. How much of this color ultimately makes it into the vertex is dependant on how we set the attenuation values, the falloff values, and the inner and outer cone angles in the **D3DLIGHT9** structure. It also depends on the angle between the vertex normal and the light direction vector (excluding the ambient color which is not scaled by this angle but still has range and attenuation applied).

### **Position**

A spot light is assigned a position in the 3D world that describes the origin of the light source.

### **Direction**

A spot light must be assigned a direction vector that describes the direction the spot light beam travels in the 3D world.

### **Range**

Like a point light source, the Range member of the D3DLIGHT9 structure is interpreted as the radius of a sphere of influence. Values outside this range are not affected by the light. If a vertex is within range of the light, then another test is performed to test if it is within the outer cone of light. If the vertex is outside the outer cone, it will not receive any of the light color. If the vertex is within the outer cone and is not facing away from the light source, then the vertex will receive at least some of the light color.

### **Attenuation**

The Attenuation1, Attenuation2, and Attenuation3 members of the D3DLIGHT9 structure are the same for spotlights as they are for point lights. They can be configured so that the light intensity diminishes in relation to the vertex distance from the light source. The distance value considered is the length of a vector from the light source position to the vertex.

### **Phi and Theta**

The Phi and Theta members of the D3DLIGHT9 structure describe the angle (in radians) of the outer and inner spotlight cones respectively. Vertices that fall within the inner cone receive the color of the light scaled by distance using the attenuation values. Vertices that are outside the inner cone but within the outer cone receive the color as well, but it is scaled by another form of attenuation called *falloff*.

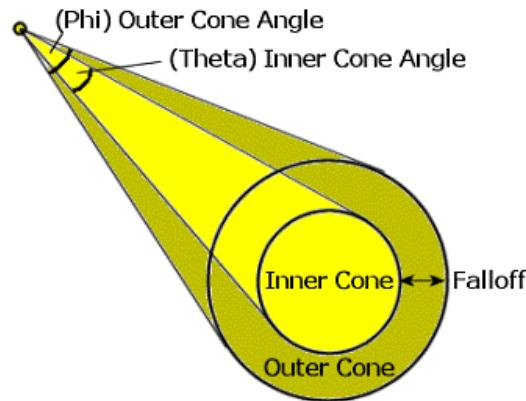


Figure 5.16

Fig 5.16 shows that Phi describes the angle of the outer cone and Theta describes the angle of the inner cone, both projecting outwards from the light source along the direction vector. You can set up the inner and outer cones of the spot light so that vertices within the inner cone are lit more brightly than vertices outside the inner cone but still inside the outer cone. Vertices in the outer cone can slowly fade in intensity as they approach the boundary of the outer cone. Obviously, if you set both angles to the same value then you will see no falloff on the vertices. By setting the Falloff member of the **D3DLIGHT9** structure we can model an additional attenuation.

### FallOff

The FallOff member of the **D3DLIGHT9** structure attenuates the light color for vertices falling within the outer cone of the light but outside the inner cone. To calculate falloff the pipeline first measures the angle between the direction vector and a vector from the light to the vertex to return the cosine of the angle between them.

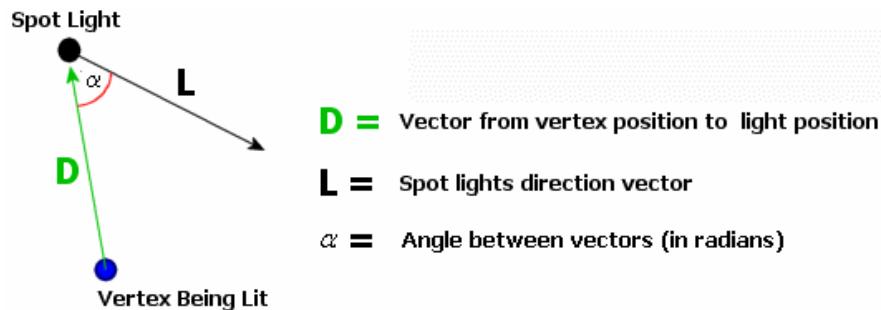


Figure 5.17

The vector  $\mathbf{L}$  is negated and the dot product performed to return the cosine of the angle between the two vectors.

$$\cos \alpha = -\mathbf{L} \bullet \mathbf{D}$$



This angle can then be directly compared to the outer and inner cone angles (Phi and Theta) specified in the D3DLIGHT9 structure to determine if the vertex is within the inner or outer cones. The Phi and Theta values are divided by 2 prior to being compared with  $\alpha$  because they describe total cone angles when we actually want the angle relative to the direction vector  $\mathbf{L}$  at the center of the inner cone. The cones have an angle that sweeps out  $\alpha/2$  on each side of vector  $\mathbf{L}$ .

After Phi and Theta have been divided by 2, the pipeline compares them with  $\alpha$  to see if the vertex is within the cones. If  $\alpha$  is larger than the Phi/2 then it is totally outside the outer cone and is rejected from further consideration for this light. If this is not the case then  $\alpha$  is compared against Theta/2. If  $\alpha$  is less than Theta/2, then it is within the inner cone and will not have the additional attenuation (FallOff) applied to it. Remember that *Falloff* is separate from the three attenuation settings that scaled the intensity of the light based on the distance from the light source. Even if the vertex is at the center of the cone, it will still have the distance based attenuation equation applied to it. However, it will not be scaled by the falloff equation. If the vertex does lie within the outer cone but is outside the inner cone, the color is additionally scaled by the result of the following equation:

$$S = \left( \frac{\cos(\alpha) - \cos(\phi)}{\cos(\theta) - \cos(\phi)} \right)^{Falloff}$$

$\alpha$  = Angle between vectors  $\mathbf{L}$  and  $\mathbf{D}$

$\phi$  = Phi / 2 (Half the outer cone angle)

$\theta$  = Theta / 2 (Half the inner cone angle)

*Falloff* = Falloff property of the **D3DLIGHT9** structure

The equation produces an  $S$  value between 0.0 and 1.0. This is used to further scale the light color to account for falloff. This is referred to as the **spot factor**. By altering the *Falloff* property of the **D3DLIGHT9** structure we can adjust the falloff curve. Unlike the distance based attenuation that we discussed earlier (which is also applied to the color before reaching the vertex), the falloff curve is not nearly as difficult to manipulate since it does not rely on world space distances. This means for example that a falloff value of 1.0 will always give a linear falloff regardless of the other properties of the light. This should be clear enough given that an exponential value of 1.0 in the above equation does not affect the ratio. This approach allows you to create consistent falloff curves that can be used by all of your spot lights.

The graph in Fig 5.18 shows three examples of falloff values: 1.0, 0.2, and 5.0.



## Falloff Graph

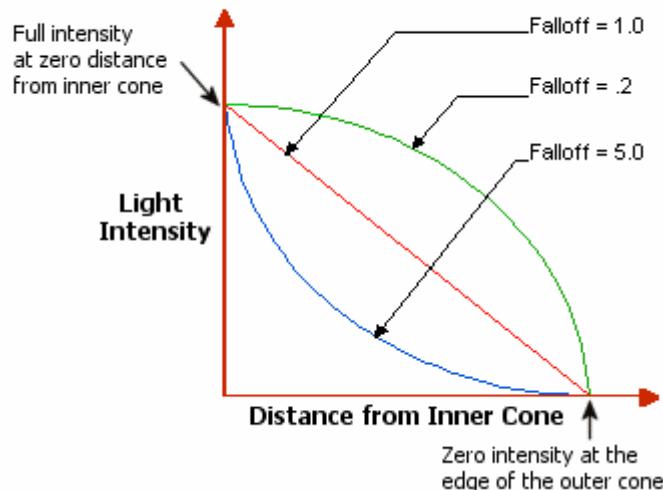


Figure 5.18

Your applications will usually set the Falloff member to 1.0 which causes linear attenuation between the inner and outer cones to be applied to vertices in the outer cone. There is a small performance penalty incurred when using falloff values other than 1.0 and as the attenuation effect between the inner and out cones is usually extremely subtle it is often not worth doing.

In the next code example we create a spot light at position(10, 50, 10) with a direction vector that orients it down the positive Z axis (0, 0, 1). We set the range of the light to 400 and use the same attenuation values we used in our point light example. We set the falloff value of the structure to 1.0 to provide linear attenuation between the inner and outer cones, where the angle of the outer cone is 60 degrees and the inner cone is 30 degrees. The light emits green diffuse light, white specular light, and no ambient light (black ambient color). Finally we set the light as light index 1. If we used this code after the preceding point light code we would have the point light as light 0 and the spot light as light 1.

```
// Setup a point light
D3DLIGHT9 MyLight;
ZeroMemory(&MyLight, sizeof(D3DLIGHT9));

MyLight.Type      = D3DLIGHT_SPOT;           // Spot Light
MyLight.Diffuse.a = 1.0f;                     // Green Diffuse Light
MyLight.Diffuse.r = 0.0f;
MyLight.Diffuse.g = 1.0f;
MyLight.Diffuse.b = 1.0f;

MyLight.Specular.a = 1.0f;                    // White Specular Light
MyLight.Specular.r = 1.0f;
MyLight.Specular.g = 1.0f;
MyLight.Specular.b = 1.0f;
```



```
MyLight.Ambient.a      = 1.0f;           // No ambient light
MyLight.Ambient.r      = 0.0f;
MyLight.Ambient.g      = 0.0f;
MyLight.Ambient.b      = 0.0f;

MyLight.Position       = D3DXVECTOR (10.0f , 50.0f , 10.0f);    // Position
MyLight.Direction       = D3DXVECTOR3( 0.0f , 0.0f ,1.0);      // Direction vector

MyLight.Range          = 400;            // Range

MyLight.Attenuation0   = 1.0f;           // Attenuation Coefficients
MyLight.Attenuation1   = 0.004;
MyLight.Attenuation2   = 0.0007;

MyLight.Theta          = D3DXToRadian (30);        // Inner Cone angle (30 degrees)
MyLight.Phi             = D3DXToRadian (60);        // Outer Cone angle (60 degrees)
MyLight.Falloff         = 1.0f;           // Falloff (linear = 1.0)

m_pDevice->SetLight (1, &MyLight);           // Set this as light index 1
m_pDevice->LightEnable(1, TRUE);           // Turn the light on
```

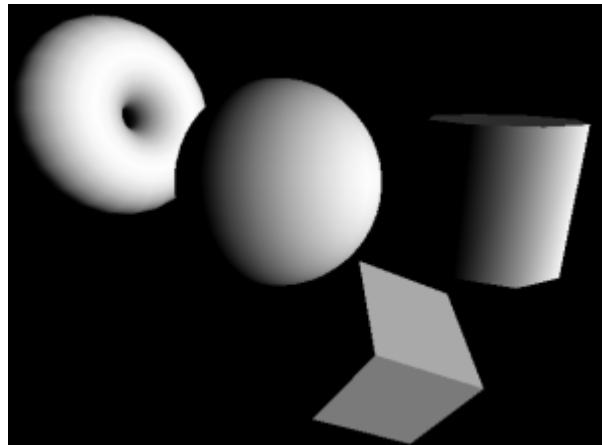
Spot lights are generally more computationally expensive than point lights because of the extra calculations involved for both determining if the vertex is within the inner or outer cones and for calculating the falloff between the cones. While this is true on a per-vertex basis, these unique properties of a spot light are such that they will generally affect fewer vertices in the scene than other light types. As a result, using spot lights can sometimes result in better performance than point lights under certain circumstances. Nevertheless, as a general rule, they are the most computationally expensive light type, followed by point lights, and then directional lights.

## Directional Lights

Directional lights are the least computationally expensive light source and are ideal for simulating far away light sources such as the sun. A directional light does not have a position in the 3D world and does not have a limited range. A directional light is essentially nothing more than a unit length direction vector describing the direction in which the light is shining on all vertices in the scene. A directional light can emit ambient, diffuse, and specular colors just like point and spot lights, but the Position, Range, Falloff, Theta, Phi and the three attenuation members of the D3DLIGHT9 structure are all unused.

This is very much like the type of lighting that we used in Lab Project 3.2 when we calculated the colors of the terrain vertices. When there is a directional light in the scene, all vertices have their vertex normals compared to its direction vector and the diffuse and specular colors of the light are scaled by the cosine of the angle between them. Its ambient color is applied without prejudice to all vertices in the scene. Because the diffuse and specular colors are scaled by the angle between the vertex normal and the light direction vector, this simplified lighting calculation still provides smooth lighting. As vertex normals become less parallel to the light source direction vector, the colors received by the vertex are scaled down to a greater degree. We can think of the directional light source as emitting an

infinite number of rays of light parallel to the direction vector throughout the 3D world. Fig 5.19 shows four meshes that are lit by a bright white directional light shining from right to left.



**Figure 5.19**

Remember that a directional light has no position. Every vertex in the world perceives the directional light to be an infinite distance away shining rays along the same vector. In Fig 5.19, notice that there are no ambient light sources in the scene, so vertices facing away from the direction vector do not receive any light. This creates a smooth fade to black effect for the sphere, cylinder, and torus. Since the cube has many fewer faces, the change from light to dark is much less gradual. It looks as if the back face of the cube (with relation to the light vector) is missing, but it is there. It simply receives no light because the vertices of that face have normals that are facing away from the direction vector. This is a good example of why we might consider setting a global ambient color; so that such faces are rendered using at least a very low intensity color. Again, a possible alternative is to use a low intensity directional light that shines along the view vector. You will typically use only 1 or 2 directional light sources in your scene (if at all) since they will affect all vertices.

The following code creates a red diffuse directional light with white specular light and a small amount of blue ambient light. The light has a direction vector aligned with the world's negative X axis (i.e. the light is emitting parallel rays from right to left in world space) and is set at index 2.

```
// Setup a directional light
D3DLIGHT9 MyLight;
ZeroMemory(&MyLight, sizeof(D3DLIGHT9));

MyLight.Type      = D3DLIGHT_DIRECTIONAL;           // Spot Light
MyLight.Diffuse.a = 1.0f;                            // Green Diffuse Light
MyLight.Diffuse.r = 0.0f;
MyLight.Diffuse.g = 1.0f;
MyLight.Diffuse.b = 1.0f;

MyLight.Specular.a = 1.0f;                           // White Specular Light
MyLight.Specular.r = 1.0f;
```



```
MyLight.Specular.g = 1.0f;
MyLight.Specular.b = 1.0f;

MyLight.Ambient.a = 1.0f;                                // No ambient light
MyLight.Ambient.r = 0.0f;
MyLight.Ambient.g = 0.0f;
MyLight.Ambient.b = 0.0f;

MyLight.Direction = D3DXVECTOR3( 0.0f , 0.0f ,1.0);        // Direction vector

m_pDevice->SetLight (2 , &MyLight);                      // Set as light index 2
m_pDevice->LightEnable(2, TRUE);                         // Turn the light on
```

At this point we know how to set up the three different light types. We also know (roughly) how the pipeline scales the colors emitted from a light source using the angle between the vertex normal and the light source. We examined the attenuation equation, which further scales the light source based on a distance from the light, and in the case of a spot light, the colors may be further attenuated by a falloff equation. So a vertex may receive only a fraction of a light's full intensity when its position, orientation, and distance from the light source are taken into account.

Note that the color components of a **D3DLIGHT9** structure are not limited to the [0.0, 1.0] range, although it is the range we will most commonly use. We could for example create a light that has negative color components which would actually 'remove' light from the scene. This could be used to create shadowy areas within the level. Simply place a 'dark light' where you want the shadowy area to appear and any vertices that are within the influence of the light will have the light's color subtracted from their total color. The following snippet of code shows how you might setup the diffuse members of a **D3DLIGHT9** structure to create a light that removes diffuse lighting from the scene.

```
//dark light
MyLight.Diffuse.r = -0.5;
MyLight.Diffuse.g = -0.5;
MyLight.Diffuse.b = -0.5;
MyLight.Diffuse.a = 1.0;
```

You could also use this strategy to create a light that removes only certain color components from a scene (ex. a diffuse color with a negative red component to subtract only the redness from vertices). Note as well that we can also use intensity values that exceed 1.0 to create an oversaturation effect (i.e. a 'bright light'). Because the color of a vertex is clamped to its maximum color intensity (bright white), creating a bright light can cause vertices to have a much higher intensity color added to them. This causes them to be clamped to their maximum color much sooner. This causes a light saturation effect where every vertex within range is lit to full intensity.

**Note:** When you pass the D3DLIGHT9 structure into the IDirect3DDevice9::SetLight function, a copy of the information is made by the device and this copy is used. Therefore, after calling SetLight, you can safely delete the D3DLIGHT9 variable (or let it go out of scope) without it deleting or corrupting the light in the scene. After the call to SetLight, the passed structure is NOT attached to the physical light. It is just used to create it.

## Materials

Materials are used to determine how a vertex reacts to incoming light and ultimately, what color it is perceived to be. They do so by specifying which color components are absorbed and which are reflected. In Fig 5.20 we see a white directional light shining on the objects from right to left.

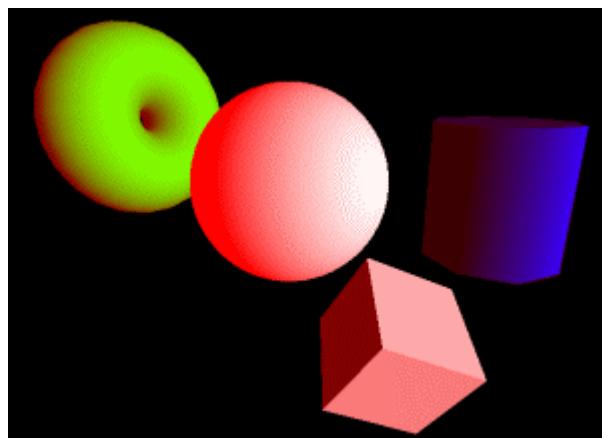


Figure 5.20

Although all of the objects in Fig 5.20 receive the same white light, each object is a different color because they have different materials that define how they react to the light. All of the objects above use a material that reflects red ambient light and we can see that the faces that point away from the light source take on a red coloring to some degree. The torus has a material that only reflects green diffuse light; the red and blue color components of the white directional light are absorbed by the material and only the green is reflected. The sphere uses a material that reflects all diffuse lighting so it appears white when fully lit and fades to red as faces leave the influence of the direct light source and have only the red ambient color applied. The material the cylinder uses reflects only blue diffuse light. The cube reflects all three color components but reflects the red color of a light source to a much stronger degree. It appears as a light red color since it also reflects some blue and green.

The device object has a memory slot for exactly one material, so we can only set one material at a time for rendering. Usually you will have a polygon structure that contains an index into an array of **D3DMATERIAL9** structures created by your application. When a triangle is rendered, we will need to make sure that the material it uses is set as the current device material by calling the IDirect3DDevice9::SetMaterial member function.

For each vertex, its final color is calculated by the pipeline as follows:



$$\text{Vertex Color} = (\text{AmbientLight} * \mathbf{A}) + (\text{DiffuseLight} * \mathbf{D}) + (\text{SpecularLight} * \mathbf{S}) + \mathbf{E}$$

**AmbientLight** –This is the total ambient light color that has reached a vertex after attenuation and falloff have been taken into account for direct light sources. The global ambient light color is also added to this color.

**DiffuseLight** –This is the total diffuse light color that has reached the vertex after angle, attenuation, and falloff have been taken into account.

**SpecularLight** –This is the total specular light color that has reached a vertex after angle, attenuation, and falloff have been taken into account.

**A, D, S** and **E** are colors that we can specify using either materials or vertex colors. We will ignore using vertex colors in this equation for the time being and concentrate on materials. Later we will see how we can use colors stored in the vertex as variables in the above equation.

Each value in the above equation is an ARGB group of floats. We are modulating inbound light color with colors stored in the material. The above equation could be rewritten as:

$$\begin{aligned}\text{VertexColor} = & (\text{AmbientLight}(a,r,g,b) * A(a,r,g,b)) + \\ & (\text{DiffuseLight } (a,r,g,b) * D(a,r,g,b)) + \\ & (\text{SpecularLight}(a,r,g,b) * S(a,r,g,b)) + E(a,r,g,b)\end{aligned}$$

If for example the color of A was (0.5, 0.5, 0.5, 0.5) then the incoming ambient light color would be scaled by half, giving the impression that the surface has absorbed half of the ambient light. If the incoming ambient light was bright white (1.0, 1.0, 1.0, 1.0) and the color A was (1.0, 0.0, 0.5, 0.0) we can see that the following ambient light reflectance calculation would occur:

$$\begin{aligned}\text{Vertex Color} &= (\text{AmbientLight} * \mathbf{A}) \dots \\ \text{Vertex Color} &= (1.0, 1.0, 1.0, 1.0) * (1.0, 0.0, 0.5, 0.0) \\ \text{Vertex Color} &= (1.0, 0.0, 0.5, 0.0)\end{aligned}$$

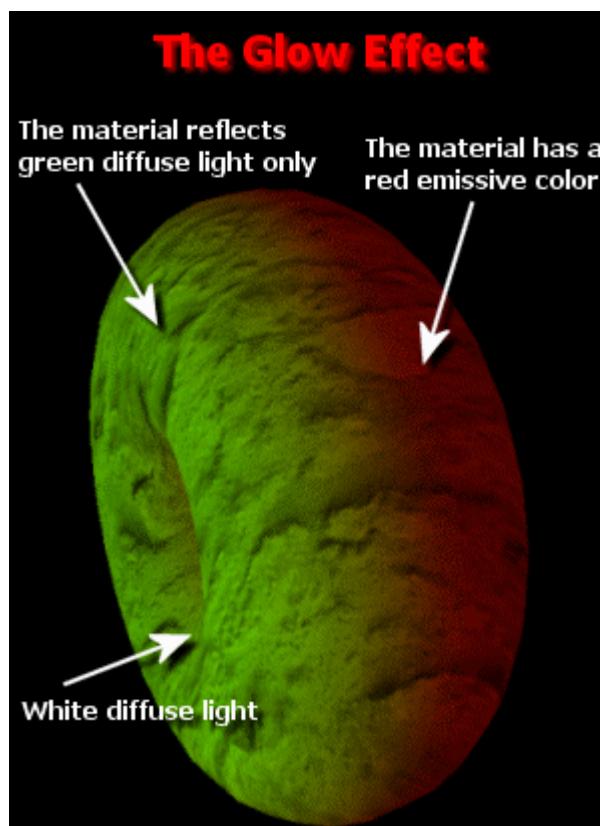
The red and blue ambient light would be completely absorbed by the surface and we would reflect green ambient light at half intensity. The same logic obviously holds true for the other color types.

So a material is really nothing more than a collection of four ARGB colors with color components in the range of 0.0 to 1.0. The structure used to represent materials in DirectX Graphics is called **D3DMATERIAL9**. We store an ambient, diffuse and specular color in this structure which maps to A, D and S in the above we equation respectively. Although you can think of these as colors that control the color of the vertex that uses it, you can also think of them as simple being a collection of four component values in the range of 0.0 to 1.0 that are used to scale the incoming ambient, diffuse and specular light.

The **D3DMATERIAL9** structure contains a fourth color called Emissive (**E** in the above equation). The emissive color of a material is a color that is always added to the vertex even if it is not receiving any

light whatsoever. We can think of this as a color emitted by the vertex itself. Unlike light sources, the color emitted by the vertex is not projected onto neighboring vertices.

Fig 5.21 shows a torus rendered using an emissive red color and a green diffuse color. There is a single bright white diffuse directional light source in the scene. Notice that the vertices that are facing the directional light source reflect only the green component of the light color. Also notice how the vertices that are facing away from the light source receive no light but that the material always emits a red emissive color.



**Figure 5.21**

Emissive colors in materials can look really nice. You often see them used in space combat games to emit a glow effect on the engines of a space craft. As the space ship leaves the sun's illumination, its hull fades into darkness. But its engines still glow fully bright. This can be done using a material with an emissive property when rendering the engines. You might also use emissive colors in a first-person shooter game on the mesh of a fireplace so that the bricks glow an orange color even when there is no light shining on the fireplace.

The **D3DMATERIAL9** structure is shown below. The Power floating point member is used to control the application of specular highlights and will be discussed momentarily.



```
typedef struct _D3DMATERIAL9
{
    D3DCOLORVALUE Diffuse;    // D in the above equation
    D3DCOLORVALUE Ambient;   // A in the above equation
    D3DCOLORVALUE Specular;  // S in the above equation
    D3DCOLORVALUE Emissive;  // E in the above equation
    float Power;           // Controls the sharpness of highlights
} D3DMATERIAL9;
```

The following code example creates a material that reflects only green diffuse light and emits its own red color as seen in Fig 5.21. No specular or ambient light are reflected.

```
D3DMATERIAL9 mat;

mat.Diffuse.a = 1.0f;
mat.Diffuse.r = 0.0f;
mat.Diffuse.g = 1.0f; // reflects only green diffuse light
mat.Diffuse.b = 0.0f;

// reflects no ambient or specular light
mat.ambient.a = mat.ambient.r = mat.ambient.g = mat.ambient.b = 0.0f;
mat.specular.a = mat.specular.r = mat.specular.g = mat.specular.b = 0.0f;

mat.Emissive.a = 1.0f;
mat.Emissive.r = 1.0f; // the material emits its own red color onto vertices that use it
mat.Emissive.g = 0.0f;
mat.Emissive.b = 0.0f;

mat.Power = 0.0f;      // Not reflecting specular highlights so set power to zero
```

Once we have the material structure filled out, we can set the material as the current device material using the `IDirect3DDevice9::SetMaterial` function:

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

We pass in the address of our material structure so that the device can copy the values into its local material properties memory. Our code would do something like this:

```
m_pd3dDevice->SetMaterial( &mat );
```

**Note:** After the call to `SetMaterial`, a copy is made of the material properties by the device. The material structure can be safely deleted (or allowed to go out of scope) without affecting the material settings of the device.

At this point we can render all of the polygons that use this material and they will have the correct reflectance properties applied in their color calculations. Every triangle that is rendered thereafter will use this material until such a time that the material is changed to a new material or the lighting pipeline is disabled. Therefore, you could create many material structures in an array and store indices into this array in your object or polygon structures to set the required material before rendering. In our workbook we will discuss batch rendering so that we are not calling `SetMaterial` before we render



every polygon. This would be very slow and render state changes should be minimized as they can be expensive.

### Specular and Power

We know that the specular color of the material is used as a multiplier with the specular color received from the light source to generate the specular color that will be applied to the vertex. However, specular highlights on objects are spread across many vertices depending on the position of the camera and the light source with respect to the object. The Power member is used to control the dispersion of highlights across the range of vertices. Specular highlights are more computationally expensive than calculating the diffuse lighting for a vertex because we need to factor in view dependant information. As a result, specular highlights are disabled by default and must be enabled by setting the following renderstate:

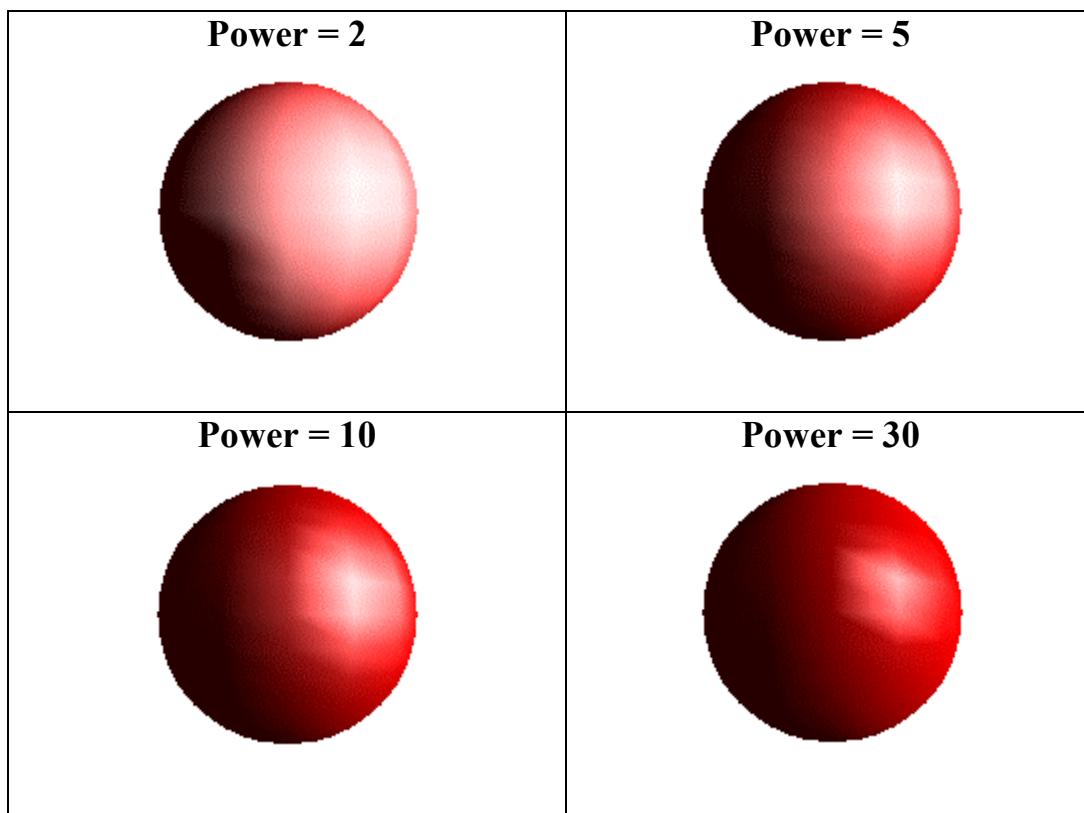
```
m_pDevice->SetRenderState (D3DRS_SPECULARENABLE , TRUE);
```

If you set this condition to false, specular highlights will not be calculated and the specular material properties will be ignored. This speeds up the lighting pipeline but the visual results are not as pleasing.

Although the code we saw earlier set Power to 0.0 and the specular color to black, you must be sure to disable the **D3DRS\_SPECULARENABLE** renderstate to prevent the specular calculation from occurring unnecessarily.

Let us look at how the Power member of the material can be used to alter the spread and intensity of specular highlights. Higher powers yield sharper specular highlights and vice versa. The range of values that you can use in the Power member is the range of a float. You will usually wind up using values between 0.0 and 250.0.

Fig 5.22 depicts a sphere illuminated by a white (diffuse and ambient) directional light. The material used to render the sphere reflects red diffuse light, white specular light, and dark red ambient light. Altering the Power of the material changes the size of the specular highlight:



**Figure 5.22**

At this point we have now covered the requirements for using DirectX Lighting:

- Create materials for your objects.
- Add your lights to the scene.
- Enable lighting.
- Enable specular highlights if desired.
- Begin render loop.
  - Set material for current triangles to be rendered.
  - Render triangles that use the currently set material.
  - Repeat until all triangles have been rendered.
- End render loop.
- Present scene.

Notice that we render the triangles in batches based on the materials they use. This is important because we want to reduce number of calls to DrawPrimitive and render (within reason) as many triangles as is optimal. The problem is that we can only set a different material between DrawPrimitive calls. Therefore, in order to reduce the number of calls to DrawPrimitive, you will want to reduce the number of times a new material has to be set. In order to do this you will want to (in a pre-process)



batch your triangles together into groups depending on the material they use. Then you can just set material 1 and render all triangles that use material 1, then set material 2 and render all triangles that use material 2, and so on. This allows you to render the scene with the minimum amount of DrawPrimitive calls and a minimal number of material changes. Changing state assigned objects like materials, textures, and vertex buffers can be expensive operations inside of a render loop (although some are certainly far more expensive than others). We should always look to minimize state changes as best we can.

## Material Sources

What happens if we enable lighting and give our vertices a vertex normal, but also include diffuse and specular color components at the same time? In previous weeks we treated the inclusion of these colors as a sign that the vertices were pre-lit. We might assume that when using the lighting pipeline, any color stored in the vertex would be ignored. As we will soon see, this assumption may or may not be true depending on how we set up our material source render states. Let us look again at our simplified equation for calculating the color of a vertex:

$$\text{Vertex Color} = (\text{AmbientLight} * \mathbf{A}) + (\text{DiffuseLight} * \mathbf{D}) + (\text{SpecularLight} * \mathbf{S}) + \mathbf{E}$$

Recall that in this equation, AmbientLight, DiffuseLight and SpecularLight represent the total amount of light that hits the vertex collected from all light sources. This light was scaled by distance attenuation and the angle of the vertex normal with the incident light vector. Then each sub-group of the total light is scaled by the corresponding material component ( $\mathbf{A}$ ,  $\mathbf{D}$ ,  $\mathbf{S}$ , and  $\mathbf{E}$ ).

We can change these relationships so that if desired, we could store a color (or two colors) in the vertex itself and can instruct the device not to use a particular material component as the reflectance property but instead use the color in the vertex. This means for example, that you could instruct the device to use the currently set material members for calculating ambient ( $\mathbf{A}$ ) and diffuse ( $\mathbf{D}$ ) reflection, but use the two colors stored in each vertex as the specular ( $\mathbf{S}$ ) and emissive ( $\mathbf{E}$ ) reflectance properties. This would allow a per-vertex emissive property which is simply not possible using materials. This is just one example and there are many combinations available.

There are four device render states that allow the application to configure the source of  $\mathbf{A}$ ,  $\mathbf{D}$ ,  $\mathbf{S}$  and  $\mathbf{E}$  in the above equation. They determine whether a reflectance property is taken from the currently set material or from one of the two color components that can be stored in a vertex.

```
SetRenderState ( D3DRS_DIFFUSEMATERIALSOURCE , D3DMATERIALCOLORSOURCE)
SetRenderState ( D3DRS_AMBIENTMATERIALSOURCE , D3DMATERIALCOLORSOURCE)
SetRenderState ( D3DRS_SPECULARMATERIALSOURCE , D3DMATERIALCOLORSOURCE)
SetRenderState ( D3DRS_EMISSIVEMATERIAL SOURCE, D3DMATERIALCOLORSOURCE)
```

We pass in a member of the D3DMATERIALCOLORSOURCE enumerated type:



```
typedef enum _D3DMATERIALCOLORSOURCE
{
    D3DMCS_MATERIAL = 0,
    D3DMCS_COLOR1 = 1,
    D3DMCS_COLOR2 = 2,
    D3DMCS_FORCE_DWORD = 0x7fffffff
} D3DMATERIALCOLORSOURCE;
```

**D3DMCS\_MATERIAL** - This instructs the device that the color for the current material source being set is found in the material. If we were currently setting the **D3DRS\_AMBIENTMATERIALSOURCE** render state and used **D3DMCS\_MATERIAL**, then the lighting calculations would use the ambient color stored in the material as **A** in our lighting equation to reflect incoming ambient light.

**D3DMCS\_COLOR1** – This instructs the device that the color for the material source being set is to be taken from the diffuse color member of the vertex. For this to work you must have defined your vertex structure to have a color component and must set your FVF flags so that they include the **D3DFVF\_DIFFUSE** flag. If we were setting the **D3DRS\_AMBIENTMATERIALSOURCE** renderstate and used **D3DMCS\_COLOR1**, then the lighting calculations would use the color stored in the diffuse member of the vertex as **A** in our lighting equation to reflect incoming ambient light.

**D3DMCS\_COLOR2** – This instructs the device that the color for the material source is to be taken from the specular color member of the vertex. For this to work you must have defined your vertex structure with a color component and set your FVF flags to include the **D3DFVF\_SPECULAR** flag. If we were currently setting the **D3DRS\_AMBIENTMATERIALSOURCE** renderstate and were using **D3DMCS\_COLOR2**, then the lighting calculations would use the color stored in the specular member of the vertex as **A** in our lighting equation to reflect incoming ambient light.

In summary, we can configure **A**, **D**, **S**, and **E** to be taken from the corresponding color in the currently set material or from one of two possible colors that could be stored at each vertex. Let us examine some examples. We start with a vertex structure that will contain a vertex normal and two color values which could be used as material sources in lighting calculations.

```
CVertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Normal;
    DWORD Diffuse;
    DWORD Specular;
};

m_pDevice->SetFVF( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE | D3DFVF_SPECULAR)
```

Notice that the two color components in the vertex structure are described as diffuse and specular members by the FVF flags. This is slightly misleading in the general case. When we are not using DirectX lighting, then these two colors do indeed store the diffuse and specular components of the vertex. The colors are passed straight through to the rasterizer which combines these colors to create the color of the vertex which is then interpolated over the surface. In this case, DirectX Graphics expects that you have calculated the correct diffuse color and specular colors yourself and stored them



in the vertex. Referring to the color members of the vertex as diffuse and specular components is accurate and makes perfect sense.

When using the lighting pipeline however, referring to the two color members as diffuse and specular is not quite as accurate since these members are no longer used specifically to store the diffuse and specular colors. These are colors used as material reflectance properties. With the render states covered above, we can use the two colors in the vertex as either A, D, S or E in the lighting equation. This means of course, that in FVF terms, the diffuse member of the vertex could be used as the emissive reflectance property in place of the material emissive property, or the specular member could be used as the diffuse reflectance property in place of the material diffuse member. While there is not much we can do to avoid this unfortunate naming dilemma at the FVF level, we can at least modify our structure to more accurately reflect what we are trying to do:

```
CVertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Normal;
    DWORD Color1;
    DWORD Color2;
};
```

Of course, you can call the color members whatever you like. We simply chose names that were generic enough to make sense when we wish to store material colors in the vertex. Note that even if you store reflectance properties in the vertex, you will still need to use them alongside materials if you wish to model all four material properties (ambient, diffuse, specular, and emissive) since we can only store two colors in the vertex.

The vertex structure we will use in Lab Project 5.1 will not have colors stored at the vertex. This means that **A**, **D**, **S** and **E** in the lighting equation will be taken from the currently set material Ambient, Diffuse, Specular and Emissive members. We will not explicitly set any of the material source renderstates. This would lead us to believe that the default state of the device is to simply take **A**, **D**, **S** and **E** in the lighting equation from the currently set material and ignore any color components stored in the vertex unless the device is set otherwise. This is not actually the case. In fact, the default state of the device is to take the material reflectance properties from the sources listed in the following table when they exist.

Material State	Color Source	Render	Default Value	Description
D3DRS_DIFFUSEMATERIALSOURCE			D3DMCS_COLOR1	If the vertex has a diffuse component ( <b>D3DFVF_DIFFUSE</b> ), this color is used as the diffuse reflectance property ( <b>D</b> in the lighting equation). If the vertex does not have a diffuse member then the currently set material diffuse member is used instead.



D3DRS_SPECULARMATERIALSOURCE	D3DMCS_COLOR2	If the vertex has a specular component ( <b>D3DFVF_SPECULAR</b> ), this color is used as the specular reflectance property ( <b>S</b> in the lighting equation). If the vertex does not have a specular component then the currently set material specular member is used instead.
D3DRS_AMBIENTMATERIALSOURCE	D3DMCS_MATERIAL	The Ambient member of the currently set material is used as <b>A</b> in the lighting equation.
D3DRS_EMISSIVEMATERIALSOURCE	D3DMCS_MATERIAL	The Emissive member of the currently set material is used as <b>E</b> in the lighting equation.

As you can see, the default device state uses the diffuse and specular color components stored in the vertex instead of the material. If the vertex does not contain the appropriate color members, then the device uses the material members. However if our vertex did have one or two color components then we would need to explicitly set the diffuse and specular material color sources to that of the material if we wanted the material members to be used.

The main difference between the two concepts is that vertex colors are stored as DWORDs and material colors are stored as D3DCOLORVALUES (4 floats). Therefore, when storing colors in the vertex you must combine them into a DWORD. Fortunately, there is a DirectX macro that allows us to pass in four floats and returns a DWORD representation of that color. This macro is shown below and is found in d3d9types.h.

```
#define D3DCOLOR_COLORVALUE(r,g,b,a) \
D3DCOLOR_RGBA((DWORD)((r)*255.f),(DWORD)((g)*255.f),(DWORD)((b)*255.f),(DWORD)((a)*255.f))
```

This system provides a good deal of flexibility to our application since it allows per-vertex control of reflectance properties. Normally we would set the current material and render the triangle(s) that apply the material properties to all of the vertices within the triangle(s). By storing reflectance properties in the vertex also, we can render a triangle where each vertex within that triangle has a unique set of reflectance properties.

**// Example 1:** In this example, if the vertex has a diffuse or specular color component (or both) they are completely ignored by the lighting calculations. The reflectance properties for the diffuse, ambient, specular and emissive light are all taken from the relevant members of the currently set material.

```
m_pDevice->SetRenderState(D3DRS_AMBIENTMATERIALSOURCE, D3DMCS_MATERIAL);  
m_pDevice->SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_MATERIAL);  
m_pDevice->SetRenderState(D3DRS_SPECULARMATERIALSOURCE, D3DMCS_MATERIAL);  
m_pDevice->SetRenderState(D3DRS_EMISSIVEMATERIALSOURCE, D3DMCS_MATERIAL);
```



With these device states:

**A = Material.Ambient**

**D = Material.Diffuse**

**S = Material.Specular**

**E = Material.Emissive**

// **Example 2:** In this example, the ambient and diffuse reflectance properties used in the lighting calculations are stored in the diffuse and specular color components of the vertex. The specular and emissive reflectance properties are taken from the specular and emissive members of the currently set material.

```
m_pDevice->SetRenderState( D3DRS_AMBIENTMATERIALSOURCE , D3DMCS_COLOR1);
m_pDevice->SetRenderState( D3DRS_DIFFUSEMATERIALSOURCE , D3DMCS_COLOR2);
m_pDevice->SetRenderState( D3DRS_SPECULARMATERIALSOURCE , D3DMCS_MATERIAL);
m_pDevice->SetRenderState( D3DRS_EMISIVEMATERIALSOURCE , D3DMCS_MATERIAL);
```

With these device states:

**A = Vertex.Diffuse**

**D = Vertex.Specular**

**S = Material.Specular**

**E = Material.Emissive**

// **Example 3:** This is a combination which you will probably never use, but it does make a good example. In this next example none of the currently set material members are used and the diffuse color of the vertex is used to reflect the incoming ambient and specular light as well as being used as the emissive color. The specular color of the vertex is used to reflect incoming diffuse light.

```
m_pDevice->SetRenderState( D3DRS_AMBIENTMATERIALSOURCE , D3DMCS_COLOR1;
m_pDevice->SetRenderState( D3DRS_DIFFUSEMATERIALSOURCE , D3DMCS_COLOR2);
m_pDevice->SetRenderState( D3DRS_SPECULARMATERIALSOURCE , D3DMCS_COLOR1);
m_pDevice->SetRenderState( D3DRS_EMISIVEMATERIALSOURCE , D3DMCS_COLOR1);
```

With these device states:

**A = Vertex.Diffuse**

**D = Vertex.Specular**

**S = Vertex.Diffuse**

**E = Vertex.Diffuse**

// **Example 4:** The final example shows ambient, diffuse and specular reflectance properties taken from the currently set material. The diffuse color component of the vertex is used as the emissive color.

```
m_pDevice->SetRenderState( D3DRS_AMBIENTMATERIALSOURCE , D3DMCS_MATERIAL);
m_pDevice->SetRenderState( D3DRS_DIFFUSEMATERIALSOURCE , D3DMCS_MATERIAL);
```



```
m_pDevice->SetRenderState( D3DRS_SPECULARMATERIALSOURCE , D3DMCS_MATERIAL);  
m_pDevice->SetRenderState( D3DRS_EMISSIVEMATERIALSOURCE , D3DMCS_COLOR1);
```

With these device states:

**A = Material.Ambient**

**D = Material.Diffuse**

**S = Material.Specular**

**E = Vertex.Diffuse**

Unless you require per-vertex reflectance properties (probably not very often), you will use materials almost exclusively and will not need to store colors in your vertices.

### DirectX Vertex Lighting Advantages

- Speed: On nearly all modern 3D graphics cards, lighting calculations are done in hardware on the GPU.
- Dynamic: Unlike many lighting techniques which are usually done as a pre-process, vertex lighting is calculated each frame as the vertices are passed through the transformation and lighting pipeline. So vertex lighting is a viable choice for implementing dynamic lighting effects. By simply updating a light source's properties each frame, we can animate its position, orientation, and even its color to create dynamic lighting effects.
- Simplicity: Vertex lighting is easy to use. All we need is a way to record the positions of where each light is placed in the scene and a way to apply materials to different polygons. Almost all 3D modeling packages provide this functionality. The GILEST™ level editor certainly allows you to do all these things quickly and easily. Because GILEST™ was written to compliment this course and DirectX specifically (although it is in no way limited to being used for DirectX applications) you will notice that setting materials and lights in GILEST™ is almost identical to setting them in DirectX. So you can more easily understand all of our discussions simply by firing up GILEST™ and experimenting. Using GILEST™ to create your game world and to set your lights and materials makes using DirectX lighting very easy. In Lab Project 5.2 we will load in a simple level made in GILEST™ and look at all of these features.
- Clarity: Vertex lighting is ideal for beginners to 3D graphics. It helps them more easily see how light sources affect geometry and add realism to a scene.

### DirectX Vertex Lighting Disadvantages

- Resolution: One of the major disadvantages of vertex lighting is its fixed relationship to the geometric level of detail. Changes in illumination are only reflected from vertex to vertex rather than at the pixel level, so in order to have nice looking vertex lighting, we need meshes that use a lot of vertices. If a mesh contains only a few vertices (such as a cube) much detail

and nuance is lost. Fig 5.23 depicts a cylinder and a sphere made from numerous triangles (and therefore many vertices). The ambient color in the scene is blue and each meshes have a spotlight shining on them.

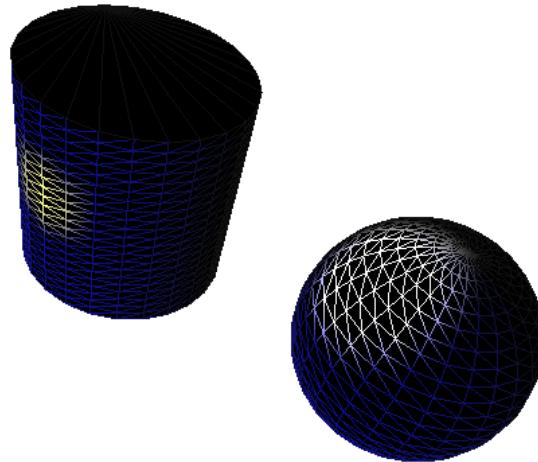


Figure 5.23

Keep in mind that although we view these meshes as solid triangle based objects, the lighting pipeline works only with the vertices of the mesh. Therefore, what is really happening to the vertices is shown in Fig 5.24:

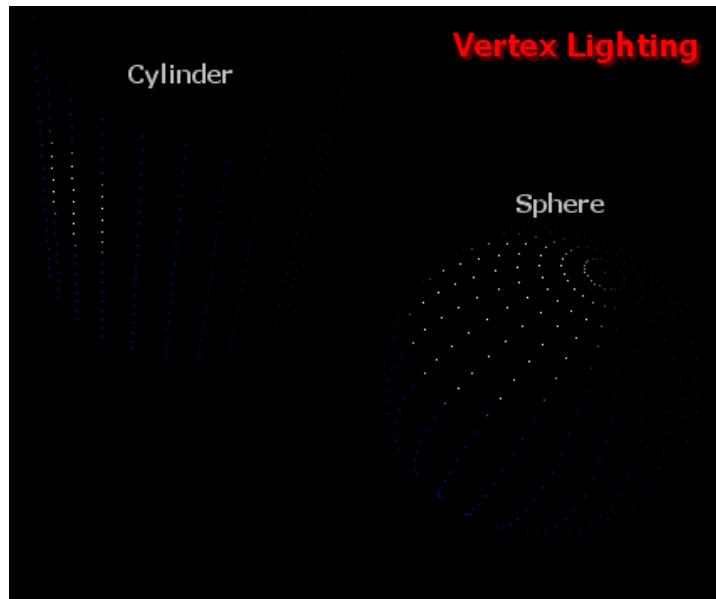
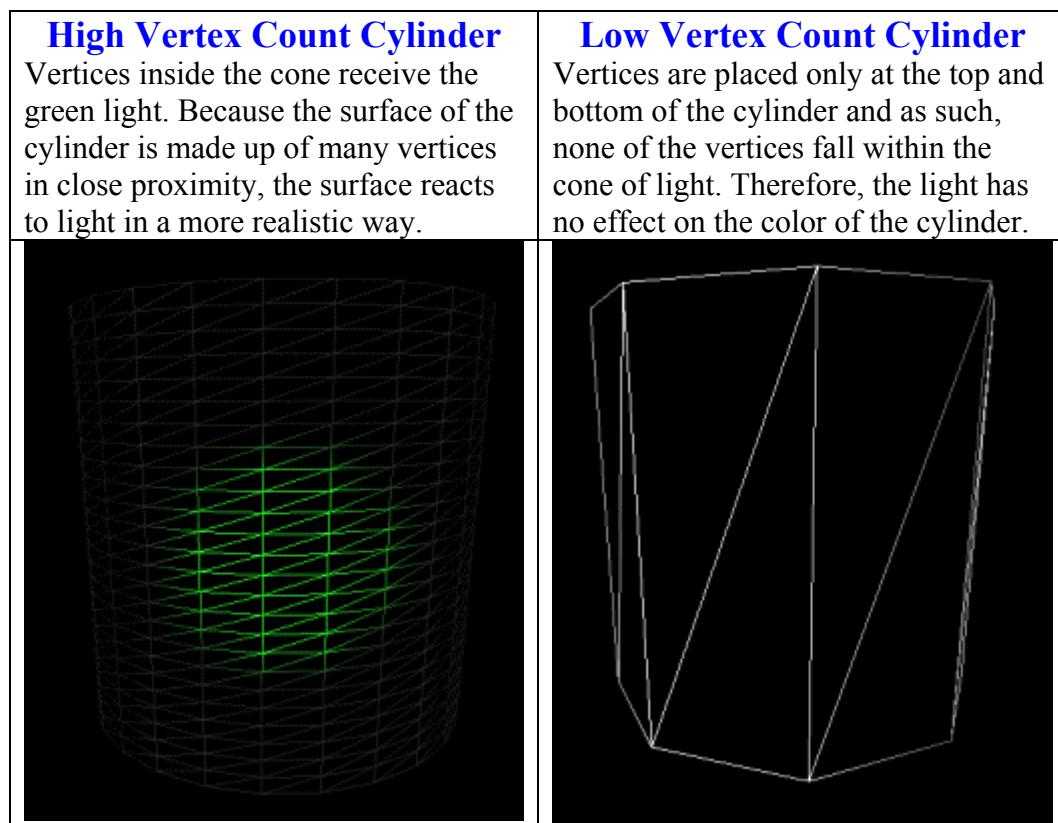


Figure 5.24

If a light is placed such that it does not influence any vertices, the light will have no effect. Now this may seem like an obvious point to make, but think about a situation where we have a cube face. Imagine that we created a bright white spot light with a narrow cone such that it

shone on the center of the quad but that the corner points were outside the cone. Although the quad should have a circle of white light at its center, the light would have no effect. The only way we could get this effect would be to subdivide the quad into smaller quads, perhaps using a 10x10 vertex grid for example. This gives us a lit quad, but at the expense of having many more vertices than we need. Each of these additional vertices will need to be transformed and lit by the pipeline thus slowing the performance of our game engine.

Fig 5.25 shows cylinders with low and high polygon counts and a green spot light shining on their surfaces.

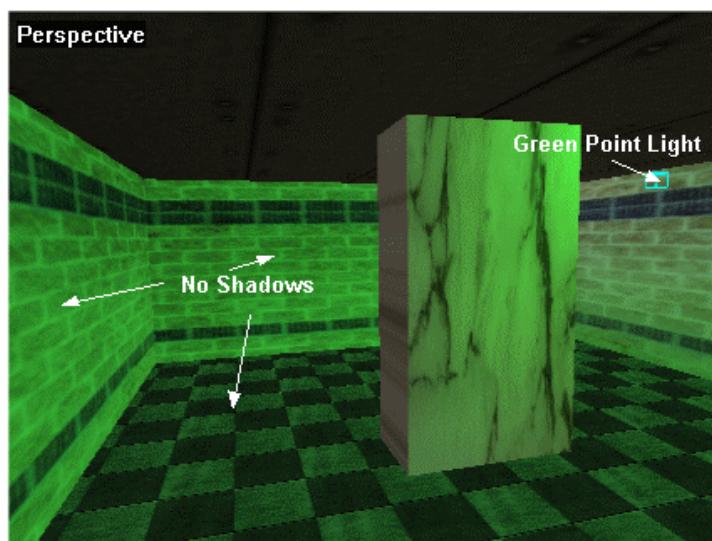


**Figure 5.25**

This is a very common and constant problem when using vertex lighting. The answer could be to make sure that your entire world is highly tessellated, but usually this is in direct opposition to what you are trying to accomplish. Level designers are generally encouraged to restrict polygon budget so that the game engine does not have to transform, light, and render too many triangles each frame. It is often for this reason that vertex lighting is not usually solely relied upon for the illumination of game worlds in modern commercial games.

- No Occlusion: Another disadvantage of vertex lighting is a lack of accounting for occlusion. It illuminates vertices without considering the positions of other triangles in the world. If polygon B is positioned between polygon A and the light source, polygon B will not block the light and cast Polygon A into shadow as would be expected in real life. The lighting pipeline does not cast shadows. The result is a major loss of scene realism and mood.

In Fig 5.26 there is a green point light behind the pillar. In real life, the pillar would block the light and cast certain parts of the wall into shadow. As you can see, the light passes right through the pillar.



**Figure 5.26**

If you have very tessellated geometry (high polygon counts) you can simulate area shadowing by placing dark lights (negative color values) in appropriate places. The problem with the technique of course (even beyond the extra polygons) is that it would be difficult or nearly impossible to recreate the shape of a particular object's shadow. However, it may be good enough for your game to use only pools of shadow like this.

- The final disadvantage is the limit on the number of simultaneously active light sources. The problem is not that vertices will not receive adequate coloring (in practice, only two or three light sources are needed for that), it is that we will have to create a light management system that knows how to activate only the lights that influence the polygon(s) being rendered and disable the rest.



## Conclusion

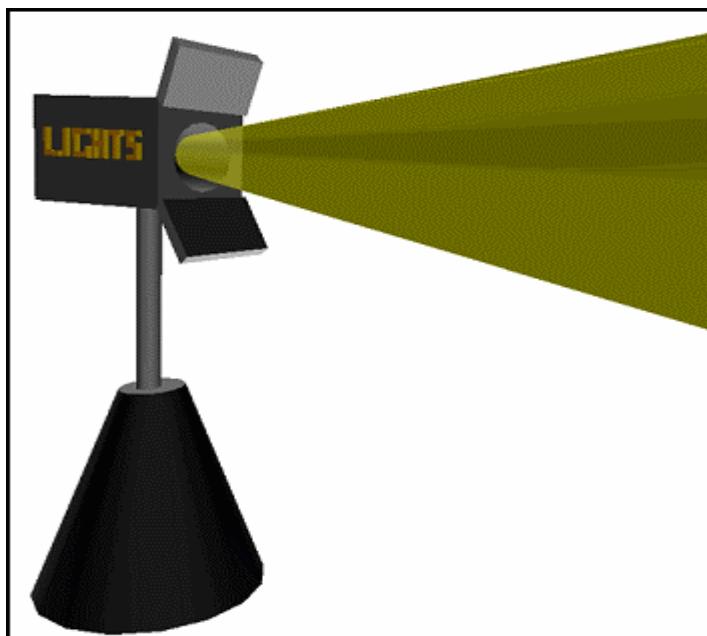
This lesson taught us the basics of DirectX lighting and materials. We understand the different light types available to our application as well as some of the mathematics of the lighting pipeline. In addition to reviewing many of these concepts, our lecture will focus in on the mathematics in greater detail, so be sure to take the time to go through it and take notes. Also, make sure that you spend time exploring the projects in your workbook since we have only covered the high level theory here in the text. The lab projects for this lesson will teach you more practical concepts such as how to:

- emulate the lighting equations yourself to record light influences at each vertex
- work around the simultaneous light limit imposed by hardware vertex processing devices
- use batching strategies to keep render state changes and DrawPrimitive calls minimized
- use the IWF SDK helper classes to load geometry from GILEST™

# **Workbook Chapter Five:**

## **Lighting**

---



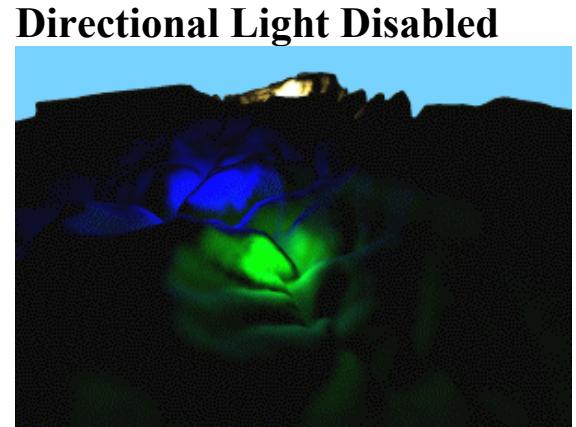
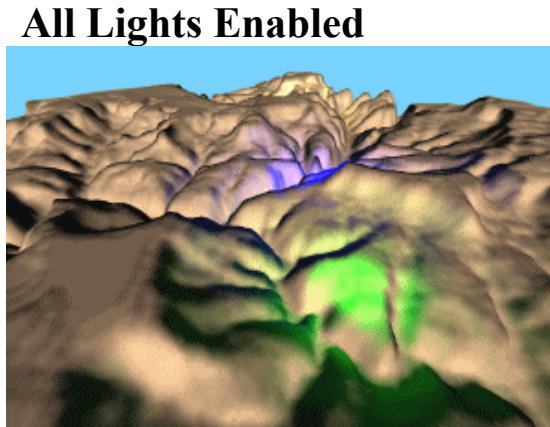
© 2003, eInstitute, Inc.

**You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.**

## 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.



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 );

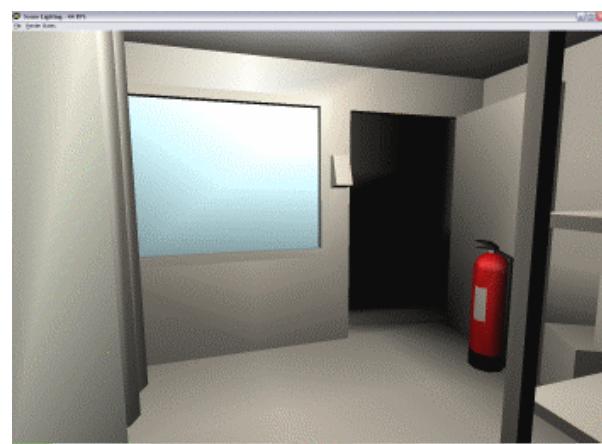
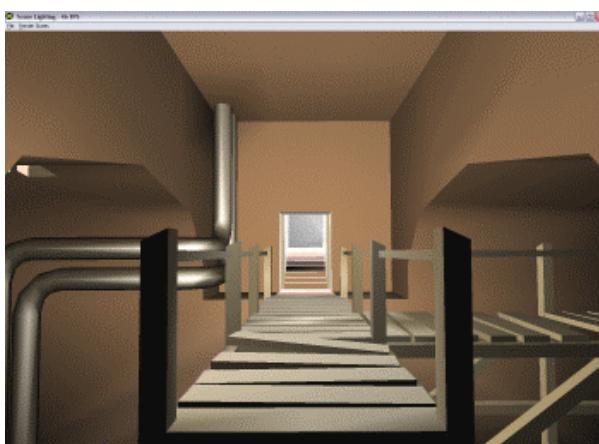
break;
```

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 GILEST™ 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 GILEST™ 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
```

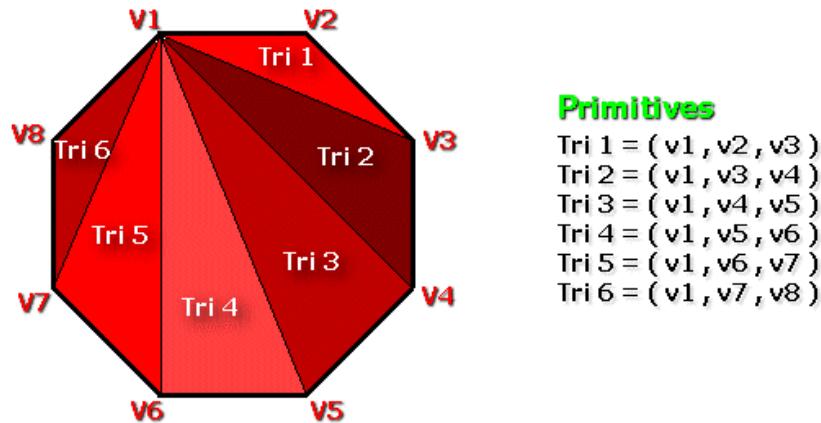
```
..
```

```
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
<b>VertexBuff = Av1, Av2, Av3, Av4, Av5, Av6, Av7, Av8, Bv1, Bv2, Bv3, Bv4, Bv5, Bv6, Bv7, Bv8</b>																

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?

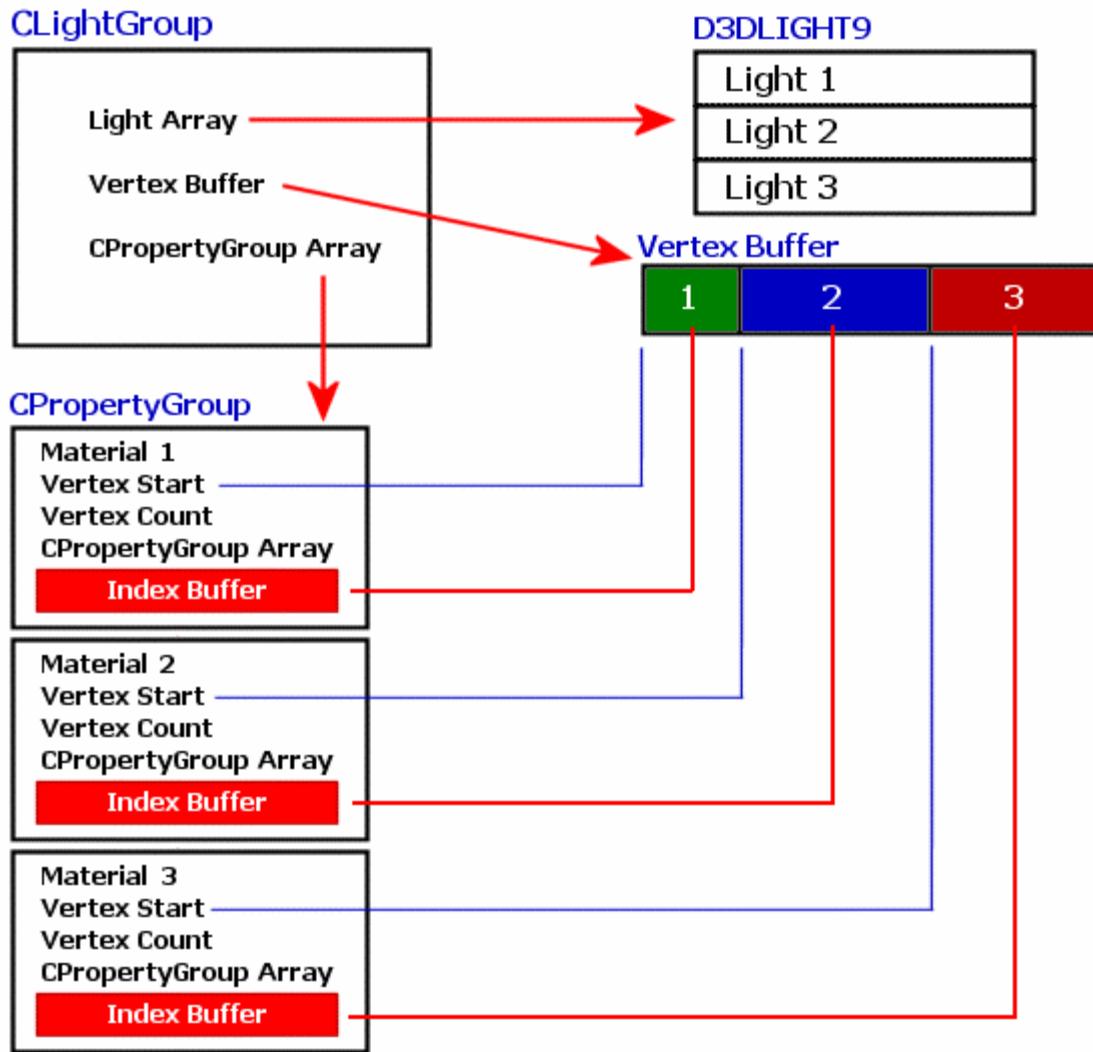
Our new rendering strategy will look like this:

- 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 may 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;
}

// Success
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\_nVerexStart, 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;

    // Success!
    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
};


```

### Public Member Variables

#### **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.

#### Private Member Variables

**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\_nReservedLights**.

**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 4<sup>th</sup> row of the matrix and the direction vector from the 3<sup>rd</sup> 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 3<sup>rd</sup> 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;
```

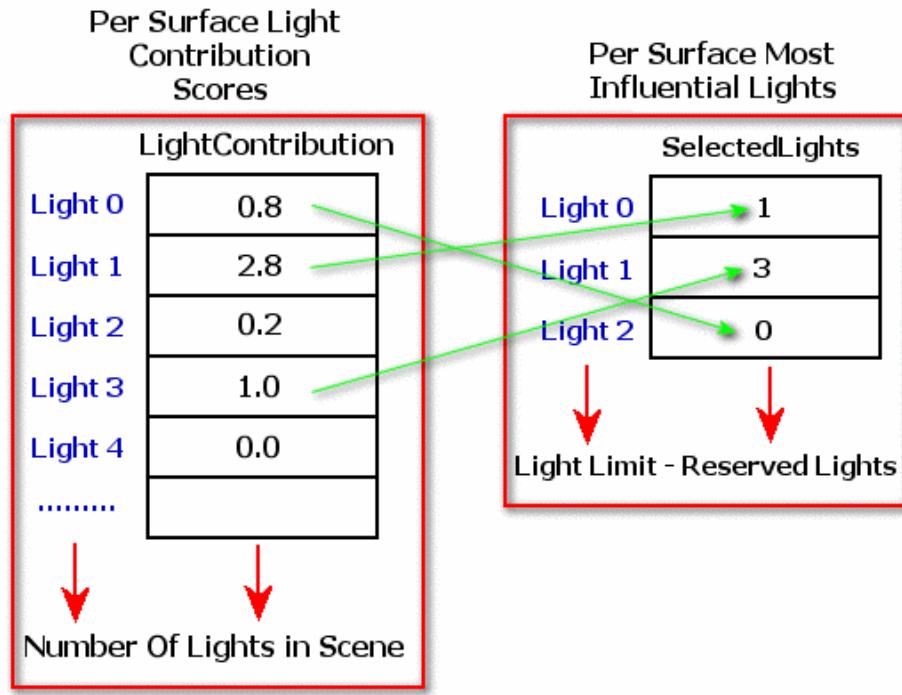
m\_nLightLimit holds the maximum number of simultaneous lights allowed by the device. We must subtract the number of lights the application would like to reserve for its own uses to obtain how many lights can exist in a single light group.

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 it's 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.

$$\text{AttFactor} = \frac{1}{\text{Attenuation1} + (\text{Attenuation2} \times D) + (\text{Attenuation3} \times D^2)}$$

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.

$\alpha$  = Angle between Vertex Normal and VertexToLight direction vector

$\phi$  = Phi / 2 (Half the outer cone angle)

$\theta$  = Theta / 2 (Half the inner cone angle)

*Falloff* = Falloff property of the **D3DLIGHT9** structure

$$\text{SpotFactor} = \left( \frac{\cos(\alpha) - \cos(\phi)}{\cos(\theta) - \cos(\phi)} \right)^{\text{Falloff}}$$

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 by 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 calculate 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;
else
```

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.

```
{
    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 left 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.

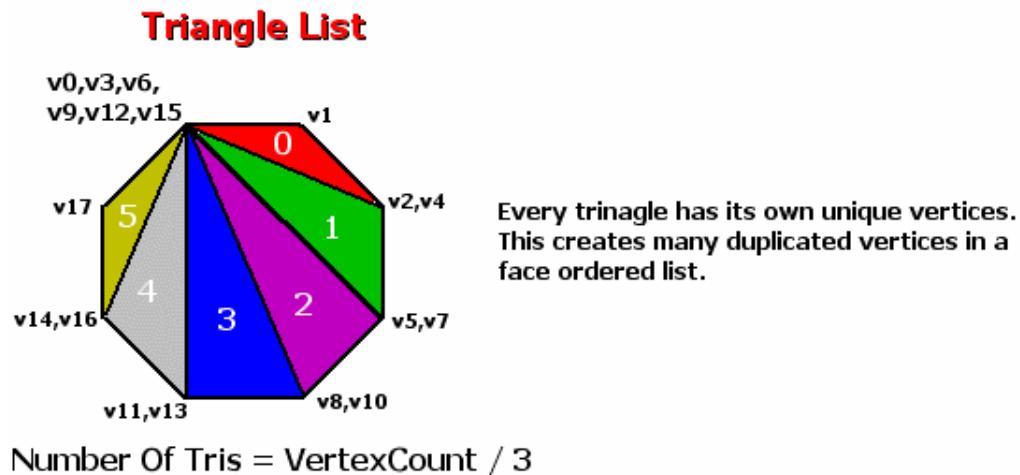
There is one problem to consider: in some cases, the surface passed into this function may not have indices pre-generated for it or even if it does, they may be stored as strips or fans. Bearing this in mind, this function has to allow for these cases. If the surface passed into the function does not 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.

### Generating an Indexed Triangle List from a Triangle List

If a surface is stored as a triangle list, it means that there 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;
```

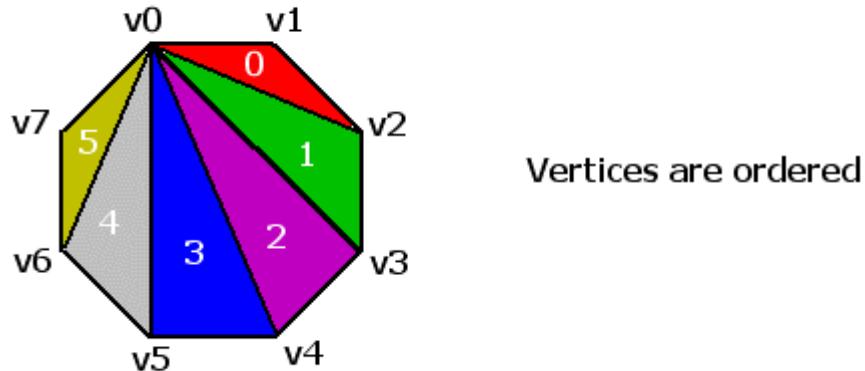
When the passed surface is a triangle list we call the AddIndex function to resize the property group index array using the vertex count of the surface. Next, we loop through each vertex in the face and the element number of the vertex in the array becomes the index in the index list.

**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.

### Generating and Indexed Triangle List from a Triangle Fan

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 to generate one. Because we use the first vertex ( $v_0$ ) for every face, the code for generating the indexed triangle list is straightforward.

## Triangle Fan



$$\text{Number Of Tris} = \text{VertexCount} - 2$$

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 ) ;

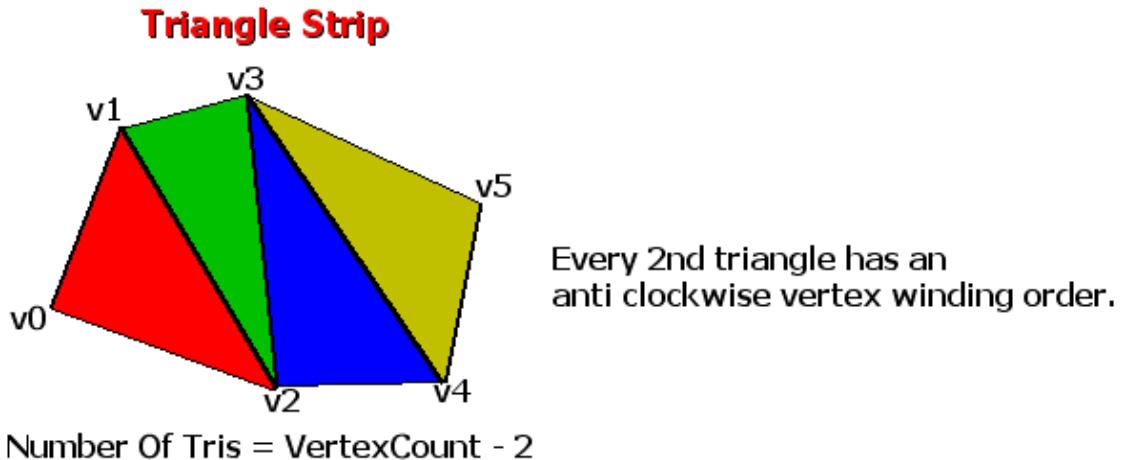
    for ( Counter = IndexCount, i = 1; i < pFace->VertexCount - 1; i++ )
    {
        pPropertyGroup->m_pIndex[ Counter++ ] = VertexStart;
        pPropertyGroup->m_pIndex[ Counter++ ] = i + VertexStart;
        pPropertyGroup->m_pIndex[ Counter++ ] = i + 1 + VertexStart;
    } // Next triangle
break;
```

---

**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.

## Generating an Indexed Triangle List from a Triangle Strip

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 ( $v_1, v_2, v_3$ ) and the fourth triangle ( $v_3, v_4, v_5$ ) 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;
```

## Generating a Indexed Triangle List from an Indexed Triangle List

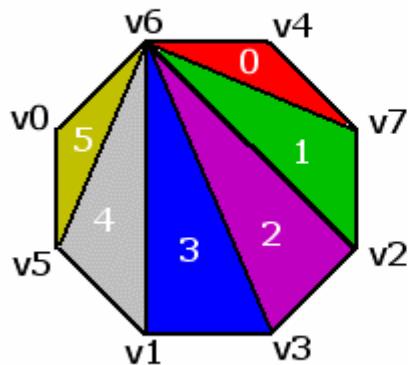
If the surface we are assigning to this property group already has an array of indices in indexed triangle list format, we can simply copy the indices from the surface straight into the index array as shown below. We must remember to add on the VertexStart value to each index to be sure that the index 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;
```

## Generating an Indexed Triangle List from an Indexed Triangle Fan

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.

### Indexed Triangle Fan



Vertices are in no particular order  
but indices describe the triangles.

IndexList = { 6,4,7,2,3,1,5,0 }

Number Of Tris = 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:

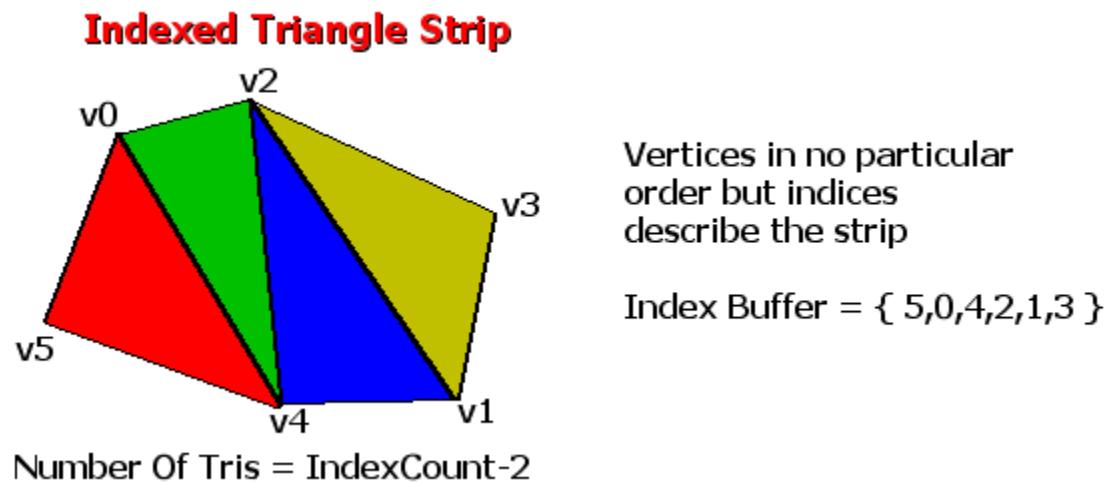
```

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.

### Generating an Indexed Triangle List from an Indexed Triangle Strip



With an indexed triangle strip, the vertices are 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  $\text{IndexCount} - 2$ . 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 appear 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.

# Chapter Six:

## Texture Mapping

---



© 2003, eInstitute Inc.

You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.



## Table of Contents

Introduction.....	3
Texture Memory .....	6
Texture Formats .....	8
Textures and Memory Pools .....	16
MIP Maps.....	19
Loading Textures .....	23
Creating Textures Manually .....	31
Setting a Texture .....	32
Texture Coordinates.....	33
Sampler States.....	40
Texture Addressing Modes.....	41
Texture Coordinate Wrapping .....	45
Texture Filtering .....	48
Texture Stages.....	57
Multi-Texturing.....	68
Color Blending.....	71
Compressed Textures.....	78
Texture Coordinate Transformation .....	89
The IDirect3DTexture9 Interface .....	94
The IDirect3DSurface9 Interface.....	97
D3DX Texture Functions.....	106
Conclusion .....	108



## Introduction

In this lesson we will learn:

- what a texture is
- how it is stored in memory
- how to create textures
- how to load texture images from files
- how to store texture coordinates in our vertex structure
- how blend multiple textures together
- how to use compressed textures

In nature, a surface such as a wooden board or a metal sheet can have a near infinite amount of detail and randomness caused by surface imperfections, wear and tear over time, etc. Moreover, it is generally true that two surfaces are never exactly the same. Consider two planks of wood purchased from the same timber merchant. Even if the wood is of the same type, each plank of wood will have its own unique grain pattern and color.

We cannot possibly hope to model this behavior in our applications in real-time, but we can give the surfaces of our objects much more detail than we have to date. One approach is to use many more vertices in our models. If we wanted to render a rectangle that looked like a piece of wood, instead of building it from four vertices, we might make it from a grid of quads (like our terrain grid in Chapter 3) and give each little quad its own color. This way the parent rectangle would look like it was made up from many hundreds or thousands of shades of brown and more closely resemble a piece of wood. The problem with this approach is that at the scene level it would potentially require million of vertices that will need to be processed (transformed and lit).

This is why the concept known as texturing is so important. It is a method of ‘painting’ image detail (usually a 2D image created in a paint program or with the use of a digital camera) onto a polygon as it is rendered. Instead of creating thousands of tiny quads to give the wood surface many different colors, we can instead use an image of a piece of wood to achieve the same end. If we had some means to keep the rectangle as a simple 4 vertex construct but paint the 2D image onto the surface of the polygon as it was rendered, we would experience the best of both worlds -- we would have a very low polygon object with a highly detailed surface. Fig 6.1 shows a quad with a wood image applied.



Figure 6.1

The quad in Fig 6.1 does not exhibit the true imperfections of a real piece of wood and the surface is still perfectly flat. Nevertheless to the naked eye, the surface does not appear completely flat and does indeed look like a piece of wood planking.

The process of mapping images to polygons is referred to as **texturing** (or texture mapping). The image which is being mapped onto these polygons is referred to as a **texture** (or a texture map).

**Note:** The word texture usually describes the way something feels to touch. In 3D graphics programming, a texture refers to an image that is used by the rendering pipeline to provide per-pixel color information for a polygon during the rendering process. The two are not totally unrelated. The idea is to give objects the appearance of a surface that has texture.

Recall the way that color is stored at the vertices and then interpolated across the triangle as it is rendered. Imagine if we also stored 2D coordinates (referred to as **texture coordinates**) at each vertex that described a pixel in the texture such that the color of that pixel in the texture is assigned as the color of the vertex. Now imagine that the 2D coordinates themselves are interpolated across the surface for each pixel. Each pixel of the polygon now has its own set of 2D coordinates which describe a pixel location in the texture.

**Note:** To avoid confusion when referring to both pixels on the screen (or in the frame buffer) and pixels within a texture image, a pixel within a texture is referred to as a **texel**.

The images in Fig 6.2 show our scene from Lab Project 5.3 both with and without textures.



Figure 6.2

For most of our scenes we will use many texture images. Creating these textures is usually the responsibility of the project texture artist or someone else on the artistic team. You can see in the Fig 6.2 that this small section of a level has a different texture for the door, walls, floor, ceiling, window frame and even a transparent texture applied to the glass of the window.

Fig 6.3 shows some 2D images that might be used as textures in a game level. These were created using the popular paint programs Adobe Photoshop™ and Jasc's Paint Shop Pro™.



**Figure 6.3**

Looking at the visuals for some of the latest games, it should be clear that a modern 3D application has to load and maintain many texture maps. Generating quality textures can be quite difficult and time consuming work for the artists. Fortunately for us, as game programmers our job is actually much less difficult than theirs in this respect. We simply need to load the textures they create into memory and be sure to apply them to the correct polygons in a given scene. We will even be receiving assistance from the DirectX API to make that job easier. DirectX Graphics will allow us to bind textures to polygons with ease and even automates loading the textures from file and managing them in memory.

Although we will cover the complete the texturing process in this chapter, the following list describes a brief summary of the steps involved:

- Use a level editor or modeling program to assign textures to faces.
- Have the level editor or modeling program generate texture coordinates at the vertices.
- Load the vertex data from the file containing texture coordinate set(s) per vertex.
- Load the texture images from files.
- Set the texture(s) as the device texture(s).



- Render all the triangles that use the current texture(s).
- Repeat last two steps for each texture in the scene.

Since the first two steps are typically not our responsibility, we can concentrate on allocating and loading textures properly, and setting the device to get the best texture image quality when rendering. We will examine later in the lesson how to calculate texture coordinates for a vertex. Although most of the time the texture coordinates will be generated for us in the level editing package, there will be times when we may want to calculate texture coordinates ourselves in code.

## Texture Memory

Because of the way our textures will be loaded, it is important that we discuss how textures are stored in memory first. When we load texture images from a file, we can specify several parameters that describe to the D3DX texture loading functions how we would like them stored in memory.

Before the release of DirectX 8.0, there was a DirectX API called DirectDraw. It was used to perform 2D operations and render 2D graphics to surfaces. We would store 2D image data on DirectDraw surfaces which if possible, would be stored in video memory for optimal performance. These surfaces would typically be used to hold bitmap data and the DirectDraw surface could be blitted to the screen or to another DirectDraw surface very quickly with the aid of hardware acceleration.

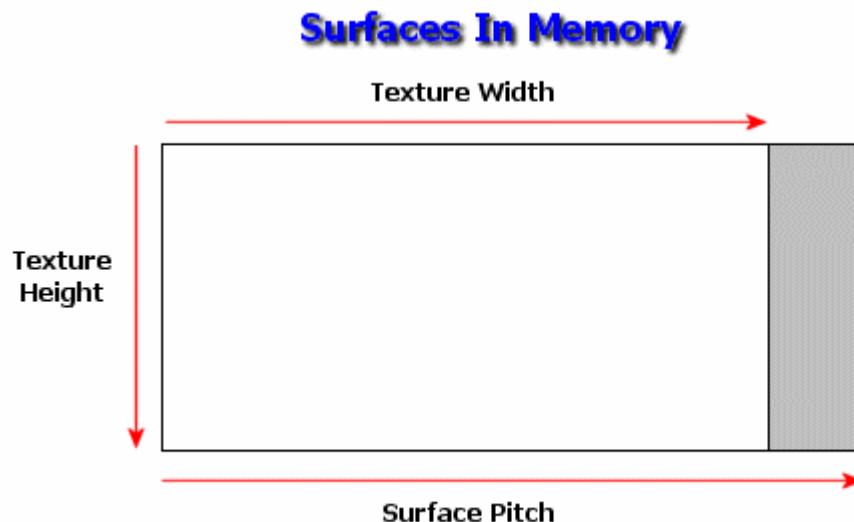
While the notion of a surface being used as the main means for transporting and storing image data has been diminished in favor of the IDirect3DTexture9 interface, at the driver level DirectDraw remains a core asset. In fact, DirectDraw surfaces are still used to hold our texture image data. The Direct3DTexture9 object is used to manage and work with textures and completely encapsulates the underlying surface object containing the texture image data. There is a Direct3DSurface9 object as well which allows us to store and work with surfaces directly. These surface objects are typically used for performing 2D operations like blitting title screens for example, rather than being used directly for texture work. Because the texture image data is stored in a surface within the texture object, we need to look at how surfaces are stored in memory. This will enable us to work with both texture and surface objects alike.

**Note:** It may seem a strange fact that under the bonnet DirectX also stores vertex buffer data as DirectDraw surfaces. This is not so strange however when you consider that they are simply blocks of linear memory as we shall soon see.

Regardless of where a surface is stored, we can think of it as a rectangle with a width and height and a memory area for the image data. In reality, this memory is just a linear block of bytes. You might think that you could calculate how many bytes a surface is using by multiplying the width by the height and then multiplying the result by the number of bytes used by each pixel, but this is not the case. When a surface is created, the driver may choose to insert extra bytes in the surface at the end of each row of image data such that the data is aligned to memory address boundaries. This is done so that the driver can work with or copy the image data in an optimal way. Since each row of the surface may have one



or more extra bytes of data allocated, the length of a row in the surface is referred to as the **pitch**. So the number of bytes used by a surface can accurately be calculated as Pitch x Height x Bytes Per Pixel (BPP) as shown in Fig 6.4.



**Figure 6.4**

Although these extra bytes on the end of each row are not rendered when the texture is drawn and are not considered to be part of the image data, we still have to be aware of this fact when we work with image data. Just like vertex buffers, the IDirect3DTexture9 and the IDirect3DSurface9 interfaces both include Lock methods to lock the resource and return a pointer to the image data for reading and/or writing pixels. When we have a pointer to image data, we normally think to advance to the next line in the image by:

**pImageData += ImageWidth**

But with a DirectX texture/surface, because of the padding bytes on the end, this approach might leave the pointer stranded in the wrong location. Therefore, when we have a pointer to the image data of a DirectX surface, we must advance to the next line of image data by:

**pImageData += ImagePitch**

**Note:** We have been using the terms surface and texture interchangeably. Technically, the surface is the actual memory holding the image data and the Direct3DTexture9 object is an object that encapsulates the surface and allows us to work with it. The IDirect3DSurface9 object is another object that encapsulates image data, but we will discuss surfaces in the context of the data encapsulated by the Direct3DTexture9 object for now.



## Texture Formats

When we initialized our rendering device in Chapter 2, we had to choose a format for the frame buffer. We did this using the **D3DFORMAT** enumerated type. While this type has many members, during frame buffer creation we were limited to only a small subset of formats. When creating textures, we have a much wider range of formats that we can choose from. We use the **D3DFORMAT** type to create a texture or load a texture such that its image data is stored in memory using the specified color component arrangement for each pixel.

Not all of the possible texture formats are shown in Table 6.1, since many are obscure and not widely supported. However, we do list the common formats that you are likely to use along with a description of how the color information is stored in a single pixel within the surface. Typically you will be using one of the 32-bit or 16-bit formats when creating your textures. Popular texture formats supported by most hardware are highlighted in grey.

**Table 6.1 Texture Formats**

<b>D3DFORMAT Member</b>	<b>Description</b>
<b>32-bit Surface Formats</b>	
<b>D3DFMT_A8R8G8B8</b>	Each pixel in the surface will be a 32-bit value. 8 bits are used for alpha (transparency information), and 8 bits each for red, green and blue.
<b>D3DFMT_X8R8G8B8</b>	8 bits for red, 8 bits for green and 8 bits for blue (8 bits are unused -- X8). This gives 24 bit color resolution on a 32-bit surface. This is useful because many graphics adapters do not support 24 bit textures so this allows us to use a 32-bit surfaces to store 24 bit color.
<b>D3DFMT_A2B10G10R10</b>	2 bits for alpha, 10 bits for each red, green and blue.
<b>D3DFMT_A8B8G8R8</b>	8 bits for alpha followed by 8 bits for each blue, green and red. The components are arranged in memory in ABGR format as opposed to 32-bit ARGB.
<b>D3DFMT_X8B8G8R8</b>	8 bits for red, green and blue stored in XBGR format. There are 8 bits of each pixel not used (X8)
<b>D3DFMT_G16R16</b>	A rarely used format where each pixel contains 16-bits for green and 16-bits for red. You will rarely ever use this for textures because of the lack of its blue color component limiting the colors that can be stored.
<b>D3DFMT_A2R10G10B10</b>	2 bits for alpha and 10 bits for each red, green and blue component. Rarely used.
<b>16-bit Surface Formats</b>	
<b>D3DFMT_R5G6B5</b>	This 16-bit RGB format has 5 bits reserved for red, 6 bits for green and 5 bits for blue.



<b>D3DFMT_X1R5G5B5</b>	5 bits for each red, green and blue component with one bit unused.
<b>D3DFMT_A1R5G5B5</b>	This ARGB surface stores 1 bit for alpha and 5 bits for each red, green and blue component.
<b>D3DFMT_A4R4G4B4</b>	4 bits for each alpha, 4 bits for red, 4 bits for green and 4 bits for blue.
<b>D3DFMT_A8R3G3B2</b>	Rarely used ARGB format with 8 bits for alpha, 3 bits each for both red and green color components and 2 bits for blue.
<b>D3DFMT_X4R4G4B4</b>	4 bits unused and 4 bits each for red, green and blue color components.

In addition to the more common surface formats described above, DirectX Graphics also supports compressed texture formats. Compressed textures allow the application to use much larger and more detailed textures that otherwise might not fit in video memory. These large textures retain an amazing amount of detail even when the player is standing very close to them. They can also be used to optimize smaller textures by saving video memory. The formats are **D3DFMT\_DXT1** through **D3DFMT\_DXT5**. Compressed textures are now quite widely supported and all future 3D cards released will certainly support them. We will examine compressed textures in detail later on in the lesson.

## Validating Texture Formats

It is likely that not all of the 16 and 32-bits formats listed in Table 6.1 will be supported on all devices. We have highlighted the most common formats and the ones you will most likely use in your own applications, but you should always check the support for any format before using it. It is possible that a format may be supported by the current hardware but perhaps not with the current frame buffer/depth buffer format.

Once we have created a device, we know the frame buffer and depth buffer formats we are using. With this information, we can check a given format against the device using `IDirect3D9::CheckDeviceFormat` to see if this is a valid texture format for the current device.

We used this function in the enumeration code in Chapter 2 to check if the depth buffer and frame buffer could work together on the device we were trying to create.

```
HRESULT CheckDeviceFormat
(
    UINT Adapter,
    D3DDEVTYPE DeviceType,
    D3DFORMAT AdapterFormat,
    DWORD Usage,
    D3DRESOURCETYPE RType,
    D3DFORMAT CheckFormat
);
```



In the following example we are trying to find a supported 32-bit format that we can tell DirectX to use when creating our textures. As each one fails, we check the next in line until we find one which is supported. Otherwise we move on to testing 16-bit formats. The following code assumes we are using the primary display adapter and a HAL device. It also assumes that our adapter is in 32-bit X8R8G8B8 format and that we are testing to see if we can find support for a 32-bit texture format.

```
D3DFORMAT SupportedFormat = D3DFMT_UNKNOWN

if(SUCCEEDED(pD3D->CheckDeviceFormat(D3DADAPTER_DEFAULT,           //Adapter
                                       D3DDEVTYPE_HAL,                //Device Type
                                       D3DFMT_X8R8G8B8,              //Adapter Format
                                       0,                            //Usage flags
                                       D3DRTYPE_TEXTURE,             //Resource Type
                                       D3DFMT_A8R8G8B8               //Requested Format
                                       )))
SupportedFormat = D3DFMT_A8R8G8B8;

else
if(SUCCEEDED(pD3D->CheckDeviceFormat(D3DADAPTER_DEFAULT,
                                       D3DDEVTYPE_HAL,
                                       D3DFMT_X8R8G8B8,
                                       0,
                                       D3DRTYPE_TEXTURE,
                                       D3DFMT_X8R8G8B8 // Requested Format?
                                       )))
SupportedFormat = D3DFMT_X8R8G8B8;

else
if(SUCCEEDED(pD3D->CheckDeviceFormat(D3DADAPTER_DEFAULT,
                                       D3DDEVTYPE_HAL,
                                       D3DFMT_X8R8G8B8,
                                       0,
                                       D3DRTYPE_TEXTURE,
                                       D3DFMT_X8B8G8R8 //Requested format?
                                       )))
SupportedFormat = D3DFMT_X8R8G8B8;
```

We perform this process for all formats we are interested in. Fortunately, D3DX includes a texture loading function that simply loads the file with the file name you supply and automatically creates a compatible texture format that most closely matches the bit depth and pixel format of the image in the file. The CheckDeviceFormat function is useful if we are looking for a very specific set of formats. For example, if we want our textures to use an alpha channel, then we could use the code above to check all formats that have alpha components until a suitable one is found.



**Note:** Unlike choosing a back buffer and front buffer format in fullscreen device mode, we are not limited to choosing a texture format that matches either the front buffer or the frame buffer. It is acceptable to run the graphics card in 16-bit mode while still creating and using 32-bit texture formats. The textures, unlike the frame buffer that needs to be flipped to an identical front buffer format in full screen mode, are containers of per-pixel information accessed on a per-texel basis by the renderer. The performance overhead of using a 32-bit texture while the adapter (and frame buffer) is in 16-bit mode (or vice versa) is typically quite small. The color is converted during rendering fairly quickly on the hardware. However, for maximum performance you should try to use a texture format that matches your frame buffer format whenever it is possible or convenient.

D3DX also includes a function called D3DXLoadSurfaceFromSurface which allows you to copy image data in one surface to a destination surface of the same or different format. For example, if we have a source surface of **D3DFMT\_X8R8G8B8** and a destination surface in **D3DFMT\_DXT1** (compressed format), the function will take the image data in the source surface and copy it into the compressed format required by the destination surface. This gives us a mechanism to convert between arbitrary surface formats with ease. We will take a look at the D3DX texture functions shortly when we discuss the various ways of creating and filling textures.

**Note:** Often we will not need to manipulate texture data at the pixel level. Letting the D3DX loader functions choose a texture format for us is generally fine. If we do need to lock the texture and work with image data directly, we will need to be aware of the format so that we know how the pixels are arranged in memory.

## Understanding Surface Formats

We will briefly cover the **D3DFORMAT** type and the way texels are arranged in memory since there will be times where you will want to manipulate the surface images directly. Let us use as our first example, the process of reading and writing to the individual pixels within a texture that has been created with a 32-bit ARGB format (**D3DFMT\_A8R8G8B8**). Although each of the formats in Table 6.1 store colors in a different configuration for each texel, the A8R8G8B8 case should shed some light on accessing texels in texture surfaces stored in other formats.

A **D3DFMT\_A8R8G8B8** surface means that every 32 bits (4 bytes) of surface data represents a single texel color. The name of this format makes sense when we consider that we have one byte (8 bits) for each color component. Although we have not discussed what the alpha component of a color is at this point in the text (see Chapter 7), just know that a pixel can include an alpha component that describes how transparent it is. This value determines how the pixel will be blended with another color already in the frame buffer at the same pixel location. If the alpha component is at full intensity then the pixel is considered opaque and should overwrite any pixel in the frame buffer at that location, assuming it passes the appropriate tests (depth, etc.).

As you may recall, this idea of packing four color components into a 32-bit ARGB DWORD can be found in cases such as the Diffuse and Specular vertex color components. In a moment we will look at how to extract the individual color values from this type of variable.



Although we do not yet know how to create a Direct3DTexture9 object, assume for now that we have created one that stores a 32-bit image surface that we wish to access. As is the case with vertex buffers, the IDirect3DTexture9 interface does not expose the surface data directly, so we have to lock the surface first. We do this using the IDirect3DTexture9::LockRect method shown below.

```
IDirect3DTexture9::LockRect  
(  
    UINT Level,  
    D3DLOCKED_RECT *pLockedRect,  
    CONST RECT *pRect,  
    DWORD Flags  
) ;
```

Ignore the first parameter for now -- we will discuss it in the next section. The second parameter accepts the address of a **D3DLOCKED\_RECT** structure. If the lock call is successful it will contain a pointer to the image data that our application can use to read from or write to the surface. The **D3DLOCKED\_RECT** will also contain an integer value describing the ***pitch*** of the surface in bytes.

```
typedef struct _D3DLOCKED_RECT  
{  
    INT Pitch;  
    void *pBits;  
} D3DLOCKED_RECT;
```

The third parameter to the LockRect function allows our application to pass in a rectangle specifying the region of the texture surface that we would like to lock. If you set this to NULL then the pointer returned in the **D3DLOCKED\_RECT** structure will contain a pointer to the first byte of the first pixel in the surface. This can be understood as the pixel in the top left corner of the image. If you specify a rectangle on the surface, then the pointer returned will point to the first byte of the pixel in the top left corner of the rectangle on the surface.

The fourth parameter allows us to specify a series of flags that can be used to supply DirectX Graphics with hints regarding the most efficient locking strategy. Unlike vertex buffers which can always be locked (even static ones in video memory at a severe performance cost), not all textures are lockable. Typically textures that exist in video memory are not lockable and ideally most of our required textures will exist there. However if a texture is created with the **D3DUSAGE\_DYNAMIC** flag, then it can be locked even if it does exist in video memory.

The Flags parameter of the lock function can be set to one or more of the following:

- **D3DLOCK\_DISCARD**
- **D3DLOCK\_NO\_DIRTY\_UPDATE**
- **D3DLOCK\_NO\_SYSSLOCK**
- **D3DLOCK\_READONLY**



These flags will not be discussed just yet, although you might recognize them from Chapter 3 when we locked our vertex buffers. Their functioning is very similar for textures since, at the driver level, both vertex buffers and texture resources are stored on DirectDraw surfaces.

The following example will lock a 32-bit surface and extract the individual color components of the texel at coordinates (x =10, y = 4). We use the surface pitch to calculate the correct row width.

```
IDirect3DTexture9 * pTexture;  
...  
...  
D3DLOCKED_RECT LockInfo;  
  
// Lock the texture to retrieve its lock info.  
pTexture->LockRect( 0 , &LockInfo , NULL , 0 );
```

We want to cast the void pointer returned in the **D3DLOCKED\_RECT** structure to a 32-bit pointer to access the data on a per-pixel level.

```
// create a 32-bit pointer to the surface pixel data  
unsigned long *pPixelData = NULL;  
pPixelData = (unsigned long *)LockInfo->pBits;
```

To retrieve the pixel at location (10, 4) we calculate the offset of that pixel as:

```
Row = LockInfo.Pitch * 4;    (Y=4 so we need to move 4 rows down)  
TotalOffset = Row + 10;      (X=10, so we move to 10th pixel in 4th row)
```

```
// Adjust pointer to point at correct pixel  
pPixelData += TotalOffset;
```

Now simply dereference the pointer to extract the pixel:

```
unsigned long SrcColor = *pPixelData;
```

When specifying a 32-bit color using hexadecimal, each full byte is represented as 0xFF (255). The hexadecimal layout of the color in code allows us to clearly see the two digits for each color component. Now let us see how we can break the DWORD into its separate A, R, G and B byte components using bitwise AND operations. Assume that SrcColor has a value of 0xFF326C94. To break the color into its separate BYTE components:

```
// SrcColor = 0xFF326C94;  
unsigned long DestColor = 0;  
  
// Lets extract each individual color component  
unsigned char Alpha = (SrcColor & 0xFF000000) >> 24; // 0xFF  
unsigned char Red = (SrcColor & 0x00FF0000) >> 16; // 0x32  
unsigned char Green = (SrcColor & 0x0000FF00) >> 8; // 0x6C  
unsigned char Blue = (SrcColor & 0x000000FF); // 0x94
```



At this point we have the alpha, red, green and blue components stored separately as byte values between 0 and 255. To pack this information back into the DWORD and write it to our 32-bit surface we use a series of bitshifting operations. A new color (if desired) could then be directly written to our surface as follows:

```
// Lets now take each of these values and rebuild it
DestColour = (Alpha << 24) | (Red << 16) | (Green << 8) | (Blue);

// Write Pixel to surface
pPixelData = DestColor;
```

So a color component is extracted by masking off the byte in which that component is contained and then shifting it to the right until it occupies the low (first) byte. Since bit masking is sometimes confusing for newcomers to programming, let us extract the red color component as a quick example.

```
(Bit 32) .....(Bit1)
SrcColor = 11111111001100100110110010010100 // 0xFF326C94
```

We mask out the component by &'ing the value with a bit mask of 0x00FF0000:

```
BitMask = 000000011111110000000000000000 // 0x00FF0000
          &
SrcColor = 11111111001100100110110010010100 // 0xFF326C94
          =
TempColor = 000000000011001000000000000000 // 0x00320000
```

All of the values contained within the alpha, green and blue components now equal 0. Only red bits remain set. The value now stored in the variable ‘TempColor’ has a hex value of 0x00320000. Since there are now 16 empty bits to the right of our red component value we will shift it so that it lines up with bit locations 1 through 8 (the bits which describe values between 0 and 255).

```
TempColor = 000000000011001000000000000000 // 0x00320000
          >> 16
Result = 00000000000000000000000000110010 // 0x00000032
```



The green and blue components essentially drop off the end leaving a red value of 0x00000032 (hex) -- 50 in decimal.

The same operation is applied to the other three color components. The only differences between the processes used to extract the colour components is that they each utilize unique bit masks and bit shifting values in order to extract that component's information. Try out the remaining three for yourself and see what you come up with.

We should be able to adjust the above code to extract color components from a 16-bit A4R4G4B4 surface just as easily. The only difference is that there are two bytes in total to store the colors (each color component is represented by 4 bits instead of 8). This of course reduces the available colors significantly. At 32-bits we had 255 intensity levels per component, but with a 4444 surface we have only 16 levels of intensity per color. This does not mean that the components cannot be as bright as a 32-bit color, but that between zero intensity and full intensity there are only 16 different shades of that color component available. When separated out, each component should have a value between 0 and 15. Finally, it should be clear that the color masks for each format would have to be constructed differently based on the color format. The following example shows how to extract the colors from an A4R4G4B 16-bit surface just for completeness -- the other formats you should be able to figure yourself by looking at the two examples provided.

For a single component to be at full intensity, all four bits would have to be set (1111). We know from the binary number system that this is equal to 15 in decimal, or 'F' in hexadecimal. We will use an example color of 0xF8C4 (ARGB = (15, 8, 12, 4)).

```
(Bit 16) .....(Bit1)
SrcColor = 1111001011000100 // 0xF8C4
```

To retrieve the red color component we need to isolate the first 4 bits in the high byte of the word. To mask off the green color value, we need to mask off the last 4 bits in the low word, and so on.

```
// WORD SrcColor = 0xF8C4;
WORD DestColor = 0;

// Lets extract each individual color component
unsigned char Alpha = (SrcColor & 0xF000) >> 12; // 0xF
unsigned char Red = (SrcColor & 0x0F00) >> 8; // 0x8
unsigned char Green = (SrcColor & 0x00F0) >> 4; // 0xC
unsigned char Blue = (SrcColor & 0x000F); // 0x4
```

Notice that we mask off the 4 bits of interest and reduce the number of bits that need to be shifted. In the 32-bit example the red bits started at bit 16, now they start at bit 8. So we only have to shift down by 8 bits this time to convert it to a byte value in the range of 0 - 15.



```
BitMask = 000011100000000 // 0x0F00  
      &  
SrcColor = 1111001011000100 // 0xF8C4  
      =  
TempColor = 0000001000000000// 0x0800
```

```
TempColor = 0000001000000000 // 0x0800  
      >> 8  
Result = 0000000000000010 // 0x08
```

If we wish to build a color with this format we use the same approach and shift the bytes into position.

```
// Lets now take each of these values and rebuild it  
DestColour = (Alpha << 12) | (Red << 8) | (Green << 4) | (Blue);  
  
// Write Pixel to surface  
pPixelData = DestColor;
```

## Textures and Memory Pools

Like vertex and index buffers, textures are resources that derive their interface from `IDirect3DResource9`. As with all resources, when they are created, we must choose the memory pool where we want them to be stored. We covered the `D3DPOOL` enumerated type when we discussed vertex buffers in Chapter 3, and most of the same rules apply. We will briefly examine memory pool usage as it applies to textures.

### `D3DPOOL_DEFAULT`

When textures are placed in `D3DPOOL_DEFAULT` we are indicating that we would like the driver to place the texture in the memory pool it considers optimal for rendering performance. This will typically be local video memory or non-local video memory (AGP memory). Unlike vertex and index buffers created in this pool, a `D3DPOOL_DEFAULT` texture cannot be locked unless it is created with the `D3DUSAGE_DYNAMIC` usage flag. The reason is that a driver may manipulate and rearrange the bits of data so that it can work with the texture data using its own format for maximum speed. This is called **swizzling**. Once the texture data has been swizzled, locking the surface could return a pointer to a texture format we no longer understand – one that does not correspond to the DirectX standard formats. If we specify a dynamic texture, we inform the driver that we will want to lock it at some



point. It now knows that the texture data should not be converted to an unknown format because we expect to read/write from the texture using the format that it was created with. This can carry a performance penalty because the driver will typically more efficiently with swizzled data.

Note that this is also the only pool type you can choose if you intend to use the `IDirect3DDevice9::StretchRect` function to copy one texture surface to another with automatic scaling. The same is true of the `IDirect3DDevice9::ColorFill` function which can be used to fill a texture surface with a color. We will examine both of these functions later in the lesson.

Video memory management falls to the application in this case. If there is not enough video memory to create a texture, you may need to evict other (perhaps least recently used) textures to make room for it. You should also create all of your `D3DPOOL_DEFAULT` resources before creating any `D3DPOOL_MANAGED` resources. Otherwise the memory management system employed by DirectX9 for your `D3DPOOL_MANAGED` resources will not be able to accurately track available memory.

Like other resources in this pool, when the device is lost, all textures in the `D3DPOOL_DEFAULT` pool are lost also and must be recreated.

### **D3DPOOL\_MANAGED**

When a texture is created in the `D3DPOOL_MANAGED` pool a copy is created first in system memory and then the data is uploaded to device memory as it is needed. The DirectX resource management system will make sure the texture data exists in the optimal device memory pool and that it can be locked. When the texture is locked we get back a pointer to the system memory copy of the surface. This makes reads and writes relatively fast. When the surface is unlocked, the modified image data is uploaded from the system memory copy into the actual texture surface (typically in video memory) and the changes to the texture will take effect.

One of the biggest advantages of using this pool is that the DirectX memory management system will remove least recently used textures from video memory and promote more recently used ones into that space. Each texture is given a time stamp describing the last time it was used. When a polygon(s) is about to be rendered that uses a texture not already in video memory, the texture with the oldest time stamped is removed and the required texture has its data uploaded from the system memory copy into video memory. The application can assign each texture a priority if desired. This way if two textures in memory currently have the same time stamp, the texture with the lowest priority gets evicted from video memory first. This allows us to hint that certain textures are more important than others. We might do this if the texture is large -- where constantly evicting and uploading it would affect performance.

Like all managed resources, these textures will not have to be recreated when a lost device is recovered. The device can automatically recreate the texture surface when the device is reset and upload the texture data from the system memory copy. This is done automatically; making `D3DPOOL_MANAGED` a reasonable default memory pool for applications that simply create textures from images loaded from files and use them to render texture mapped polygons.



There are several cases where you may not want to use **D3DPOOL\_MANAGED** textures. This pool cannot be used if you need to call any of the following IDirect3DDevice9 functions:

- StretchRect
- ColorFill
- UpdateSurface
- UpdateTexture

### **D3DPOOL\_SYSTEMMEM**

Textures created with this pool are placed in system memory and do not need to be recreated when the device is reset. Hardware accelerated devices generally cannot usually use system memory textures directly for rendering. You can check whether this is the case by checking the D3DCAPS9::DevCaps member returned from the IDirect3DDevice9::GetDeviceCaps function. This member is a bit field so you can check to see if it has the **D3DDEVCAPS\_TEXTURESYSTEMMEMORY** flag set as follows:

```
D3DCAPS9 caps;
pDevice->GetDeviceCaps(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &caps);
if(cap.DevCaps & D3DDEVCAPS_TEXTURESYSTEMMEMORY)
{
    //Texturing from system memory is supported
}
```

Typically, software devices (such as the reference rasterizer) are limited to only using system memory textures and cannot use video memory textures. You can check if a device can or can not support video memory textures by checking the DevCaps member for the **D3DDEVCAPS\_TEXTUREVIDEOMEMORY** flag. This is not something your application will usually need to do because if you use the **D3DPOOL\_MANAGED** or **D3DPOOL\_DEFAULT** pool types then the driver will create the texture in the memory pool preferred by the device. This will normally be in video memory for hardware devices and system memory for software devices.

Since most applications will be using a hardware device you might wonder if there would ever be a need to create system memory texture resources on a hardware device if they cannot be used directly for texturing. As it happens there is a need for this resource type if you intend to manage texture memory yourself (using **D3DPOOL\_DEFAULT**) and need to frequently alter the contents of the texture.

When managing textures yourself, your application will typically want to create a system memory copy of all of the **D3DPOOL\_DEFAULT** created textures. If the application needs to update the contents of the texture, it is usually fastest to make the alterations to the system memory copy and then transfer the contents up to the video memory version with the IDirect3DDevice9::UpdateTexture member function. This function is designed specifically for this purpose. It accepts two textures (a source and a destination texture) where the source texture must be a system memory texture and the destination texture must be a **D3DPOOL\_DEFAULT** texture. These copies are also useful when the device becomes lost because **D3DPOOL\_DEFAULT** textures will need to be recreated. Since **D3DPOOL\_SYSTEMMEM** textures persist, they are available for copying back up to the hardware.



## D3DPOOL\_SCRATCH

In this pool the texture is created in system memory but is not accessible to the device. These textures can be understood as simple data containers. While they can be locked, and their bits copied to and from the surfaces of other textures, we cannot use a texture in this pool to render textured polygons.

We will shortly see that devices often put restrictions on textures used for rendering. A common prerequisite is that the texture dimensions be a power of 2 (and on some very old hardware textures must be perfectly square). Some hardware devices may impose a maximum texture size such as 256x256 or 512x512. Textures created in the `D3DPOOL_SCRATCH` pool do not have any of these restrictions – but that is hardly surprising since these restrictions are imposed by the device and the device cannot access textures in the `D3DPOOL_SCRATCH` pool.

## MIP Maps

We continue our exploration of textures and texture memory with a brief discussion of MIP maps. Let us begin by examining their origins and move on to storage implications.

When working exclusively with 2D graphic images, we can choose to maintain precise dimensions when copying images to the screen. For example, we could sample an image with a pixel/texel ratio of 1:1 -- every texel in the source image mapping precisely to one pixel on the display. But when working in 3D we generally will not have this ability. We may have textures that are mapped to a polygon far in the distance taking up fewer pixels on the screen than there are texels in the texture. The polygon may have screen space dimensions of 64x64 pixels, while the texture mapped to it has 128x128 texels. When this is the case, it is clear that we can no longer copy all of the texture detail onto the polygon. As each screen space pixel is mapped back to a 2D texel coordinate by the rasterizer, some texels will have to be left out (every other texel in the aforementioned case).

If you have ever done any sound recording, you can liken this process to sampling a sound. Provided we use a high enough sample rate, the recorded sound will be very close to the original -- as is the case with music recorded on compact discs. Reducing the sample rate results in the loss of accuracy and detail. Texture mapping a polygon on the display is much like this. The fewer pixels we have to work with on the display, the less detail we will be able to preserve.

Simply skipping texels like this can result in the loss of important detail information that we may not be prepared to sacrifice. But if you have ever scaled down an image in a paint package, you know that detail preservation is attempted. Most paint packages include filters that use pixel averages to reduce the picture in such a way that it remains a good approximation of the original image. If you use a pixel discarding process, you will notice that when scaling down by large amounts the image can be quickly distorted -- especially noticeable if there is text on the original image. In the downsampled result, the loss of detail may be significant. Fig 6.5 shows a 128x128 texture (left) mapped to a (64x64) quad. A simple pixel skipping algorithm is used to make the texture fit the polygon:



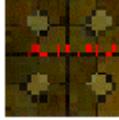
128x128 Texture	64x64 Quad
	

Figure 6.5

The scaled down image in Fig 6.5 looks rather bad and the text has become totally unreadable. This would only get worse as the polygon gets smaller. Further, it will also produce a shimmering artifact as the interpolated floating point texture coordinate for each pixel in the polygon is mapped into the 2D texture space and snapped to the nearest integer texture coordinate. The floating point rounding causes a given pixel's corresponding texel in the texture map to fluctuate between neighboring texels and pixel colors can appear to change as the object moves nearer or further away from the viewer.

There are also problems with the opposite scenario. As polygons approach the camera and begin to occupy more pixels than the texture, we reach a point where we are mapping 1 texel to multiple pixels. This stretches the texture image to fill the surface of the quad causing an undesirable blocky appearance (Fig 6.6).

A 64x64 texture map will look fine when applied to a polygon of approximately the same size.	A blockiness and lack of detail appears when the polygon is larger than the texture and many pixels map to a single texel.
	

Figure 6.6

A common solution to both of these problems is **MIP mapping**. MIP is short for Multum In Parvo -- a Latin phrase meaning ‘much in small’ (i.e. many things in a small place). A MIP map is essentially an

ordered series of texture surfaces where each one is half the size in each dimension as its predecessor. For example, we could create a texture to hold a 256x256 image. If we instruct our texture loading function to create a MIP map chain, it should create an array of surfaces such that the next one in the chain is 128x128 and the next is 64x64 and the next is 32x32 – all the way down to 1x1 if that many MIP levels are desired. Since we will use D3DX texture loading functions for this procedure, in memory, each one of these surfaces will be a separate surface object that is managed by the main Direct3DTexture9 object. We instruct the D3DX texture loading functions to take the image stored in the top level surface (this is our base texture image) and sample the image down into all of the descending MIP surfaces as shown in Fig 6.7.

## A MIP map Texture ( 4 Levels)



Figure 6.7

**Note:** In Fig 6.7 all of the textures are square but this does not have to be the case. The MIP levels reflect the scaled dimensions of the top level surface. Some older 3D graphics cards do insist on square textures, so we will discuss how to check the device for that limitation later in the lesson.

The D3DX texture loading functions can use filtering algorithms to generate a high quality downsampled image. They do this by calculating the color of every texel in a MIP surface using a weighted average of neighboring pixels in the preceding surface in the chain. Most paint packages such as Jasc's Paint Shop Pro™ use similar filters that allow you to scale the image down by a fair amount before the image starts to become conspicuously corrupt. The following set of images show a 128x128 texture, followed by this same texture downsampled onto a 64x64 quad by the rasterizer. The third image is a 64x64 MIP level generated by a D3DX texture loading function.

	<p>A 128x128 image. When the texture has MIP surfaces, this will be the dimensions and image in the top level MIP surface (Level 0). When the texture does not have MIP surfaces, this will be the only surface used for texturing the polygons that use it regardless of their distance from the camera.</p>
---	---



	When the texture does not have MIP surfaces the image is downsampled onto a 64x64 quad by the rasterizer using a fast but crude algorithm causing crude sampling artifacts.
	If the texture has MIP surfaces the correct MIP surface will be used to texture the quad. The D3DX loading function has done a much nicer job of downsampling the image to this resolution

We can access each surface separately by using the `IDirect3DTexture9::LockRect` function and specifying the MIP level (zero based indices) that we want to retrieve a pointer to. This is what the first parameter in the `LockRect` function is used for. For example, your artist may think that despite the advanced filtering algorithms provided by D3DX, some important detail has become unacceptably blurred. In such case they may want to create a new touched up image for a given MIP level. Although this is probably not something that will happen very often, the ability to access each MIP surface individually and copy data directly provides you with this capability.

The next example shows a 128x128 texture downsampled by the D3DX texture loading functions to fit on a 64x64 MIP surface. Compare that to the same image sampled down by the artist and copied into the 64x64 MIP level. In the example, the floor tile looks basically the same and there has been no real benefit over the automatic downsampled version. However, the artist has chosen to use a smaller, sharper font for the text which makes it appear cleaner.

	In this image we have let the D3DX texture loading function generate the 64x64 MIP level by downsampling the original 128x128 image using filtering. It has done a fair job although the text is slightly blurred and hard to read.
	In this image, the 64x64 image was created by the artist in a paint package and a different font was added that was cleaner and looked better at a smaller resolution. This image could be loaded and copied into the 64x64 MIP surface of our texture.

If you decide that the texture you are creating will not have MIP maps, then the first `LockRect` parameter will always be 0 to specify the top level (and in that case, the only) surface.

When the device is rendering the scene, it will perform a distance calculation to determine which MIP level should be used to texture a polygon (pixel). If we are rendering a polygon that is close to the camera, it might use the top level surface of the texture (perhaps a 512x512 texture with lots of detail). As the polygon gets further away and becomes increasingly smaller, then the device will automatically select a smaller MIP surface that more closely matches the size of the polygon being rendered. This is more efficient and minimizes aliasing artifacts.

We can also gain a performance boost using MIP maps at the cost of additional memory footprint. Rather than render a single texture with one resolution, it is faster to use multiple textures at varying resolutions. When rendering a small polygon with MIP maps, a smaller MIP level will be chosen.



Many more of the texels we intend to use will fit into the cache memory during rendering and fewer will have to travel across the bus when uploading to the hardware is necessary. In the best case when we are using the lowest MIP levels (1x1, 2x2, 4x4, etc.), the entire MIP surface could be cached for the rendering of distant polygons.

## Loading Textures

Let us now look at how to load image data in from a file. Fortunately, we can accomplish this with a single function call using D3DX. There are two functions for creating and loading textures in DirectX Graphics (D3DXCreateTextureFromFile and D3DXCreateTextureFromFileEx). The first is the easiest to use and takes fewer parameters as it assumes certain default values. The second function provides much more flexibility but includes a larger parameter list. We will cover the latter first so that we can better understand the default values used by the former.

### D3DXCreateTextureFromFileEx

```
HRESULT D3DXCreateTextureFromFileEx
(
    LPDIRECT3DDEVICE9 pDevice,
    LPCSTR 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
);
```

#### LPDIRECT3DDEVICE9 pDevice

This parameter is a pointer to the device interface. It is needed because the device owns the texture memory and because this function calls the D3DXCheckTextureRequirements function to check that the other inputs to this function are valid (i.e. that properties such as the width, height, and format are supported by the current device). If the any of the inputs are invalid, then they are adjusted to find the best supported match.

#### LPCSTR pSrcFile

This is a pointer to a string storing the texture filename. It can include relative or absolute paths -- or no path at all to load a texture from the application's current working directory. D3DX supports a wide range of image formats (bmp, tga, jpg, png, dib, dds). This function automatically handles the creation of the texture object and the conversion of image data from the file into the texture surface.



### **UINT Width**

This is the width of the desired texture. It does not have to be the same as the width of the image that we are loading because the function will scale the image to fit using one of the filter types specified in the **Filter** parameter to this function. If we specify **D3DX\_DEFAULT** for this parameter, then the width is taken from the image in the file. If this width is not a supported texture width, then the closest supported width is chosen and the image is scaled to fit.

### **UINT Height**

This is the height of the desired texture. Same rules as above.

### **UINT MipLevels**

This is where we specify the desired number of MIP map levels for the texture. A value of 1 indicates a texture with a top level surface only. If the value we specify is not a valid number it will be adjusted to create a texture with a MIP level count supported by the device. **D3DX\_DEFAULT** (or a value of 0) will create a texture with a MIP chain down to 1x1 in dimensions. This means a 128x128 texture would have 8 levels with dimensions shown below:

<b>128 x 128</b>	<b>Level 0</b>
<b>64 x 64</b>	<b>Level 1</b>
<b>32 x 32</b>	<b>Level 2</b>
<b>16 x 16</b>	<b>Level 3</b>
<b>8 x 8</b>	<b>Level 4</b>
<b>4 x 4</b>	<b>Level 5</b>
<b>2 x 2</b>	<b>Level 6</b>
<b>1 x 1</b>	<b>Level 7</b>

Once the image has been loaded into the top level surface and scaled (if necessary) using the filtering algorithm specified in the **Filter** parameter, it is sampled from the top level down through to the bottom level automatically. The filtering algorithm used to resize the image from one MIP level to the next does not need to be the same as the filtering algorithm used to scale the image file at the top level surface. We specify the filtering algorithm we would like the function to use when generating the MIP map images with the **MipFilter** parameter to this function.

### **DWORD Usage**

For general texture use, you will usually set this flag to zero. You can however specify the **D3DUSAGE\_DYNAMIC** flag if you wish the function to create a dynamic texture. You can check to see if the device supports dynamic textures using the **IDirect3DDevice9::CheckDeviceFormat** function, passing in the texture format and the **D3DUSAGE\_DYNAMIC** flag. The other flag that we can specify is the **D3DUSAGE\_RENDERTARGET** flag. If a texture is created as a render target, we can tell the device to render the scene to the texture surface instead of the frame buffer.

### **D3DFORMAT Format**

This parameter indicates the pixel format that we desire. If the format is not supported by the current device then the function will find the next closest match. This format does not need to be the same



format or bit depth as the image we are loading. Once the texture has been created, the function will copy the image data to the surface performing the appropriate color conversion. This is useful because it allows us to request a texture format that is identical to the frame buffer format. When we do so and then use this texture for rendering, no color conversion has to be performed and we can render at maximum speed. If we specify `D3DFMT_UNKNOWN` then the texture surface created will be the closest match to the pixel format of the image data in the file. Because of this, the function will very rarely fail -- unless the file name is incorrect or an invalid device was passed.

### D3DPOOL Pool

This is where we specify the memory pool that we would like the texture to be created in.

### DWORD Filter

If the size of the image being loaded does not match the size of the texture we are creating, the image will be scaled to fit the texture surface. This parameter allows us to specify the filtering technique that should be used to downsample (or upsample) the image to the top level surface. The possible algorithms that we can choose from are listed below and vary in the quality of filtering they provide. These are the more common filtering options, but be sure to refer to the SDK documentation for a more complete listing.

<code>D3DX_FILTER_NONE</code>	No filtering takes place at all. If the image from the file does not fit on the created texture surface it is simply cropped to fit. If the image is smaller than the texture surface then all unused pixels in the texture will be transparent black.
<code>D3DX_FILTER_POINT</code>	Each texel gets its color from the nearest equivalent pixel in the file image. This can cause resizing artifacts.
<code>D3DX_FILTER_LINEAR</code>	Each pixel in the texture surface is computed by averaging the four nearest pixels in the source image.
<code>D3DX_FILTER_TRIANGLE</code>	This is the slowest filter but provides the best re-sampling quality. Every pixel in the source image contributes equally to the image on the destination texture surface. This is the default filtering type used if you specify <code>D3DX_DEFAULT</code> for either the <b>Filter</b> or <b>MipFilter</b> parameters.
<code>D3DX_FILTER_BOX</code>	Each pixel is computed by averaging a $2 \times 2 (\times 2)$ box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source (as is the case with MIP maps).
<code>D3DX_FILTER_DITHER</code>	The resulting image is dithered using a 4x4 ordered dithering algorithm. This can be combined with any of the above filter types.

If we set this parameter `D3DX_DEFAULT` then the filtering method used is equivalent to specifying both `D3DX_FILTER_TRIANGLE` and `D3DX_FILTER_DITHER`.



## DWORD MipFilter

While the Filter parameter specifies the filter used to sample the source image into the top level texture surface, the MipFilter parameter allows us to specify the filter type used to filter images down through the MIP chain surfaces. All of the filters listed above are valid, and the default is a combination of **D3DX\_FILTER\_TRIANGLE** and **D3DX\_FILTER\_DITHER**.

## D3DCOLOR ColorKey

The ColorKey allows us to create textures that have transparent regions. This is useful for texture images such as windows, chain link fences, foliage, etc. where the viewer should be able to see through portions of the texture into sections of the scene that are rendered behind it. When each pixel is read from the source image and converted into the texture surface format, it is compared against the ColorKey color -- which has itself also been converted into the destination format. If the pixels match exactly, then the color of the pixel is replaced with transparent black. When alpha testing is enabled during rendering (Chapter 7), pixels with an alpha value of 0 can be ignored, even when they pass the depth test and are closer than a pixel already in the frame buffer.

## D3DXIMAGE\_INFO \*pSrcInfo

When the function returns, this structure will hold information about the original image data found in the file. NULL can be passed if your application does not require this information.

```
typedef struct _D3DXIMAGE_INFO
{
    UINT Width;
    UINT Height;
    UINT Depth;
    UINT MipLevels;
    D3DFORMAT Format;
    D3DRESOURCETYPE ResourceType;
    D3DXIMAGE_FILEFORMAT ImageFileFormat;
} D3DXIMAGE_INFO;
```

In this structure we will find the width, height, and pixel format of the image in the file that was loaded. We can also retrieve the number of MIP levels that were in the file and the resource that represents the type of the texture stored in the file (**D3DRTYPE\_TEXTURE**, **D3DRTYPE\_VOLUMETEXTURE**, or **D3DRTYPE\_CUBETEXTURE**). All files imported using the dds surface format (used by DirectX) provide this resource information. The Depth parameter is only applicable to 3D textures (volume textures). Finally, the **D3DXIMAGE\_FILEFORMAT** member describes the type of file that contained the image and is expressed as one of the members of the enumerated type shown below.

```
typedef enum _D3DXIMAGE_FILEFORMAT
{
    D3DXIFF_BMP = 0,
    D3DXIFF_JPG = 1,
    D3DXIFF_TGA = 2,
    D3DXIFF_PNG = 3,
    D3DXIFF_DDS = 4,
    D3DXIFF_PPM = 5,
    D3DXIFF_DIB = 6,
    D3DXIFF_FORCE_DWORD = 0x7fffffff
} D3DXIMAGE_FILEFORMAT;
```



### **PALETTEENTRY \*pPalette**

This member is used when we are creating 8 bit surfaces that use a color palette. Palletized surfaces contain a maximum of 256 possible pixel colors. We will not use such textures in this course as we will prefer a greater range of colors. Support for this type of texture on more modern graphics cards may even be unavailable, so be sure to check the device capabilities if you intend to use one.

### **LPDIRECT3DTEXTURE9 \*ppTexture**

This is the address of a pointer to an IDirect3DTexture9 interface. It will be assigned a valid interface if the function is successful. We use the IDirect3DTexture9 interface to work with the texture surface and to send it to the device for rendering.

The following code shows how to use this function to load a file based image into a 256x256 32-bit RGB surface in the managed memory pool:

```
IDirect3DTexture9 *pNewTexture = NULL;

D3DXCreateTextureFromFileEx(m_pDevice,
    "Brickwall.bmp",           // Our Device
    256, 256,                  // File name of texture
    D3DX_DEFAULT,               // We want a 256x256 texture
    0,                          // Create MIP chain to 1x1
    D3DFMT_X8R8G8B8,            // No special usage flags
    D3DPPOOL_MANAGED,           // Desired texture format
    D3DX_DEFAULT,                // A Managed texture
    D3DX_DEFAULT,                // Use default filtering if
    // image needs scaling to fit
    // texture
    D3DX_DEFAULT,                // Use default filtering to
    // generate mip map images
    0,                          // No Color key
    NULL,                        // Don't want file info
    NULL,                        // No palletized surface
    &pNewTexture);              // Pointer to created texture
```

That is basically all there is to loading a texture and preparing it for use. Later we will see how to send the texture to the device for rendering.

### **D3DXCreateTextureFromFile**

The D3DXCreateTextureFromFile function does not have the flexibility of the extended version of the function, but it has a much more manageable parameter list.

```
HRESULT D3DXCreateTextureFromFile
(
    LPDIRECT3DDEVICE9 pDevice,
    LPCSTR            pSrcFile,
    LPDIRECT3DTEXTURE9 *ppTexture
);
```



All we have to do is pass in our device, a filename, and a pointer to a texture interface pointer. It is the equivalent of calling D3DXCreateTextureFromFileEx with the excluded parameters set to either **D3DX\_DEFAULT** or 0. This function can be used as shown below:

```
IDirect3DTexture9 *pTexture = NULL;  
D3DXCreateTextureFromFile( pDevice , "Brickwall.bmp" , &pTexture);
```

The equivalent would be calling D3DXCreateTextureFromFileEx with the following parameters:

```
D3DXCreateTextureFromFileEx  
(  
    pDevice,           //Pass our device  
    pSrcFile,          //Image file name  
    D3DX_DEFAULT,        //Choose closest compatible width  
    D3DX_DEFAULT,        //Choose closest compatible height  
    D3DX_DEFAULT,        //Generate complete MIP map chain  
    0,                  //No special usage flags  
    D3DFMT_UNKNOWN,     // Choose closest compatible pixel format  
    D3DPOOL_MANAGED,    // Create texture in the Managed tool  
    D3DX_DEFAULT,        // Use default scaling filter  
    D3DX_DEFAULT,        // Use default Mip sampling filter  
    0,                  // No color key  
    NULL,               // No Image information returned  
    NULL,               // No Palette info returned  
    &pTexture          // Pointer to the created texture interface  
) ;
```

## D3DXCreateTexture

If we need to generate a blank texture and fill in the image data ourselves, D3DX includes functions that allow us to create a texture object without loading file image data into it. We may need to do this if we have our own texture file reading code, or if we wanted to generate texture images in our code (procedural texturing). D3DXCreateTexture includes a parameter list similar to the file loading function we saw previously.

```
HRESULT D3DXCreateTexture  
(  
    LPDIRECT3DDEVICE9  pDevice,  
    UINT Width,  
    UINT Height,  
    UINT MipLevels,  
    DWORD Usage,  
    D3DFORMAT Format,  
    D3DPOOL Pool,  
    LPDIRECT3DTEXTURE9 *ppTexture  
) ;
```

If we specify an unsupported width, height, or format, the function will find the next closest match that is supported and should not fail. To use the function:



```
IDirect3DTexture9 * pTexture = NULL;
D3DXCreateTexture( pDevice , 100 , 100 , 6 , 0 , D3DFMT_A4R4G4B4,
                  D3DPOOL_MANAGED, &pTexture );
```

**Note:** When a texture or surface has a pixel format with an alpha component, the texture is said to use an 'Alpha Channel'. This per-pixel alpha can be used when alpha blending is enabled to provide per-pixel transparency (Chapter 7).

## D3DXCreateTextureFromFileInMemoryEx

There are two D3DX functions to create textures from files stored in memory. This can be useful if you load the complete contents of the file (header and all) into a memory location. Simply provide a pointer to this memory location and the textures are created in exactly the same way with the exception that the data is read from memory rather than a file. These two functions are called D3DXCreateTextureFromFileInMemoryEx and D3DXCreateTextureFromFileInMemory. Both are analogous to D3DXCreateTextureFromFileEx and D3DXCreateTextureFromFile respectively. The function prototypes are shown below. The only difference is that we pass a void pointer to the first byte in the memory block containing the file.

```
HRESULT D3DXCreateTextureFromFileInMemoryEx
(
    LPDIRECT3DDEVICE9 pDevice,
    LPCVOID pSrcData,
    UINT SrcDataSize,
    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 will not explain the parameter list again. However, the second parameter should now be a pointer to the file in memory (as opposed to string containing the file name) and the third parameter should describe the size of the file in memory in bytes.

D3DXCreateTextureFromFileInMemory uses a simplified parameter list with `D3DX_DEFAULT`, 0, or `NULL` values as substitutes for the absent parameters.

```
HRESULT D3DXCreateTextureFromFileInMemory
(
    LPDIRECT3DDEVICE9 pDevice,
    LPCVOID pSrcData,
    UINT SrcDataSize,
    LPDIRECT3DTEXTURE9 *ppTexture
);
```



## D3DXCreateTextureFromResourceEx

There are two functions that allow you to create textures from files stored as application resources. When binding a file into your resource file you should make sure that you use either the RT\_BITMAP resource type to store bitmap files (bmp) or the RT\_DATA type to store files for other supported formats (tga, png , jpg, etc.). When we create a texture from a resource, we pass the handle of the module that contains the resource and the string identifying the resource within that module using the second and third parameters.

```
HRESULT D3DXCreateTextureFromResourceEx
(
    LPDIRECT3DDEVICE9 pDevice,
    HMODULE hSrcModule,
    LPCSTR pSrcResource,
    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
);
```

The simplified version of this function is shown below:

```
HRESULT D3DXCreateTextureFromResource
(
    LPDIRECT3DDEVICE9 pDevice,
    HMODULE hSrcModule,
    LPCSTR pSrcResource,
    LPDIRECT3DTEXTURE9 *ppTexture
);
```



## Creating Textures Manually

D3DX functions are essentially wrappers around the texture and surface creation functions exposed by the device. The `IDirect3DDevice9::CreateTexture` method can be used to create a blank texture in much the same way as `D3DXCreateTexture`:

```
HRESULT CreateTexture
(
    UINT Width,
    UINT Height,
    UINT Levels,
    DWORD Usage,
    D3DFORMAT Format,
    D3DPOOL Pool,
    IDirect3DTexture9 **ppTexture,
    HANDLE* pHandle
);
;
```

**Note:** The final parameter is reserved for future use. It should be set to NULL.

There are several parameters that are absent when compared to the `D3DXCreateTexture` function. The reason is that this function simply tries to create the specific texture object that is indicated by the parameters. The missing filtering parameters make sense since this is a blank texture (possibly with blank MIP surfaces). If you choose not use D3DX functions then you will need to write your own code to import the different image file formats and convert them into the proper surface color format. You will also need to deal with scaling, filtering, and filling MIP surfaces yourself.

The call will fail if the parameters are invalid or if the requested settings (format, width, height, or usage) are not supported on the current device. It is the application's responsibility to test for failure and adjust the parameters accordingly. Furthermore, when the application is creating textures in this way it should check the device capabilities -- especially the common ones listed below. Once you get the `D3DCAPS9` structure using the `IDirect3DDevice9::GetDeviceCaps` function, you should check the following fields to determine support:

### MaximumTextureWidth

A device will typically have a maximum texture width. If the image you are loading is wider than this limit then you will need to scale the image to fit the maximum texture size. If you try to create a texture using the above function and you specify a texture wider than this limit, the texture creation function will fail.

### MaximumTextureHeight

This specifies a value describing the maximum texture height. If you try to create texture with a larger height value than this, texture creation will fail.



## TextureCaps

The TextureCaps flag is a bit field. We will need to check the following bits to make sure that the width and height of our desired texture is supported by the device. The two bits we need to check are:

**D3DTEXTURECAPS\_POW2** – If this bit is set then it means the width and height of the texture must be a power of two (ex. 64x32, 128x256 and 512x512). If you need to adjust a value so that it is a power of 2, simply start with a value of 1 and shift it left, each time testing whether it is greater than or equal to the original number. The following function would accept a width or height parameter and if it is not a power of 2, round up to the nearest power of 2.

```
int GetPowerOfTwo (int Number)
{
    int n = 1;
    while (n < Number) n<<1;
    return n;
}
```

**D3DTEXTURECAPS\_SQUARE** – If this bit is set then the device only supports textures that are perfectly square (width = height).

Lab Project 6.5 examines how to create textures using IDirect3DDevice9::CreateTexture rather than a D3DX helper function. It will check the capabilities of the device and modify the texture creation parameters when they are not supported. For all other demos we will use the D3DX texture loading functions so that all of this labor is handled automatically.

## Setting a Texture

A typical game level will use many textures and your faces will probably contain indices into a (global) scene level texture array. The application can create all required textures at application initialization or as needed – although initialization is preferred. Before rendering a face or a group of faces, we will call IDirect3DDevice9::SetTexture to tell the device which texture to use with the next DrawPrimitive call. For the time being, you can think of it as analogous to the IDirect3DDevice9::SetMaterial function. Like that function, the texture will persist for DrawPrimitive calls until changed.

```
HRESULT SetTexture (DWORD Stage, IDirect3DBaseTexture9 *pTexture);
```

If we created an array of three textures:

```
IDirect3DTexture9 *Textures[3];
D3DXLoadTextureFromFile (m_pDevice , "Texture1.bmp" , &Texture[0]);
D3DXLoadTextureFromFile (m_pDevice , "Texture2.bmp" , &Texture[1]);
D3DXLoadTextureFromFile (m_pDevice , "Texture3.bmp" , &Texture[2]);
```

We could then render all of the faces that use these textures as follows:



```
m_pDevice->BeginScene  
    m_pDevice->SetTexture ( 0 , Textures[0]);  
    m_pDevice->DrawPrimitive ( Render all triangles that use texture 1 );  
    m_pDevice->SetTexture ( 0 , Textures[1]);  
    m_pDevice->DrawPrimitive ( Render all triangles that use texture 2 );  
    m_pDevice->SetTexture ( 0 , Textures[2]);  
    m_pDevice->DrawPrimitive ( Render all triangles that use texture 3 );  
m_pDevice->EndScene
```

Note that we see polygon state batching once again. Later in our workbook we will revisit the IWF level loaded in the last chapter and see to how load in textures referenced from an IWF file. We will also adjust the CLightGroup class used in Lab Project 5.3 to batch by both texture and material.

**Note:** Batching polygons to reduce the number of SetTexture calls is very important since the SetTexture device state change can be one of the most expensive. This is especially true if there is not enough video memory on the card to contain all of the textures used to render a single frame. For example, when using managed textures on a hardware device, if a texture used for rendering is not currently in video memory because there is not sufficient room, then a texture currently in video memory will need to be evicted to make room for the new texture. The system memory copy of the texture will then be uploaded to the hardware so that the rasterizer can access it. Passing texture data over the bus can be slow given the number of potential pixels involved. If we do not batch calls to SetTexture, this could result in a texture being evicted and then uploaded to video memory several times in a single frame. This is minimized if we batch by texture, because once all of the polygons that use a texture have been rendered, the texture can be evicted and will not be needed until the next frame -- at which point it will be uploaded again. Ideally, you will have enough video memory available to fit the textures for entire regions in a level (or perhaps even entire scenes) so that textures are only evicted and uploaded as new areas are entered by the player.

The Stage parameter above will be discussed later in the lesson. For now just know that you can simultaneously set multiple textures that will be blended together during rendering and that each texture (along with certain settings) is assigned to a texture stage. We will forget about multiple textures for the time being and simply set the Stage member to 0. This is the top level texture stage that must be used when we are only using single textured polygons.

## Texture Coordinates

Texture coordinates are ordered tuples that define a mapping from locations on the polygon surface back to pixel locations in texture space. 2D coordinates are by far the most common since almost all texture maps we use will be two dimensional images. However 1D and 3D texture coordinates are also possible for special circumstances and texture types. For 2D coordinates we use a normalized coordinate system where both the horizontal (U) and vertical (V) axis range is [0.0, 1.0] regardless of texture size (Fig 6.8). Every pixel in the image can be accessed using UV coordinates in this range.

The process is similar to what we saw with colors stored at vertices. Recall that when Gouraud shading is enabled, the vertex positions are converted into screen space coordinates and then the color of each

pixel is calculated by calculating its distance from each of the three vertices. We then used these distances to calculate a weighted blend of the three vertex colors to generate a final pixel color.

The rasterizer will calculate a 2D set of texture coordinates for each pixel that can be used to index into the texture surface and retrieve a color. Just as we needed to supply a per-vertex color that was interpolated across the surface, we need to supply a set of per-vertex texture coordinates for surface interpolation as well. This interpolation will generate a per-pixel texture coordinate by calculating the distance of the pixel from each vertex in the polygon. Using these distances, we perform a weighted interpolation for the current pixel. Once a per-pixel texture coordinate is found, a color is retrieved from the texture and used to fill the pixel. This is a simplified description of the process, but serves our purposes for this discussion.

## Texture Coordinate System

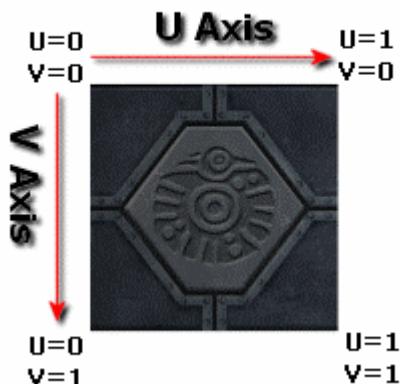
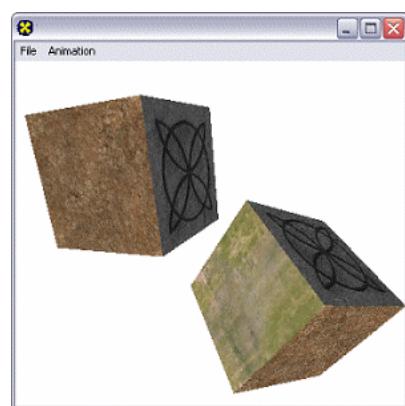


Figure 6.8

Texture coordinates will usually be generated in a level editing package and the data simply loading in along with the rest of the vertex data.

Let us briefly look at how to calculate a correct UV coordinate for any texel in a texture using a simple calculation. In Lab Project 6.1 we return to the infamous spinning cubes (right image). This time the faces of our cubes will have textures mapped to them. We will apply a different one to every face in the cube mesh.

In the texture coordinate system, the UV coordinate  $(0, 0)$  is considered to be the top left texel in the image and  $(1, 1)$  is the texel in the bottom right corner. Fig 6.10 shows how a texture might be mapped to a quad. Each vertex in the quad stores a coordinate pair that maps to the corresponding corner in the texture map. The texture is assumed to be 256x256 texels in this example but it does not matter what size the texture is. These texture coordinates will map the entire contents of the texture to the surface of the quad when it is rendered.



## Vertex Coordinates

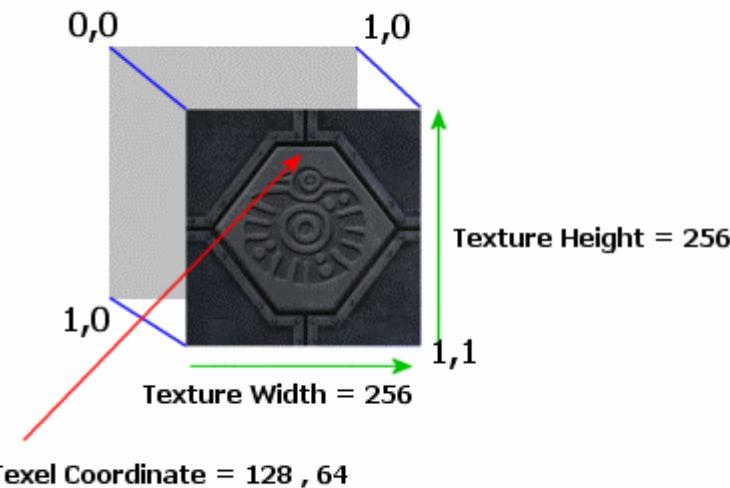
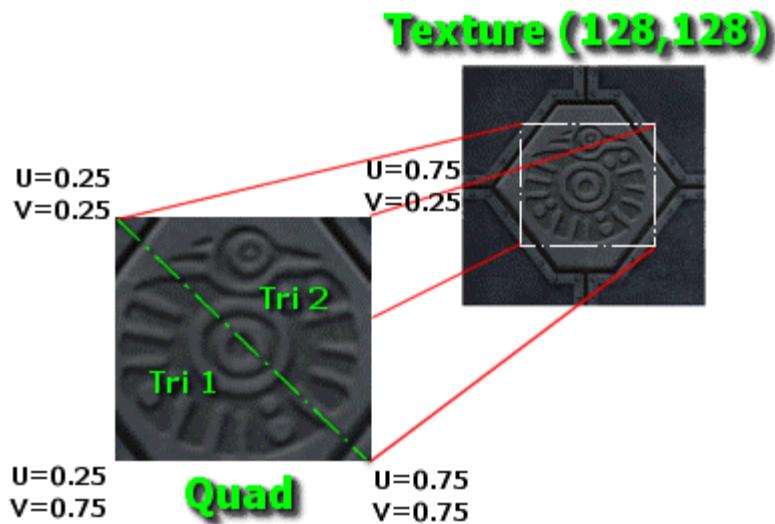


Figure 6.10

In Fig 6.10 we see how to calculate the UV coordinates for a pixel in the image at location (128, 64). Note that we must take the dimensions of the image into account using a division. The coordinate (128, 64) in the above 256x256 image would generate a UV coordinate pair of (0.5, 0.25).

Fig 6.11 is a bit more revealing. Here we see a 128x128 texture mapped to a quad. This time however, the texture coordinates stored in the vertices do not map to the four corners of the texture but rather to only a particular section of the image. This section is them mapped over the entire surface when the quad is rendered, and scaled to fit. The green diagonal line reminds us that the quad is really two triangles:



### Mapping a Quad to Part of a Texture

Figure 6.11

The texture coordinates stored at each vertex in Fig 6.11 describe a rectangle on the texture surface. This is the section used to map to the polygon. Note that if we were to alter these texture coordinates between frames, we could give the appearance of making the texture slide across the surface.

Ultimately we can think of texture coordinates as defining a window. Anything on the texture surface that falls within the window is mapped to the polygon and texels outside the window are not.

Although we have looked at rectangular regions, this need not be the case. Typically, you will want the window described by the polygon vertices to be the shape of the polygon itself so that the image on the texture is not unevenly squished out of shape -- but this is not a requirement. Your texture coordinates can define any shape, or even map to the same texel.

In Fig 6.12 we are mapping a texture to a triangle. We will usually want the texture coordinates of its vertices to define a triangular region on the texture of similar proportion so that the image does not look too distorted when its texels are interpolated across the surface.

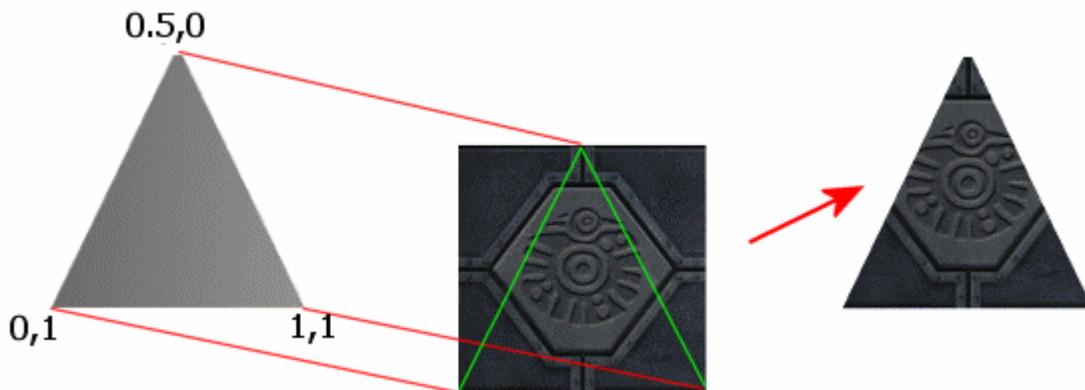


Figure 6.12

## Vertex Texture Coordinates

The texture coordinate set for a vertex is stored in the vertex itself and is typically represented by two floating point values. As discussed in Chapter 2, we use flags to tell the pipeline that our vertex structure has one (or more) sets of texture coordinates. The Flexible Vertex Format flags are listed again below with the ones we are currently interested in highlighted.

Common FVF Flags	Description
<b>D3DFVF_XYZ</b>	Informs the device that the vertices are untransformed and will need to be sent through the transformation pipeline.
<b>D3DFVF_XYZRHW</b>	Informs the device that this vertex is pre-transformed and should not be sent through the transformation and lighting pipeline. The X and Y members of the vertex describe the screen space coordinates and the Z member describes the depth buffer value between 0.0 and 1.0.
<b>D3DFVF_NORMAL</b>	This flag can be used to inform the device that the vertex contains a normal vector that is used by the lighting pipeline to calculate diffuse and specular lighting.
<b>D3DFVF_DIFFUSE</b>	If lighting is disabled the vertex have a diffuse color component. If lighting is enabled this may be used as a material reflectance property. If lighting is not enabled then this contains the vertex diffuse color. If the vertex also contains a specular color these are added together to become the final color of the vertex.
<b>D3DFVF_SPECULAR</b>	The vertex has a specular component. When lighting is enabled this can store a material reflectance property. When lighting is disabled this describes the specular color of the vertex.
<b>D3DFVF_TEX0 through D3DFVF_TEX8</b>	DirectX Graphics supports vertices with up to 8 sets of texture coordinates. Many graphics cards available at this time however do not



support single pass blending of as many as 8 textures. You can check the MaxSimultaneousTextures member of the D3DCAPS9 structure returned from the IDirect3D9::GetDeviceCaps function to inquire about a device's texture blending capabilities. Although many 3D graphics cards will only support 2 to 4 textures being blended simultaneously, this does not limit the ability to store 8 texture coordinates in a single vertex. This is because you may wish to store the texture coordinates in the vertex and render the polygon several times using different sets.

So we will specify one of the D3DFVF\_TEX0 through D3DFVF\_TEX8 flags to tell the pipeline how many sets of texture coordinates the vertex includes. As mentioned earlier, we can set different textures in different texture stages for blending onto a single polygon. Thus we can store multiple sets of textures coordinates in a vertex so that the polygon can map its vertices to separate independent regions of the textures being blending. Lab Project 6.1 uses only a single texture with a single pair of texture coordinates per vertex.

### Untransformed, Pre-Lit Vertex with Texture Coordinates

```
#define PreLitVertex D3DFVF_XYZ | D3DFVF_TEX1
```

When we set this vertex format, the device knows to expect a vertex with a position that needs to be transformed and a pair of UV coordinates. Our vertex structure would look something like this:

```
struct MyTexVertex
{
    float x; float y; float z; // Model/World space position
    float u;           // U Texture Coordinate
    float v;           // V Texture Coordinate
};
```

We would use this vertex type with lighting disabled. If no diffuse or specular color is specified in the vertex when lighting is disabled, the pipeline treats the vertex as if it has bright white diffuse and specular colors (0xFFFFFFFF). We will discuss in detail how to enable blending between the texture color and the polygon color later in the lesson.

In this next example, we create another pre-lit vertex with texture coordinates. We use a vertex with a diffuse color (instead of relying on a default white diffuse color) and a set of texture coordinates so that the interpolated texel and diffuse colors can be blended together.

```
#define PreLitVertex2 D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1

struct MyPreLitVertex2
{
    float x; float y; float z; // Object/World space position
    DWORD diffuse;           // Pre-lit diffuse color of vertex
    float u;           // U Texture Coordinate
    float v;           // V Texture Coordinate
};
```



## Untransformed, Unlit Vertex with Texture Coordinates

In this next example we see a vertex structure with texture coordinates designed to work with the DirectX lighting pipeline (includes a vertex normal).

```
#define UnLitVertex D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1

struct MyUnlitVertex
{
    float x; float y; float z; // Object/World space position
    D3DXVECTOR Normal;        // Normal used for lighting
    float u;                  // U Texture Coordinate
    float v;                  // V Texture Coordinate
};
```

The lighting pipeline calculates the per-vertex color (Chapter 5) and the vertex colors are interpolated to provide a per-pixel diffuse color. We can instruct the pipeline to blend this color with the texel color in the texture for a given pixel. If a vertex is outside the range of all lights, it will have a black color interpolated across the surface, thus darkening the texels mapped to the polygon.

In the next example we see an unlit vertex with a diffuse vertex color that could be used as a material reflectance property (Chapter 5). You might use a structure like this if you wanted each vertex in the object to have a per-vertex emissive property (instead of per-face).

```
#define UnLitVertex D3DFVF_XYZ| D3DFVF_NORMAL | D3DFVF_DIFFUSE | D3DFVF_TEX1

struct MyUnlitVertex
{
    float x; float y; float z;      // Object/World space position
    D3DXVECTOR Normal;            // Normal used for lighting
    DWORD Diffuse;               // Diffuse Color
    float u;                     // U Texture Coordinate
    float v;                     // V Texture Coordinate
};
```

**Note:** To be technically correct we should refer to a pixel that has not yet been rendered in the frame buffer as a fragment (or color fragment). This is because a pixel is used to describe a displayed point on the monitor screen and any fragment passing through the pipeline may be rejected by a depth test or some other test. Therefore, a fragment can be thought of as a *potential* pixel because it is a color that will be plotted on the screen as a pixel only if it does not get rejected at some point in the pipeline.



## Sampler States

IDirect3DDevice9::SetSamplerState configures the way the device samples texels during rendering. Sampler states are analogous to render states. When a sampler state is set, it remains set until it is either unset or changed to some other state.

```
HRESULT SetSamplerState
(
    DWORD Stage,
    D3DSAMPLERSTATETYPE Type,
    DWORD Value
);
```

Sampler states can be used to control what happens when the U or V coordinate of a vertex is outside the [0.0, 1.0] range. They can also be used to modify the way the device maps texels to pixels. Other settings are possible and we will see such examples as the lesson progresses.

### DWORD Stage

This parameter is the zero-based integer index of the texture stage that we are setting the sampler state for.

### D3DSAMPLERSTATETYPE Type

This is the sampler state that we would like to set or change the property for. We specify one of the **D3DSAMPLERSTATETYPE** enumerated types defined below.

```
typedef enum _D3DSAMPLERSTATETYPE
{
    D3DSAMP_ADDRESSU = 1,
    D3DSAMP_ADDRESSV = 2,
    D3DSAMP_ADDRESSW = 3,
    D3DSAMP_BORDERCOLOR = 4,
    D3DSAMP_MAGFILTER = 5,
    D3DSAMP_MINFILTER = 6,
    D3DSAMP_MIPFILTER = 7,
    D3DSAMP_MIPMAPLODBIAS = 8,
    D3DSAMP_MAXMIPLEVEL = 9,
    D3DSAMP_MAXANISOTROPY = 10,
    D3DSAMP_SRGBTEXTURE = 11,
    D3DSAMP_ELEMENTINDEX = 12,
    D3DSAMP_DMAPOFFSET = 13,
    D3DSAMP_FORCE_DWORD = 0x7fffffff
} D3DSAMPLERSTATETYPE;
```

### DWORD Value

This value is interpreted based on the sampler state being set. For one state this might contain a **D3DCOLOR** while another may use the value to determine which MIP level is used for texturing. We will see more alternatives for this parameter as we examine various sampler states.



As one quick example, the following code could be used to limit stage 0 to only use the first three MIP maps (MIP levels 0, 1, and 2 -- which are the largest) in the chain when rendering. Even if the texture in stage 0 had 16 MIP levels, levels 3 through 15 would not be used, even when the polygon being rendered is very small. While this is not something you will usually want to do because it could cause aliasing artifacts, it does show us how to set a sampler state:

```
m_pDevice->SetSamplerState(0, D3DSAMP_MAXMIPLEVEL, 2);
```

Like the SetRenderState function, this function has a partner function to retrieve the current sampler states for a given stage on the device:

```
HRESULT GetSamplerState
(
    DWORD Sampler,
    D3DSAMPLERSTATETYPE Type,
    DWORD* pValue
);
```

### Texture Addressing Modes

The job of fetching the texel from the current texture using the UV coordinate pair belongs to the sampler unit. How the sampler interprets texture coordinates outside the [0.0, 1.0] range depends on the texture addressing algorithm used. We will look at these algorithms in a moment.

The sampler states we set or modify to change the addressing mode are **D3DSAMP\_ADDRESSU** and **D3DSAMP\_ADDRESSV** for the U and V coordinates respectively. The value passed will be a member of the **D3DTTEXTUREADDRESS** enumerated type:

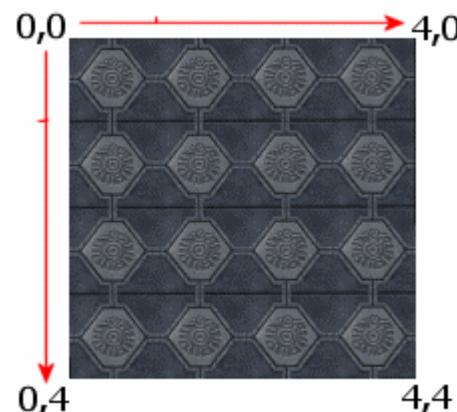
```
typedef enum _D3DTTEXTUREADDRESS
{
    D3DTADDRESS_WRAP = 1,
    D3DTADDRESS_MIRROR = 2,
    D3DTADDRESS_CLAMP = 3,
    D3DTADDRESS_BORDER = 4,
    D3DTADDRESS_MIRRORONCE = 5,
    D3DTADDRESS_FORCE_DWORD = 0x7fffffff
} D3DTTEXTUREADDRESS;
```

### Wrapping (D3DTADDRESS\_WRAP)

This is the default addressing mode used for both the U and V texture coordinates when either (or both) is outside the [0, 1] range. UV coordinates outside the range cause the texture to be tiled across the surface.

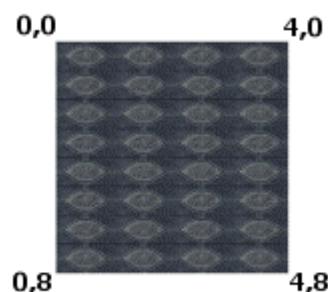
## The Texture

The Texture is repeated over the quad surface using WRAP address mode.



**Figure 6.13**

Wrapping works simply by taking the interpolated texture coordinate and dropping the integer term. This can be affected in the U or V direction separately (or both as in Fig 6.13). When texture coordinates are negative, the tiling would still work in the same way, only in the opposite direction. The GILEST™ level editor uses this addressing mode to allow you to scale and tile your textures across faces. In Fig 6.13, if we were to change the bottom left coordinate to (0,8) and the bottom right to (4,8), the textures would tile four times across the surface and eight times down as seen in Fig 6.14.



**Figure 6.14**

You will normally want to use textures that will tile without visible seams. For example, making your own textures with a digital camera is easy to do, but generally they do not tile properly and will require some touching up in a photo editing package.

### Setting WRAP addressing mode for both U and V

```
m_pDevice->SetSamplerState( 0 , D3DSAMP_ADDRESSU , D3DADDRESS_WRAP );
m_pDevice->SetSamplerState( 0 , D3DSAMP_ADDRESSV , D3DADDRESS_WRAP );
```



## Mirroring (**D3DTADDRESS\_MIRROR**)

When we set either the U or V address modes to **D3DTADDRESS\_MIRROR**, any coordinates outside the [0, 1] range are tiled much like **D3DTADDRESS\_WRAP** except that every time the texture repeats along that axis, the coordinates are flipped. The best way to understand this is to see it in action. In Fig 6.15 we have a texture of the planet Earth mapped to a quad with UV coordinates in the range [0.0, 2.0]. The texture is tiled as in the previous mode but this time it is mirrored as it is repeated.

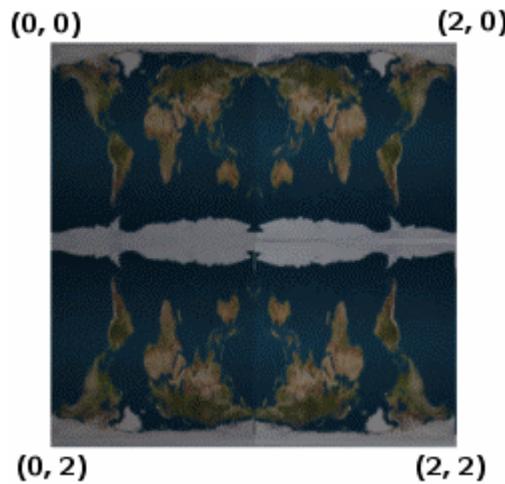


Figure 6.15

The flipping of the image happens at the texture boundary. If Fig 6.15 had coordinates in the 0.0 to 4.0 range, it would be repeated 4 times along the U and V axes with the 2<sup>nd</sup> and 4<sup>th</sup> tiles mirrored and the 1<sup>st</sup> and 3<sup>rd</sup> tiles the same as the original source image.

Sometimes using mirror mode can help break up repeating patterns when applying a texture over a large area. This makes the results appear somewhat more random to the viewer

### Setting MIRROR addressing mode for both U and V

```
m_pDevice->SetSamplerState( 0 , D3DSAMP_ADDRESSU , D3DTADDRESS_MIRROR );
m_pDevice->SetSamplerState( 0 , D3DSAMP_ADDRESSV , D3DTADDRESS_MIRROR );
```

## Bordering (**D3DTADDRESS\_BORDER**)

Unlike other texture addressing modes which involve a single state change per axis to set that mode, border addressing mode requires that we set an additional sampler state. This second state will be a color that to be used to generate a border beyond the [0, 1] range. When a pixel maps to a texture coordinate outside this range, the sampler returns the border color. In the following example, we set the border color to opaque red using the **D3DSAMP\_BORDERCOLOR** sampler state and then map



the texture to a quad so that some of its pixels fall outside the  $[0, 1]$  range. We can see that these pixels are colored red (Fig 6.16).

#### Setting BORDER addressing mode and color for both U and V

```
m_pDevice->SetSamplerState( 0 , D3DSAMP_ADDRESSU , D3DADDRESS_BORDER );
m_pDevice->SetSamplerState( 0 , D3DSAMP_ADDRESSV , D3DADDRESS_BORDER );
m_pDevice->SetSamplerState( 0 , D3DSAMP_BORDERCOLOR , 0xFFFF0000 );
```

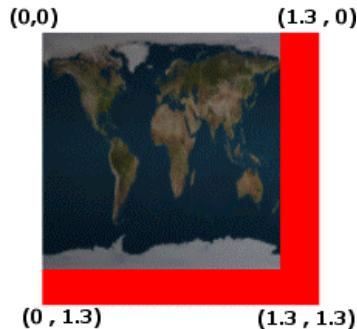


Figure 6.16

You can use this mode to make sure that only one copy of the texture is assigned to each polygon rendered with it.

#### **Clamping (D3DTADDRESS\_CLAMP)**

Clamping (like the border address mode) is also useful when you want only one copy of the texture to appear on a polygon. U coordinates for pixels outside the 0.0 to 1.0 range are clamped to the color of the last (or first if  $U < 0$ ) texel color in the given row. V coordinates outside the range are clamped to the last (or first if  $V < 0$ ) texel in the given column (Fig 6.17).

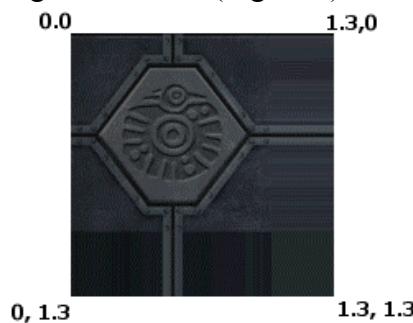


Figure 6.17

#### **Mirror Once (D3DTADDRESS\_MIRRORONCE)**

This is analogous to using D3DTADDRESS\_MIRROR and D3DTADDRESS\_CLAMP addressing modes. It takes the absolute value of the texture coordinate (thus, mirroring around 0), and then clamps it to the maximum value.

## Texture Coordinate Wrapping

In addition to the sampler being assigned a texture addressing mode, we can also assign each set of texture coordinates a wrapping mode. This is often confused with the D3DTADDRESS\_WRAP texture addressing mode described above, but texture wrapping modes are quite different.

To understand wrapping modes we first have to understand how texture coordinates are interpolated when wrapping is disabled. The following diagram shows how two vertices belonging to an edge of a triangle may be mapped to texels in the current texture being used. As you would expect, the interpolated per-pixel UV coordinates of the edge step across the texture and maintain the rule that higher U values are to the right of lower U values and higher V values are below lower V values. The texels along the edge of the line are returned to the renderer by the sampler unit for each fragment along the edge of the polygon.



**Figure 6.18**

When wrapping is enabled along either the U or V axis of the texture coordinate system, we interpolate along the edge formed by the per-vertex texture coordinates using the shortest distance between the two coordinates along the given axis.

In Fig 6.19, we see an edge that will be interpolated between the same two points when wrapping is enabled along the U axis. Notice that it takes the shortest distance along the U axis by wrapping off the left hand side of the texture and back again onto the right hand side.



**Figure 6.19**

Notice that only U wrapping is enabled in Fig 6.19. The V interpolation still carries on vertically down the texture as usual. The edge is only wrapped if the length of the edge will be shorter by doing so. Otherwise, the edge will be interpolated across the face of the texture in the normal fashion.

We can also enable texture wrapping along the V axis in the same way. The difference here is that if the edge is vertically shorter by wrapping off the top/bottom of the texture, then this approach is used (Fig 6.20).

**V Wrapping Enabled**



**Figure 6.20**

Note that the U edge is not wrapped and steps along the edge of the texture from left to right. The V coordinate is wrapped and the edge is shorter vertically if the edge is wrapped off the top edge of the texture and up through the bottom.

Finally, we can enable texture wrapping on both the U and V axes of the texture coordinate system which, in Fig 6.21, causes the edge to be wrapped both horizontally and vertically:

**UV Wrapping Enabled**



**Figure 6.21**

Wrapping can be very useful when generating texture coordinates to wrap a texture around a sphere or cylinder (Fig 6.22).

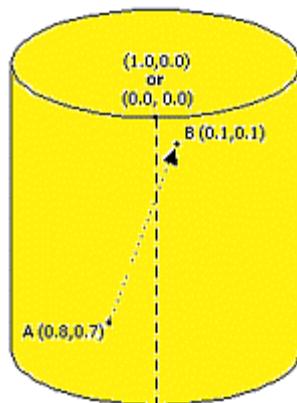


Figure 6.22

Because this wrapping is really a manipulation of the interpolation of per-vertex texture coordinates, it is performed by the renderer and not in the texture stage sampler unit. Therefore, this kind of wrapping is actually a render state and not a sampler state.

Wrapping is not enabled on a per-texture state basis, but rather on sets of texture coordinates within the vertices. Earlier we discussed how a vertex may have more than one set of texture coordinates so that they can be used to address multiple textures (or into the same texture more than once). If we enable texture wrapping for texture coordinate 3 for example, then all vertices that are rendered with three sets of texture coordinates (or more) will have their 3<sup>rd</sup> set wrapped by the renderer when they are used to access the texture into which they are indexing. This will make more sense later in the lesson when we cover multiple texture blending and multiple texture coordinate sets.

To enable wrapping we will use the `IDirect3DDevice9::SetRenderState` function. The `D3DRS_WRAP0 - D3DRS_WRAP15` render state types will enable wrapping for texture coordinate sets 0-15 respectively. The second parameter specifies the axis we wish to enable wrapping for (`D3DWRAPCOORD_0` or `D3DWRAPCOORD_1` for U and V respectively and `D3DWRAPCOORD_2` and `D3DWRAPCOORD_3` for wrapping 3D or 4D texture coordinates).

The following example shows how to enable wrapping on the V axis for the 1<sup>st</sup> texture coordinate set stored at a vertex.

```
d3dDevice->SetRenderState(D3DRS_WRAP0, D3DWRAPCOORD_1);
```

The next example shows how to enable texture wrapping for the 3<sup>rd</sup> set of texture coordinates (for vertices that have them) on the U axis.

```
d3dDevice->SetRenderState(D3DRS_WRAP2, D3DWRAPCOORD_0);
```

## Texture Filtering

Texture filtering improves the visual quality of our rendered image. Earlier we saw that using MIP maps can help reduce aliasing artifacts that occur when textures are mapped to polygons that are much smaller or larger than the dimensions of the texture.

The process of mapping a texture to a polygon such that it has to be reduced (use fewer texels) is referred to a **minification**. When a texture image is reduced in size, ideally each pixel rendered would be sampled using a weighted average of all texels in the texture image. However this would be far too expensive to be done in real-time and this is where MIP maps can really help. MIP maps provide pre-generated images that are scaled down using this exact filtering technique (by default). Nevertheless the MIP map downsampling is only available at discrete intervals. We can still suffer minification artifacts when a polygon is at a distance that places it between two MIP levels.

As discussed earlier, minification is only half the story. When the polygon is very close to the viewer, it occupies many more pixels on the screen than are in the texture. This process is called **magnification** -- where many pixels are mapped to the same texel. Magnification gives a blocky result as shown in Fig 6.23.

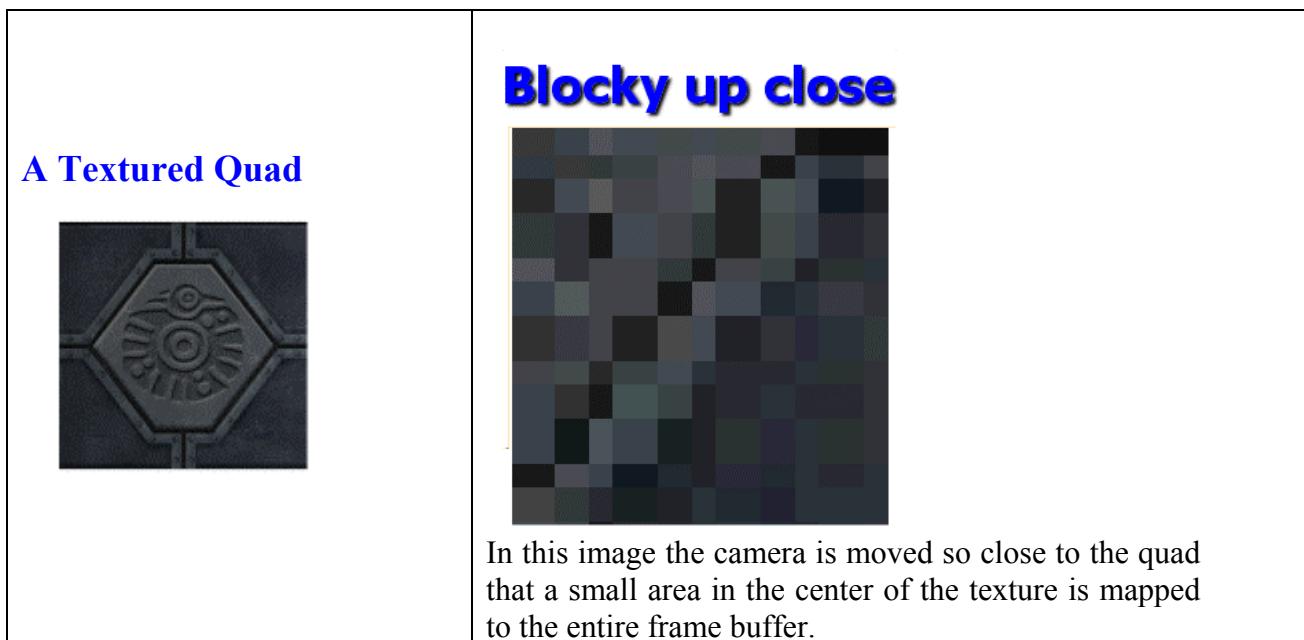


Figure 6.23

DirectX 9 supports several filters that can be used independently for both minification and magnification artifacts. These filters affect the way a floating point per-pixel UV coordinate is used to sample the texel in the texture. Forgetting about MIP maps for the time being, we will describe the filtering techniques from the perspective of using a one level texture (non-MIP mapped). Then we will



discuss how the minification and magnification filters can be used with MIP maps to provide additional image quality.

If you are an avid gamer or games programmer you have probably heard terms like bilinear filtering, trilinear filtering, and anisotropic filtering. Using sampler states to set the minification, magnification and MIP map filters is how we get our applications to use these well-known filtering techniques. If a texture has to be scaled down to fit the polygon being rendered then the minification filter is used to fetch the correct texel from the texture for the corresponding pixel. If the texture has to be magnified to fill the on-screen region then the magnification filter is used. (Often you will use the same filtering technique for both.)

## Magnification Filters & Minification Filters

There are two sampler states that we can set (using SetSamplerState) to independently set the filtering technique used for magnification and/or minification.

```
m_pDevice->SetSamplerState( stage , D3DSAMP_MINFILTER , D3DTEXTUREFILTERTYPE );
m_pDevice->SetSamplerState( stage , D3DSAMP_MAXFILTER , D3DTEXTUREFILTERTYPE );
```

In the above code, *stage* should be set to the texture stage for which you wish to set the filter. If we have multiple textures being used (in different stages) we can tell the device to use a different filtering technique for each stage. The third parameter must be set to one of the **D3DTEXTUREFILTERTYPE** enumerated type members:

```
typedef enum _D3DTEXTUREFILTERTYPE
{
    D3DTEXF_NONE = 0,
    D3DTEXF_POINT = 1,
    D3DTEXF_LINEAR = 2,
    D3DTEXF_ANISOTROPIC = 3,
    D3DTEXF_PYRAMIDALQUAD = 6,
    D3DTEXF_GAUSSIANQUAD = 7,
    D3DTEXF_FORCE_DWORD = 0x7fffffff
} D3DTEXTUREFILTERTYPE;
```

The enumeration is used to set the minification and magnification types as well as the MIP filter type which we will explain in a moment. Because of this, not all types are valid for all three sampler states.

### No Filtering (D3DTEXF\_NONE)

This is only a valid filter type when setting a MIP filter. It should not be used with minification and magnification filters. When tested with minification and magnification filters the results can differ across hardware and drivers. Some hardware defaults to **D3DTEXF\_POINT** whilst others default to **D3DTEXF\_LINEAR**. As **D3DTEXF\_POINT** is basically no filtering at all, you should use point filters to disable filtering as seen next.



## Point Filtering (D3DTEXF\_POINT)

This is the default minification and magnification filter for all texture states. Point filtering essentially equates to no filtering at all; the per-pixel UV coordinate is truncated to an integer value used to index the correct texel in the texture. For example, if a pixel's UV coordinates were (10.2, 15.3) then this would be snapped to the integer set (10, 15).

**Note:** A UV coordinate is not simply snapped to an integer coordinate. It is first mapped from texture coordinate space into an image space UV coordinate using the following formula:

$$U = u \times \text{ImageWidth} - 0.5$$

$$V = v \times \text{Image Height} - 0.5$$

Where  $u$  and  $v$  are the floating point texture coordinates and  $U$  and  $V$  are floating point coordinates mapped into image space but not yet snapped to an integer texel coordinate. Therefore, in the above paragraph and in future discussions in this section, when we talk about the UV coordinates being snapped to an integer, we are referring to the coordinates after they have been transformed into image space using the above equation. It is this snapped coordinate that is then considered the true integer coordinate of the nearest texel in the texture.

When the texture is magnified, many pixels in the polygon being rendered will be snapped to the same integer texel and therefore will have exactly the same color. This is what causes the blocky appearance in Fig 6.23. When the texture is minified, this filtering technique leads to texels in the source image being skipped. Since the skipped texels may have been important contributors to the integrity of the image, aliasing artifacts occur as discussed earlier.

## Bilinear Filtering (D3DTEXF\_LINEAR)

After the integer UV coordinate is found, the color of the corresponding texel and the four neighboring texels are combined together to create the final color returned from the sampler. The amount of weight that each texel contributes to the final color is based on the distance from that integer texel coordinate to the ideal float point UV coordinate (Fig 6.24).



## Bilinear Filtering

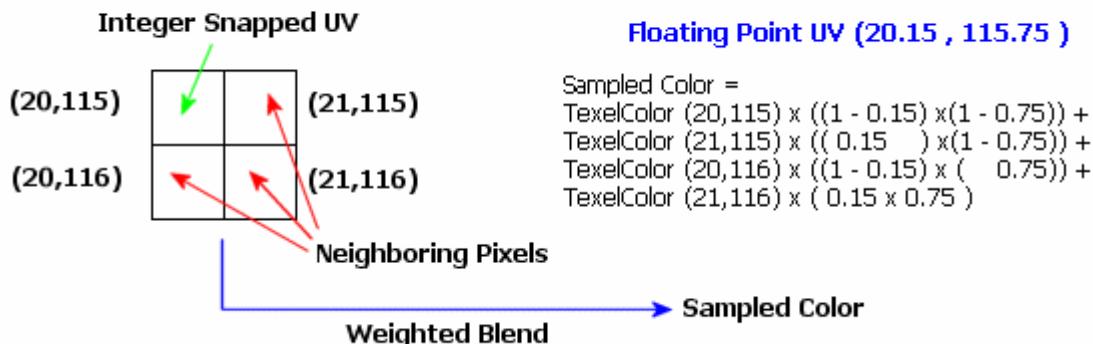


Figure 6.24

In Fig 6.24 an image space floating point UV coordinate of (20.15, 115.75) is passed to the sampler unit from the rasterizer. The ideal image space coordinate is somewhere between the four texels. The offset between the ideal UV image space coordinate and each integer texel coordinate is used to scale the contribution of that texel to the final color. In this case, the color returned will be a blend between the colors found at integer texture coordinates (20,115), (21,115), (20,116) and (21,116).

Using this filter (especially during magnification) means that abrupt color changes in the surface caused by the snapping of floating point UV coordinates to a single integer texture coordinate are smoothed (Fig 6.25).

## Point Filtering



## Bilinear Filtering

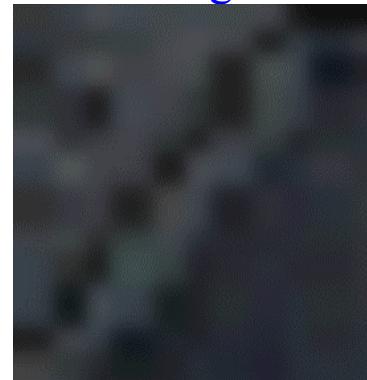


Figure 6.25

When used for minification, this helps alleviate the problems caused by skipping pixels in the source image. Pixels that would otherwise be completely skipped with point filtering now contribute to the neighboring pixel color. Bilinear minification works exceptionally well when used alongside MIP

maps because the smaller images sampled at far distances means that there is a lower chance of any texel in the source image not contributing to the screen representation of the image in some way.

## Anisotropic Filtering (D3DTEXF\_ANISOTROPIC)

Anisotropic means ‘no equal shape’. When anisotropic filtering is enabled, the shape of the filter in texture space and the number of texels sampled take polygon orientation into account. To understand the need for anisotropic filtering we must first understand the problems associated with using bilinear filtering (or trilinear filtering discussed later). Bilinear filtering is done using a 2x2 square sampling grid. Using a square grid is fine when the plane of the polygon is oriented directly with the view plane, but as the orientation of the polygon changes with respect to the viewer, this perspective should be taken into account when choosing the shape of the filter used for sampling the texture map.

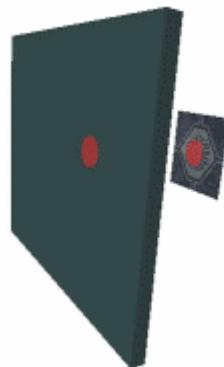
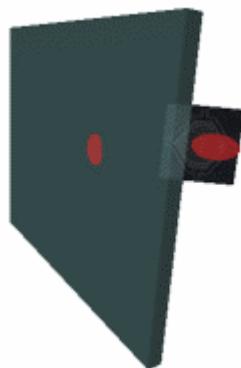


Figure 6.26

In Fig 6.26, the gray slab represents the monitor screen and the textured quad behind it is directly facing the viewer. In this example we have used a circular spot to indicate a region of pixels on the screen that would be mapped back into texture space. We can see that the shapes are the same, as they should be. Imagine that the red circle on the screen is a spotlight shining on the screen which is then projected onto the texture. When the polygon is facing the viewer, using a box shaped filter to sample textures is ideal, but look at what happens to our spotlight projection when the polygon is oriented away from the viewer (Fig 6.27).



**Figure 6.27**

The spotlight shining through the screen onto the quad is now elongated because of perspective. Using this as an analogy, we can see that when the texture/viewer angle is large, a box shaped filter no longer describes the correct region in the texture to sample. When the angle is large, many more texel colors should be taken into account to produce the final perspective correct samples for each pixel. Because of this, when using bilinear filtering, polygons at large angles with respect to the view plane can appear exceptionally blurry. Anisotropic filters use this perspective concept to provide a more accurate filter shape when sampling (much more like the projected shape of our spotlight).

More recent 3D cards support anisotropic filtering in hardware. Some earlier cards (the TNT™ generation) did support some level of anisotropic filtering but it typically resulted in significant performance degradation. Typically a graphics card that supports anisotropic filtering will support multiple levels of anisotropy between 1 and *MaxAnisotropy*. *MaxAnisotropy* can be found by querying the device capabilities and examining the **D3DCAPS9::MaxAnisotropy** member. This is typically between 1 and 16 but can be higher on the latest graphics hardware. A level of 1 will provide no visual improvement whilst the higher levels will provide better results (at a higher performance cost). Additionally, some cards only support anisotropy for specific filters. For example, the early models of the nVidia GForce3™ supported only anisotropic filtering as a minification filter.

Although it can be expensive on older cards, anisotropic filtering looks about as good as we can expect things to get at this time. There are other filters in the **D3DTEXTUREFILBERTYPE** enumerated type not mentioned above, but at the time of this writing they are unsupported in hardware and are terribly slow in software. Therefore, the applications in this chapter will use anisotropic filtering as the top level filtering technique that you can enable. If you would like more information on the **D3DTEXF\_PYRAMIDALQUAD** and **D3DTEXF\_GAUSSIANQUAD** filters, please consult the DirectX 9 SDK documentation.

## Setting Minification and Magnification Filters

The following code shows us how to set the minification filter to sample texels using bilinear filtering and the magnification filter to sample using an anisotropic filter of the highest level supported by the graphics hardware. These settings are applied to texture stage 0.



```
// Get the devices maximum supported anisotropy level
D3DCAPS9 caps;
m_pDevice->GetDeviceCaps ( &caps );
DWORD MaxLevel = caps.MaxAnisotropy;

// Set Sampler filters in State0
m_pDevice->SetSamplerState ( 0, D3DSAMP_MINFILTER ,D3DTEXF_LINEAR );
m_pDevice->SetSamplerState ( 0, D3DSAMP_MAXFILTER ,D3DTEXF_ANISOTROPIC);
m_pDevice->SetSamplerState ( 0, D3DSAMP_MAXANISOTROPY, MaxLevel );
```

It is important to realize that these filters are applied in conjunction with MIP mapping when it is enabled. Minification and magnification filters will be applied to the current MIP level being used for rendering. This means that if we have a polygon in the distance such that its closest match in size is MIP level 8 in the texture surface, the minification and magnification filters will be applied to MIP surface 8 to sample the color.

## Enabling MIP maps

We talked earlier about how MIP maps can be used to provide pre-filtered images of a texture at different size resolutions. They also provide an even greater benefit when a given MIP level is filtered using the minification and magnification filters discussed above. Not only do we have a pre-filtered image of approximately the size of the rendered polygon, but the slight aliasing that would occur between the closest-match MIP level sizes will further be reduced using bilinear or anisotropic filtering when sampling the MIP level.

It is worth noting that the correct MIP map is chosen for a given polygon at an arbitrary distance partly based on the MIP filter selected. This is in addition to the minification and magnification filters for MIP surface texels. To set the MIP filter, we use the SetSamplerState function with the following sampler state:

```
m_pDevice->SetSamplerState(stage, D3DSAMP_MIPFILTER , D3DTEXTUREFILTERTYPE);
```

We pass in the stage which we are setting the filter for, along with a member of the **D3DTEXTUREFILTERTYPE** enumerated type (whose members we used to set the minification and magnification filters in the last section). Let us briefly discuss how these filters affect the selection of the MIP level and ultimately the color sent back from the sampler.

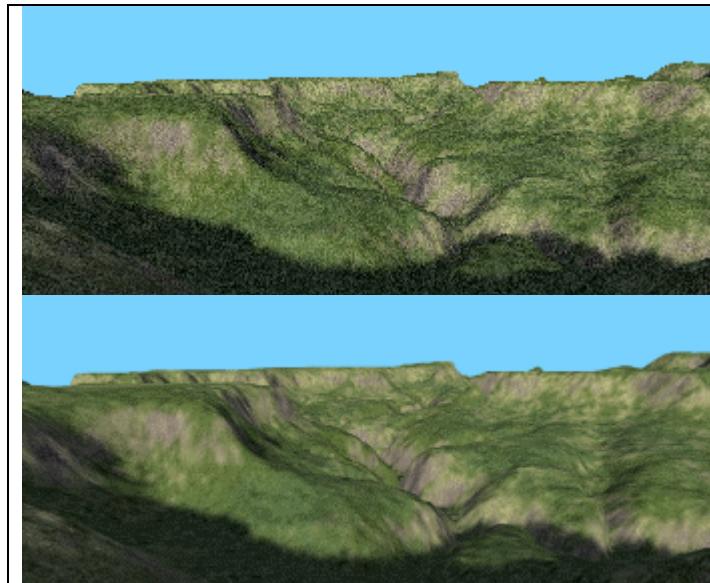
### **D3DTEXF\_NONE**

When this is the MIP filter, the MIP mapping mechanism for the texture stage is disabled. Whether the texture has MIP levels or not, the top level surface will always be used. This can cause aliasing artifacts, even with the minification filter set to bilinear or anisotropic.

### **D3DTEXF\_POINT**

If the texture includes MIP levels, then the correct MIP map is selected to fetch texels using the currently active minification and magnification filters. In Fig 6.28 both terrains use bilinear filtering for minification and magnification filters. In the bottom image, MIP map textures are used with the

**D3DTEXF\_POINT** MIP filter state. As you can see, even with the minification and magnification filters turned on, the top image, which does not use MIP maps, still suffers aliasing artifacts. It looks even worse when the camera is moving because the pixels in the distance noticeably shimmer.



**Figure 6.28**

In the bottom image in Fig 6.28, the distant hill polygons do not suffer significant aliasing because a higher level (lower resolution) MIP map surface is being used for sampling. As the polygons get progressively further from the viewer, the correct MIP map is used that most closely matches the ideal size of a 1:1 pixel-to-texel ratio. Because these MIP maps have been pre-filtered, they look much better than the top image, which is simply downsampled from the maximum texture size for each polygon.

**D3DTEXF\_POINT** is adequate for sampling from the closest ideal MIP map, but it is far from perfect. Because the number of MIP values is so low in comparison to the number of possible distance values, the MIP map generally selected for sampling will only be a closest match. Let us say for example that a polygon is at a distance such that its ideal MIP level is 1.2 (i.e. 20% between levels 1 and 2). With a point MIP filter, this fraction will not be taken into account and level 1 would be used exclusively. When walking closer to a wall you might even see its texture change as it is snapped up to the next MIP level. Likewise, when walking away from the wall you will see its MIP level visibly switch as distance increases.

### **D3DTEXF\_LINEAR**

When we enable linear MIP filtering we are performing a sampling between the two closest MIP levels. In Fig 6.29 we can see that the ideal MIP level of 1.2 places the pixel between MIP levels 1 and 2. The texel is sampled from both MIP maps using the bilinear minification and magnification filters

on each, and then the resulting colors from both MIP levels are blended together based on the distance from the ideal MIP level. In this particular case, the final color contains 80% of the level 1 sampled color and 20% of the level 2 color. This is called **trilinear MIP map interpolation** or simply, **trilinear filtering**.

### Trilinear MIP Map Interpolation

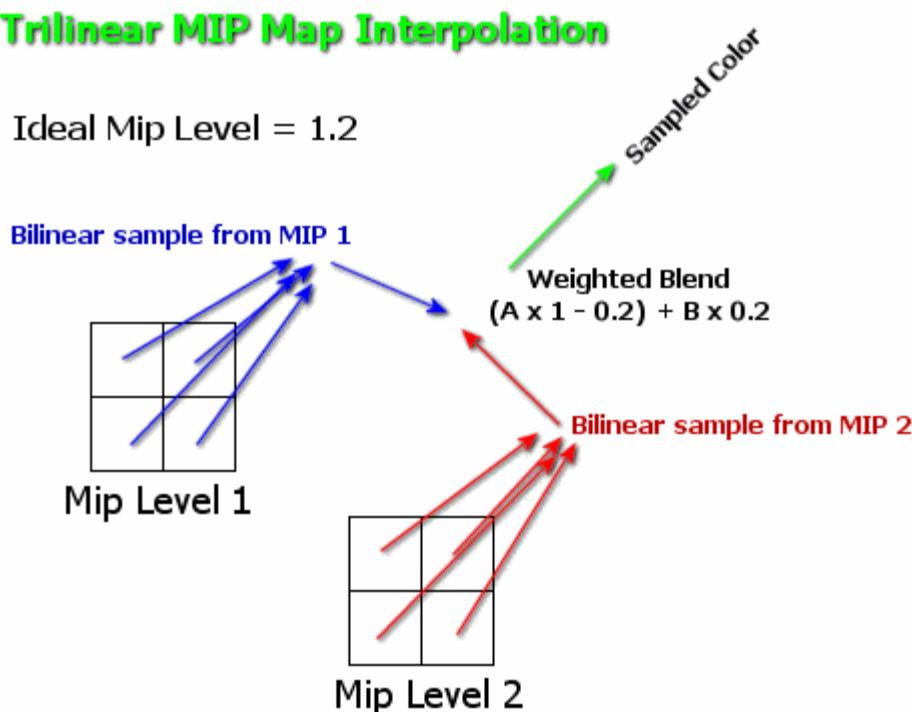


Figure 6.29

Fig 6.29 could have used anisotropic filtering for magnification and/or minification to further increase visual quality. Alternatively a point filter could have been used to extract the nearest texel from each level and blend them together.

When we enable trilinear MIP filtering, we no longer see transitions between MIP levels when the viewer moves through the world. The two MIP levels are taken into account and gradually interpolate from one to the other, making for a smoother transition.

#### D3DTEXF\_ANISOTROPIC

This is not a valid MIP filter.



## Texture Stages

The texture stages form the core of the DirectX blending cascade. The cascade determines the color and opacity of a fragment as it passes through the pipeline on its way to becoming a pixel in the frame buffer. When we call the SetTexture function, we specify a texture stage that the given texture will be assigned to. Texture stages are fed texel colors from the sampler unit. The sampler determines how texels are sampled from the texture bound to that stage. DirectX 9 has eight texture stages, so theoretically eight textures can be stored and subsequently blended together in a single pass. In practice however, most hardware supports somewhere between two and four stages -- although the most recent cards support eight or more. When using only one texture at a time, we will bind that texture to stage 0.

## Texture Color

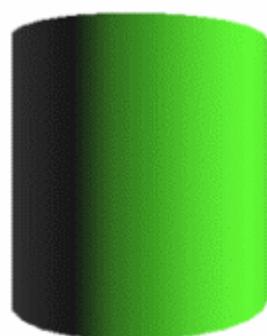
If we have a texture in stage 0 and our vertices include texture coordinates, then the sampler unit will retrieve the color for each pixel in the polygon from the corresponding texel in the texture. This color fragment is forwarded to the texture stage where it can be used as an argument to a blending function to produce the final color. In previous lessons we saw how to store vertex colors and interpolate them across each pixel in a polygon. Once the interpolated color for each pixel is determined, it can be sent to a texture stage.

**Note:** If lighting is enabled then we do not calculate the vertex colors ourselves since the lighting calculations in the pipeline will determine those values. From that point on, vertex colors calculated by the pipeline are interpolated to generate a per-pixel color. This color can be sent to a texture stage.

Texture colors and vertex colors are not mutually exclusive. When a green light is placed near a vertex for example, the vertex will have a green color generated for it (assuming proper material conditions). This ultimately leads to the pixel colors generated via interpolation being green as well. The cylinder on the right has a green light shining on it. If we were unable to use this per-pixel color information when texturing was enabled, we would effectively lose the lighting pipeline completely.



The cylinder to the left, while textured, lacks shading. A world without shading would look very flat and boring indeed.



As it happens, when texturing is enabled, every pixel in the polygon being rendered has both a texel color sampled for it and a diffuse and a specular color generated for it. These colors are passed to the texture stages where we can choose how we wish to blend them together.

In the next cylinder image we see the result of modulating the texel color sampled for each pixel with the diffuse color generated for each pixel. Notice how the brick wall texture detail can still be seen, but that the cylinder is also affected by the green light nearby.



Every polygon pixel is sent through the texture blending cascade. In fact, it is perhaps more precise to say that the possible pixel ingredients are sent to the stages. By this we mean that the texture stages can be sent the diffuse and specular colors, the texel color, and even a constant color to construct the final pixel value.

## Setting Texture Stage State

Our application can configure the behavior of a texture stage through state settings. The function that controls this is called `SetTextureStageState` and is a member of the `IDirect3DDevice9` interface.

```
HRESULT IDirect3DDevice9::SetTextureStageState
(
    DWORD Stage,
    D3DTEXTURESTAGESTATETYPE Type,
    DWORD Value
);
```

Just as `IDirect3DDevice9::SetRenderState` and `IDirect3DDevice9::SetSamplerState` can be used to set render states and sampler states respectively, the `SetTextureStageState` function can be used to set the states of any textures stage. We simply pass in the stage we wish the state change to apply to, the state type we wish to set, and a state specific value.

**Note:** We do not bind a texture to a stage using the `SetTextureStageState` function. There is a separate function for binding textures to stages called `SetTexture`.

Fig 6.30 shows what a texture stage looks like internally.

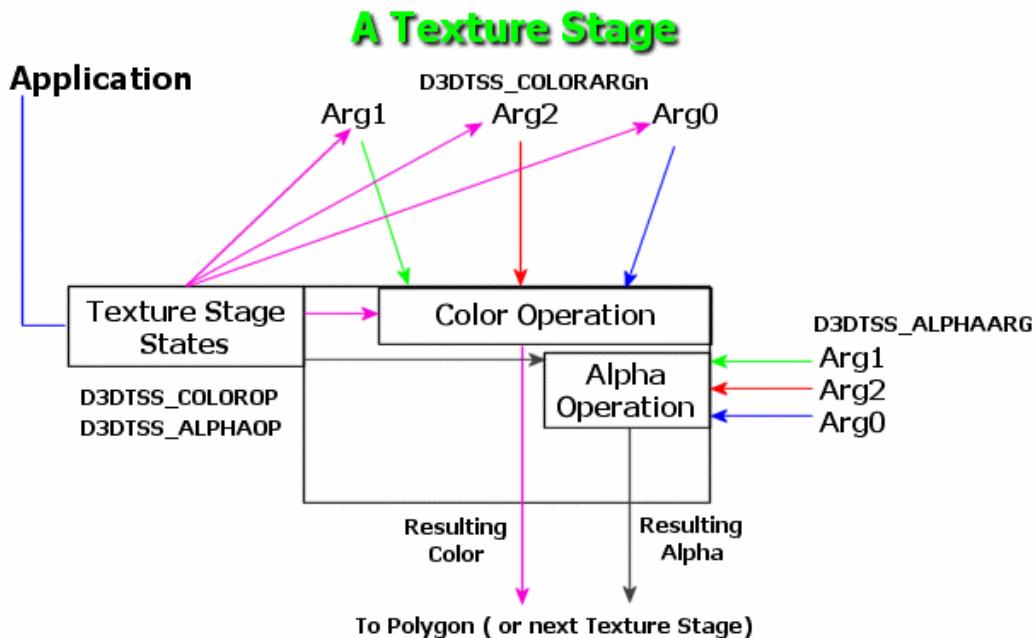


Figure 6.30

Once the ARGB color has been calculated for each pixel and the texture has been sampled, the color and alpha components are separated (Fig 6.30). The texture stages actually use two pipelines so that RGB and A can be processed separately. The Color unit and the Alpha unit are each fed three inputs called Arg0, Arg1, and Arg2. Our application will configure which information is routed through these inputs. We could decide for example, that Arg0 should be the diffuse pixel color and Arg1 should be the sampled texel color. We can then set the color operation unit to multiply these two colors together. The final result is the output for that texture stage. In the simplest case, using only one stage and no alpha blending, the output of the texture stage becomes the final fragment color that will become a pixel if it passes the depth test. The alpha unit will also have its own arguments and operations (Chapter 7).

## Texture Stage States

Let us now examine the different states that can be set using this function. We will first look at a list of all available states with a brief description of each. This will be followed by much closer examination of the states that we are interested in for this lesson. The rest of the texture stage states will be covered throughout the remainder of this course and in the next course in this series.



```
typedef enum _D3DTEXTURESTAGESTATETYPE
{
    D3DTSS_COLOROP = 1,
    D3DTSS_COLORARG1 = 2,
    D3DTSS_COLORARG2 = 3,
    D3DTSS_ALPHAOP = 4,
    D3DTSS_ALPHAARG1 = 5,
    D3DTSS_ALPHAARG2 = 6,
    D3DTSS_BUMPENVMAT00 = 7,
    D3DTSS_BUMPENVMAT01 = 8,
    D3DTSS_BUMPENVMAT10 = 9,
    D3DTSS_BUMPENVMAT11 = 10,
    D3DTSS_TEXCOORDINDEX = 11,
    D3DTSS_BUMPENVLSCALE = 22,
    D3DTSS_BUMPENVLOFFSET = 23,
    D3DTSS_TEXTURETRANSFORMFLAGS = 24,
    D3DTSS_COLORARG0 = 26,
    D3DTSS_ALPHAARG0 = 27,
    D3DTSS_RESULTARG = 28,
    D3DTSS_CONSTANT = 32,
    D3DTSS_FORCE_DWORD = 0x7fffffff
} D3DTEXTURESTAGESTATETYPE;
```

Below we discuss the texture stage states listed above but not necessarily in the order specified.

### D3DTSS\_COLORARG0

### D3DTSS\_COLORARG1

### D3DTSS\_COLORARG2

These states configure the inputs for the color operation specified with the `D3DTSS_COLOROP` texture stage state. We use them to assign the diffuse pixel color to ARG1 for example, or the texel color to ARG2, etc. These states are used for configuring the arguments to the color unit only. When setting one of these states using the `SetTextureStageState` function, the third parameter should be one of the DirectX defined `D3DTA_` values listed below with a description of its meaning. (TA is short for Texture Argument). Most of the available color blending functions use only two arguments (ARG1 and ARG2) but there are a few that use ARG0.

### D3DTA\_DIFFUSE

The interpolated per-pixel diffuse color is routed to the specified input. For example, the following code snippet would specify that the ARG1 input to the color unit in texture stage 0 should be the interpolated diffuse color.

```
pDevice->SetTextureStateState(0, D3DTSS_COLORARG1 , D3DTA_DIFFUSE);
```

Using the above code, any color operations that use ARG1 during blending will use the diffuse color. The next example shows how you would set the diffuse color to ARG2 for the color unit in texture stage 3.

```
pDevice->SetTextureStateState(3, D3DTSS_COLORARG2 , D3DTA_DIFFUSE);
```



## D3DTA\_TEXTURE

Binds the texel color sampled for the current pixel as an argument to one of the blending equations. To use this as an argument for a color unit you should have a texture assigned to the stage. The texel color used as an argument will be the one returned from the sampler unit. The following example shows how to configure ARG1 in the color unit to receive the texture color.

```
pDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
```

## D3DTA\_SPECULAR

If we are using pre-lit vertices then we can store a specular color in our vertex structure -- if we do not store it, then the specular component in our vertex defaults to black. When using the lighting pipeline, the specular color is calculated by DirectX. This state allows us to assign the interpolated per-pixel specular color to one of the arguments of a color operation. The following code shows to assign the specular color as an input argument.

```
pDevice->SetTextureStageState(0, D3DTSS_COLORARG0, D3DTA_SPECULAR);
```

## D3DTA\_CURRENT

This argument is the key to using multiple stages. When we assign this value to an argument in a stage that is not stage 0, the color of that argument is the output from the previous texture stage. This means for example, that we could modulate the diffuse color and the texture color in stage 0 so that the result is output to stage 1 where it is used in another blending operation, perhaps with sampled texels from the texture assigned to stage 1.

When used as an argument for stage 0 it simply defaults to the same behaviour as **D3DTA\_DIFFUSE**. This is because stage 0 is the first stage in the cascade and there is no previous stage data to use as an input.

The following code shows how we might set ARG1 in the second texture stage such that it uses the color output by texture stage 0.

```
pDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_CURRENT);
```

## D3DTA\_TFACTOR

We can provide texture stages with access to a constant color. This can be useful for a number of tasks. Perhaps we might decide to perform an ADD operation so that the texture factor color is added to the color input into the stage. Or perhaps we wish to darken some pixels by modulating with a half-intensity color like (128, 128, 128, 128). These values will be converted to floating point numbers (0.5, 0.5, 0.5, 0.5) for use in blending operations. The application will set the color using a render state rather than a texture stage state and it will be global across all texture stages.

```
pDevice->SetRenderState(D3DRS_TEXTUREFACTOR, 0xFF0000FF);
pDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_TFACTOR);
```



## D3DTA\_CONSTANT

This allows a particular stage to use an application set constant color in its blending operations. Unlike the **TFACTOR** argument, this color will only be available to that stage.

```
pDevice->SetTextureStageState(0 , D3DTSS_CONSTANT , 0xFF0000FF);  
pDevice->SetTextureStageState(0 , D3DTSS_COLORARG1 , D3DTA_CONSTANT);
```

If you intend to use a per-stage constant color then you will need to check the device capabilities. Such constants are not supported on most hardware, at least not at the time of this writing. If this feature is supported, then the **D3DP MISCCAPS\_PERSTAGECONSTANT** bit will be set in the PrimitiveMiscCaps member of the D3DCAPS9 structure:

```
D3DCAPS9 caps;  
pDevice->GetDeviceCaps (&caps);  
if(Caps.PrimitiveMiscCaps & D3DP MISCCAPS_PERSTAGECONSTANT)  
    Supported = TRUE;
```

## D3DTSS\_TEXCOORDINDEX

By default, if we are using stage 0, then the first set of texture coordinates in the vertex describe the mapping of the texture in stage 0 to the polygon. If there is a texture in stage 1, then the second set of vertex texture coordinates describe how the texture in stage two is mapped to the polygon. And so on up until the final stage.

We can use this texture stage state to change these defaults and instruct a given stage to use any of the available coordinate sets in the vertex. For example, you might have 2 textures, one in stage 0 and one in stage 1. Assume that the same texture coordinates describe how they are mapped to the polygon. In this case, rather than store a duplicate set, you could set both stages to use texture coordinate set 0.

```
pDevice->SetTextureStageState(0 , D3DTSS_TEXCOORDINDEX , 0);  
pDevice->SetTextureStageState(1 , D3DTSS_TEXCOORDINDEX , 0);
```

## D3DTSS\_CONSTANT

Set the per-stage constant color.

```
pDevice->SetTextureStageState(0 , D3DTSS_CONSTANT , 0xFF0000FF);
```

## D3DTSS\_ALPHAARG1

## D3DTSS\_ALPHAARG2

## D3DTSS\_ALPHAARG3

These states allow us to specify inputs for the alpha pipeline and its blending functions. We will examine alpha blending in Chapter 7 so for now, just be aware that D3DTA\_ values allow us to pass parameters to the alpha pipeline as well as the color pipeline.

```
pDevice->SetTextureStageState(0 , D3DTSS_ALPHAARG1 , D3DTA_DIFFUSE);
```



## D3DTSS\_COLOROP

We use this state to configure the color blending operation for the texture stage. The third parameter to the SetTextureStageState function should be one of the members of the **D3DTEXTUREOP** enumerated type shown below.

```
typedef enum _D3DTEXTUREOP
{
    D3DTOP_DISABLE = 1,
    D3DTOP_SELECTARG1 = 2,
    D3DTOP_SELECTARG2 = 3,
    D3DTOP_MODULATE = 4,
    D3DTOP_MODULATE2X = 5,
    D3DTOP_MODULATE4X = 6,
    D3DTOP_ADD = 7,
    D3DTOP_ADDSIGNED = 8,
    D3DTOP_ADDSIGNED2X = 9,
    D3DTOP_SUBTRACT = 10,
    D3DTOP_ADDSMOOTH = 11,
    D3DTOP_BLENDDIFFUSEALPHA = 12,
    D3DTOP_BLENDTEXTUREALPHA = 13,
    D3DTOP_BLENDFACTORALPHA = 14,
    D3DTOP_BLENDTEXTUREALPHAPM = 15,
    D3DTOP_BLENDCURRENTALPHA = 16,
    D3DTOP_PREMODULATE = 17,
    D3DTOP_MODULATEALPHA_ADDCOLOR = 18,
    D3DTOP_MODULATECOLOR_ADDALPHA = 19,
    D3DTOP_MODULATEINVALPHA_ADDCOLOR = 20,
    D3DTOP_MODULATEINVCOLOR_ADDALPHA = 21,
    D3DTOP_BUMPPENVMAP = 22,
    D3DTOP_BUMPPENVMAPLUMINANCE = 23,
    D3DTOP_DOTPRODUCT3 = 24,
    D3DTOP_MULTIPLYADD = 25,
    D3DTOP_LERP = 26,
    D3DTOP_FORCE_DWORD = 0x7fffffff
} D3DTEXTUREOP;
```

**Note:** The default color operation for texture stage 0 is diffuse/texture color modulation.

Most of the operations work exclusively with arguments 1 and 2 and are relatively simple mathematical operations like add, subtract, multiply (modulate), and so on. Some of the modes are more advanced and will be covered later in the course and in the next course in this series.

In the following descriptions, R is used to indicate the result of the color operation. ARG1, ARG2 and ARG3 represent color inputs as arguments to the color or alpha pipelines. We will not discuss all the above texture operations but will concentrate on the ones relevant to us at this time.



## D3DTOP\_SELECTARG1

The operation routes the ARG1 input straight to the output of the stage, ignoring all other inputs.

$$R = ARG1$$

## D3DTOP\_SELECTARG2

The operation routes the ARG2 input straight to the output of the stage, ignoring all other inputs.

$$R = ARG2$$

## D3DTOP\_MODULATE

The inputs to the texture stage are multiplied together. This is the default color operation for stage 0. It can be used to modulate the diffuse color and the texture color as discussed above. Note that because this is floating point multiplication, the result is actually darker than one or both input colors -- which may or may not be desirable. For example,  $(0.9, 0.9, 0.9) \times (0.5, 0.5, 0.5) = (0.45, 0.45, 0.45)$

$$R = ARG1 \times ARG2$$

## D3DTOP\_MODULATE2X

This is a useful operation if `D3DTOP_MODULATE` creates undesirably dark results. It performs the multiplication and then doubles the result -- therefore doubling its brightness.

$$R = (ARG1 \times ARG2) \ll 1$$

## D3DTOP\_MODULATE4X

Modulate the colors and then multiply the result by four. This can be useful when blending two very dark textures or where you need to over-brighten. For general use, this will often cause a washed out look as the resulting color components exceed the 1.0 max limit and are clamped.

$$R = (ARG1 \times ARG2) \ll 2$$

## D3DTOP\_ADD

This operation adds the two colors together. If the ARG1 and ARG2 matching components are both over 0.5, they will exceed the 1.0 range and be clamped to 1.0 (bright white).

$$R = ARG1 + ARG2$$



## D3DTOP\_ADDSIGNED

Add the two arguments together and subtract 0.5. This makes the effective range of values [-0.5, 0.5].

$$R = ARG1 + ARG2 - 0.5$$

This allows us to blend two textures together without suffering from over brightening. The results are comparable to **D3DTOP\_MODULATE2X**.

## D3DTOP\_ADDSIGNED2X

ADDSIGNED with the result multiplied by two.

$$R = (ARG1 + ARG2 - 0.5) \ll 1$$

## D3DTOP\_SUBTRACT

Subtract ARG2 from ARG1.

$$R = (ARG1 - ARG2)$$

## D3DTOP\_MULTIPLYADD

This blending operation uses all 3 arguments. ARG2 and ARG3 are multiplied and ARG1 is added to the result. ARG3 is set using **D3DTSS\_COLORARG0** and **D3DTSS\_ALPHAARG0** for the respective pipelines.

$$R = ARG1 + (ARG2 * ARG3)$$

## D3DTOP\_LERP

Linearly interpolates using all three input arguments. ARG3 is set using **D3DTSS\_COLORARG0** and **D3DTSS\_ALPHAARG0** for the respective pipelines.

$$R = (ARG1) * ARG2 + (1 - ARG1) * ARG3.$$

## D3DTOP\_DISABLE

This disables the stage and it will output no color results. Because the texture stages are connected in ascending order, disabling the color operations in a given texture stage will also disable color operations in subsequent stages. By default, when the device is first created, stages 1 – 7 are disabled and only stage 0 is enabled.

**Note:** Alpha operations should not be disabled when color operations are enabled as this can result in undefined behavior. Make sure to set D3DTOP\_DISABLE in both the color and alpha pipelines at the same time for a given stage.



## Texture Stage Usage

In these first examples we will only be using stage 0. After you have studied the code for Lab Project 6.1, we can examine using multiple stages to perform single-pass and multi-pass texture blending.

### Example 1: Blending Diffuse and Texture Color

These settings are actually the default states for texture 0. We use a color modulate operation to blend the diffuse color with the texture color currently set at that stage.

```
pDevice->SetTexture      ( 0 ,  pBrickWallTexture);
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLOROP , D3DTOP_MODULATE );
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLORARG1 , D3DTA_DIFFUSE);
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLORARG2 , D3DTA_TEXTURE);
```

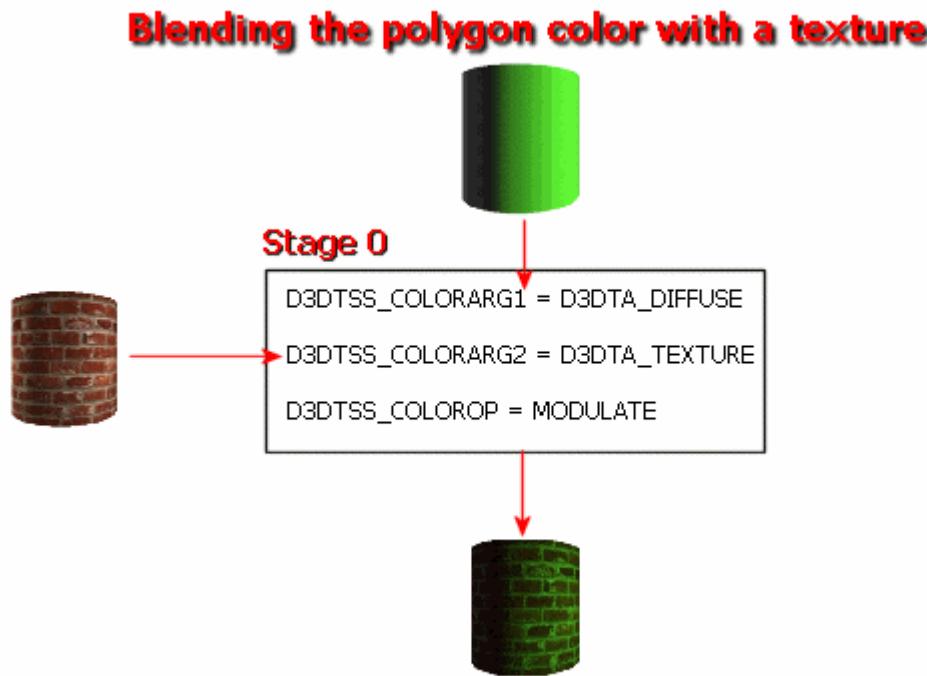


Figure 6.31

### Example 2: Texture Color Only

In this example, the diffuse fragment color will be ignored and only the color of the texture will be used.

```
pDevice->SetTexture      ( 0 ,  pBrickWallTexture);
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLOROP , D3DTOP_SELECTARG1 );
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLORARG1 , D3DTA_TEXTURE);
```



### Example 3: Adding Texture and Diffuse colors

The texture color is added to the diffuse color, which would cause the scene to be rendered brighter.

```
pDevice->SetTexture           ( 0 ,  pBrickWallTexture);
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLOROP , D3DTOP_ADD );
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLORARG1 , D3DTA_DIFFUSE);
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLORARG2 , D3DTA_TEXTURE);
```

**Note:** Burnout happens when color overflows the allowed range and is clamped. In the extreme case, this would cause the scene to be rendered completely white.

### Example 4: Adding Diffuse to a Constant Color

In some ways it might be preferable thinking of texture stages simply as color stages. As this next example shows, we do not need to use textures in order to benefit from the color operations the texture stages supply. In this case we will not use a texture at all. Instead we will set a constant color which will then be added to the diffuse color of all objects rendered.

```
pDevice->SetRenderState      ( D3DRS_TEXTUREFACTOR , 0xFF008800);
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLOROP , D3DTOP_ADD );
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLORARG1 , D3DTA_DIFFUSE);
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLORARG2 , D3DTA_TFACTOR);
```

### Example 5: Modulating the Texture with the Texture Factor

In this example we take the texture color for each pixel and scale it by the texture factor. In this case it has half intensity RGB components which will scale the texture color by half.

```
pDevice->SetRenderState      ( D3DRS_TEXTUREFACTOR , 0xFF888888 );
pDevice->SetTexture          ( 0 ,  pBrickWallTexture );
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLOROP , D3DTOP_MODULATE );
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLORARG1 , D3DTA_TEXTURE);
pDevice->SetTextureStageState ( 0 ,  D3DTSS_COLORARG2 , D3DTA_TFACTOR);
```



## Multi-Texturing

As discussed earlier in the lesson, the texture stages form a texture blending cascade. Input starts at the top of the cascade (stage 0) and the result of operations conducted in that stage can be passed to the next stage as an input argument. In this next stage, it can be blended with the color of the texture assigned to that stage (or some other argument) and passed on yet again. Fig 6.32 shows the texture blending cascade using the first four stages to perform four color operations.

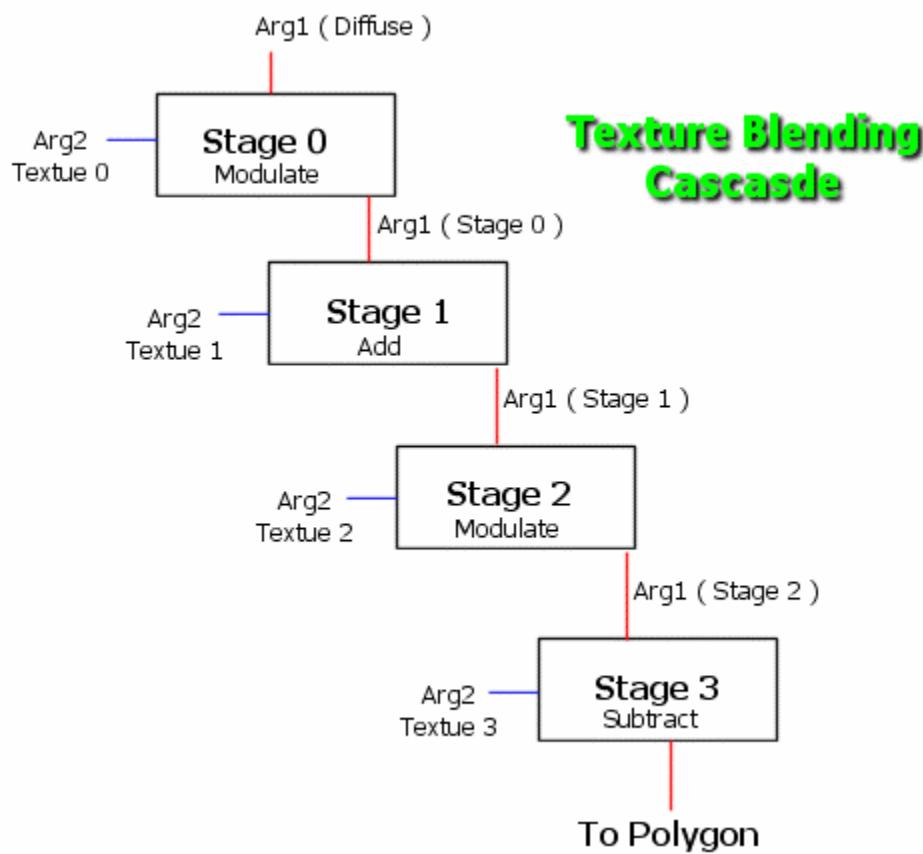


Figure 6.32

In Fig 6.32, the diffuse color and the texture are assigned to stage 0 as input arguments. They are modulated and the result is passed as stage 1's first argument. In this stage, the texture assigned to stage 1 is used as the second argument and added to argument 1 -- the output from the previous stage. The result is passed to stage 2 where it is modulated with the texture assigned there. The result of this operation is passed to stage 3 where the texture color there is subtracted from it. The result is the pixel color sent to be rendered. While this may be an unlikely set of operations, it does get the point across. Note as well that if you use multiple stages, you must use them in order. You cannot just use stages 0, 1, and 5 for example.



By default, only stage 0 is active and the color operations and alpha operations in all other stages are set to **D3DTOP\_DISABLE**. To enable consecutive stages, we simply set the color (or alpha) operation we desire in that stage and assign the inputs. You will usually want one of the inputs to these texture stages set to **D3DTA\_CURRENT** to access the result of a previous stage.

Fig 6.33 sets up two texture stages: a brick wall texture in the first stage and a yellow light texture (a simple light map) in the second stage. We pass the stage 0 texture color to stage 1 where it will be added to the color of the light texture.

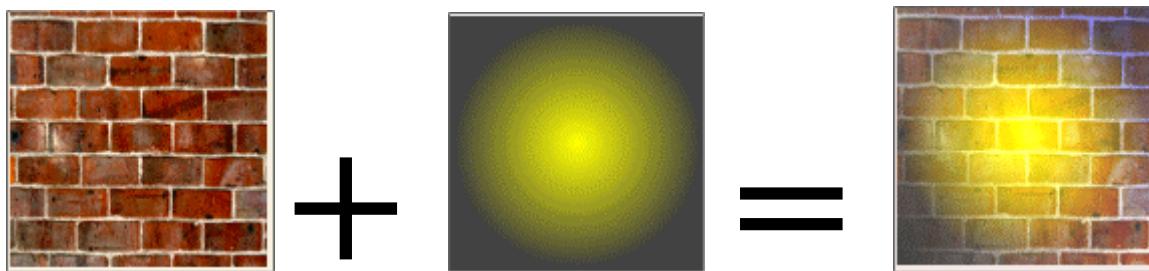


Figure 6.33

The texture stage settings to achieve this effect are shown below.

```
pDevice->SetTexture      ( 0 , pBrickWallTexture );
pDevice->SetTextureStageState( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE );
pDevice->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_SELECTARG1 );

pDevice->SetTexture      ( 1 , pYellowLightTexture );
pDevice->SetTextureStageState( 1 , D3DTSS_COLORARG1 , D3DTA_CURRENT );
pDevice->SetTextureStageState( 1 , D3DTSS_COLORARG2 , D3DTA_TEXTURE );
pDevice->SetTextureStageState( 1 , D3DTSS_COLOROP , D3DTOP_ADD );
```

Mastering multiple texture blending techniques is critical to making visually compelling games. In this course, we only begin to scratch the surface. The next course in this series will spend a great deal of time examining multi-texturing techniques to achieve realistic scene lighting and other interesting visual effects.

The next example adjusts the previous texture stage states so that stage 0 modulates the texture with the diffuse vertex color before sending it on to stage 1. The resulting color is passed to stage 2 where its color is scaled in half by modulating all components by the texture factor.

```
pDevice->SetTexture      ( 0 , pBrickWallTexture );
pDevice->SetTextureStageState( 0 , D3DTSS_COLORARG1 , D3DTA_DIFFUSE );
pDevice->SetTextureStageState( 0 , D3DTSS_COLORARG2 , D3DTA_TEXTURE );
pDevice->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_MODULATE );
```



```
pDevice->SetTexture           ( 1 , pYellowLightTexture);
pDevice->SetTextureStageState( 1 , D3DTSS_COLORARG1 , D3DTA_CURRENT);
pDevice->SetTextureStageState( 1 , D3DTSS_COLORARG2 , D3DTA_TEXTURE);
pDevice->SetTextureStageState( 1 , D3DTSS_COLOROP , D3DTOP_ADD );

pDevice->SetRenderState      ( D3DRS_TEXTUREFACTOR,0x88888888 );
pDevice->SetTextureStageState( 2 , D3DTSS_COLORARG1 , D3DTA_CURRENT);
pDevice->SetTextureStageState( 2 , D3DTSS_COLORARG2 , D3DTA_TFACTOR);
pDevice->SetTextureStageState( 2 , D3DTSS_COLOROP , D3DTOP_MODULATE);
```

**Note:** Video cards have varying levels of support for the number of allowable texture stages. Some even impose limitations on the order of the arguments. For example, some drivers prefer the D3DTA\_TEXTURE argument to always be ARG1 for a given stage. Tom Forsyth's blogspot is an excellent resource for analysis of the texture stage support on various hardware.

<http://tomsdxfaq.blogspot.com/>

If your texture stage state configurations perform strangely on certain video cards, this website should be your first port of call in identifying whether there is a card or driver specific problem. Often, there is a way to re-order your texture stage operations to find a configuration that works.

Our application should check the capabilities of the device to see how many texture stages are supported and how many different textures can be blended simultaneously. A device might support four texture stages for example, but only allow you to blend three textures. There are two members of the D3DCAPS9 structure which give us this information: **MaxTextureBlendStages** and **MaxSimultaneousTextures**.

```
D3DCAPS9 Caps;
m_pDevice->GetDeviceCaps(&Caps);
DWORD MaxTextures = Caps.MaxSimultaneousTextures;
DWORD MaxStages   = Caps.MaxTextureBlendStages;
```

You should also test to make sure that the color operations and alpha operations you intend to use are supported by the current device. The IDirect3DDevice9 interface exposes the ValidateDevice function for this purpose. First set up the texture stages and render states as they would be used in the render loop. Then call this function to test whether the current stages will render in a single pass.

```
HRESULT ValidateDevice(DWORD *pNumPasses);
```

The HRESULT returned will be one of the values in the following table.



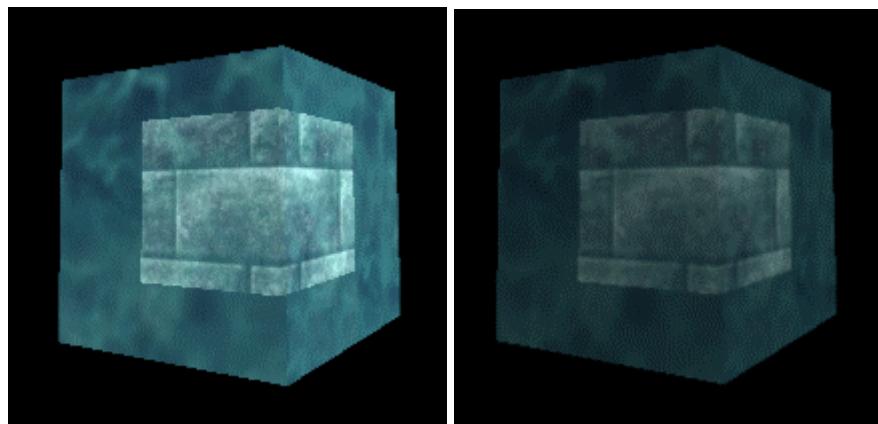
D3DERR_CONFLICTINGTEXTUREFILTER	The current texture filters cannot be used together.
D3DERR_DEVICELOST	The device has been lost but cannot be reset at this time. Therefore, rendering is not possible.
D3DERR_DRIVERINTERNALERROR	Internal driver error. Applications should generally shut down when receiving this error.
D3DERR_TOOMANYOPERATIONS	The application is requesting more texture filtering operations than the device supports.
D3DERR_UNSUPPORTEDALPHAARG	The device does not support a specified texture-blending argument for the alpha channel.
D3DERR_UNSUPPORTEDALPHAOPERATION	The device does not support a specified texture-blending operation for the alpha channel.
D3DERR_UNSUPPORTEDCOLORARG	The device does not support a specified texture-blending argument for color values.
D3DERR_UNSUPPORTEDCOLOROPERATION	The device does not support a specified texture-blending operation for color values.
D3DERR_UNSUPPORTEDFACTORVALUE	The device does not support the specified texture factor value.
D3DERR_UNSUPPORTEDTEXTUREFILTER	The device does not support the specified texture filter.
D3DERR_WRONGTEXTUREFORMAT	The pixel format of the texture surface is not valid.

Notice that we pass in the address of a DWORD. The function will fill it with the number of passes needed to render with the current state setup. If the value  $> 1$ , then we are trying to use either more textures than the maximum amount or more stages. We will need to break the texture stage configuration down into smaller sub-configurations and render the polygon multiple times using a subset of the states each time.

## Color Blending

DirectX allows us to blend the polygon pixels we are about to render with the pixels already rendered in the frame buffer -- rather than simply overwriting them. This process is commonly referred to as alpha blending and the result is a transparent effect. This choice of wording is somewhat unfortunate because we are not limited to blending based only on the alpha component of a color. The technique can also be used quite successfully for general RGB color blending. In Chapter 7 we will examine alpha blending based solely on the alpha component of a color. In this chapter, we will discuss RGB color blending.

In Fig 6.34 the smaller cube was rendered using our standard approach. Then we rendered the larger cube with color blending enabled and a color blending operation setup. The final color of each pixel rendered to the frame buffer was calculated by combining the color of the fragment output from the texture stages with the color of the pixel already in the frame buffer (the smaller cube pixels). The left image used an addition operation to add the pixel to the current contents of the frame buffer. The right image used a modulate operation so that each pixel was multiplied with the current pixel color in the frame buffer.



**Figure 6.34**

After we have finished rendering the color blended polygons, we can disable color blending and continue to render any other polygons normally. Usually you will want to sort your polygons such that you can render all opaque polygons first followed by those that need to be color blended. This will cut down on the number of render state changes.

To enable color blending, we will set three render states. The first state (**D3DRS\_ALPHABLENDENABLE**) informs the device to enable blending with the frame buffer. We set it to either true or false. Do not be put off by the name, it can be used to perform both alpha blending *and* color blending.

```
m_pDevice->(D3DRS_ALPHABLENDENABLE , TRUE); //enable blending
m_pDevice->(D3DRS_ALPHABLENDENABLE , FALSE); //disable blending
```

Once blending has been enabled, the device will take the values generated by the color and alpha pipelines in the texture stage cascade and use them as input to a blending operation. Fig 6.35 shows the color and alpha components output from the texture stages moved into the rasterizer module. By default, alpha blending is disabled and alpha values output from the texture stages will be ignored. In this case, the color will be copied directly into the frame buffer overwriting anything that exists there (assuming the pixel has passed the depth test). When we enable blending however, the rasterizer can use these inputs (alpha, color, or both) as variables in a blending function that blends the pixel we are about to render with the pixel already in the frame buffer.



### How Render States Work

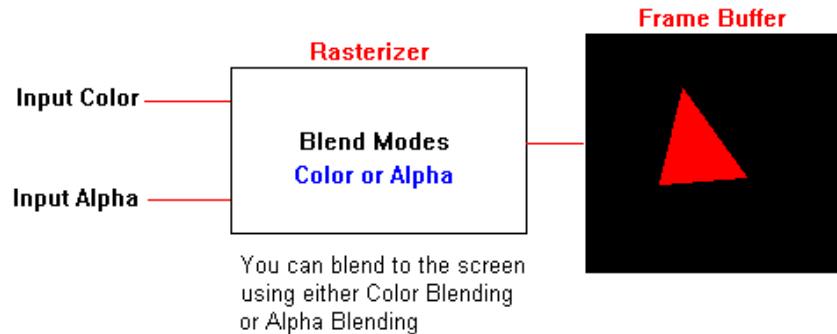


Figure 6.35

When blending is enabled, the following formula calculates the final color written to the frame buffer:

$$\text{FinalColor} = (\text{SrcColor} * \text{SrcBlend}) + (\text{DestColor} * \text{DestBlend})$$

**SrcColor** is the color output from the texture stage cascade (Input Color in Fig 6.35) and **DestColor** is the color of the pixel that is already in the frame buffer. The **SrcBlend** and **DestBlend** variables control the behavior of the blending equation using a set of blend modes. To set the source and destination blend modes we use the **D3DRS\_SRCBLEND** and **D3DRS\_DESTBLEND** render states:

```
m_pDevice->SetRenderState(D3DRS_SRCBLEND , SourceBlendMode);  
m_pDevice->SetRenderState(D3DRS_DESTBLEND , DestBlendMode );
```

For the second parameter, we pass a member of the **D3DBLEND** enumerated type:

```
typedef enum _D3DBLEND  
{  
    D3DBLEND_ZERO = 1,  
    D3DBLEND_ONE = 2,  
    D3DBLEND_SRCCOLOR = 3,  
    D3DBLEND_INVSRCOLOR = 4,  
    D3DBLEND_SRCALPHA = 5,  
    D3DBLEND_INVSRCALPHA = 6,  
    D3DBLEND_DESTALPHA = 7,  
    D3DBLEND_INVDESTALPHA = 8,  
    D3DBLEND_DESTCOLOR = 9,  
    D3DBLEND_INVDESTCOLOR = 10,  
    D3DBLEND_SRCALPHASAT = 11,  
    D3DBLEND_BOTHSRCALPHA = 12,  
    D3DBLEND_BOTHINVSRCALPHA = 13,  
    D3DBLEND_BLENDFACTOR = 14,  
    D3DBLEND_INVBLENDFACTOR = 15,  
    D3DBLEND_FORCE_DWORD = 0x7fffffff  
} D3DBLEND;
```



These blend modes are used to scale the source color and the destination color by some given amount. Let us take a look at a few of them to see what they do.

**D3DBLEND\_ZERO** = ARGB (0, 0, 0, 0)

Multiply all components of the color by 0. The color will not contribute to the final color in any way.

**D3DBLEND\_ONE** = ARGB(1, 1, 1, 1)

Multiply all components of the color by 1. The color is not diminished at all by the operation and the final color will be at least as bright as the color for which this scale factor is used.

**D3DBLEND\_SRCCOLOR** = ARGB (S\_alpha, S\_red, S\_green, S\_blue)

The ARGB components will be multiplied by the source color ARGB components.

**D3DBLEND\_INVSRCOLOR** = ARGB (1-S\_alpha, 1-S\_red, 1-S\_green, 1-S\_blue)

This multiplies each component of the color by the inverse of the source color.

**D3DBLEND\_DESTCOLOR** = ARGB(D\_alpha, D\_red, D\_green, D\_blue)

This multiplies the components of the color by the components of the frame buffer color.

**D3DBLEND\_INVDESTCOLOR** = ARGB (1-D\_alpha, 1-D\_red, 1-D\_green, 1-D\_blue)

Each component of the color is multiplied by one minus the corresponding component of the frame buffer color.

### Example 1 (Source Pixel Overwrites Frame Buffer Pixel)

```
m_pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);  
m_pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ZERO);
```

In this example we multiply each component of the source color by 1, so it is unaltered. The current frame buffer color has been multiplied by 0 so of course it will not play a part in the final pixel color when the two are added. This is the same result we would see if alpha blending was disabled. This is for example only. We would not want to use this mode for standard rendering since blending will be slower. Using these blend modes, the blending equation would be as follows:

```
FrameBufferColor = Source(ARGB) * 1 + Dest(ARGB) * 0  
FrameBufferColor = Source(ARGB) * (1,1,1,1) + Dest(ARGB) * (0,0,0,0)  
FrameBufferColor = Source(ARGB) + (0,0,0,0)  
= Source(ARGB)
```

### Example 2 (Frame Buffer Unchanged by Source Pixel)

```
m_pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO);  
m_pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
```



The final color is the source color multiplied by 0 plus the frame buffer pixel color multiplied by 1. So, the source pixel does not alter the frame buffer in any way.

$$\begin{aligned}\text{FrameBufferColor} &= \text{Source(ARGB)} * 0 + \text{Dest(ARGB)} * 1 \\ \text{FrameBufferColor} &= \text{Source(ARGB)} * (0,0,0,0) + \text{Dest(ARGB)} * (1,1,1,1) \\ \text{FrameBufferColor} &= (0,0,0,0) + \text{Dest(ARGB)} \\ &= \text{Dest(ARGB)}\end{aligned}$$

### Example 3 (Additive Blend between Source and Frame Buffer Pixels)

```
m_pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);  
m_pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
```

Let us imagine that we have a source pixel ARGB (0.5, 0.5, 0.5, 0.5) and a frame buffer pixel ARGB (0.2, 0.2, 0.2, 0.2). The final color in the frame buffer would be:

$$\begin{aligned}\text{FrameBufferColor} &= \text{Source(ARGB)} * 0 + \text{Dest(ARGB)} * 1 \\ \text{FrameBufferColor} &= (0.5,0.5,0.5,0.5) * (1,1,1,1) + (0.2,0.2,0.2,0.2) * (1,1,1,1) \\ \text{FrameBufferColor} &= (0.5, 0.5, 0.5, 0.5) + (0.2, 0.2, 0.2, 0.2) \\ &= (0.7, 0.7, 0.7, 0.7)\end{aligned}$$

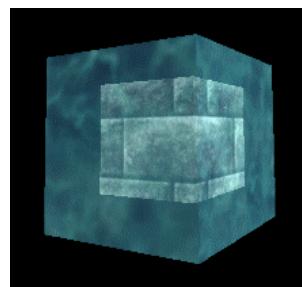
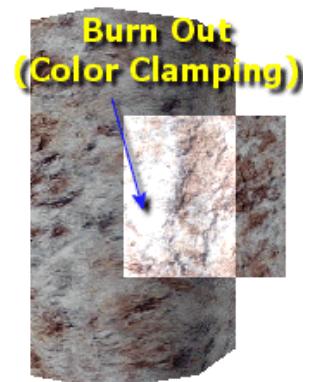


Figure 6.36

In Fig 6.36 we see the results of an addition operation similar to the equation above.

When using addition to blend already bright pixels, this can result in burnout. Burnout happens when detail is lost because many of the pixels have been clamped to white. Neighboring pixels that would normally be subtly different shades of color will now all be clamped to the same color. The image on the right shows this happening when the cylinder is rendered after the cube with color blending enabled. Because both objects have fairly bright textures applied already, we can see that where they overlap, the result is washed out.



#### Example 4 (Source Color Intensity Blend)

```
m_pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
m_pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCOLOR);
```



This blending configuration uses the intensity of the source color to control the weight of the frame buffer color. As the source color increases, the destination color contributes to a lesser extent. When the source color is 0, the frame buffer color is used. When the source color is full intensity (1.0) the frame buffer color does not contribute at all.

SourceBlend = SrcColor  
DestBlend = 1 - SrcColor

$$\begin{aligned}\text{FrameBufferColor} &= \text{Src(ARGB)} * \text{Src(ARGB)} + \text{Dest(ARGB)} * (1 - \text{Src(ARGB)}) \\ \text{FrameBufferColor} &= (0.25, 0.25, 0.25, 0.25) * (0.25, 0.25, 0.25, 0.25) \\ &\quad + (0.5, 0.5, 0.5, 0.5) * (1-0.25, 1-0.25, 1-0.25, 1-0.25) \\ \text{FrameBufferColor} &= (0.062, 0.062, 0.062, 0.062) + (0.75, 0.75, 0.75, 0.75) \\ \text{FrameBufferColor} &= (0.812, 0.812, 0.812, 0.812)\end{aligned}$$

#### Example 5 (Emulating the Modulate2x Texture Stage Color Operation)

```
m_pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_DESTCOLOR);
m_pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_SRCCOLOR);
```

This example is useful, especially for multiple render passes, because it works like the modulate2x color operation in the texture stages. In effect we are modulating the source color and the destination color together twice and adding the results.

$$\begin{aligned}\text{FrameBufferColor} &= (\text{SourceColor} * \text{DestColor}) + (\text{DestColor} * \text{SourceColor}) \\ \text{FrameBufferColor} &= (\text{SourceColor} * \text{DestColor}) * 2\end{aligned}$$

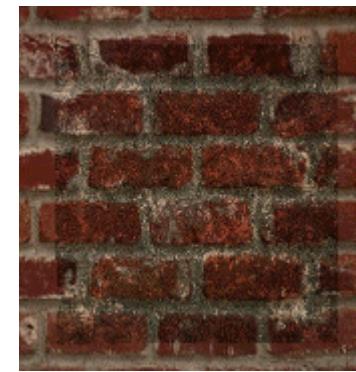
Let us assume that we have a source pixel color of (0.5, 0.5, 0.5, 0.5) and a destination color of (0.75, 0.75, 0.75).

$$\begin{aligned}\text{FrameBufferColor} &= (0.5, 0.5, 0.5, 0.5) * (0.75, 0.75, 0.75, 0.75) \\ &\quad + (0.75, 0.75, 0.75, 0.75) * (0.5, 0.5, 0.5, 0.5) \\ \text{FrameBufferColor} &= (0.375, 0.375, 0.375, 0.375) + (0.375, 0.375, 0.375, 0.375) \\ \text{FrameBufferColor} &= (0.75, 0.75, 0.75, 0.75)\end{aligned}$$

If the source color is exactly half intensity (0.5), then the frame buffer is unchanged by the operation. If the source color is less than half intensity, then the frame buffer result will be darkened to some degree. If the source color is greater than half intensity then the frame buffer result will be lightened to some degree. This is the blending mode we will use in Lab Project 6.2 for detail texturing.



The image on the left shows a brick texture and a high-detail gray cement texture, each mapped to its own quad. The quads are rendered without any alpha blending. As expected, the second polygon (the gray one) overwrites the first one. In the image on the right, we render the gray polygon with the blending operation just discussed. We note that it has added a grainy detail to the otherwise smooth brick wall texture. Also notice that because we are doing what is essentially a modulate2x operation, the intensity of the blended pixels has been mostly maintained.



**Note:** One of the blend modes that we can specify is D3DBLEND\_BLENDFACTOR. This allows us to use a constant color as a blend factor much like the TextureFactor constant discussed earlier. It is set with a call to SetRenderState. The following example sets a half-intensity color which can be used in the alpha blending equation during frame buffer blending.

```
m_pDevice->SetRenderState(D3DRS_BLENDFACTOR, 0x88888888);
```

This functionality is available if the D3DPBLENDCAPS\_BLENDFACTOR capabilities bit is set in the SrcBlendCaps member of D3DCAPS9 or the DestBlendCaps member of D3DCAPS9. This identifies whether the blend factor can be used as a source blend mode or a destination blend mode respectively.

Please open your workbook now to Lab Project 6.2. This project will use multi-texturing to blend two textures onto a terrain in a single pass. If multi-texturing is not available on the current hardware, we will implement the texture blending using multiple passes. This means that we will render the terrain with the first texture, and then render it again with the second texture using alpha blending to color blend the second pass with the results of the first.



## Compressed Textures

Compressed textures reduce the storage requirements for texture resources. This allows us to store more textures in video memory at any one time, and also to reduce the amount of bandwidth required to upload the data to the card during texture swapping. While there is some small amount of overhead involved when rendering with compressed textures due to on-the-fly decompression, this overhead is generally minimal compared to the cost of storing uncompressed textures and transferring them across the bus to the graphics card. The potential downside to compression is lossiness -- some image data may be discarded to save space -- which might affect texture detail and quality. However, when using texture compression, we can work with much larger textures and, for the most part, this addresses any potential loss in quality.

The D3DXCreateTextureFromFileEx function makes loading and compression simple and easy. Let us imagine that we would like to use compressed textures of the type **D3DFMT\_DXT1** (do not worry what this actually means for the moment). There are two steps that we must perform. First, we will use **IDirect3DDevice9::CheckDeviceFormat** to determine whether or not the desired compressed format can be used with our current device. If so, then we can proceed to the second step and load the texture.

```
ULONG      Ordinal = pSettings->AdapterOrdinal;
D3DDEVTYPE Type    = pSettings->DeviceType;
D3DFORMAT AFormat = pSettings->DisplayMode.Format;

if ( SUCCEEDED( m_pD3D->CheckDeviceFormat( Ordinal, Type, AFormat, 0,
                                              D3DRTYPE_TEXTURE, D3DFMT_DXT1) ) )
{
    D3DXCreateTextureFromFileEx(m_pD3DDevice, FileName, D3DX_DEFAULT, D3DX_DEFAULT,
                               D3DX_DEFAULT, 0, D3DFMT_DXT1, D3DPOOL_MANAGED,
                               D3DX_DEFAULT, D3DX_DEFAULT, 0, NULL, NULL, &pTexture);
}
```

We used the D3DXCreateTextureFromFileEx form of the function because we want to specify the precise surface format that we want the source image converted to. If we had just used the D3DXCreateTextureFromFile function, then it would have chosen a texture format that most closely matched the image format in the file. When the function returns, we will have a pointer to our compressed texture and we can use it just like a normal texture from this point on. The texture can even be locked and the pixel data manipulated, although this will cause the compressed data to be decompressed into a temporary surface for reads and writes. After we unlock the surface, the modified image data will be compressed again.



## Compressed Texture Formats

There are five compressed texture formats available for use in DirectX Graphics. They are members of the D3DFORMAT enumerated type and are listed below along with brief descriptions. We will go on to examine each in further detail in the next few sections.

### D3DFMT\_DXT1

The DXT1 format is primarily used when the texture is not required to store an alpha component, or in cases where we might only require a single alpha level (i.e. on/off). This will probably be the compression format used most within your scene because in most cases textures are either fully opaque or are being used for entities such as billboards for rendering trees, grass, etc. In the former case, no transparency is required. In the latter, it is often the case that we are only required to trace the outline of the billboard texture itself.

### D3DFMT\_DXT2 & D3DFMT\_DXT3

These two formats are commonly used in situations where the texture requires several levels of alpha. This is often the case with textures used for effects like lens flares, user interface elements, and other special effects. Although the two provide similar functionality, there is one key difference between them. When a texture is created using the DXT2 format, each pixel's color data is multiplied by any per-pixel alpha information stored within the texture. This is done as a pre-process, before texture creation is completed. For the DXT3 format, no such multiplication takes place.

### D3DFMT\_DXT4 & D3DFMT\_DXT5

Like DXT2 and DXT3, these formats also provide variable levels of alpha information. As with the previous two, DXT4 is the pre-multiplied format, and DXT5 is the standard format. The only difference between these two formats and the previous two formats is that the DXT2 and DXT3 both store their alpha information *explicitly*. This means that each pixel within the texture maintains its own alpha value. However, when using DXT4 or DXT5, much of the alpha information is discarded and is instead interpolated over a wide area of pixels. As a result, you would likely only use these formats in situations where the accuracy of the alpha information is not extremely important.

## Pre-multiplied Alpha Texture Formats

Probably one of the most confusing aspects about compressed texture formats is why we need the option to create texture surfaces that have their colors pre-multiplied by their alpha components. Although the alpha information is not discarded after the multiplication takes place, and we gain no compression/bandwidth benefits from this alone, the advantage to this approach is that it will speed up



the per-pixel calculations required to perform certain color blending operations. To understand why this is the case, we first need to take a look at the most typical alpha blending formula. It blends two colors together, taking into account the texture alpha:

$$R_{(RGB)} = (C1_{(RGB)} * Alpha) + (C2_{(RGB)} * (1 - Alpha))$$

In this formula, R is the resulting color, C1 and C2 are the two colors we are blending together, and Alpha is the alpha value being used. In this case, the alpha value is the one stored within the compressed texture. It is assumed that each component (R,G,B,A) is within the range [0.0, 1.0].

Note that we perform two multiplications. But if both C1 and Alpha originate in the same texture, we can speed up the formula by moving part of it out of the per-pixel operation and into a pre-process. This is where the pre-multiplication concept comes into play. Take a look at the formula again, and assume that Alpha originates in the same texture as C1:

$$R_{(RGB)} = (C1_{(RGB)} * C1_{(Alpha)}) + (C2_{(RGB)} * (1 - C1_{(Alpha)}))$$

This is exactly the same formula with the exception that the alpha value is taken from the same location as the first color input. We notice that the first portion of the runtime per-pixel processing ( $C1_{(RGB)} * C1_{(Alpha)}$ ) can be optimized. Rather than waste precious cycles, we can do this multiplication once as a pre-process when we create the texture and remove it from the formula altogether. So when we use a pre-multiplied texture for our C1 input value, we know in advance that the color components have already undergone this multiplication. Thus, we can then reduce the formula to:

$$R_{(RGB)} = C1_{(RGB)} + (C2_{(RGB)} * (1 - C1_{(Alpha)}))$$

Bear in mind that this is a per-pixel operation, so if we use a 1280x1024 frame buffer for example, we just eliminated up to 1,310,270 potential multiplications. This can be a significant savings indeed.

## Texture Compression Interpolation

One of the primary methods by which DirectX Graphics reduces texture storage space when using compressed textures is interpolation. This is true for color as well as alpha values. Let us start with an example. Take a look at the following 300 pixel wide gradient:



This block of color slowly changes from blue to red as we move from left to right. This transition takes place over 300 pixels, and thus requires 300 sets of color values to correctly express the gradient. Interestingly, we can store this entire image with as few as two color values if we were to use interpolation. Let us simplify this a bit and take a look at how interpolation works using an example which is only 3 pixels wide:



We can see here that the image uses 3 distinct colors. On the left, the first pixel has a bright blue color: RGB (0.0, 0.0, 1.0). On the right the third pixel has a bright red color: RGB (1.0, 0.0, 0.0). The pixel in the middle is a purple color: RGB (0.5, 0.0, 0.5). This pixel is positioned between the blue and red pixels and is rendered using a color value that is also directly between the neighboring pixel values. If we were to use interpolation to interpret the above image, we could store the color values for pixels 1 and 3 only, and then interpolate between the two to come up with the color for pixel 2. This concept can be applied to a much wider span of pixels of course. We could then render this arbitrarily wide gradient using these two stored colors with some code that is similar to the following:

```
// Calculate the color shift between each pixel of the gradient  
// We subtract one from GradientWidth so that the last pixel = Color_2  
// this ensures that during the loop (GradientWidth-1)/(GradientWidth-1) = 1.0  
Color_Shift = (Color_2 - Color_1) / (GradientWidth - 1);  
  
// Render the gradient  
for ( i = 0; i < GradientWidth + 1; i++ )  
    SetPixel( i, 0, Color_1 + (Color_Shift * i) );
```

This is the concept used for the majority of the texture compression techniques used by DirectX Graphics. Let us now examine how DirectX Graphics stores the color data within a compressed texture format and how interpolation applies.

## Compressed Data Blocks - Color Data Layout

Texture compression divides the texture into a series of 4x4 texel blocks. This allows the interpolation to be performed across each individual block of 4x4 texels, and works to maintain image data integrity. Each block is essentially a mini palletized image. Each of the 16 individual texels stores their information using 2 bits. These bits serve as an index (between 0 and 3), that look up a color in what we might consider a **virtual palette**.

Color_0
Color_1
XX XX XX XX

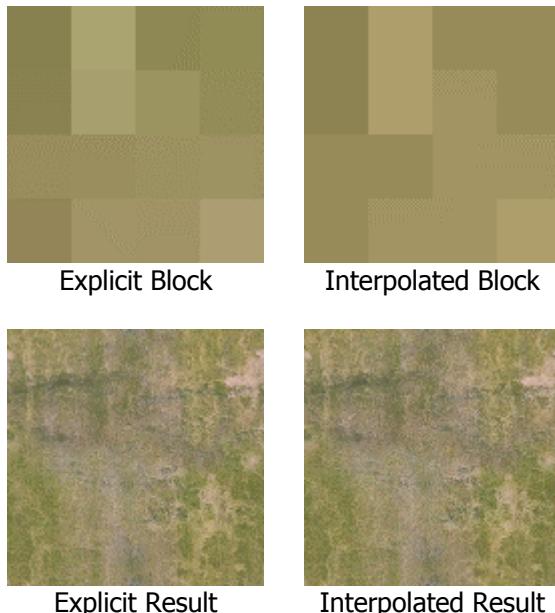
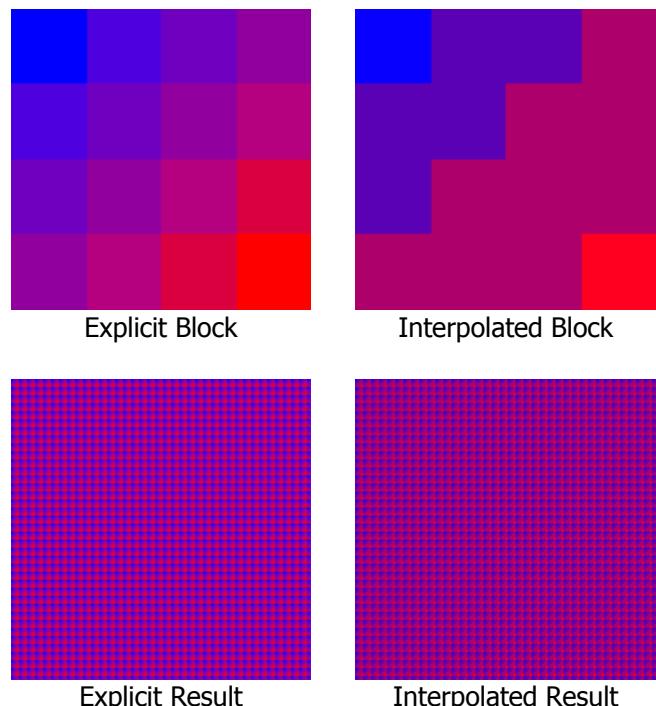
As we can see in the image to the left, each block contains **two** colors (not four as might be expected given the index range). We referred to the palette as ‘virtual’ because the other two colors to be indexed are generated at runtime by interpolating between the two colors provided within the block.

Referring back to the previous example, let us assume for a moment that Color\_0 contains the bright blue color and Color\_1 the bright red color. The third and fourth color values would now be calculated such that the third color is 2/3<sup>rd</sup> blue and 1/3<sup>rd</sup> red, and the fourth color is 1/3<sup>rd</sup> blue and 2/3<sup>rd</sup> red. We can think of this as generating a four pixel wide gradient using Color\_0 and Color\_1



as the outer bounds. This can reduce image quality to some degree, but in most cases, unless the texture is extremely small and scaled over a large area, the effect is hardly noticeable.

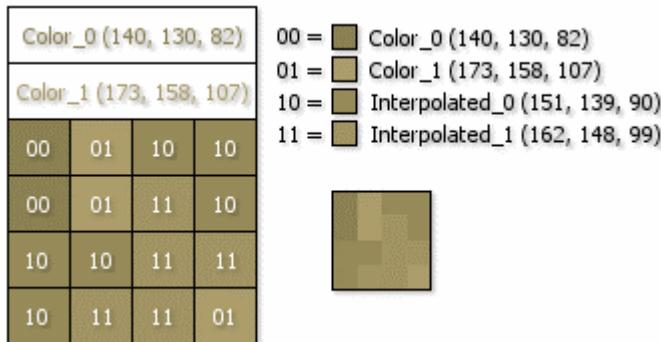
The images on the right show comparisons of the uncompressed (explicit) 4x4 block and the block using interpolated color values. We also see the resulting images (actual size) making use of each type. When we inspect the block colors close up, we can see a significant difference between the two. In the explicit block, we are using seven distinct colors and in the interpolated case we used four. The interpolated block (50X magnification) would not be of sufficient quality as an image on its own. However, the key to this technique is that each block of 16 texels can have its own unique virtual palette. So we can vary the range of colors used by each block in the image. If we take a look at the two results, we note that even in this extreme case, the result is a barely noticeable difference in the overall color between them.



In the more practical example on the left, even though it is possible to spot the difference between the two blocks when magnified, the resulting images are practically indistinguishable even at this relatively low resolution of 128x128 (non-filtered, actual size). The larger the texture, the less you will be able to see the visible signs of compression. This is due to the fact that more and more texels will be mapped to the polygon currently being rendered. This is really an ideal situation because it encourages us to increase the size and quality of our textures. We gain significant benefits in doing so because we can provide a far richer game environment without requiring more video memory on the card.



The compressed texture block is laid out and interpolated as follows:



The first portion holds the actual color data. These are the two extents between which our two interpolated colors will be computed. They are stored in a 16-bit R5G6B5 format. The 4x4 grid values specify the two bit binary code used as the index into the virtual palette. Recall that  $00_{(bin)} = 0_{(dec)}$ ,  $01_{(bin)} = 1_{(dec)}$ ,  $10_{(bin)} = 2_{(dec)}$  and  $11_{(bin)} = 3_{(dec)}$ . Thus, each color block uses only 64 bits (8 bytes) of memory, and provides reasonably good image quality.

Let us now interpolate the color values. As discussed earlier, the first interpolated color uses  $2/3^{\text{rd}}$  of Color\_0 and  $1/3^{\text{rd}}$  of Color\_1. The second interpolated color uses  $1/3^{\text{rd}}$  of Color\_0 and  $2/3^{\text{rd}}$  of Color\_1.

```
Interpolated_0(red) = (2 * (Color_0(red) / 3)) + (1 * (Color_1(red) / 3));  
Interpolated_0(red) = (2 * Color_0(red) + Color_1(red)) / 3;
```

```
Interpolated_1(red) = (1 * (Color_0(red) / 3)) + (2 * (Color_1(red) / 3));  
Interpolated_1(red) = (Color_0(red) + 2 * Color_1(red)) / 3;
```

Ex.

```
Color_0(red) = 140  
Color_1(red) = 173
```

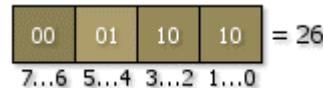
```
Interpolated_0(red) = (2 * 140 + 173) / 3;  
Interpolated_1(red) = (140 + 2 * 173) / 3;
```

```
Interpolated_0(red) == 151  
Interpolated_1(red) == 162
```

Referring back to the previous diagram, we realize that this works out exactly as our results demonstrated for the red components. If you follow this calculation through, you should find that it also works out for both the green and blue components too.



When decoding the 4x4 bit-map area of the block, we will extract the data for each row one byte at a time.



Here we see a single row of our example color data block. The row consists of 8 bits of data. In binary form, the byte consists of the bits 00 01 10 10. When converted to decimal, this value = 26. The code snippet below demonstrates how we might extract the indices from each 2 bit section of this byte. Remember that when using a little-endian system (as is the case with the Intel 80x86 architecture), bit 7 is furthest left and bit 0 is furthest right, as labeled above.

```
RowByte = 26; // Assume this is the first row
Texel[0][0] = Color[(RowByte & 0x03)];           // 2 (dec) = 10 (bin)
Texel[1][0] = Color[(RowByte & 0x0C) >> 2];     // 2 (dec) = 10 (bin)
Texel[2][0] = Color[(RowByte & 0x30) >> 4];     // 1 (dec) = 01 (bin)
Texel[3][0] = Color[(RowByte & 0xC0) >> 6];     // 0 (dec) = 00 (bin)
```

This code extracts the 2 bit index from the row's bit data and uses it to look up one of the four colors calculated earlier. The result is stored in a 4x4 texel color array. We would obviously have to do this three more times (for the three remaining rows) to complete the full decoding of the block.

**Note:** You might have noticed that the least significant bits in the byte describe the first texel for that image's row. This is not a typo, the image data is actually encoded in this way.

Take a moment to examine the following tables. They compare memory footprint between standard non-compressed textures, and their opaque DXT1 compressed counterparts. Remember that each DXT1 color block is 8 bytes in size.

Standard 16bit / 32bit Uncompressed Textures

Dimensions	Bits Per Pixel	Pixel Count	Size (Bytes)
128x128	16	16,384	32,768
128x128	32	16,384	65,536
256x256	16	65,536	131,072
256x256	32	65,536	262,144
512x512	16	262,144	524,288
512x512	32	262,144	1,048,576
1024x1024	16	1,048,576	2,097,152
1024x1024	32	1,048,576	4,194,304
2048x2048	16	4,194,304	8,388,608
2048x2048	32	4,194,304	16,777,216
4096x4096	16	16,777,216	33,554,432
4096x4096	32	16,777,216	67,108,864



Opaque DXT1 Compressed Textures

Dimensions	Block Count	Size (Bytes)
128x128	1,024 (32x32)	8,192
256x256	4,096 (64x64)	32,768
512x512	16,384 (128x128)	131,072
1024x1024	65,536 (256x256)	524,288
2048x2048	262,144 (512x512)	2,097,152
4096x4096	1,048,576 (1024x1024)	8,388,608

This is a significant reduction. Note as well that the storage space and bandwidth savings are exponential as the size of the texture increases. In addition, each block is exactly 64 bits in size. This allows for optimal (practically best case) transfer rate when sending the texture data to the card. As a general rule, we can use a compressed texture with dimensions (Width\*2) x (Height\*2) beyond those of an uncompressed 16-bit texture and take up no additional storage, with acceptable lossiness.

**Note:** Due to the fact that each block describes a 4x4 texel area, and because of other optimizations that may be performed, compressed textures must always use dimensions equal to  $(N^2)$ . This is true even in cases where the graphics adapter supports non-'power of two' image dimensions for uncompressed textures.

## Compressed Data Blocks - Alpha Data Layout

Although the layout of the color data stays basically the same between compressed texture formats (with one exception to be discussed shortly), we have several options when it comes to how we store alpha information and how it is interpolated.

### D3DFMT\_DXT1 - *Opaque or 1 Bit Alpha*

The DXT1 format has the ability to store a single bit of alpha information. This basically describes whether or not the resulting pixel should be rendered (i.e. totally transparent or totally opaque).

As discussed previously, we store 2 bits for each texel stored in the 4x4 color data block. In order to provide transparency, we will discard one of the interpolated color values in favor of having a 'free' index available (11<sub>(bin)</sub> or 3<sub>(dec)</sub>). Here we will specify whether a texel is transparent or not. So in this case we now have three color values: the two encoded into the block itself and a single interpolated value (created using  $\frac{1}{2}$  Color\_1 and  $\frac{1}{2}$  Color\_2). The consequence is some additional lossiness. Fortunately, this is determined on a block-by-block basis during compression, so only those blocks which happen to contain transparent pixels will suffer as a result.

To determine whether a compressed data block contains transparency information during decompression / rendering, rather than use a flag, this format uses a little hardcoded logic. If the block does not contain transparency information then Color\_0 will be greater than Color\_1. If it does contain transparency information then the situation is reversed and Color\_1 will be greater than Color\_0.



```
if ( (Color_0 > Color_1) || Format != D3DFMT_DXT1 )
{
    // No transparency, this block uses all four colors
    Color_2 = (2 * Color_0 + Color_1) / 3;
    Color_3 = (Color_0 + 2 * Color_1) / 3;
}
else
{
    // Contains transparency, use only three colors
    Color_2 = (Color_0 + Color_1) / 2;
    Color_3 = transparent;
}
```

Notice that the above ‘if’ statement takes into account the fact that this transparency technique applies **only** to the DXT1 compressed texture format. All other formats always assume four color encoding.

### D3DFMT\_DXT2 & D3DFMT\_DXT3 - *Explicit Alpha*

AAAA	AAAA	AAAA	AAAA
AAAA	AAAA	AAAA	AAAA
AAAA	AAAA	AAAA	AAAA
AAAA	AAAA	AAAA	AAAA
Color_0			
Color_1			
XX	XX	XX	XX
XX	XX	XX	XX
XX	XX	XX	XX
XX	XX	XX	XX

These two formats both store their color data in exactly the same way as the *opaque* version of the DXT1 format. However, they store their alpha values separately. These explicit alpha formats provide a unique 4 bit alpha value for each texel in the block. This data is stored in a separate 64-bit 4x4 **transparency block**. It is encoded *before* the color block, as seen in the image on the left.

When explicit alpha is used, the size of each data block (transparency block + color block) adds up to a total of 128 bits (16 bytes). This is obviously not as significant a space savings as DXT1, but when we need relatively accurate, explicit per-texel alpha values, these formats provide the best of both worlds.

For DXT2 and DXT3 formats, the 4-bit alpha encoding for each texel can be achieved using several methods, such as dithering or simply truncation using only the 4 most significant bits (4 through 7). The latter is the method DirectX Graphics uses when compressing alpha information using either of these two formats. Since we are truncating the alpha information into 4 bits, the texture is only capable of describing 16 unique alpha levels. During compression, any original alpha values that are less than 16 will become completely transparent (0).

Because the alpha values are explicit, and use 4 bits per texel, decoding the transparency block is relatively easy when using these formats. We read the entire row into a single WORD (2 bytes).

Let us assume that the row we are processing is split up into 4 bit chunks as [1101 0010 1100 0101]



```
RowWord = 53957; // Assume this is the first row
Alpha[0][0] = (RowWord & 0x000F) << 4;           // 80(dec) = 01010000(bin)
Alpha[1][0] = ((RowWord & 0x00F0) >> 4) << 4; // 192(dec) = 11000000(bin)
Alpha[2][0] = ((RowWord & 0x0F00) >> 8) << 4; // 32(dec) = 00100000(bin)
Alpha[3][0] = ((RowWord & 0xF000) >> 12) << 4; // 208(dec) = 11010000(bin)
```

As with the code used to extract the color indices, this process must also be repeated for the remaining three rows. Shifting each value to the left by 4 bits converts the 4-bit value back into a byte within the range [0, 255]. One potential optimization is to simplify the right and left shifts to one operation: (ex.  $>> 12$  followed by a  $<< 4$  is changed to a  $>> 8$ ).

**Note:** The least significant bits in the word describe the first alpha value for that physical texture row. This is not a typo, the alpha data is actually encoded in this way.

Remember that DXT2 has its color data encoded using pre-multiplied alpha and DXT3 does not. If you are encoding these formats manually, do not forget to do this multiplication.

### D3DFMT\_DXT4 & D3DFMT\_DXT5 - *Interpolated Alpha*

Alpha_0			
Alpha_1			
AAA	AAA	AAA	AAA
AAA	AAA	AAA	AAA
AAA	AAA	AAA	AAA
AAA	AAA	AAA	AAA
Color_0			
Color_1			
XX	XX	XX	XX
XX	XX	XX	XX
XX	XX	XX	XX
XX	XX	XX	XX

DXT4 and DXT5 also store their own transparency block encoded before the color block. For alpha, they will use interpolation in much the same way as the color block. There are two key differences. First, the two alpha values used to generate the interpolated values are stored as 8-bit (one byte) values. This allows a full range between 0 and 255. Second, each index in the bit-map uses 3 bits, providing up to 8 interpolated alpha values to be specified (0 through 7).

As with the interpolated color values, we will need to generate a certain number of alpha values that fall within the range specified by the Alpha\_0 and Alpha\_1 values contained within the transparency block. Because our data area uses 3-bit indices, we can address a maximum of 8 unique values. When these formats are decoded, we will generate 6 alpha values interpolated between the two values specified within the block. The following code snippet generates each of these 6 alpha values.

```
// Alpha_0 = First 8 bits of transparency block      - Bit code 000
// Alpha_1 = Second 8 bits of transparency block     - Bit Code 001
Alpha_2 = ( 6 * Alpha_0 +      Alpha_1 ) / 7; //    - Bit Code 010
Alpha_3 = ( 5 * Alpha_0 + 2 * Alpha_1 ) / 7; //    - Bit Code 011
Alpha_4 = ( 4 * Alpha_0 + 3 * Alpha_1 ) / 7; //    - Bit Code 100
Alpha_5 = ( 3 * Alpha_0 + 4 * Alpha_1 ) / 7; //    - Bit Code 101
Alpha_6 = ( 2 * Alpha_0 + 5 * Alpha_1 ) / 7; //    - Bit Code 110
Alpha_7 = (      Alpha_0 + 6 * Alpha_1 ) / 7; //    - Bit Code 111
```



Each of these values is distributed evenly between the Alpha\_0 and Alpha\_1 such that:

Alpha\_2 uses 6/7<sup>th</sup> Alpha\_0 and 1/7<sup>th</sup> Alpha\_1,  
Alpha\_3 uses 5/7<sup>th</sup> Alpha\_0 and 2/7<sup>th</sup> Alpha\_1,  
Alpha\_4 uses 4/7<sup>th</sup> Alpha\_0 and 3/7<sup>th</sup> Alpha\_1,  
...  
As with the color data, there is a special case which can be used on a block-by-block basis. If Alpha\_1 is greater than Alpha\_0, we assume a 6-alpha block, as opposed to the 8-alpha block described above:

```
// Alpha_0 = First 8 bits of transparency block      - Bit code 000
// Alpha_1 = Second 8 bits of transparency block     - Bit Code 001
if ( Alpha_0 > Alpha_1 )
{
    Alpha_2 = ( 6 * Alpha_0 +     Alpha_1 ) / 7; //  - Bit Code 010
    Alpha_3 = ( 5 * Alpha_0 + 2 * Alpha_1 ) / 7; //  - Bit Code 011
    Alpha_4 = ( 4 * Alpha_0 + 3 * Alpha_1 ) / 7; //  - Bit Code 100
    Alpha_5 = ( 3 * Alpha_0 + 4 * Alpha_1 ) / 7; //  - Bit Code 101
    Alpha_6 = ( 2 * Alpha_0 + 5 * Alpha_1 ) / 7; //  - Bit Code 110
    Alpha_7 = (     Alpha_0 + 6 * Alpha_1 ) / 7; //  - Bit Code 111
}
else
{
    Alpha_2 = ( 4 * Alpha_0 +     Alpha_1 ) / 5; //  - Bit Code 010
    Alpha_3 = ( 3 * Alpha_0 + 2 * Alpha_1 ) / 5; //  - Bit Code 011
    Alpha_4 = ( 2 * Alpha_0 + 3 * Alpha_1 ) / 5; //  - Bit Code 100
    Alpha_5 = (     Alpha_0 + 4 * Alpha_1 ) / 5; //  - Bit Code 101
    Alpha_6 = 0;                                //  - Bit Code 110
    Alpha_7 = 255;                             //  - Bit Code 111
}
```

As you can see, this approach creates only 4 interpolated values and explicitly defines the extremes. This can be used in cases where we want no blending to occur (total opacity or transparency).

Decoding the data area for three-bits per texel can be a little bit tricky. The easiest way is to read the entire data area (48 bits) into a single 64-bit variable (using an `_int64`) and parse it.

```
_int64 BlockData = 195010219826458; // Assume this is the first row

// In this example, BlockData binary is shown below
// 101 100 010 101 110 001 011 010 111 000 010 101 110 100 011 010
TAlpha[0][0] = Alpha[(BlockData & 0x7)]; // 2(dec) = 010(bin)
BlockData >>=3;
TAlpha[1][0] = Alpha[(BlockData & 0x7)]; // 3(dec) = 011(bin)
BlockData >>=3;
TAlpha[2][0] = Alpha[(BlockData & 0x7)]; // 4(dec) = 100(bin)
BlockData >>=3;
TAlpha[3][0] = Alpha[(BlockData & 0x7)]; // 6(dec) = 110(bin)
BlockData >>=3;
...
```



Again, this process repeats for the remaining three rows. As we saw earlier, we start with the least significant bits of the data type being processed. We strip off the 3 least significant bits used in the previous calculation, so that the next 3 bits in the data area become the least significant bits in their place. This is also a valid way of decoding the other data blocks and is extremely useful if you wish to place your extraction code in a loop.

Finally, remember that DXT4 color data is encoded using pre-multiplied alpha and DXT5 is not.

### Texture Coordinate Transformation

Each of the eight texture stages owns a 4x4 matrix that can be used to apply transformations to the texture coordinates associated with that stage. This provides an easy way to animate texture coordinates at run-time as they are pumped through the cascade. We can think of the texture coordinates as normal 2D vectors, such that multiplying them with the texture matrix works in exactly the same way as multiplying our vertex positions with a 4x4 matrix (see Chapter 1). By default, texture coordinate transformations are disabled for each stage and the matrix stored at each stage is an identity matrix.

Our first concern is figuring out how to multiply a 2D texture coordinate with a 4x4 matrix (the 2D coordinate is a 1x2 matrix and the inner dimensions do not match the 4x4 matrix). Recall that we used homogeneous coordinates in the form (x,y,z,1) to solve this problem for 3D vectors. This also gave us the ability to place addition/subtraction into the matrix for translations. The DirectX pipeline uses a similar approach with texture coordinates. Note that even though we have only used 2D texture coordinates so far, you can also use 1D, 3D, and even 4D coordinates with DirectX. If N is the dimension of the texture coordinates, the coordinate set will be padded to a 4D vector prior to multiplication with the texture matrix such that component N+1 will have the value 1. The remaining values in the vector are padded to 0.

Let us assume that we have a 2D set of texture coordinates in our vertex structure ( $u, v$ ). If texture transformations are enabled in the stage that uses those coordinates, the texture coordinates will be padded to 4D texture coordinates for the matrix multiplication. The 2D texture coordinate would now be  $(u, v, 1, 0)$ . A 1D texture coordinate ( $u$ ) would be padded to  $(u, 1, 0, 0)$ . A 3D coordinate  $(u, v, w)$  would be padded to  $(u, v, w, 1)$ .

### 2D Texture Coordinate Translation Matrix

**Texture Matrix**

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$



Let us imagine that we have a texture matrix assigned to stage 0 as an identity matrix.

#### Texture Matrix

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

When we are using 3D vertex positions ( $x, y, z, 1$ ) we know that the translation vector should go in the fourth row. However, when using 2D texture coordinates ( $u, v, 1, 0$ ), this will not suffice:

$$U = u * m_{11} + v * m_{21} + 1 * m_{31} + 0 * m_{41}$$

$$V = u * m_{12} + v * m_{22} + 1 * m_{32} + 0 * m_{41}$$

$$U = u * m_{11} + v * m_{21} + 1 * m_{31}$$

$$V = u * m_{12} + v * m_{22} + 1 * m_{32}$$

Instead we see that since the third component of the texture coordinate is set to 1 prior to the multiplication. Thus, the third row in the matrix provides us with the ability to add or subtract values from the input components.

Assume UV components of (0.5, 0.9). If we wanted to translate our U coordinates by 2 and our V coordinates by -5 we would generate the following matrix:

#### 2D Texture Translation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 2 & -5 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$U = 0.5 * 1 + 0.9 * 0 + 1 * 2$$

$$U = \mathbf{2.5}$$

$$V = 0.5 * 0 + 0.9 * 1 + 1 * -5$$

$$V = \mathbf{-4.1}$$



Next we see how to setup a matrix to translate a 1D texture ( $u, 1, 0, 0$ ) by  $\mathbf{du}$  and  $\mathbf{dv}$ :

### 1D Texture Translation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ du & dv & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



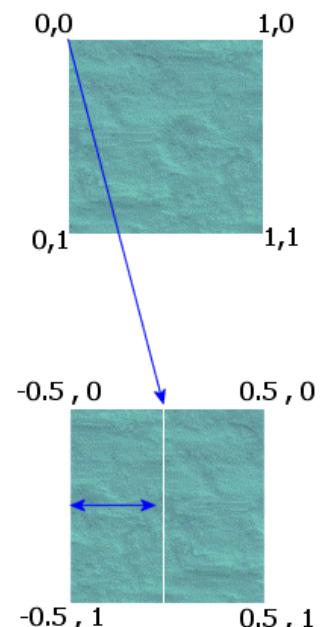
In Lab Project 6.3 we will use texture coordinate animation to make the water underneath the wooden bridge look like it is flowing. We do this by setting the texture stage matrix to a translation matrix that continuously increments the translation amount of the texture coordinates each time the face is rendered. This offsets the texture coordinates a little more each frame.

Before rendering each water polygon, we will set the texture stage texture matrix to increment the U component of the texture

coordinates. Since the default texture addressing mode wraps coordinates outside the  $[0, 1]$  range, as the texture is tiled we will see a scrolling effect. The top part of the image on the right shows the water texture mapped to a quad in its entirety. The bottom image shows what the quad would look like if we subtracted 0.5 from the U coordinate using the translation matrix. The white line lets us see where one tile ends and the next one begins, but is for the purposes of the diagram only – it will be a seamless join in the application.

If we subtract a small amount from the U coordinate each frame, the white line would scroll slowly from right to left (from 0.0 to 1.0). Once it reached 1.0 it would just wrap around to 0.0. This means that our water texture will look like it is endlessly scrolling. In the demo we only translate the U coordinates because the room design is such that the water flow is aligned with our world X axis. If the room was oriented diagonally between the X and Z axes, we would have translated both the U and V coordinates by equal measure.

### Scrolling the Texture





We can also perform texture scaling and rotations on our UV coordinates. The following matrix scales the u coordinate by 5 and the v coordinate by 2.

#### A 2D texture coordinate scale matrix

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Setting up the Texture Transformation

To enable texture transformation we first use the `D3DTTS_TEXTURETRANSFORMFLAGS` texture stage state as shown below:

```
pDevice->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, flags);
```

We use this function to inform the device about the texture coordinate size. The 4D texture coordinate will be padded and the output will be trimmed accordingly. We specify one of the members of the `D3DTEXTURETRANSFORMFLAGS` enumeration:

```
typedef enum _D3DTEXTURETRANSFORMFLAGS
{
    D3DTTFF_DISABLE = 0,
    D3DTTFF_COUNT1 = 1,
    D3DTTFF_COUNT2 = 2,
    D3DTTFF_COUNT3 = 3,
    D3DTTFF_COUNT4 = 4,
    D3DTTFF_PROJECTED = 256,
    D3DTTFF_FORCE_DWORD = 0x7fffffff
} D3DTEXTURETRANSFORMFLAGS;
```

### **D3DTTFF\_DISABLE**

Texture transformations are disabled. This is the default state for each texture stage.

### **D3DTTFF\_COUNT1**

This tells the device to trim the 4D texture coordinate output from the texture matrix multiplication to its first component. All other components are discarded. The rasterizer now knows to expect 1D coordinates.



### D3DTFF\_COUNT2

This tells the device to trim the 4D texture coordinate output from the texture matrix multiply to its first two components. All other components are discarded. The rasterizer now knows to expect 2D coordinates.

### D3DTFF\_COUNT3

This tells the device to trim the 4D texture coordinate output from the texture matrix multiply to its first three components. The 4th component is discarded. The rasterizer now knows to expect 3D texture coordinates.

### D3DTFF\_COUNT4

The 4D vector output from the matrix multiplication is passed straight to the rasterizer.

### D3DTFF\_PROJECTED

This flag can be combined with any of the above flags. The 4D texture coordinates output from the matrix are all divided by the last component before being passed to the rasterizer. For example, if this flag is specified with the `D3DTFF_COUNT3` flag, the first and second texture coordinates are divided by the third coordinate before being passed to the rasterizer. This flag is used for projective texturing. We will discuss this feature in detail during the next course in this series.

Thus, to enable transformations in stage 0 for 2D texture coordinates:

```
pDevice->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTFF_COUNT2);
```

To set the transformation matrix we use the `SetTransform` function, passing one of the `D3DTS_TEXTURE0 - D3DTS_TEXTURE7` transform states. The number on the end describes which texture stage we are setting matrix for. The following code creates a texture translation matrix that scrolls the U coordinates by 0.1 and assigns it to stage 0.

```
D3DXMATRIX mat;
D3DXMatrixIdentity (&mat);
Mat._31 = 0.1;
pDevice->SetTransform ( D3DTS_TEXTURE0 , &Mat );
```



## The IDirect3DTexture9 Interface

It is tempting to think of the IDirect3DTexture9 interface as an interface to an object that encapsulates an image stored on a surface. However, this is not quite true. As we have already discovered, the texture object can actually manage multiple surfaces called MIP maps. (It would be appropriate however to think of the IDirect3DSurface9 interface as an object that encapsulates a single image surface.) Each MIP mapped texture object is really an object that manages an array of surface objects. Each of these surfaces can be accessed and operated on using the methods of the IDirect3DSurface9 interface. With the release of DirectX 9.0 the IDirect3DTexture9 interface was extended, allowing you to lock any surface belonging to a texture without the need to use an intermediate IDirect3DSurface9 interface. The IDirect3DSurface9 interface is nevertheless quite important. Not only is it used to operate on surfaces that are not textures (such as the frame buffer for example), there are still D3DX functions that operate on IDirect3DSurface9 rather than IDirect3DTexture9. Since IDirect3DTexture9 includes a function to return an IDirect3DSurface9 interface for any one of its MIP levels, we can use these D3DX functions with individual texture surfaces with no trouble.

Let us briefly examine some of the methods exposed by the IDirect3DTexture9 interface. Some of these you will use very rarely, while others may be used quite regularly.

### **LockRect(UINT Level, D3DLOCKED\_RECT \*pLockedRect, CONST RECT \*pRect, DWORD Flags);**

This function locks a rectangular area on the texture and returns a pointer to the first pixel in the top left corner of the rectangle. If the RECT pointer is set to NULL, a pointer to the first pixel in the top left corner of the entire surface is returned. This pointer can be used to read and write directly to or from the surface. The Level parameter specifies the MIP map level you wish to lock. The **D3DLOCKED\_RECT** structure is filled with a pointer to the bits of the image surface and the pitch of the surface.

### **UnlockRect(UINT Level);**

When a surface has been locked you must remember to unlock it after you have finished reading or writing. We pass in the MIP level we wish to unlock.

### **DWORD GetLevelCount(VOID);**

This function returns the number of MIP levels in the texture.

### **GetLevelDesc(UINT Level, D3DSURFACE\_DESC \*pDesc);**

This function is used to get specific information about a particular surface level in a texture. The Level parameter specifies the MIP level we wish to inquire about. The **D3DSURFACE\_DESC** will be filled with the width, height and format information about that surface level. The **D3DSURFACE\_DESC** structure is shown below.



```
typedef struct _D3DSURFACE_DESC
{
    D3DFORMAT Format;
    D3DRESOURCETYPE Type;
    DWORD Usage;
    D3DPOOL Pool;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    DWORD MultiSampleQuality;
    UINT Width;
    UINT Height;
} D3DSURFACE_DESC;
```

The `D3DSURFACE_DESC` is used to contain information about all sorts of surfaces. For example, it could be used for frame buffer information as well as texture information. The Format, Width, Height, Usage, and Pool members have been discussed in detail already. The MultiSampleType and MultiSampleQuality members are only relevant for surfaces that can be used as render targets (i.e. frame buffer or render target textures). The Type parameter identifies the type of surface this is. For a standard 2D texture surface, this will always be `D3DRTYPE_TEXTURE`. The other `D3DRESOURCETYPE` members are shown below and some of the following types we have not yet covered.

```
typedef enum _D3DRESOURCETYPE
{
    D3DRTYPE_SURFACE = 1,
    D3DRTYPE_VOLUME = 2,
    D3DRTYPE_TEXTURE = 3,
    D3DRTYPE_VOLUMETEXTURE = 4,
    D3DRTYPE_CUBETEXTURE = 5,
    D3DRTYPE_VERTEXBUFFER = 6,
    D3DRTYPE_INDEXBUFFER = 7,
    D3DRTYPE_FORCE_DWORD = 0x7fffffff
} D3DRESOURCETYPE;
```

### **GetSurfaceLevel(UINT Level, IDirect3DSurface9 \*\*ppSurfaceLevel);**

Individual texture MIP surfaces can be retrieved using this function. It will return a pointer to an `IDirect3DSurface9` interface based on the MIP surface level specified in the `Level` parameter.

### **void PreLoad (void);**

We use this function to inform the DirectX memory manager that the texture will soon be used for rendering. This function is used for managed textures only. A managed texture resource may have been temporarily unloaded from video memory to make room for other textures. Calling this function forces the texture manager to upload the texture to video memory immediately. Often when textures are first created, they will not be loaded up to video memory until the first time they are used. This can result in stutters or delays at the start of an application. Calling `PreLoad` on all of your managed textures can help minimize this occurrence by making sure the textures are in video memory before rendering.



### **void AddDirtyRect(CONST RECT \*pDirtyRect);**

When using managed textures, whenever a texture is locked, the rectangle that was locked is flagged as dirty. When the texture is unlocked, the texture manager updates the video memory copy of the texture with the new image data from the system memory copy. When we are not using managed textures, we manage this system memory copy of the texture ourselves. The IDirect3DDevice9::UpdateTexture function can be used to copy data from the system memory texture to the video memory texture. This function copies all the dirty rectangles of the system memory texture. We can use the AddDirtyRect function to specify additional dirty regions. This forces either the memory manager or the IDirect3DDevice9::UpdateTexture function to push the system memory copy to the card even if the system does not officially consider them to be dirty.

### **DWORD SetPriority(DWORD PriorityNew);**

This function is used to set the priority of a managed texture. By default, all managed textures begin life with a priority level of 0. When the memory manager is deciding which textures to remove from video memory to make room for newly requested ones, it will choose textures with a lower priority. This means that we can assign a priority number  $> 0$  to make sure that a texture is not removed from memory. This can be useful if we know a texture will be used lots of times throughout the scene. The function returns the previous priority level of the texture. This is only used with managed textures.

### **HRESULT IDirect3DDevice9::UpdateTexture(IDirect3DBaseTexture9 \*pSourceTexture, IDirect3DBaseTexture9 \*pDestinationTexture);**

This function is used to copy a system memory texture to a texture in the default memory pool. Typically you will use this function when updating textures that are not in the **D3DPOOL\_MANAGED** resource pool. This function will copy only the dirty regions of the texture. Locking a texture makes the locked rectangle dirty, provided that the texture rectangle was not locked using the **D3DLOCK\_NO\_DIRTY\_UPDATE** or **D3DLOCK\_READONLY** flags. Note that using the IDirect3DDevice9::AddRect function is another way to add dirty rectangles to a texture.

The source texture must have been created in the **D3DPOOL\_SYSTEMMEM** memory pool and the destination surface must have been created in the **D3DPOOL\_DEFAULT** memory pool. This function cannot be used for managed textures. There are a few other semantics that must be obeyed when using this function and the full details are discussed in the DirectX SDK documentation.



## The IDirect3DSurface9 Interface

IDirect3DSurface9 is an interface used to interact with an individual surface, such as a single MIP surface in a texture object. This is an important interface because not all surfaces are texture surfaces. The frame buffer, front buffer, and depth buffer are good examples. They are not necessarily limited to the same size and shape restrictions that texture surfaces may be limited to. Note that although we cannot physically write to the front buffer for example, the IDirect3DDevice9::GetFrontBufferData makes a copy of the front buffer and returns an IDirect3DSurface9 interface. This is ideal for grabbing a screen shot and saving it to a file.

The IDirect3DSurface9 interface also includes many useful methods that are not available through IDirect3DTexture9. For example, let us imagine that we have a texture with 10 MIP levels and we wish to write some text to the top level surface. It would be handy if we could use the Win32 GDI text output functions since they are familiar and easy to use (assuming speed is not a concern). IDirect3DSurface9 exposes a GetDC function that allows us to do just that. Once you have the device context, you can use GDI functions to render to the surface as though it were a window. For example:

```
// assumes that pTexture is a valid IDirect3DTexture9 interface...
IDirect3DSurface9 *pSurface;
HDC hDC;

pTexture->GetSurfaceLevel(0, &pSurface);
pSurface->GetDC( &hDC );

::SetTextColor( hDC, 0xFFFFFFFF );
::TextOut( hDC , 10 , 10 , "Hello World" , 11);

pSurface->ReleaseDC( hDC );
pSurface->Release();

D3DXFilterTexture(pTexture, NULL, 0, D3DX_DEFAULT);
```

The first thing we do is retrieve an IDirect3DSurface9 for the current texture level (0 in this example). Once we have the surface interface, we use the GetDC function to retrieve a device context handle for drawing to the surface. In this example we set the text color of the device context to white and then print the text ‘Hello World’ using the 2D device context coordinates X=10, Y=10. Once we are done writing to the surface, we release the device context and the surface interface as they are no longer needed. Finally, we called the D3DXFilterTexture function (see next section) to ensure that the changes to the top level surface are downsampled to all subsequent MIP levels.

**Note:** While the GDI is too slow to be used in a time critical situation, it can be used during initialization of your application to write to or copy from texture surfaces.



## IDirect3DDevice9 Surface Functions

The IDirect3DDevice9 interface has a few useful methods for working with surfaces. They are listed below along with a description of their use, followed by a description of their parameters.

### **HRESULT ColorFill(IDirect3DSurface9 \*pSurface, CONST RECT \*pRect, D3DCOLOR color);**

The IDirect3DDevice9::ColorFill function is used to fill a surface (or a rectangular portion thereof) with a specified color.

#### *IDirect3DSurface9 \*pSurface*

We use this parameter to pass an IDirect3DSurface9 pointer to the surface we wish to fill.

#### *CONST RECT \*pRect*

The rectangular region on the surface that we wish to be filled. If we pass NULL, then the entire surface will be filled with the color.

#### *D3DCOLOR Color*

The color we will use for the fill operation.

```
//ex.Fill the entire surface with opaque green.  
pDevice->ColorFill (pSurface, NULL, 0xFF00FF00);
```

### **HRESULT CreateOffscreenPlainSurface(UINT Width, UINT Height, D3DFORMAT Format, DWORD Pool, IDirect3DSurface9\*\* ppSurface, HANDLE\* pHandle);**

This function creates a new surface object which we will typically want to place in system memory. When we create an offscreen plain surface, we are not limited by any of the device restrictions that apply to textures. This makes them ideal for storing images that are too large to fit into a texture.

For example, if we had a title screen image that was 1024x768 and we loaded this as a texture on a device that was limited to textures of 256x256 in size, the D3DXCreateTextureFromFile function would automatically resize the image so that it would fit onto a 256x256 texture surface. When the image was stretched over the entire screen, magnification artifacts would be the result. So instead, CreateOffscreenPlainSurface should be used to create a blank surface of the correct size. We can then use the D3DXLoadSurfaceFromFile function to load the image data and the IDirect3DDevice9::UpdateSurface function to copy the image from the offscreen surface to the frame buffer surface. If the frame buffer and offscreen surface are different sizes, we can use the IDirect3DDevice9::StretchRect function to copy from the offscreen surface to the frame buffer. This performs the color conversion and image resizing to fit the frame buffer.

Remember that this function only creates the surface. Your application is responsible for filling it with image data.



### ***UINT Width***

### ***UINT Height***

The first two parameters specify the pixel width and pixel height of the surface to be created.

### ***D3DFORMAT Format***

This is the pixel format of the surface and it must be a valid format that the device supports. We verify this using the IDirect3D9::CheckDeviceType function as shown below.

```
DWORD UsageFlags = 0;
D3DFORMAT CheckFormat = D3DFMT_X4R4G4B4;
pD3D->CheckDeviceFormat(Adapter, DeviceType, AdapterFormat, UsageFlags,
                         D3DRTYPE_SURFACE, CheckFormat);
```

### ***DWORD Pool***

This is the memory pool we would like the surface to be created in. This will be either **D3DPOOL\_SYSTEMMEM** or **D3DPOOL\_DEFAULT** depending on how we intend to use the surface.

### ***IDirect3DSurface9 \*\*pSurface***

If surface creation is successful, this will point to the new IDirect3DSurface9 interface.

### ***HANDLE \*pHandle***

Reserved. Set this parameter to NULL

Typically you will find that you will be creating offscreen surfaces for holding images like title screens. In this case, you will want the offscreen surface format to match the back buffer surface format because no color conversion would have to take place. Also, no stretching or shrinking of the image is performed so we must copy an area of MxN pixels from the source surface to an area of MxN pixels on the destination surface. If you wanted to use IDirect3DDevice9::UpdateSurface to copy the offscreen surface to the frame buffer, then you would probably want to create your offscreen surface using an approach like this:

```
IDirect3DSurface9 pMySurface;
pDevice->CreateOffscreenPlainSurface(BackBufferWidth, BackBufferHeight,
                                       BackBufferFormat, D3DPOOL_SYSTEMMEM,
                                       &pMySurface, NULL);
```

Note that when we use the UpdateSurface function, we must create our offscreen surface in the **D3DPOOL\_SYSTEMMEM** pool.

The problem with this using this method to present a title screen is that the surface is bound to the dimensions frame buffer. If the frame buffer is resized, we will need to create a new offscreen surface matching the new dimensions. The IDirect3DDevice9 interface has another surface-to-surface copying function called StretchRect which is a lot more flexible. We will cover this function shortly.



---

**HRESULT UpdateSurface(IDirect3DSurface9\* pSourceSurface, CONST RECT\* pSourceRect,  
IDirect3DSurface9\* pDestinationSurface, CONST POINT\* pDestinationPoint);**

If you have ever used DirectX prior to version 9.0, you may recall a function called CopyRects. It was used to do a direct bit copy from one image surface to another. This function has been replaced in DirectX 9 with the ID3DDevice9::UpdateSurface function. The pixel formats of the source and destination surfaces must be the same or the copy will fail. The function performs no color conversion between surface formats and no shrinking or stretching to fit the destination surface dimensions.

***IDirect3DSurface9\* pSourceSurface***

This is a pointer to the surface that will be the source for the copy operation. This surface must have been created in the **D3DPOOL\_SYSTEMMEM** pool for the copy to be successful. It must not be currently locked or have any outstanding device contexts.

***CONST RECT\* pSourceRect***

If this parameter is NULL then the entire source surface will be copied to the destination surface. If not, then this is a pointer to a RECT structure that defines a rectangular region on the surface that should be copied. The top left corner of the source rectangle will be mapped to the specified point on the destination surface (4<sup>th</sup> parameter). Be sure to verify that the copied pixels will fit onto the destination surface starting at this position.

***IDirect3DSurface9\* pDestinationSurface***

This is a pointer to the destination surface. The surface must have been created in the **D3DPOOL\_DEFAULT** pool. This surface must not be currently locked or have any outstanding device contexts.

***CONST POINT\* pDestinationPoint***

This is a 2D point on the destination surface where the source pixels will be copied. The top left corner pixel of the source rectangle in the source image will be mapped to this point on the destination surface. If you specify NULL, then the source rectangle will be copied starting at the top left corner of the destination surface.



## Surface Types

The following table contains permissible source/destination surface combinations for various surface types. It is assumed that the source surfaces are all created in the **D3DPOOL\_SYSTEMMEM** pool and the destination surfaces have been created in the **D3DPOOL\_DEFAULT** pool.

**Key:** Source Surface Type      Destination Surface Type

	Texture Surface	Texture Render Target	Render Target Surface	Off-screen Plain
Texture Surface	Yes	Yes	Yes*	Yes
TextureRender Target	No	No	No	No
Render Target Surface	No	No	No	No
Off-screen Plain	Yes	Yes	Yes	Yes

\* If the device does not support copying from a texture surface to a render target surface (such as the frame buffer) then this will be emulated using a lock and pixel copy operation.

Finally, there are two additional rules to keep in mind when copying surfaces:

- Neither the source surface nor the destination surface can have been created with multisampling capabilities. The only valid flag for both surfaces is **D3DMULTISAMPLE\_NONE**.
- The pixel formats of both surfaces must match and they must not be a depth stencil format.

---

**HRESULT StretchRect(IDirect3DSurface9 \**pSourceSurface*, CONST RECT \**pSourceRect*,  
IDirect3DSurface9 \**pDestSurface*, CONST RECT \**pDestRect*,  
D3DTEXTUREFILTERTYPE *Filter*);**

---

DirectX 9.0 introduced a new 2D surface copying function called `IDirect3DDevice9::StretchRect`. This function is used primarily for copying images from surfaces to render target surfaces. Unlike the `UpdateSurface` function, the pixel formats of the source and destination pixel formats do not have to match because color conversion will be applied if necessary. Also, the source and destination rectangles do not require matching dimensions either. This allows us to magnify/minify the source image onto the destination surface using a specified filtering technique. While the function looks simple enough, there are some rules and restrictions that must be considered. We will discuss these as we move along.

***IDirect3DSurface9 \*pSourceSurface***

The source surface of the copy operation. This surface must have been created in the **D3DPOOL\_DEFAULT** memory resource pool.



#### ***CONST RECT \*pSourceRect***

The source rectangle that marks a region on the source surface that is to be copied onto the destination surface. If this parameter is NULL, the entire contents of the surface will be copied.

#### ***IDirect3DSurface9 \*pDestSurface***

This is the destination surface for the copy operation. This surface must have been created in the **D3DPOOL\_DEFAULT** memory resource pool and will typically be a render target such as the frame buffer or a texture that has been created as a render target. This surface need not match the pixel format of the source surface. This cannot be the same surface as the source surface.

#### ***CONST RECT \*pDestRect***

The destination rectangle on the destination surface. The pixels that fall within the source rectangle on the source surface will be copied, color converted if necessary, and resized to fit within this rectangle on the destination surface. If this is set to NULL, the source rectangle will be resized to fill the entire destination surface.

#### ***D3DTEXTUREFILTERTYPE Filter***

If the pixels copied from the source image need to be resized to fit the destination rectangle, this parameter describes one of three possible filter types used to reduce aliasing. Possible values are **D3DTEXF\_NONE**, **D3DTEXF\_POINT** or **D3DTEXF\_LINEAR**. If **D3DTEXF\_NONE** is specified, the driver will choose a filtering algorithm. If you intend to explicitly set a filter type you should check that the filter type is supported for the device. The **D3DCAPS9** structure has a member called **StretchRectFilterCaps**. You can check one of the following filter flags for support:

D3DPFILTERCAPS_MINPOINT	Device supports point-sample filtering for minifying rectangles. This filter type is requested by calling <b>IDirect3DDevice9::StretchRect</b> using <b>D3DTEXF_POINT</b> .
D3DPFILTERCAPS_MAGPOINT	Device supports point-sample filtering for magnifying rectangles. This filter type is requested by calling <b>IDirect3DDevice9::StretchRect</b> using <b>D3DTEXF_POINT</b> .
D3DPFILTERCAPS_MINLINEAR	Device supports bilinear interpolation filtering for minifying rectangles. This filter type is requested by calling <b>IDirect3DDevice9::StretchRect</b> using <b>D3DTEXF_LINEAR</b> .
D3DPFILTERCAPS_MAGLINEAR	Device supports bilinear interpolation filtering for magnifying rectangles. This filter type is requested by calling <b>IDirect3DDevice9::StretchRect</b> using <b>D3DTEXF_LINEAR</b> .



The following code shows how we could perform a copy from one surface to another using a driver selected filtering type.

```
pDevice->StretchRect(pSrcSurface, NULL, pDestSurface, NULL, D3DTEXF_NONE);
```

There are a few issues we need to look out for if we are converting between different image formats. First, once we have checked that the surface formats are supported by the device using the CheckDeviceFormat function, we must check that the device can handle color conversion between the two formats that we are using. Just because the device supports both formats does not mean that it supports copying from one to the other. The following code shows how we can use the IDirect3D9::CheckDeviceFormatConversion function to test if color conversion is supported between an A8R8G8B8 and a X4R4G4B4 surface.

```
if(SUCCEEDED(pDevice->CheckDeviceFormatConversion(Adapter, DeviceType,
                                                    D3DFMT_X8R8G8B8, D3DFMT_X4R4G4B4)))
{
    // Success
}
```

There are also restrictions on the combinations of source and destination surfaces that can be used together. Some of this depends on the driver. DX8 drivers cannot use a normal texture surface as a source surface for the copy operation but a DX9 driver can. Furthermore, a DX8 driver can only use render target textures as source surfaces if no stretching/shrinking of the image is required. While there is no easy way to know whether our application is running on DX8 or DX9 drivers, we can check the D3DCAPS9::DevCaps2 member for **D3DDEVCAPS2\_CAN\_STRETCHRECT\_FROM\_TEXTURES**. This tells us whether the device supports using a texture surface as a source surface when calling StretchRect. DX8 drivers will not, whilst DX9 drivers will.

```
D3DCAPS9 Caps;
pDevice->GetDeviceCaps( &Caps );
if(Caps.DevCaps2 & D3DDEVCAPS2_CAN_STRETCHRECT_FROM_TEXTURES)
{
    // DirectX9 Driver -- supports texture surfaces as source surfaces
}
```

The following table shows the surface type combinations that can be used for both DX8 and DX9 drivers, with and without pixel resizing.



Key : Source Surface Type      Destination Surface Type

DX8 Driver (No Stretching)	Texture Surface	Texture Target	Render Surface	Target	Off-screen Plain
Texture Surface	No	No	No	No	No
Texture Render Target	No	Yes	Yes	No	No
Render Target Surface	No	Yes	Yes	No	No
Off-screen Plain	Yes	Yes	Yes	Yes	Yes
DX8 Driver (Stretching)	Texture Surface	Texture Target	Render Surface	Target	Off-screen Plain
Texture Surface	No	No	No	No	No
Texture Render Target	No	No	No	No	No
Render Target Surface	No	Yes	Yes	No	No
Off-screen Plain	No	Yes	Yes	No	No
DX9 Driver (No Stretching)	Texture Surface	Texture Target	Render Surface	Target	Off-screen Plain
Texture Surface	No	No	No	No	No
Texture Render Target	No	Yes	Yes	No	No
Render Target Surface	No	Yes	Yes	No	No
Off-screen Plain	Yes	Yes	Yes	Yes	Yes
DX9 Driver (Stretching)	Texture Surface	Texture Target	Render Surface	Target	Off-screen Plain
Texture Surface	No	Yes	Yes	No	No
Texture Render Target	No	Yes	Yes	No	No
Render Target Surface	No	Yes	Yes	No	No
Off-screen Plain	No	Yes	Yes	No	No

The IDirect3DDevice9::StretchRect function can fail for a number of reasons:

- If pSourceSurface and pDestSurface are the same.
- If stretching or shrinking is involved and either surface has a DXTn compressed format.
- If the source surface is multisampled.
- If the destination surface is an off-screen plain surface but the source is not.
- If the destination surface is an off-screen plain surface and stretching is involved.

For more information, please consult the DirectX 9 SDK documentation.

**HRESULT GetBackBuffer(UINT iSwapChain, UINT BackBuffer,  
D3DBACKBUFFER\_TYPE Type, IDirect3DSurface9 \*\*ppBackBuffer);**

We use the IDirect3DDevice9::GetBackBuffer function to retrieve a surface interface to the frame buffer or to any surface in the back buffer swap chain. You could then call the surface interface GetDC function to write some text to the back buffer or perhaps call the IDirect3DDevice9::StretchRect function to copy an image surface to the frame buffer.



#### ***UINT iSwapChain***

This is an integer specifying the swap chain. It is possible to create multiple swap chains that can be used as rendering targets. Specifying 0 selects the swap chain connected to the device which contains the frame buffer setup at device creation.

#### ***UINT BackBuffer***

This parameter specifies the number of the surface in the swap chain that we wish to retrieve. Specifying index 0 returns an interface for the current frame buffer.

#### ***D3DBACKBUFFER\_TYPE Type***

This parameter must be set to `D3DBACKBUFFER_TYPE_MONO` in DirectX 9.

#### ***IDirect3DSurface9 \*\*ppBackBuffer***

The address of an `IDirect3DSurface9` interface that will point to a valid surface interface if the function is successful. Remember to release the surface interface after you have finished using it or else you will cause a memory leak.

### **HRESULT GetFrontBufferData(UINT iSwapChain, IDirect3DSurface9 \*pDestSurface);**

This function retrieves a copy of the front buffer. Due to its slow speed given the video memory read, this function should not be used in performance critical code. Remember to release the surface interface after you have finished using it or else you will cause a memory leak.

#### ***UINT iSwapChain***

Integer specifying the swap chain. This will usually be zero.

#### ***IDirect3DSurface9 \*pDestSurface***

This is an interface to a surface which has already had its buffer created by the application. This function does not create the surface memory buffer, it only copies the data into it. Thus it is the application's responsibility to allocate the surface correctly. It should be a surface created in the `D3DPOOL_SYSTEMMEM` resource pool and the pixel format of this surface should be 32-bit `D3DFMT_A8R8G8B8`. It must also be large enough to hold the image data. If this is a windowed device then the surface should be the size of the entire desktop, if it is a fullscreen device, it should match the dimensions of the current adapter mode.



## D3DX Texture Functions

D3DX includes a number of functions that can be used to assist with common texture housekeeping tasks. We already discussed the D3DX texture loading functions, but there are a few more worth mentioning.

**D3DXFilterTexture(LPDIRECT3DBASETEXTURE9 *pBaseTexture*,  
CONST PALETTEENTRY \**pPalette*, UINT *SrcLevel*, DWORD *MipFilter*);**

There may be times when you need to force MIP map image regeneration. This will be the case in Lab Project 6.4 when we use GDI to write to a texture surface. Once we write to the top level surface, we need the changes to be filtered down to all subsequent MIP levels. This function provides the solution. We pass a pointer to the texture we wish to have refiltered as the first parameter.

**Note:** LPDIRECT3DBASETEXTURE9 is a pointer to an IDirect3DBaseTexture9 interface. This is the interface from which all other texture interfaces are derived.

The second parameter is used only with palletized textures (we will simply set it to NULL). The third parameter specifies the MIP level at which the generation should begin. For example, if we specified 3 then the image in MIP level 3 would be filtered down to the subsequent MIP levels. Specifying 0 will ensure that the top level image is filtered down through all levels. The last parameters describes the filter we would like the function to use when downsampling. This can be a combination of one or more **D3DX\_FILTER** members as we discussed when examining the **D3DXCreateTextureFromFileEx** function. Specifying **D3DX\_DEFAULT** for this parameter is the equivalent of specifying **D3DX\_FILTER\_BOX** if the texture size is a power of two, and **D3DX\_FILTER\_BOX | D3DX\_FILTER\_DITHER** otherwise.

**D3DXSaveTextureToFile(LPCTSTR *pDestFile*, D3DXIMAGE\_FILEFORMAT *DestFormat*,  
LPDIRECT3DBASETEXTURE9 *pSrcTexture*, const PALETTEENTRY \**pSrcPalette*);**

This function is useful when you wish to save a texture image to a file. This can be useful if you want to load it into a paint program for viewing or editing. We pass in the desired file name and a member of the **D3DXIMAGE\_FILEFORMAT** enumerated type. This describes the image file format we would like the file to be saved in.

```
typedef enum _D3DXIMAGE_FILEFORMAT {
    D3DXIFF_BMP = 0,
    D3DXIFF_DDS = 4,
    D3DXIFF_DIB = 6,
    D3DXIFF_JPG = 1,
    D3DXIFF_PNG = 3,
    D3DXIFF_PPM = 5,
    D3DXIFF_TGA = 2,
    D3DXIFF_FORCE_DWORD = 0x7fffffff
} D3DXIMAGE_FILEFORMAT;
```



The third parameter is a pointer to the texture whose image we wish to save. The fourth parameter allows us to pass in an image palette if we are using palletized textures.

### **D3DXGetImageInfoFromFile(LPCSTR pSrcFile, D3DXIMAGE\_INFO \*pSrcInfo);**

The first parameter is the file name of the image we wish to extract the information for. The second parameter should be a pointer to a **D3DXIMAGE\_INFO** structure which will contain the information about the image if it is successful. The **D3DXIMAGE\_INFO** structure is defined as:

```
typedef struct _D3DXIMAGE_INFO
{
    UINT Width;
    UINT Height;
    UINT Depth;
    UINT MipLevels;
    D3DFORMAT Format;
    D3DRESOURCETYPE ResourceType;
    D3DXIMAGE_FILEFORMAT ImageFileFormat;
} D3DXIMAGE_INFO;
```

The members of this structure should be self explanatory by now as we have covered them throughout this course. In Lab Project 6.4, we are interested in the width and height members so that we can use it to create our offscreen surface. We will create the offscreen surface in the pixel format of the frame buffer so that we have an exact match when copying for maximum speed. Once we have created the surface in the format and dimensions we desire, we call D3DXLoadSurfaceFromFile to load the image into the surface we have created. The image will be color converted into the format of our surface which in our demo example is the format of the frame buffer.

### **D3DXLoadSurfaceFromFile(LPDIRECT3DSURFACE9 pDestSurface, CONST PALETTEENTRY\* pDestPalette, CONST RECT\* pDestRect, LPCTSTR pSrcFile, CONST RECT\* pSrcRect, DWORD Filter, D3DCOLOR ColorKey, D3DXIMAGE\_INFO\* pSrcInfo);**

There are several flavors of the D3DXLoadSurfaceFromXX function, just as there were for the D3DXCreateTextureFromXX functions. These functions load surfaces from files, resources, and from memory. Since they are all essentially the same, we will look at just one of them. You should consult the SDK documentation for details on the other types.

#### ***LPDIRECT3DSURFACE9 pDestSurface***

This is the pointer to the surface into which the image data will be loaded.

#### ***CONST PALETTEENTRY\* pDestPalette***

This parameter is a pointer to a palette used for palettized surfaces. We will pass in NULL for this parameter since we will not be using such textures in this course.



#### ***CONST RECT\* pDestRect***

Specifies a rectangle on the surface that the image will be loaded into. Often this will be set to NULL and the image will be loaded into the entire surface area. This function handles color conversion and resizing of the source image area to the destination rectangle.

#### ***LPCTSTR pSrcFile***

A string specifying the image file name to load.

#### ***CONST RECT\* pSrcRect***

This parameter defines a rectangular region of the source file to be loaded. Often this will be set to NULL so that the entire image is loaded into the destination surface rectangle.

#### ***DWORD Filter***

If the source image pixels have to be stretched or squashed to fit within the destination rectangle, then we can specify a filter to minimize aliasing. We pass one of the `D3DX_FILTER` values discussed earlier. Specifying `D3DX_DEFAULT` for this parameter is the equivalent of specifying `D3DX_FILTER_TRIANGLE` | `D3DX_FILTER_DITHER`.

#### ***D3DCOLOR ColorKey***

This parameter can be used to specify a 32-bit color to be replaced with transparent black pixels. We discussed color keys earlier in the lesson when examining texture loading from files.

#### ***D3DXIMAGE\_INFO\* pSrcInfo***

Pointer to a `D3DXIMAGE_INFO` structure to be filled with a description of the data in the source file. We can pass NULL if we are not interested in this information.

### Conclusion

We covered a lot of ground in this lesson. We now know quite a bit about the various texturing options at our disposal in DirectX Graphics. We know how to load them, blend them, filter them, and even how to compress and decompress them. Make sure that you thoroughly review the Lab Projects for this lesson since texturing will be an important part of all of our projects as we move forward. This will be especially true when we get to the next course in this series since it will deal almost exclusively with advanced texturing concepts to produce incredibly realistic lighting and other popular special effects.

# **Workbook Chapter Six:**

## **Texture Mapping**

---



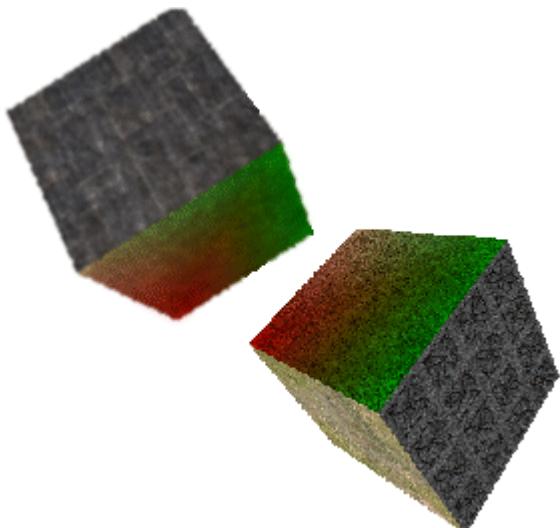
© 2003, eInstitute, Inc.

**You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.**

## 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”



one has all its vertices set to white (0xFFFFFFFF).

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

## 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
};

```

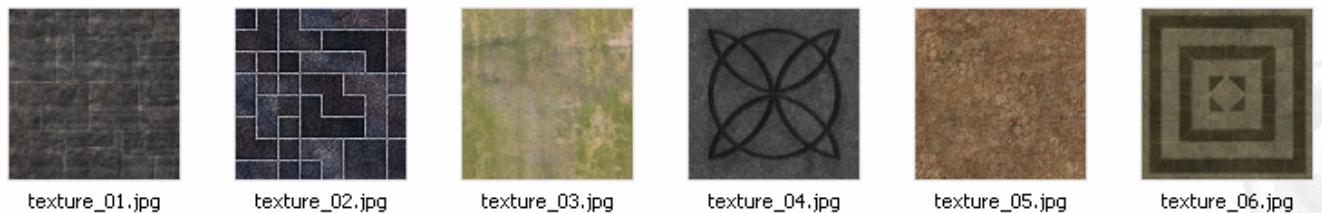
The constructors have been altered to allow input texture coordinates for each vertex created. We are not using the lighting pipeline, so we store a diffuse color at the vertex. The vertex flags are defined in CObjects.h:

```
#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 & D3DPTEXTURECAPS_MAGLINEAR) &&
   (Caps.TextureFilterCaps & D3DPTEXTURECAPS_MINLINEAR))
    m_bFilterEnabled = true;

// Determine if linear mip-mapping is supported
m_bMipEnabled = false;
if(Caps.TextureFilterCaps & D3DPTEXTURECAPS_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).

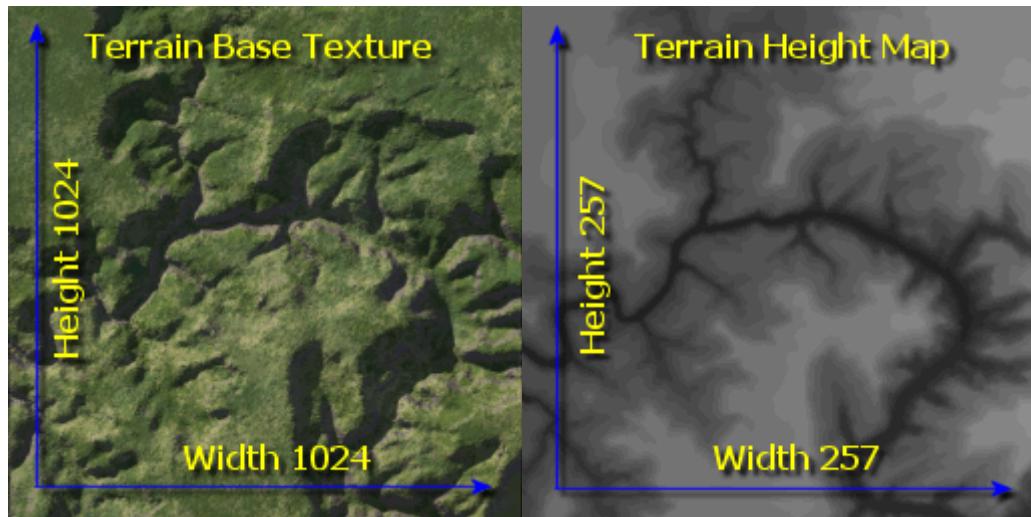
The first thing we need to do is to generate a terrain texture. This is not as hard as you might think. There is a wonderful terrain generation package called Terragen™ which will make our job very easy. You can download a shareware version for free from the following website:

<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 257<sup>th</sup> 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;
    }
}

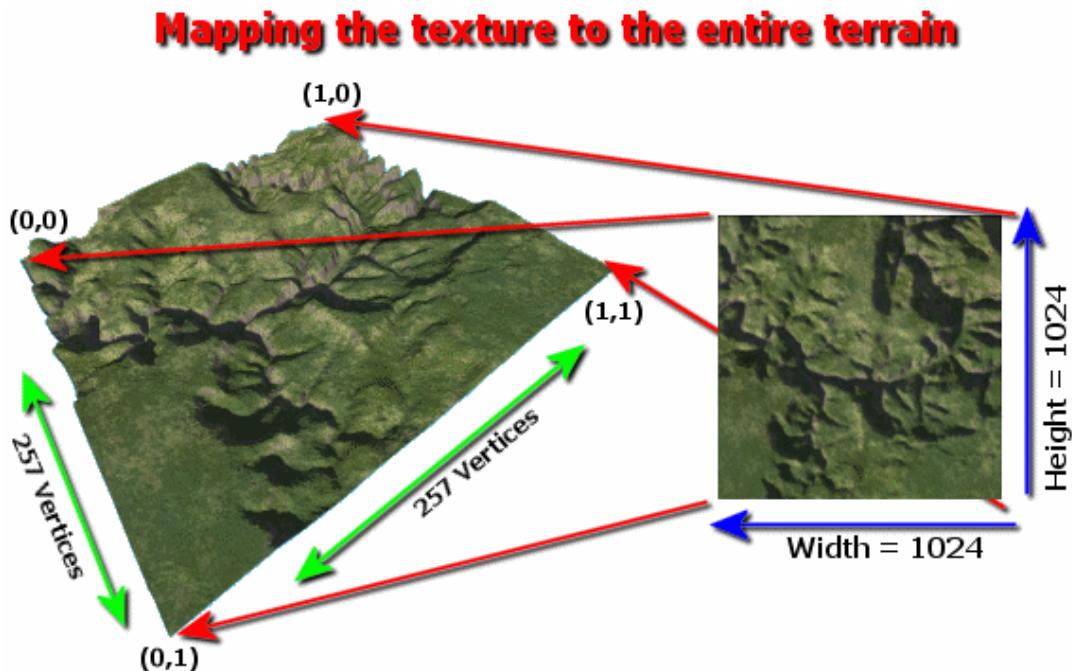
```

---

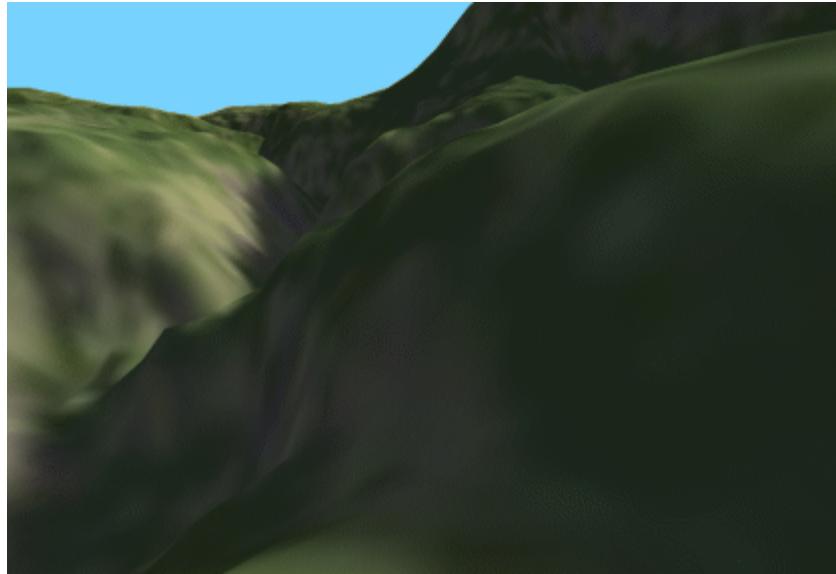
**Note:** To keep things simple for now, the code assumes that the vertices are stored in a multi-dimensional array in the form: `Vertex[Row][Column]`. In reality they will be stored in a vertex buffer.

---

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:



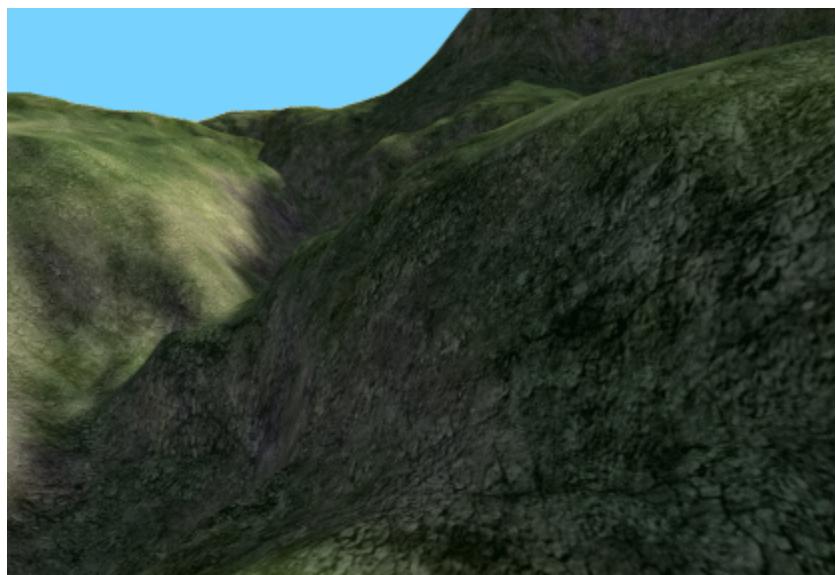
While this terrain texture is fairly large, it still is not big enough to provide enough detail when viewed up close. The solution is



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.

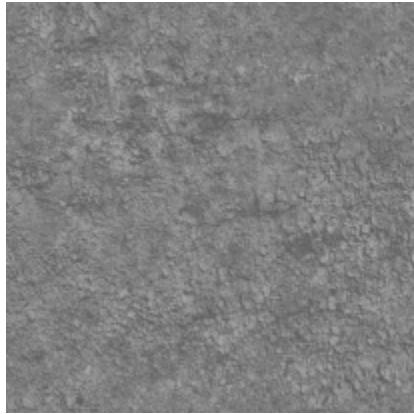
While the image above is not necessarily a terrible sight, we can certainly do much better...



This second image looks a good deal better. The solution was a simply multi-texturing technique called **detail mapping**.

### Single Pass Detail Mapping

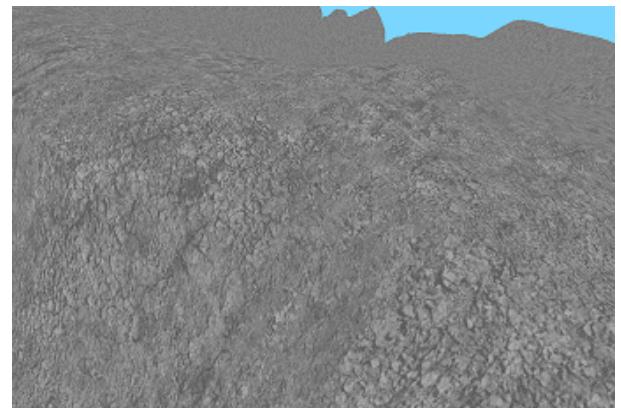
A detail map is a tileable texture with a high-frequency pattern. This project uses a 512x512 detail texture map which is shown below.



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

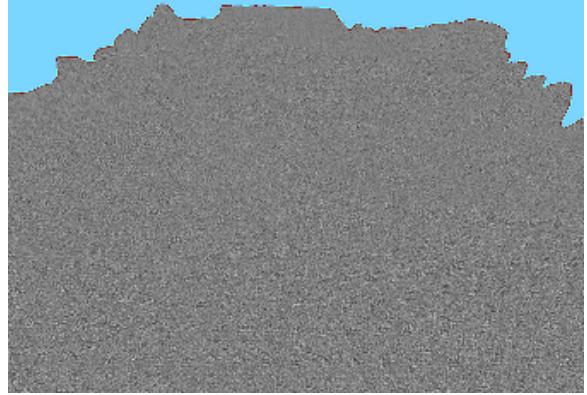
vicinity as shown below. The image to the right shows only the detail texture applied to the terrain.

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 , pDetailTexture );
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 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 only 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 ?

LPDIRECT3D 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 is 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.

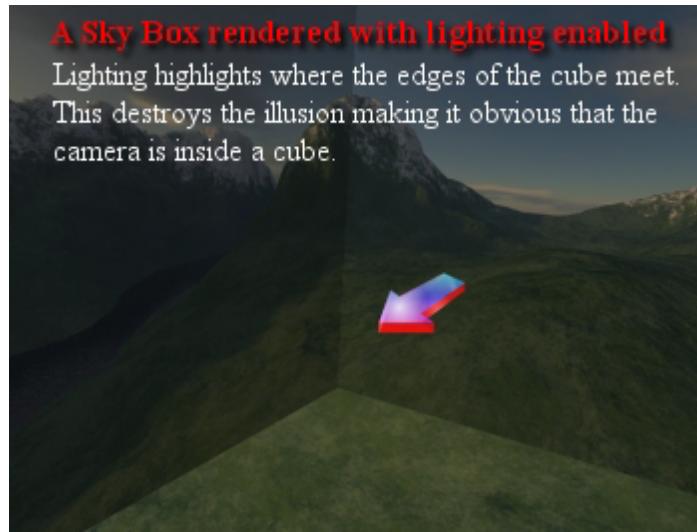
On each of the six faces of the box, a texture is placed. These textures should be seamless so that taken as a whole they depict a full panoramic view of the surrounding environment. This provides an easy way to add distant scenery and realistic atmospheric effects to outdoor environments. The sky box 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_MINPOINT)) 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 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 if 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 formula, 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 devide 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
}

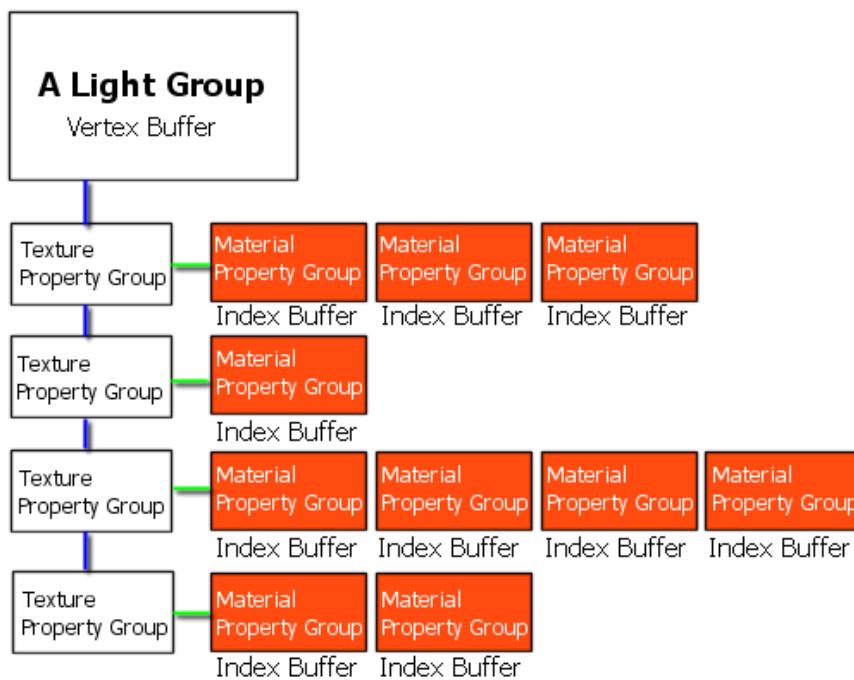
```

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 for animating our 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,
        D3DTFFF_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,
                                                D3DTFFF_DISABLE );
        } // End if Water Texture

    } // Next Property Group

} // Next Light Group
}

```

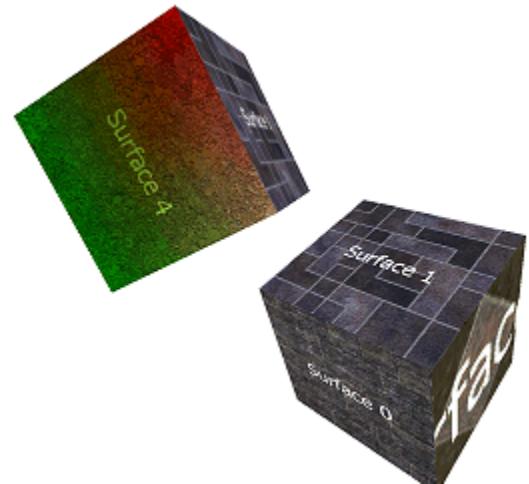
## Lab Project 6.4 Writing to Surface with GDI

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" );  
  
// Lets 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, 0xFFFFFFFF );

    // 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

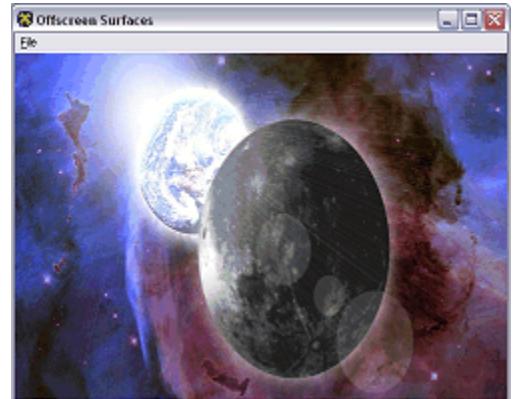
```

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 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 <b>D3DBLEND</b> enumerated 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 <b>D3DBLEND</b> enumerated type.	When alpha blending is enabled this state specifies how the destination color is blended with the source color.
D3DRS_TEXTUREFACTOR	A <b>D3DCOLOR</b> 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 <b>D3DTA_TFACTOR</b> , this color will be used. If the state is blending two colors using the <b>D3DTOP_BLENDFACTORALPHA</b> color operation, the alpha component of this color is used to blend the two input colors.
D3DRS_WRAP0 To D3DRS_WRAP15	<b>D3DWRAPCOORD0</b> To <b>D3DWRAPCOORD3</b>	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 <b>D3DBLENDOP</b> enumerated type.	This is used to set the calculation carried out when blending is enabled. The default operation is <b>D3DBLENDOP_ADD</b> as shown below: Src*SrcBlend + Dest*DestBlend

D3DRS_BLENDFACTOR	A <b>D3DCOLOR</b> 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 <b>D3DBLEND_BLENDFACTOR</b> to use this color in the blending operation.
-------------------	---	---

### New Texture Stage States Table

Texture Stage State	Parameters	Description
D3DTSS_COLOROP	A member of the <b>D3DTEXTUREOP</b> enumarated 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 <b>D3DTOP_MODULATE</b> and for all other stages the default is <b>D3DTOP_DISABLE</b> .
D3DTSS_COLORARG1	One of the <b>D3DTA</b> constant values should passed	This is used to set the first color argument of a texture stage. The default is <b>D3DTA_TEXTURE</b> meaning that the first input color to the texture stage is the color sampled from the texture.
D3DTSS_COLORARG2	One of the <b>D3DTA</b> constant values should passed	This is used to set the second color argument of a texture stage. The default is <b>D3DTA_CURRENT</b> 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 <b>D3DTA</b> 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 <b>D3DTEXTUREOP</b> enumarated 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 <b>D3DTOP_SELECTARG1</b> meaning the 1 <sup>st</sup> 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 <b>D3DTOP_DISABLE</b> .
D3DTSS_ALPHAARG1	One of the <b>D3DTA</b> constant values should passed	This is used to set the first alpha argument of a texture stage alpha pipeline. The default is <b>D3DTA_TEXTURE</b> 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 <b>D3DTA</b> constant values should passed	This is used to set the second alpha argument of the current texture stage. The default is <b>D3DTA_CURRENT</b> 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 <b>D3DTA</b> 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 1 <sup>st</sup> set of texture coordinates in the vertex, stage 1 uses the 2 <sup>nd</sup> , stage 2 uses the 3 <sup>rd</sup> , and so on.
D3DTSS_TEXTURETRANSFORMFLAGS	A member of the <b>D3DTEXTURETRANSFORMFLAGS</b> enumerated type.	This is used to enable the texture coordinate transformation 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 <b>D3DTTF_COUNTO2</b> .

D3DTSS_CONSTANT	A <b>D3DCOLOR</b> 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 <b>D3DTA_CONSTANT</b> , this color will be used as the color for that argument. Each stage can have its own constant color, unlike the <b>D3DRS_TEXTUREFACTOR</b> render state which is accessible from all texture stages.
-----------------	--	---

### New Sampler States Table

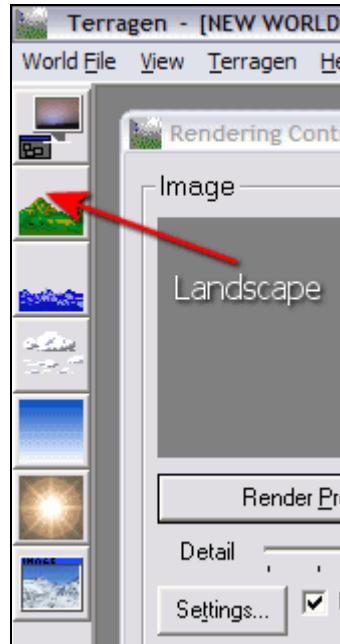
Sampler State	Parameters	Description
D3DSAMP_ADDRESSU	A member of <b>D3DTEXTUREADDRESS</b> enumerated type.	Allows you to set the addressing mode used when U component of a texture coordinate is outside the 0.0 to 1.0 range. The default value for each stage is <b>D3DTADDRESS_WRAP</b> which means the texture is repeated along the U axis.
D3DSAMP_ADDRESSV	A member of <b>D3DTEXTUREADDRESS</b> 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 <b>D3DTADDRESS_WRAP</b> which means the texture is repeated along the V axis.
D3DSAMP_ADDRESSW	A member of <b>D3DTEXTUREADDRESS</b> 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 <b>D3DTADDRESS_WRAP</b> which means the texture is repeated along the W axis.
D3DSAMP_BORDERCOLOR	A <b>D3DCOLOR</b> specifying the	If any of the addressing

	border color to be used.	modes are set to <b>D3DTSSADDRESS_BORDER</b> 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 <b>D3DTEXTUREFILTERTYPE</b> 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 <b>D3DTEXF_POINT</b> .
D3DSAMP_MAGFILTER	A member of the <b>D3DTEXTUREFILTERTYPE</b> 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 <b>D3DTEXF_POINT</b> .
D3DSAMP_MIPFILTER	A member of the <b>D3DTEXTUREFILTERTYPE</b> 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 <b>D3DTEXF_NONE</b> 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 <b>D3DCAPS9::MaxAnisotropy</b>

## Appendix B Making Terrain Textures in Terragen™

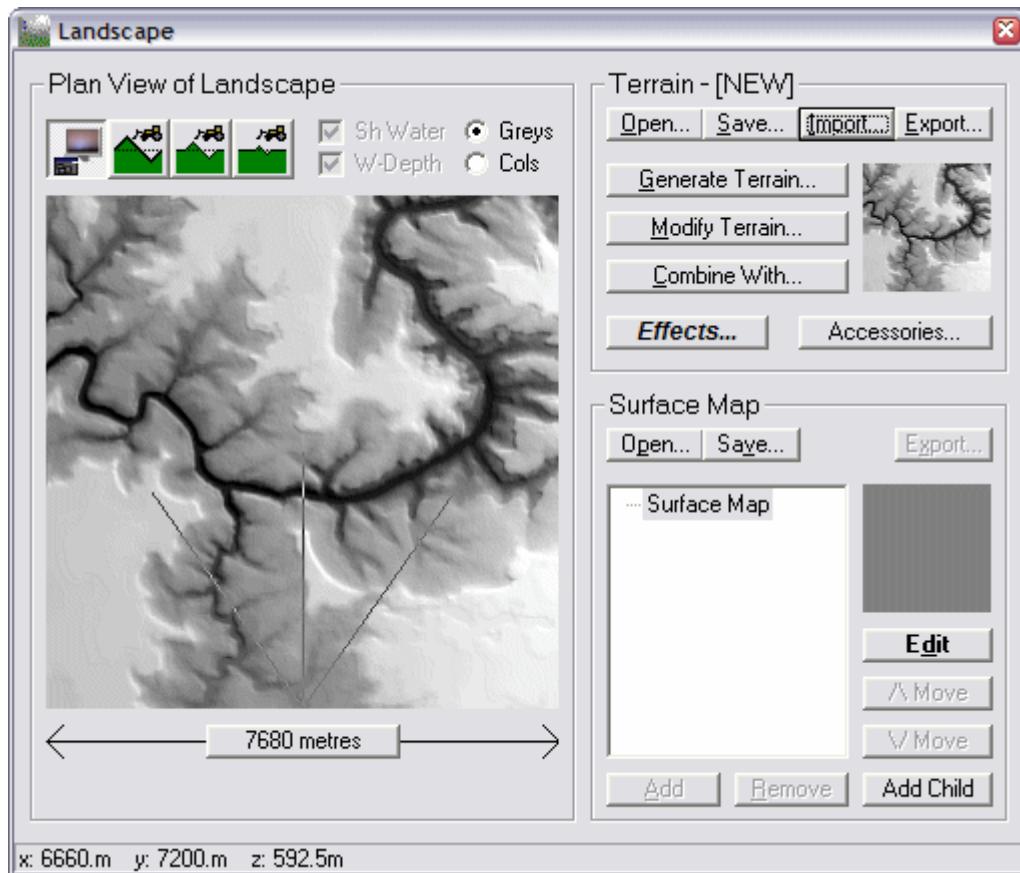
First we will need to generate or import a height-map. To do this, open the Landscape dialog. If this is not currently visible at start-up (or you have shut it down) this can be accessed by selecting the following toolbar button:



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).

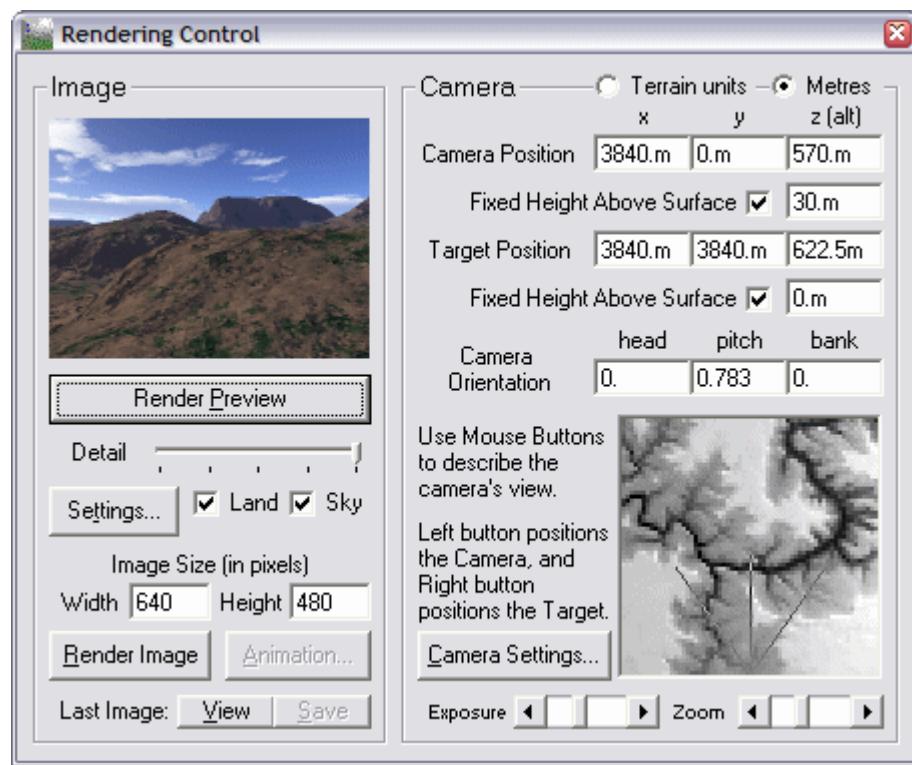
Once you have generated or imported your height-map data, the viewport on the left hand side should have been updated to reflect this as shown below:



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.

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.

The following list of steps will

- 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.

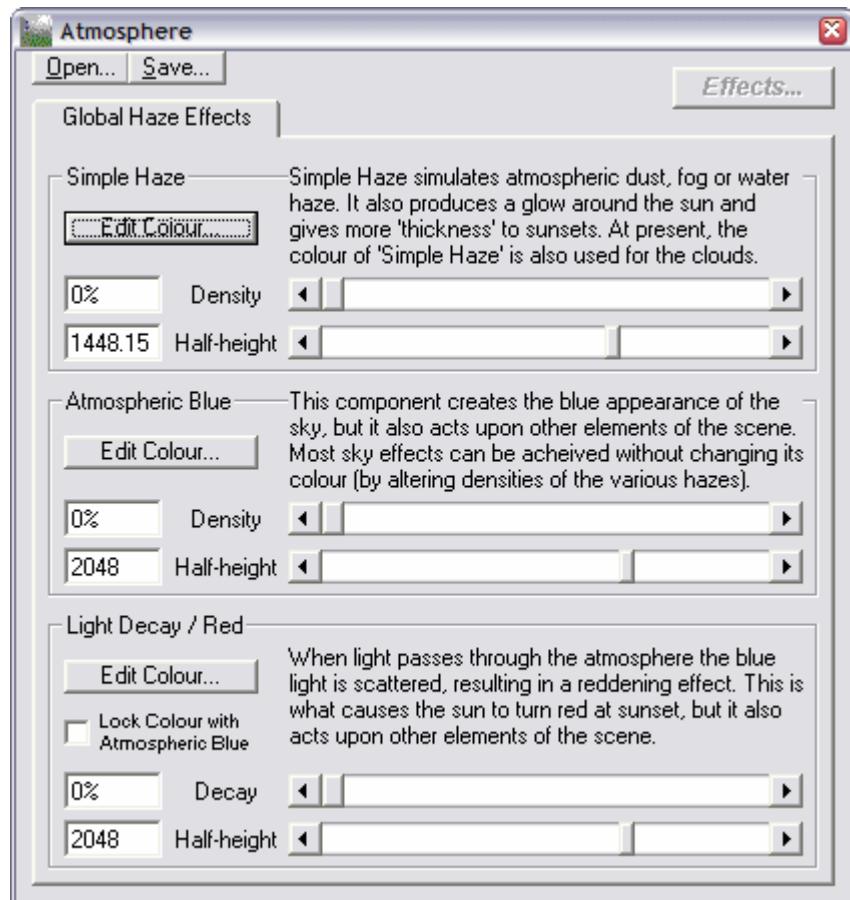
After setting up the camera, we will need to specify how our scene will be lit. For this we need to open up the Lighting Conditions dialog:



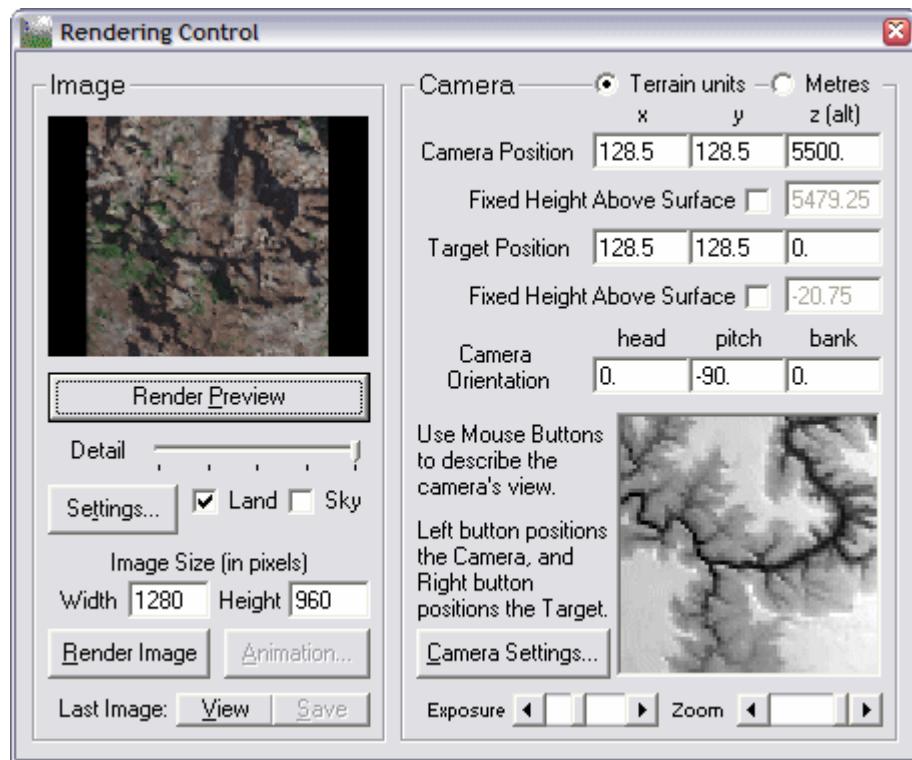
The settings you choose within this dialog are largely scene specific, but a heading of around 225 with an altitude of 30 is a good place to start. You can also enable or disable the various shadow casters in this dialog, so if you don’t want the terrain to cast shadows onto itself; then you will need 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 heightmap. 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:

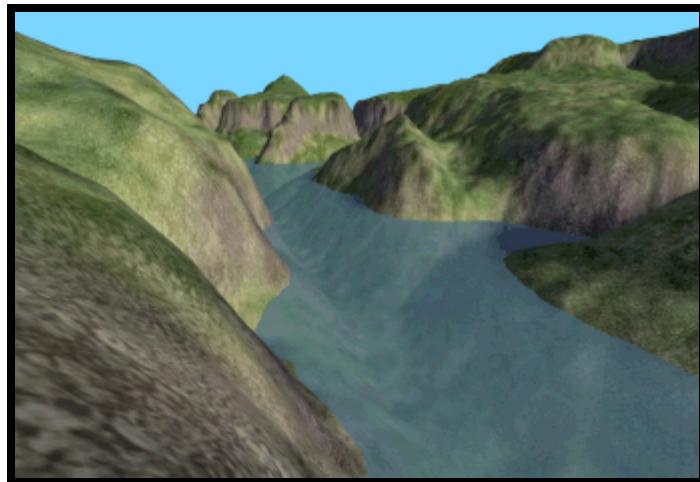
- Load settings
- Import or generate height-map
- Set surface map (if different from that saved)
- Render
- Crop, Size, Save



# Chapter 7

## Alpha Blending and Fog

---



© 2003, Game Institute Inc.

**You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.**



## Table of Contents

Alpha Components.....	3
Alpha Blending .....	5
Storing Alpha Components.....	8
Material Alpha .....	10
Vertex Alpha.....	11
Constant Alpha.....	13
Per-Stage Constant Alpha.....	14
Texture Alpha .....	14
The Texture Stage Alpha Pipeline .....	15
Alpha Blending with the Frame Buffer .....	20
Alpha Ordering .....	23
Alpha Testing.....	26
Transparent Polygon Sorting .....	28
Alpha Surfaces .....	40
Fog .....	43
Fog Types.....	46
Fog Factor Formulas.....	53
Conclusion .....	57



## Introduction

In this chapter we will examine the relationship between alpha and transparency as well as how to use our alpha components to achieve a number of important rendering effects. Our discussions and subsequent lab projects will cover:

- Vertex alpha values
- Texture alpha values (alpha channels)
- Material alpha and the DirectX Graphics lighting pipeline
- Alpha values in the texture blending cascade

### Alpha Components

In earlier lessons we saw that colors can be described as three component RGB values or four component ARGB values where the alpha value is included. Regardless of the current video mode we are working in and the current color bit depth being used, we always specify colors explicitly using one of two approaches. The first is the 32-bit four component ARGB value stored as a DWORD (**D3DCOLOR**). This is the case when specifying vertex colors. Each component uses 8 bits and has a range of [0, 255] decimal or [0, FF] hexadecimal. The second approach uses the **D3DCOLORVALUE** structure which includes four floating point member variables -- one for the alpha, red, green, and blue components respectively. We used a **D3DCOLORVALUE** structure in Chapter 5 when specifying colors for lights and materials. When using the **D3DCOLORVALUE** structure we typically specify each component in the range [0.0, 1.0]. An important point to keep in mind is that whether we are using a DWORD or a **D3DCOLORVALUE**, the colors will be converted into the bit depth currently being used for scene rendering. For example, a DWORD color will be scaled to fit into a 16-bit WORD by the renderer if the device is in 16-bit color mode.

The following example shows the DWORD representing the ARGB color (128, 255, 64, 64) or in hex, **0x80FF4040**. Hexadecimal specification makes color component assignment intuitive. The decimal value of this DWORD would be **2164211776**. This provides no visual clue as to the color being stored. With hex we see clearly that every two digits represent a BYTE of the DWORD between 0 and FF.

	Alpha	Red	Green	Blue
Hex	80	FF	40	40
Dec	128	255	64	64
Bits	31	24	16	8
	23		15	7
				0

Figure 7.1

This color has a half intensity alpha component, a full intensity red component, and quarter intensity green and blue components.



We can think of the alpha value as a packet of information that accompanies the main RGB color components. Although alpha is technically a component that can carry any information, it is used primarily to store values that will be used in the color and alpha blending pipelines in the texture stages as well as for alpha blending pixels with the existing contents of the frame buffer. If we do not perform any alpha dependant color blending in the texture stages or any alpha dependant alpha blending with the frame buffer, the alpha value of a color is simply ignored by the renderer and plays no part in the final rendered image.

The advantage to using alpha values in blending operations is that they allow us to perform color (RGB) independent blending operations. For example, color A can be blended with color B such that the alpha value stored in color A controls the ratio that each color plays in the final blended result.

**Note:** Alpha Blending technically means blending two colors using alpha values to determine the resulting color. The distinction between color blending and alpha blending has been blurred by the naming conventions of the DirectX API. When we enable alpha blending in DirectX 9, we set the following renderstate:

```
pDevice->SetRenderState ( D3DRS_ALPHABLENDENABLE , TRUE );
```

This call does not guarantee that we will be doing alpha blending specifically. Instead it enables a more generic frame buffer blending operation. Whether we actually use the alpha or color components of the source and destination colors during blending depends on how we set up the source and destination blend modes. In chapter 6 we saw how to set up the source and destination blend states to perform a modulation of the color of the pixel about to be written to the frame buffer with the color of the pixel already in the frame buffer. Recall from our lab project that this was how we blended the detail map with the base terrain texture in multi-pass rendering mode. Although we had to enable the alpha blending render state to do this, we did not technically perform alpha blending since we performed the blend by modulating RGB components and did not use alpha values at all. Therefore, the D3DRS\_ALPHABLENDENABLE simply informs the renderer that we wish to perform some form of blending with the frame buffer, be it color or alpha blending. Once alpha blending is enabled, the renderer uses the states of the D3DRS\_SRCBLEND and D3DRS\_DESTBLEND render states to determine whether to use alpha components in the blend or just RGB color components.

Further confusion is introduced by the fact that alpha blending can also be performed in the texture stages -- much earlier in the pixel pipeline. This is independent from the D3DRS\_ALPHABLENDENABLE render state. While a more fitting name for this render state might have been D3DRS\_FRAMEBUFFERBLENDENABLE, we will just have to be aware of the differences.

In keeping with the DirectX API conventions, when we refer to the process of 'enabling alpha blending' in the text, we will be referring to the process of enabling frame buffer blending using the D3DRS\_ALPHABLENDENABLE render state.



## Alpha Blending

Let us begin by looking at an example of blending two colors together using an alpha value. The most common usage of this technique is blending with the frame buffer to increase or decrease pixel opacity. Usually a source pixel about to be written will contain an alpha component and the color value in the frame buffer will not. This source alpha value governs the percentage of both the source and the destination color used to create the blended result. For this example we will assume that DestColor is a 32-bit pixel already in the frame buffer and SrcColor is a 32-bit pixel about to be written which includes an alpha component. Below we see the colors in their hexadecimal form and in their floating point equivalent form. This should make the mathematics easier to follow.

SrcColor = 0x80800000 (Alpha: 0.5; Red: 0.5; Green: 0.0; Blue:0.0)  
DestColor= 0xFF008000 (Alpha: 1.0; Red: 0.0; Green: 0.5; Blue:0.0)

Assume that we enable alpha blending and use the blend mode configurations shown next.

```
pDevice->SetRenderState( D3DRS_ALPHABLENDENABLE , TRUE );
pDevice->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
pDevice->SetRenderState( D3DRS_DESTBLEND , D3DBLEND_INVSRCALPHA );
```

Remember that these blend modes are assigned as multipliers to both the source and destination colors in the blending equation. The DirectX documentation describes these two blend modes as follows:

**D3DBLEND\_SRCALPHA** = ARGB (sA , sA , sA , sA);

Each component of the color is multiplied by the alpha component of the source color. The alpha value of the source color directly controls how much of the color is allowed into the resulting color. The higher the source alpha value, the larger contribution the color makes to the resulting color.

**D3DBLEND\_INVSRCALPHA** = ARGB (1-sA , 1-sA , 1-sA , 1-aA);

The color has each of its components multiplied by one minus the source alpha to create an inverse weighting multiplier. The higher the source alpha value, the less contribution the color makes to the resulting color.

---

**Note:** sA = source color alpha. Source color is the color about to be written to the frame buffer.

---

When alpha blending is enabled, the renderer performs the following calculation between the pixel color about to be written and the frame buffer pixel color (see Chapter 6):

Final Color = SourceColor \* SrcBlendMode + DestColor \* DestBlendMode



Using the blend modes above, this equates to the following calculation being performed:

Final Color = SourceColor \* **D3DBLEND\_SRCALPHA** + DestColor \* **D3DBLEND\_INVSRCALPHA**;

This means that we will use the alpha component of the source color to directly control the mixture of both colors in the final result. In our example, we have a source alpha value of 0.5. This should give an equal blend of both the source and destination colors (50% of each) to create the final color written to the frame buffer.

Final Color = (**sA**, 0.5, 0, 0) \* (**sA**, **sA**, **sA**, **sA**) + (1.0, 0, 0.5, 0) \* (1 -**sA**, 1-**sA**, 1-**sA**, 1-**sA**)  
Final Color = (**0.5**, 0.5, 0, 0) \* (**0.5**, **0.5**, **0.5**, **0.5**) + (1.0, 0, 0.5, 0) \* (1-**0.5**, 1-**0.5**, 1-**0.5**, 1-**0.5**)  
Final Color = (0.25, 0.25, 0, 0) + (0.5, 0, 0.25, 0)  
Final Color = (0.75, 0.25, 0.25, 0);

RGB Color = (0.25, 0.25, 0)

We can see that the final RGB color is a blend of 50% source RGB and 50% destination RGB. Figure 7.2 shows the source and destination colors used in this example and the resulting color blend in the overlapped area of the two squares.

### Transparency Effects

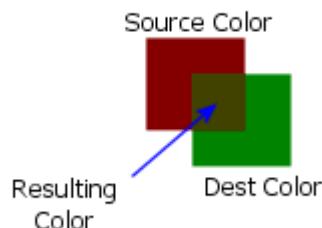


Figure 7.2

This common alpha blending equation certainly does make the source pixel appear to be transparent.

**Note:** The front buffer pixel format never supports alpha components. Although the frame buffer can, its alpha information is lost when flipping. Alpha format frame buffers are not very commonly used. The alpha component of the destination color is not used (defaults to 1.0) in the blending process if the frame buffer does not support alpha information.

We will try one more example source color. We will use the same RGB components but this time will include an alpha component of 0.75. This should cause the final color to be 75% percent of the source color and only 25% of the destination color.

SrcColor = 0xC0800000 (Alpha: 0.75; Red: 0.5; Green: 0.0; Blue: 0.0)  
DestColor = 0xFF008000 (Alpha: 1.0; Red: 0.0; Green: 0.5; Blue: 0.0)



$$\text{Final Color} = (\mathbf{sA}, 0.5, 0, 0) * (\mathbf{sA}, \mathbf{sA}, \mathbf{sA}, \mathbf{sA}) + (1.0, 0, 0.5, 0) * (1 - \mathbf{sA}, 1 - \mathbf{sA}, 1 - \mathbf{sA}, 1 - \mathbf{sA})$$

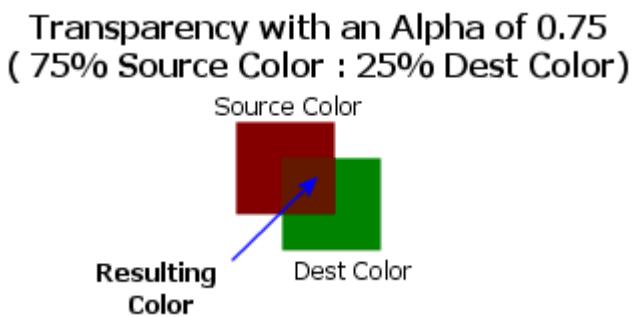
$$\text{Final Color} = (\mathbf{0.75}, 0.5, 0, 0) * (\mathbf{0.75}, \mathbf{0.75}, \mathbf{0.75}, \mathbf{0.75}) + (1.0, 0, 0.5, 0) * (1 - \mathbf{0.75}, 1 - \mathbf{0.75}, 1 - \mathbf{0.75}, 1 - \mathbf{0.75})$$

$$\text{Final Color} = (0.56, 0.375, 0, 0) + (0.25, 0, 0.125, 0)$$

$$\text{Final Color} = (0.81, 0.375, 0.125, 0);$$

$$\text{RGB Color} = (0.375, 0.125, 0)$$

Figure 7.3 shows the results. Using an alpha value of 0.75 decreases the transparency effect on the red square. The final blended color is no longer an equal mixture of both colors, but is instead 75% source color and only 25% destination color.



**Figure 7.3**

By altering the alpha value in the source color, we manipulate the color mixture in the blending equation. If the alpha value was set to 1.0 in the above example, the resulting color would be the unaltered source color and the destination color would not be blended with the source color at all. If however the source color alpha value was 0.0, then the source color would not contribute to the final color. This means that the alpha value in the source color allows direct control over how transparent the source pixel appears to be. The higher the alpha value, the more opaque the source pixel will be. Lower values result in greater pixel transparency.

This degree of transparency control was not possible in the previous chapter when we used blend modes based solely on the color information. We used blending modes such as `D3DBLEND_SRCCOLOR` and `D3DBLEND_DESTCOLOR` where the actual RGB components of source and destination pixels became multipliers in the equation to control transparency effect. It is difficult, if not impossible, to achieve certain results using this approach. When we store an alpha value in the source color, we can use it to describe very specific blending percentages regardless of the colors being blended.

Alpha blending is used to produce transparent effects that require precision. Glass, water, fire and other related game features are typical uses of this blending formula.



## Storing Alpha Components

Alpha values can be stored just about anywhere that a color is stored. Let us now examine the different places we might choose to store our alpha components. We will look more closely at each example as we progress through the lab projects accompanying this lesson.

### Vertex Alpha – Pre-Lit Vertices

This section looks at storing alpha values for vertices that are not using the lighting pipeline. As we have seen in previous chapters when using pre-lit vertices, our application explicitly stores the color of the vertex in the vertex structure. Because this is a four component color, storing the alpha value happens automatically when we specify the vertex color because the color of a vertex is specified in 32-bit ARGB format.

Let us say for example that we wanted all of the vertices in a given triangle to be full intensity green and 75% transparent. We start with a pre-lit vertex structure:

```
struct MyVertex
{
    float x;
    float y;
    float z;
    DWORD Diffuse;
};
```

Since each color component will be in the [0, 255] range, in order to make our green face  $\frac{3}{4}$  transparent, the alpha component will need to be set to 64 (0x40). If you were expecting a value of 192, remember that when the alpha value is 0, the color is totally transparent and when the alpha value is 255 the color is totally opaque. Therefore, in order to make our color  $\frac{3}{4}$  transparent we need to subtract 192 (3/4 of 256) from 256 to give 64.

```
for (int i = 0; i < Polygon.VertexCount; i++)
{
    Polygon.Vertex[i].diffuse = 0x4000FF00;
}
```

At the end of the above loop, every vertex in the polygon would be full intensity green and have an alpha value of 64 (0x40). Since every vertex in the triangle has the same color, the face would be a consistent green color across the entire surface. Since each vertex in the face would also have the same alpha value, the face would have a consistent transparency level for each pixel. The color at each vertex as it is stored in the DWORD is shown in the next table.



	Alpha	Red	Green	Blue
Hex	40	0	FF	0
Dec	64	0	255	0
Bits	31	24	16	8
	23	15	7	0

Whether or not the face is rendered transparently depends on whether alpha blending is enabled or whether alpha operations are used in the texture stages. We will address these cases in a moment. The important thing to grasp is that using this method provides the ability to store a per-vertex alpha value for each face to be used later by the vertex and pixel pipelines.

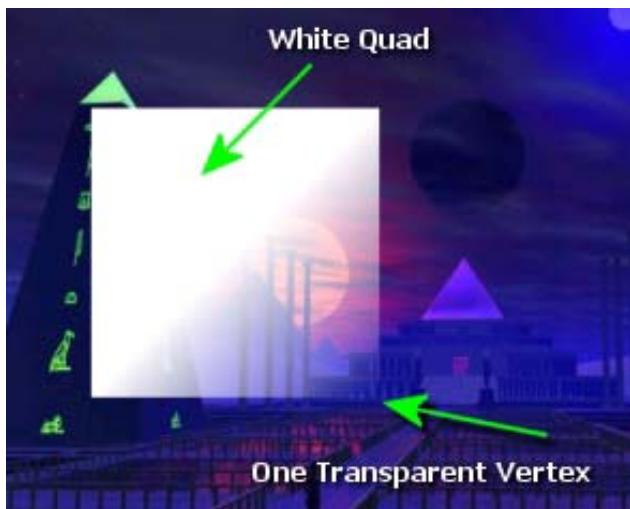


Figure 7.4

Recall that per-vertex colors are interpolated across the surface of a polygon when Gouraud shading is enabled. This generates a per-pixel color that can be accessed by the texture stages and finally passed on to the rasterizer. When the per-vertex color is interpolated, the alpha component is interpolated as well. The interpolation will generate both a per-pixel RGB value and a per-pixel alpha value. Consequently, we can have a polygon with different vertex alpha values and thus adjust the transparency so that it changes between vertices in a face. Fig 7.4 shows a simple quad rendered after the main scene

has already been rendered into the frame buffer. The quad has an opaque white color (0xFFFFFFFF) stored at three of its vertices, but the bottom right vertex has the color (0X00FFFFFF) which is a white color with an alpha value of zero (full transparency).

Notice that when the alpha values are interpolated across the quad from the top left corner to the bottom right corner, the results are a per-pixel partial transparency beginning at the lower right corner and fading as we move up and to the left. This process takes place at the same time the color components are calculated.

Whether we use the resulting per-pixel alpha values for alpha blending depends on how the texture stages are configured. Just as we can set the stages to use the vertex and/or texture color, we can also configure the texture stages to use the per-pixel alpha values generated from this interpolation. Alternatively, we could decide to ignore the alpha information generated here and use alpha information stored in a texture, as we will discuss later on in the lesson.

Finally, if you are specifying your own colors at the vertices then you may also be storing the specular highlight color at the vertex too. Just as diffuse colors are four component colors, so are specular colors. These two colors are added together at render time to create the true ARGB color of each pixel in the surface being rendered.



## Material Alpha

In Chapter 5 we learned how to use materials to store polygon colors for the DirectX lighting pipeline. The material structure determines how polygons react to incoming light and the colors that are ultimately reflected. Recall that materials use four component colors and allow us to specify alpha values for the diffuse, ambient, emissive, and specular properties in addition to the RGB values.

The **D3DCOLORVALUE** type used by materials use floats that are typically in the range [0.0, 1.0]. The following example demonstrates setting up a material for a green quad with white specular highlights. We have given the diffuse color an alpha value of 0.5 so that any pixels that receive diffuse light will be 50% transparent.

```
D3DMATERIAL9 Material;
ZeroMemory(&Material, sizeof(D3DMATERIAL9));

Material.Diffuse.a = 0.5f;
Material.Diffuse.r = 0.0f;
Material.Diffuse.g = 1.0f;
Material.Diffuse.b = 0.0f

Material.Specular.a = 1.0;
Material.Specular.r = 1.0;
Material.Specular.g = 1.0;
Material.Specular.b = 1.0;
```

In Chapter 5 we discussed how the pipeline uses the material properties to calculate a final per-vertex color. The alpha component is calculated in exactly the same way since it is just another component of the color. Thus we can give different reflective properties their own alpha values within the same material. This means that the polygon will be more or less transparent depending on the type of light that is contributing most to the final color of the vertex. For example, we know that a material can reflect white diffuse light and green ambient light to cause any polygon rendered using it to be white when lit by a light source but green when only ambient light is affecting it. Since alpha is just another color component, the same applies. We might have an alpha value of 1.0 specified in the diffuse member of the material so that polygons rendered would be completely opaque when being light by a directional light source. Then we could include an alpha value of 0.5 in the ambient color so that when the directional light is no longer shining on the object and it is only lit by ambient light, it would be semi-transparent:

```
D3DMATERIAL9 Material;
ZeroMemory(&Material, sizeof(D3DMATERIAL9));

Material.Diffuse.a = 1.0f;
Material.Diffuse.r = 1.0f;
Material.Diffuse.g = 1.0f;
Material.Diffuse.b = 1.0f

Material.Ambient.a = 0.5;
Material.Ambient.r = 1.0;
```



```
Material.Ambient.g = 1.0;  
Material.Ambient.b = 1.0;
```

You could take this idea a step further and have different alpha values for emissive and specular reflectance properties as well. However we will generally want a polygon to have a consistent transparency setting across all light types and we will set all material alpha components to the same value.

### Vertex Alpha

The problem with using only materials to inform the lighting pipeline of the surface reflectance properties is that we are limited to a polygon being a single color. This is because we set the material, and then render the polygon (or polygons) that use that material. Every vertex rendered has its color calculated by the lighting pipeline using the same reflectance properties as the currently set material. For example:

```
pDevice->SetMaterial(&Material);  
RenderFaces();
```

This approach seems to limit us to having a per-face alpha value for each reflectance property. If a material has a diffuse alpha component of 0.5 for example, then it would appear that every vertex rendered using that material will have a diffuse alpha property of 0.5 as well. However you should recall that even when using the lighting pipeline, we can store colors at the vertices and instruct the lighting pipeline to use these colors as reflectance properties in the lighting equations instead of some of the colors in the currently set material. Using this technique we can store up to two colors in each vertex which can be substituted for the reflective properties of the material. This allows us to continue to use the lighting pipeline but have per-vertex alpha properties when necessary.

In the following example we use a vertex structure with a color to store our diffuse reflectance instead of the using the diffuse material property. In order to do this, we must remember to set the diffuse material source so that the lighting pipeline takes it from the color in the vertex instead of the diffuse member of the material.

```
struct Vertex  
{  
    float x; float y; float z;  
    DWORD Color;  
};
```

Assume that we have three vertices stored in the above format and that we wish to use the lighting pipeline. Each vertex in the triangle has the same RGB color (0, 255, 0), but the first vertex has  $\frac{3}{4}$  transparency, the second vertex has  $\frac{1}{2}$  transparency and the third has  $\frac{1}{4}$  transparency.

Note that only the diffuse reflectance property is taken from the vertex. The emissive, specular and ambient reflectance properties in the material will still be used to calculate the final color of the vertex.



```
Vertex[0].Color = 0x4000FF00; // ARGB ( 64 , 0 , 255 , 0 );
Vertex[1].Color = 0x8000FF00; // ARGB ( 128, 0 ,255 , 0 );
Vertex[2].Color = 0xC000FF00; // ARGB ( 192, 0 , 255 , 0 );
```

We now enable lighting and inform the device that the diffuse reflectance property should be taken from the first color in the vertex and not the diffuse member of the currently set material.

```
pDevice->SetRenderState( D3DRS_LIGHTING , TRUE );
pDevice->SetRenderState( D3DRS_DIFFUSEMATERIALSOURCE , D3DMCS_COLOR1);
```

When the triangle is rendered, each vertex will have a different diffuse alpha value that is interpolated over the surface. This provides the same level of control that we had when using pre-lit vertices. Bear in mind that the final color of the vertex is calculated by adding the ambient, diffuse, specular and emissive light reflected by the vertex. So we could have different alpha values for different light types.

In this final example, we will look at an example of creating vertices that store a  $\frac{1}{2}$  transparent green diffuse color and a  $\frac{3}{4}$  transparent blue emissive color. This will render polygons that have a per-vertex diffuse alpha and a different per-vertex emissive alpha. When the vertices are being lit by a white diffuse light, the polygon will be  $\frac{1}{2}$  transparent and green in color. When no light is shining on the vertices, the polygon will be emissive blue and will be  $\frac{3}{4}$  transparent.

The vertex will now need to store two colors.

```
struct Vertex
{
    float x; float y; float z;
    DWORD Color1;
    DWORD Color2;
};

Vertex[0].Color1 = 0x8000FF00; // ARGB ( 128 , 0 , 255 , 0 );
Vertex[1].Color1 = 0x8000FF00; // ARGB ( 128, 0 ,255 , 0 );
Vertex[2].Color1 = 0x8000FF00; // ARGB ( 128, 0 , 255 , 0 );
Vertex[0].Color2 = 0x400000FF; // ARGB ( 64 , 0 , 0 , 255 );
Vertex[1].Color2 = 0x400000FF; // ARGB ( 64, 0 , 0 , 255 );
Vertex[2].Color2 = 0x400000FF; // ARGB ( 64, 0 , 0 , 255 );
```

Again, we need to enable lighting and inform the device that the diffuse reflectance property should be taken from the first color in the vertex and that the emissive reflectance property should be taken from the second color in the vertex:

```
pDevice->SetRenderState( D3DRS_LIGHTING , TRUE );
pDevice->SetRenderState( D3DRS_DIFFUSEMATERIALSOURCE , D3DMCS_COLOR1);
pDevice->SetRenderState( D3DRS_EMISSIVEMATERIALSOURCE , D3DMCS_COLOR2);
```



Remember that the final color of a vertex is calculated as:

$$\text{Vertex Color} = (\text{AmbientLight} * A) + (\text{DiffuseLight} * D) + (\text{SpecularLight} * S) + E$$

Where:

A = Ambient Reflectance Property

D = Diffuse Reflectance Property

S = Specular Reflectance Property

E = Emissive Property

Since the alpha value of the color is just one of its components, we know that the alpha value generated for a vertex by the lighting pipeline is therefore:

$$\text{Vertex Color.a} = (\text{AmbientLight} * A.a) + (\text{DiffuseLight} * D.a) + (\text{SpecularLight} * S.a) + E.a$$

Where:

a = the alpha component of the color

Once the lighting pipeline has calculated the per-vertex color and alpha values, the process continues as usual. The triangle is assembled and the per-vertex values are interpolated over the surface to generate per-pixel color and alpha values that will be accessible in the texture cascade.

### Constant Alpha

If we need a constant level of transparency for a set of polygons, we can use the **D3DRS\_TEXTUREFACTOR** to set a constant alpha value that is accessible in the texture stages. In Chapter 6 we discussed how to use the texture factor as a constant color and how to select its RGB values as inputs into the texture stage color pipeline. In a short while we will discuss how the alpha component of this color can also be selected as an input into the alpha pipeline of the texture cascade for blending with other alpha sources or simply passed to the rasterizer for frame buffer blending. For now we are simply trying to understand where alpha components can be stored. The following code shows how we could set a red texture factor color with  $\frac{1}{2}$  intensity alpha.

```
pDevice->SetRenderState(D3DRS_TEXTUREFACTOR, 0x80FF0000);
```

Remember that the device has only one texture factor property, so setting this color will overwrite any previous color stored.



## Per-Stage Constant Alpha

In chapter 6 we also discussed using the `D3DTSS_CONSTANT` texture stage state to set a per-stage constant color as an input argument. This color is also a four component color so we can provide each stage with an RGB constant and a constant alpha component to be used in the alpha pipeline. The following code shows how we could set a half intensity blue constant color for texture stage 2 along with a half intensity alpha component:

```
pDevice->SetTextureStageState(2, D3DTSS_CONSTANT, 0x80000080);
```

The color and alpha components of this color can be selected as input arguments to a texture stage by using the `D3DTA_CONSTANT` parameter as one of the arguments to the color or alpha pipelines of the stage.

## Texture Alpha

Alpha components can also be stored in textures. In previous chapters we looked at a number of pixel formats that contain alpha components. One such format was the 32-bit ARGB texture format `D3DFMT_A8R8G8B8`. Another was the 16-bit alpha pixel format `D3DFMT_A4R4G4B4`. There are quite a few more such formats and there are also compressed texture formats that support alpha information.

Just as each texel contains a red, green, and a blue component, we can create textures whose texels include alpha components too. When the texels in a texture also contain an alpha component, the texture is said to have an **alpha channel**. Alpha channels allow us to specify per-texel alpha. This provides a good deal more flexibility than the other storage methods we just examined.

Although per-vertex alpha is interpolated to create per-pixel alpha values, we have very little control over the per-pixel alpha values generated. Per-vertex alpha is fine if we require only that limited degree of control, but for more complex scenarios it becomes too restrictive. Consider the window texture shown below to the right. Assume that it is mapped to a quad and that there is geometry on the other side of the window that is currently being obscured. If this were a real window we would want to see the objects on the other side through the glass panes. Giving each vertex in the quad an alpha value would not work because the bars of the window on the texture would also become transparent.



So let us instead give each pixel belonging to one of the window panes an alpha value of 128 (semi-transparent) and each pixel belonging to one of the window bars an alpha value of 255. If we used the alpha values in the texture to perform frame buffer blending, only the window panes would be transparent and we would see the geometry in the distance. You can see this effect in the image on the left. Notice that the bars of the window still



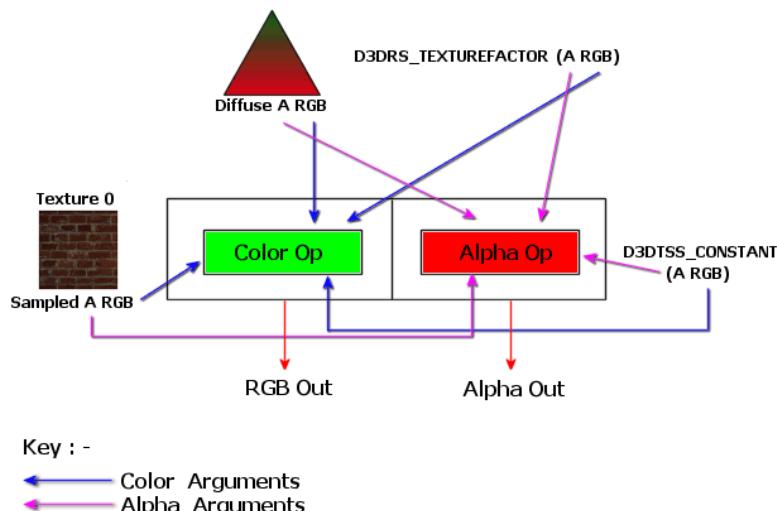
obscure the geometry behind the window but that the glass panes allow the background geometry to show through while still retaining the rough texture of window pane itself.

There are a couple of ways to add alpha information to a texture. We could lock the texture and manually set the alpha component of each pixel, but this would not be recommended in most cases. The more common approach is to use an image editing/paint package. Most paint programs like Jasc Paint Shop Pro 7™ or Adobe PhotoShop™ provide a mechanism for writing alpha values to the pixels of a texture. This makes the creation of the alpha channel much like drawing normal RGB values. These texture images can be saved out in a format that supports alpha channel images (.tga is a popular choice) and then loaded into our application using the D3DXCreateTextureFromFile function. The DirectX SDK includes a utility application in the DXUtils folder called DxTex.exe. This is a simple tool that allows you to load an image and manipulate its format. You might decide to add an alpha channel or simply to change the surface format to some other color depth. You can even load another image directly into the texture alpha channel if desired. The application exports images in .DDS format which is the DirectX native format for storing surfaces. These formats are supported by the D3DXCreateTextureFromFile functions and are stored in exactly the same format used by DirectX. The chapter appendices include a short tutorial on adding an alpha channel to a bitmap in Paint Shop Pro 7™. This is a powerful paint package that is affordably priced.

### The Texture Stage Alpha Pipeline

In Chapter 6 we looked at color blending in the color pipeline of a texture stage. We learned that there is also a separate alpha pipeline in each stage that uses nearly all of the same blending operations and input argument types as the color pipeline. Fig 7.5 shows a single texture stage with its RGB and Alpha pipelines along with possible color/alpha sources that can be used as arguments for the stage.

**A Texture Stages Dual Pipeline**



**Figure 7.5**



We know from Chapter 6 that we can set up the inputs of a texture stage to sample per-pixel colors from a variety of different color sources. In the following example we set up the color pipeline in stage 0 for a modulate2x operation.

```
pDevice->SetTextureStageState ( 0 , D3DTSS_COLORARG1 , D3DTA_DIFFUSE );
pDevice->SetTextureStageState ( 0 , D3DTSS_COLORARG2 , D3DTA_TEXTURE );
pDevice->SetTextureStageState ( 0 , D3DTSS_COLOROP , D3DTOP_MODULATE2X );
```

The RGB color components from the interpolated vertex colors will be blended with the RGB color components of the current texel in the texture to generate a new set of RGB color values that are output from the texture stage. The output is a single RGB color either sent to the next stage, or if this is the last active stage in the cascade, sent to the rasterizer as the source color for any frame buffer blending operation enabled. If alpha blending is not enabled, the source color will be written directly to the frame buffer if the depth test is passed.

The texture stage states shown above only configured the arguments and blending operations performed on the RGB values of the selected arguments. However, each texture stage also has a separate alpha unit which can be used to extract the alpha values from all of the different sources shown in the Fig 7.5. Moreover, these sources need not be the same color sources that the RGB values are being extracted from. For example, we might generate a pixel RGB color that is a modulation between the diffuse color (RGB only) and the texture color (RGB only) but decide that we want the alpha information to be taken from the texture factor color. This would extract the alpha component from the texture factor and push it into the alpha pipeline.

As discussed earlier in the lesson, the diffuse vertex color, the texture factor color, and the per-stage constant color include an alpha value -- even if it is just the fully opaque default (0xFF). The texture color source is slightly different as the texture might not contain an alpha channel (non-alpha pixel format). If this is the case then any attempt to sample an alpha value from a texel will simply return a value of (0xFF).

We will use the **D3DTSS\_ALPHAARGn** texture stage states to determine which color source(s) the alpha information is extracted from. The **D3DTSS\_ALPHAOP** texture stage state sets up a blending operation for multiple alpha input values to create a final alpha value that is output from the stage. Every stage outputs an RGB color and an Alpha value and these are either passed to the following stage as input arguments or to the rasterizer when the last active stage is reached.

**Note:** The default alpha operation is **D3DTOP\_SELECTARG1** for texture stage 0 and for all other stages it is **D3DTOP\_DISABLE**. The default value of **D3DTSS\_ALPHAARG1** is **D3DTA\_TEXTURE**, so the alpha values will be taken from the texture or equal 0xFF if no texture is bound to the stage.



## RGB = (Vertex\*Texture) : Alpha = Texture

In this code snippet we setup the texture stage states to modulate the texture color with the diffuse color and select alpha values from the texture alpha channel. This alpha setup would be useful when you wish to make sure that only certain pixels are transparent (like the window example discussed previously). We divide the texture states above into two sections to clearly show the division between setting up the RGB pipeline and the Alpha pipeline.

```
// Each pixel RGB is a combination of the texel color  
// and the interpolated vertex diffuse color  
pDevice->SetTextureStageState( 0 , D3DTSS_COLORARG1 , D3DTA_DIFFUSE );  
pDevice->SetTextureStageState( 0 , D3DTSS_COLORARG2 , D3DTA_TEXTURE );  
pDevice->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_MODULATE );  
  
// Each pixel alpha value is taken from the alpha component  
// of the texel in the texture  
pDevice->SetTextureStageState( 0 , D3DTSS_ALPHAARG1 , D3DTA_TEXTURE );  
pDevice->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_SELECTARG1 );
```

## RGB = Texture : Alpha = Vertex

In this next example, the color of each pixel is taken directly from the texel and the alpha value from the alpha component of the interpolated diffuse vertex color. This is a useful alpha pipeline setup if your texture does not include an alpha channel. It is also useful when you are only interested in a constant level of alpha across the entire polygon or if you need alpha control only at the vertex level. Lab Project 7.1 will use this alpha configuration.

```
// Each pixel RGB color is taken from the texel in the texture  
pDevice->SetTextureStageState( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE );  
pDevice->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_SELECTARG1 );  
  
// Each pixel alpha value is taken from the  
// interpolated diffuse vertex color (alpha component)  
pDevice->SetTextureStageState( 0 , D3DTSS_ALPHAARG1 , D3DTA_DIFFUSE );  
pDevice->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTA_SELECTARG1 );
```

## RGB = (Texture\*Vertex)+TextureFactor : Alpha = (Texture\*TextureFactor)

The next configuration sets up a blue texture factor color with a  $\frac{1}{4}$  intensity alpha component. The texture RGB components from stage 0 are modulated with the diffuse vertex RGB components and passed on to stage 1 where the texture factor RGB components are added to the result. The alpha component in the first stage is taken from the alpha channel of the texture and is passed to the second stage where it is modulated with the alpha component of the texture factor.

```
// Set a blue texture factor color with an  $\frac{1}{4}$  intensity alpha component  
pDevice->SetRenderState( D3DRS_TEXTUREFACTOR , 0x400000FF );  
  
// Each pixel RGB color is (texture*vertex) + texture factor  
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 );

// Each pixel alpha value is taken from the
// interpolated diffuse vertex color (alpha component)
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);
```

This example may look complex, but that is only because we are using two stages. In the color pipeline we instruct the device to modulate the diffuse vertex color and texture color in stage 0. The result of this stage is used as the first argument to the next stage (**D3DTA\_CURRENT**) where it is added to the RGB components of the texture factor that had been set previously as a render state. The resulting RGB color will be output from stage 1 and passed as an input to the rasterizer later on in the pipeline.

We can see that the alpha pipeline is setup quite differently than the color pipeline in this example. This shows us the flexibility of using the alpha and RGB pipelines together. In the first stage, we use the alpha value from the sampled texel of the texture bound to that stage as the output from stage 0. This becomes the input to the second stage alpha pipeline (**D3DTA\_CURRENT**) and we multiply it with the alpha component of the texture factor color. The resulting alpha value is output from the texture stage cascade and passed down the pipeline where it will become the source alpha value input to the rasterizer's color/alpha blending equations if frame buffer blending is enabled.

### **RGB = Texture : Alpha = (Texture\*Diffuse)+TextureFactor**

In this next example the RGB components are taken from the texture color, but the alpha component is calculated by doing a signed add between the texture alpha and diffuse alpha. The result is then added to the texture factor alpha in the second stage.

```
// Each pixel RGB color sampled from texture
pDevice->SetTextureStageState ( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE);
pDevice->SetTextureStageState ( 0 , D3DTSS_COLOROP , D3DTOP_SELECTARG1);
pDevice->SetTextureStageState ( 1 , D3DTSS_COLORARG1 , D3DTA_CURRENT);
pDevice->SetTextureStageState ( 1 , D3DTSS_COLOROP , D3DTOP_SELECTARG1);

// Alpha = (Texture*Diffuse) + TFactor
pDevice->SetTextureStageState ( 0 , D3DTSS_ALPHAARG1 , D3DTA_TEXTURE);
pDevice->SetTextureStageState ( 0 , D3DTSS_ALPHAARG2 , D3DTA_DIFFUSE);
pDevice->SetTextureStageState ( 0 , D3DTSS_ALPHAOP , D3DTOP_ADDSIGNED);
pDevice->SetTextureStageState ( 1 , D3DTSS_ALPHAARG1 , D3DTA_CURRENT);
pDevice->SetTextureStageState ( 1 , D3DTSS_ALPHAARG2 , D3DTA_TFACTOR);
pDevice->SetTextureStageState ( 1 , D3DTSS_ALPHAOP , D3DTOP_ADD );
```

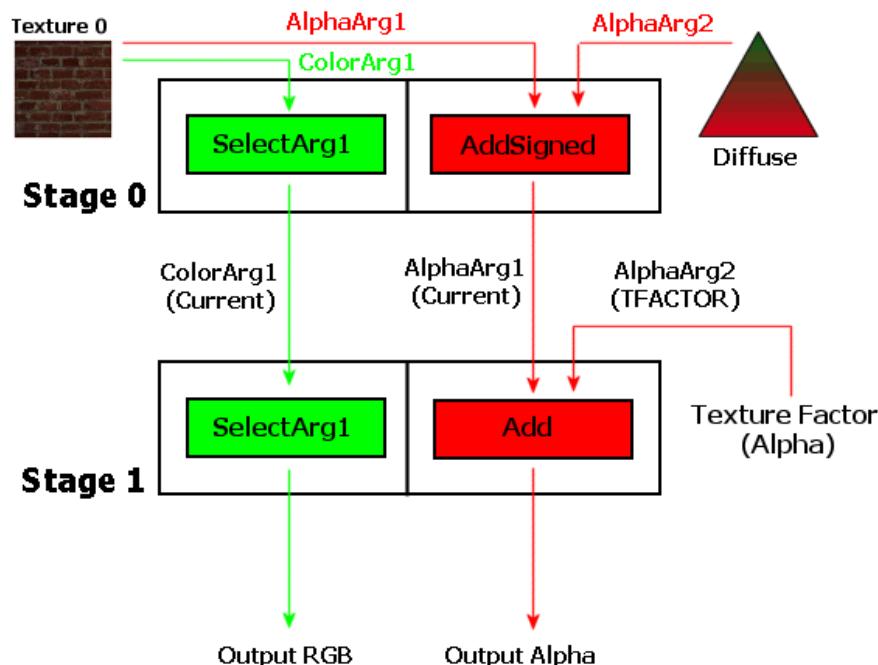


Figure 7.6

Note that the second stage in the color pipeline does not seem to be doing much beyond passing the color value from the previous stage through as output (Fig 7.6). The color pipeline technically needs only one texture stage but the alpha operations require two. This means the RGB components output from the first stage need to be passed through the second stage unaltered. The reason we have to do this is that we need the second stage enabled to do alpha operations, but we do not wish to perform any operations on the RGB components. The important concept to understand is the flow of components from one stage to the next active stage in the cascade.

There are other texture operations that can blend RGB components in a stage using the alpha components (from the other pipeline) but we will discuss these a little later in the lesson. These blend modes will be useful for performing true alpha blending inside the texture stages. Our main focus at the moment is to configure the texture stages for frame buffer blending. The RGB and Alpha components output from the texture blending cascade become the two inputs into the alpha blending equation. This will merge the color output from the stage with the frame buffer.



## Alpha Blending with the Frame Buffer

So far we have seen where we can store alpha information and how we can select this information into the texture cascade. The texture stages allow us to select which color source we will extract the alpha from and they allow us to use multiple sources to blend the alpha components together to create new alpha values. After the final stage, an RGB color and an Alpha value are output to become the **SourceColor** and **SourceAlpha** colors in the frame buffer alpha blending equation.

If the **D3DRS\_ALPHABLENDENABLE** render state has not been enabled, then the **SourceAlpha** value output from the texture stage cascade is discarded. The source color is written to the correct location in the frame buffer and (assuming the depth test is passed) overwrites any previous pixel color that may already exist there.

If the **D3DRS\_ALPHABLENDENABLE** render state has been enabled, then the source color and source alpha values output from the texture stage cascade are fed into the frame buffer blending equation as potential multipliers. Let us remind ourselves of the calculation that generates the final pixel color written to the frame buffer when alpha blending is enabled:

$$\text{Pixel Color} = \text{SourceColor} * \text{SrcBlendMode} + \text{DestColor} * \text{DestBlendMode}$$

**SourceColor** is the RGB color output from the texture stage cascade and **DestColor** is the color of the pixel already in the frame buffer. We use the **SrcBlendMode** and **DestBlendMode** values to control how much of the **SourceColor** and how much of the **DestColor** are used to create the final color.

We also briefly discussed how to use the alpha value output from the texture stage cascade. The following is the standard alpha blending equation used for controlling the ratio of the SourceColor and DestColor used to create the final color:

$$\text{PixelColor} = \text{SourceColor} * \text{SourceAlpha} + \text{DestColor} * (1 - \text{SourceAlpha})$$

The above blending approach allows us to perform RGB independent transparency effects and is the most common blend mode configuration for generating transparency effects. The equation will be configured using the following render states:

```
pDevice->SetRenderState ( D3DRS_ALPHABLENDENABLE , TRUE );
pDevice->SetRenderState ( D3DRS_SRCBLEND    , D3DBLEND_SRCALPHA );
pDevice->SetRenderState ( D3DRS_DESTBLEND   , D3DBLEND_INVSRCALPHA );
```

**Note:** It is also possible for the frame buffer or render target to have an alpha channel. You could instead choose this alpha value (called **DestAlpha**) to control blending, but this is not as common.

There are certainly many more ways that we can setup the alpha blending modes. For example, you may recall that in Chapter 6 we used blend modes that did not use the **SourceAlpha** value output from the texture stage cascade. Instead we performed frame buffer blending using the colors themselves as



scaling factors. In the following example, although alpha blending is enabled, the alpha value output from the texture stages is not used at all:

**PixelColor = SrcColor \* SrcColor + DestColor \* (1-SrcColor)**

```
pDevice->SetRenderState ( D3DRS_ALPHABLENDENABLE , TRUE);
pDevice->SetRenderState ( D3DRS_SRCBLEND     , D3DBLEND_SRCCOLOR);
pDevice->SetRenderState ( D3DRS_DESTBLEND    , D3DBLEND_INVSRCOLOR);
```

Again, there are many blending combinations available since each blend mode can be used as either the source blend mode or the destination blend (or both). We discussed many of these in Chapter 6 so please refer back to that lesson when necessary.

Let us finally look now at the blending modes available when working with the source alpha value output from the texture stages (such as `D3DBLEND_SRCALPHA` for example). We see the complete `D3DBLEND` enumerated type below. Most of these blend modes were covered in Chapter 6.

```
typedef enum _D3DBLEND
{
    D3DBLEND_ZERO = 1,
    D3DBLEND_ONE = 2,
    D3DBLEND_SRCCOLOR = 3,
    D3DBLEND_INVSRCOLOR = 4,
    D3DBLEND_SRCALPHA = 5,
    D3DBLEND_INVSRCALPHA = 6,
    D3DBLEND_DESTALPHA = 7,
    D3DBLEND_INVDESTALPHA = 8,
    D3DBLEND_DESTCOLOR = 9,
    D3DBLEND_INVDESTCOLOR = 10,
    D3DBLEND_SRCALPHASAT = 11,
    D3DBLEND_BOTHSRCALPHA = 12,
    D3DBLEND_BOTHINVSRCALPHA = 13,
    D3DBLEND_BLENDFACTOR = 14,
    D3DBLEND_INVBLENDFACTOR = 15,
    D3DBLEND_FORCE_DWORD = 0xffffffff
} D3DBLEND;
```

**D3DBLEND\_SRCALPHA** ( $A_s, A_s, A_s, A_s$ ).

This blend mode multiplies every component in the color (either source or destination color depending on whether it is being used as a source blend or destination blend mode) with the alpha component of the source color.

**D3DBLEND\_INVSRCALPHA** ( $1 - A_s, 1 - A_s, 1 - A_s, 1 - A_s$ ).

Every color component in the color (source or destination color) is multiplied with the result of 1 minus the alpha component of the source color.



### D3DBLEND\_DESTALPHA ( $A_d, A_d, A_d, A_d$ ).

If the frame buffer or render target contains an alpha channel, the destination color (the color of the pixel currently in the frame buffer) may also have an alpha value which could be used in the blending equation. This is not commonly used, but can be useful for some advanced effects.

### D3DBLEND\_INVDESTALPHA ( $1 - A_d, 1 - A_d, 1 - A_d, 1 - A_d$ ).

The source or destination color is multiplied by the result of 1 minus the destination alpha component.

### D3DBLEND\_BOTHSRCALPHA

Obsolete. You can achieve the same effect by setting the source and destination blend factors to D3DBLEND\_SRCALPHA and D3DBLEND\_INVSRCALPHA in separate calls.

### D3DBLEND\_BOTHINVSRCALPHA

Source blend factor is  $(1 - A_s, 1 - A_s, 1 - A_s, 1 - A_s)$  and destination blend factor is  $(A_s, A_s, A_s, A_s)$ . The destination blend selection is overridden. This blend mode is supported only for the D3DRS\_SRCBLEND render state. This is a single mode that performs the reverse of the **SrcAlpha** and **InvSrcAlpha** modes used for standard alpha blending. With this blend mode, the larger the alpha value the more transparent the source pixel will appear to be.

### Alpha Blending with the Frame Buffer

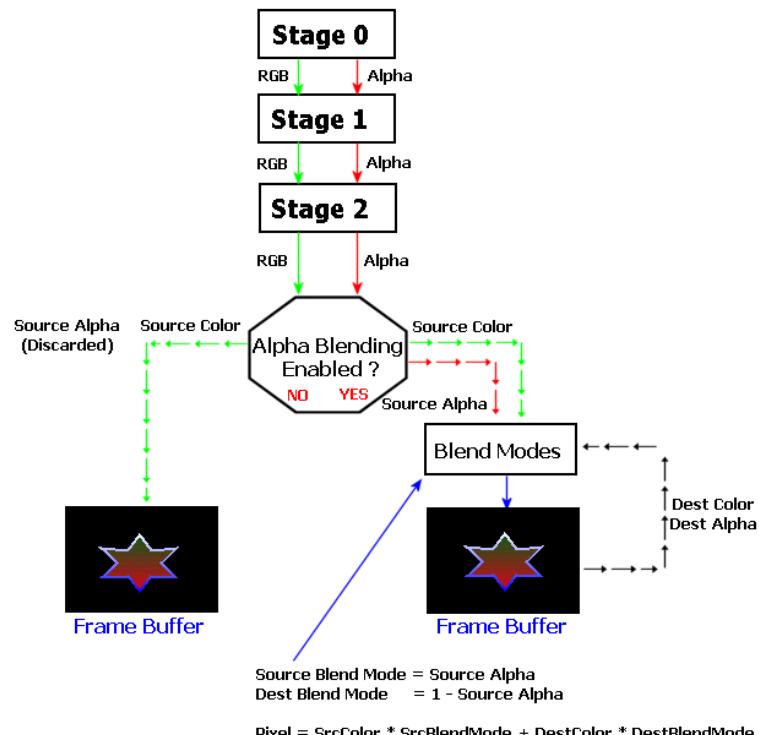


Figure 7.7



Fig 7.7 shows the complete picture of alpha blending with the frame buffer. As discussed, alpha values can be stored in vertices, textures, or constants. The texture stages manipulate the RGB and Alpha values to create a final output color consisting of RGB and Alpha components. We refer to these as SourceColor and SourceAlpha respectively.

If alpha blending is not enabled, then the SourceAlpha value is ignored and the RGB color is written to the frame buffer (and converted to the correct pixel format). If the frame buffer does have an alpha channel, then the alpha value will also be written to the frame buffer where it could be used as a color blending argument with the D3DBLEND\_DESTALPHA blending modes.

If alpha blending is enabled then the color and alpha components are fed into the alpha blending equation where the source color and destination color (the color already in the frame buffer) are blended together. The alpha value output from the stage can be used in the blending equation to scale the colors contributions for the final pixel color. This controls the transparency of the source pixel.

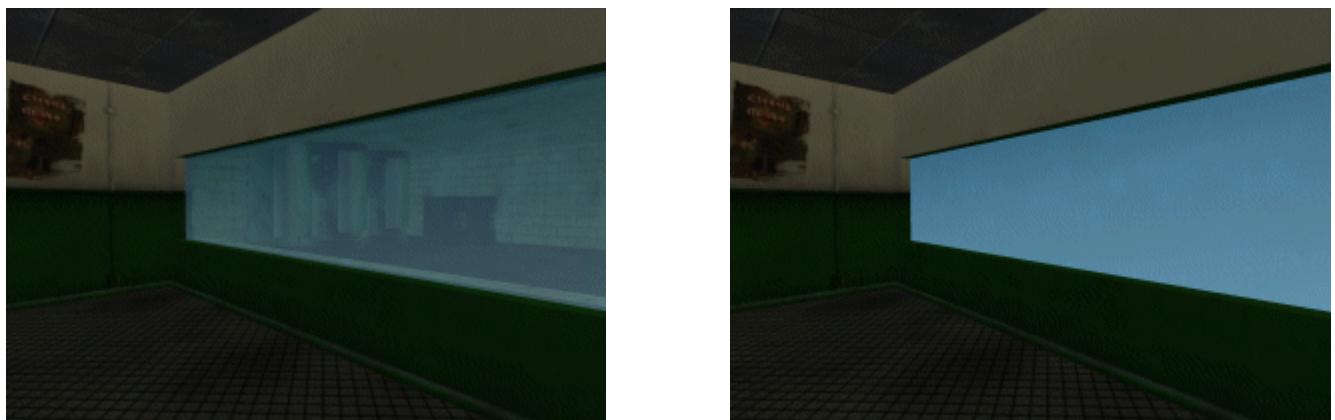
Please turn to your workbooks and examine Lab Project 7.1. This first demonstration will add a transparent water layer to our terrain application using per-vertex alpha components.

### Alpha Ordering

Transparency effects with alpha blending present certain obstacles. Regardless of how transparent or opaque the output color of the alpha blending equation, when it is written to the frame buffer, its pixel is also written to the depth buffer. We will see later in this chapter that we will take steps to render our non-alpha polygons first and then render the alpha polygons in a second pass. Additionally, we will usually have to sort the alpha polygons before they are rendered so that the furthest alpha polygons are rendered first. This is quite logical of course. Let us say for example that we have a red window polygon which is partially transparent. We know that the objects on the other side of the window from the camera should be tinted red as a result. This will only happen if the window is rendered after all of the polygons behind it, so that its color can be blended with the colors already in the frame buffer.

Consider what would happen if we did not order our rendering such that alpha polygons were rendered last. In this example our red window might be the first polygon rendered. At this point the frame buffer would be empty and may have been cleared to a black color. The alpha blending equation would blend the red color of the window with the black color of the frame buffer resulting in a very dark red color. This would in no way represent the color that it should be because in the scene there may be a bright blue polygon immediately behind the window. So in reality there should be a blend of red and blue and not red and black, but this is what we would have. In fact, the problem can get much worse. Now imagine that in this scenario the window polygon also has its Z values written to the Z-Buffer. When we then try to render the polygons on the far side of the window, they will fail the depth test because they are further away from the camera than the window polygon that is already in the frame buffer. As a result we lose the ability to render anything behind our supposedly transparent window.

Fig 7.8 shows a scene with transparent polygons. The image on the left renders all alpha polygons in the scene after it has rendered the opaque polygons. Because all opaque polygons that are behind the window are rendered first, when the window is rendered, the frame buffer contains what would truly be behind the window. The alpha blending equation then correctly blends the color of the window with the color of all objects behind the window already in the frame buffer.



**Figure 7.8**

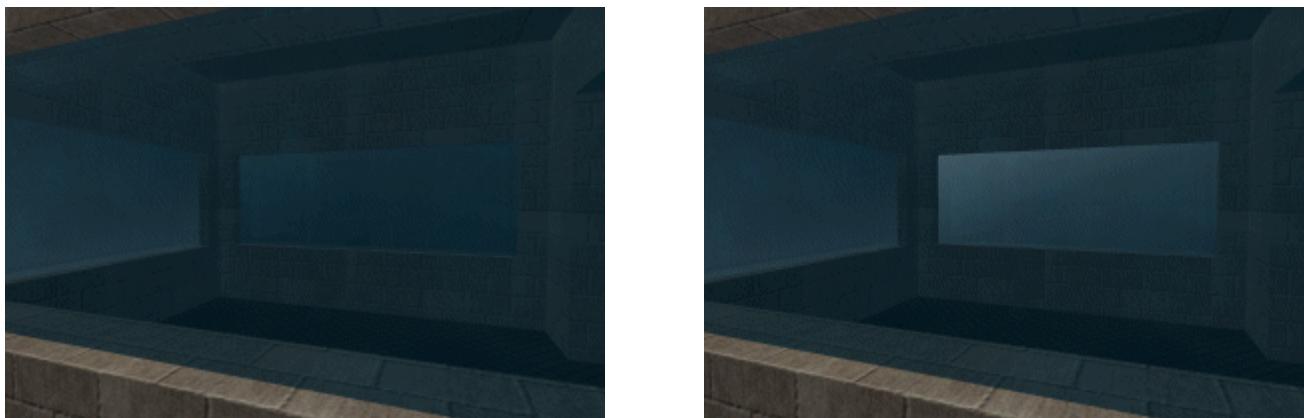
The image on the right is the same scene but with no particular rendering order applied to the polygons. We can see that the blue window has been rendered before the polygons on the other side of it. Even though the polygon is transparent, at the time it was rendered there was nothing in the frame buffer. So its pixels were simply blended with the empty frame buffer color. The depth of each pixel was stored in the depth buffer and all polygons rendered after the window that are further away from the camera are rejected by the depth test. This is an important point to remember. Just because the color of a pixel includes an alpha value, it does not alter the behaviour of the Z-Buffer. We can clearly see here why it is important to render our alpha polygons after all of the non-alpha polygons.

While rendering our alpha polygons after our opaque polygons would seem to be an easy solution to the problems discussed above, there are additional considerations. If any alpha polygons occlude each other, then we will also have to render the alpha polygons such that they are sorted and rendered back to front with respect to their distance from the camera. The artifacts that result if we do not do this are not quite as obvious as those seen in Fig 7.8 but they can still be severe under certain circumstances.

Fig 7.9 shows a case where one transparent window occludes another transparent window. Both windows have slightly different colors. In our alpha rendering pass, we should render the furthest window from the camera first and render the closest window to the camera last. This way the second window blends its color with the color of the first window -- which itself was blended with the color of the frame buffer opaque polygons rendered previously.

One might think that we could avoid any problems if we disabled Z buffer writing when rendering the alpha polygons. If we did this, an alpha polygon already in the frame buffer that is closer to the camera

would not cause another alpha polygon behind it (but rendered afterwards) to be rejected by the depth test. While this will certainly prevent transparent polygons from occluding each other, it will still result in incorrect color blending (Fig 7.9).



**Figure 7.9**

The image on the left has been rendered correctly. The furthest window was rendered first, which tinted the frame buffer pixels appropriately. The closest window was rendered last. Its pixels were blended with the pixels in the frame buffer -- which are themselves the result of a blend between the first window and the opaque polygons behind it. As we can see, when we are sorting our alpha polygons, the transparency results are correct even if two or more alpha polygons overlap from a given viewpoint.

In the image on the right, after drawing the opaque polygons, the alpha polygons are rendered in the order in which they are stored in the file. In this instance the closest window was rendered first and blended with the opaque polygons in the frame buffer. Then the second window was rendered and blended with the contents of the frame buffer also. The second window was not rejected by the Z-Buffer tests because we disabled Z writes during the rendering of the alpha polygons. So when we rendered the closest window first, its depth values were not recorded in the depth buffer. Although this certainly prevents the second window from failing the depth test, it still results in an incorrect blend because the two windows are blended in the wrong order. Remember, when we disable Z writes, the alpha polygons will still be tested against the depths that are currently in the buffer and rejected if occluded by any of the opaque polygons that were rendered in the first pass. If the alpha polygon is not occluded, it is rendered but its depth values are not written to the depth buffer. So from a depth buffer perspective, it is as though the polygon is not there.

While the artifacts in the image on the right may not initially strike you as incorrect, we can certainly see when we compare the two images that the blending results are not the same. Therefore, when we have alpha polygons, we will usually want to sort them before rendering. We will discuss sorting strategies later in this chapter.

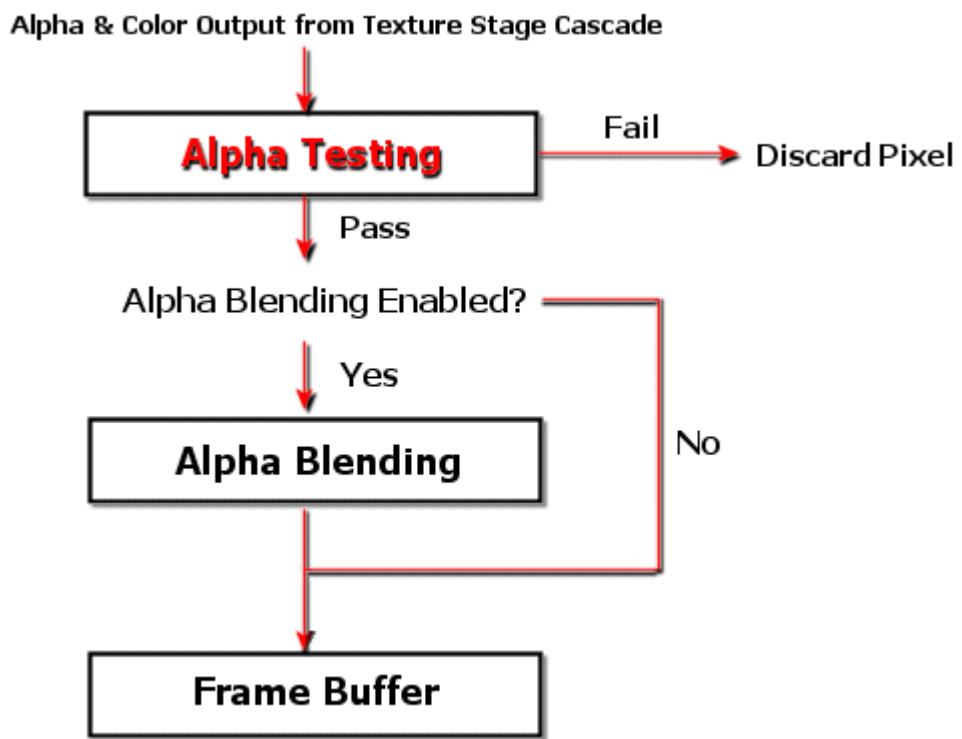


## Alpha Testing

Very often we will be rendering polygons that have a level of transparency such that the pixels are either fully opaque or fully transparent. Think about a texture of a tree mapped to a quad for example. We would want the pixels in between the leaves and branches to be totally transparent so that when rendered on top of the scene, the scene polygons between the branches are visible. The actual branches and leaves themselves however would be fully opaque, occluding anything underneath them in the frame buffer. The DirectX pipeline provides an alpha testing function which is disabled by default when the device is first created. When it is enabled, it is actually executed before the alpha blending equation that calculates the final pixel color.

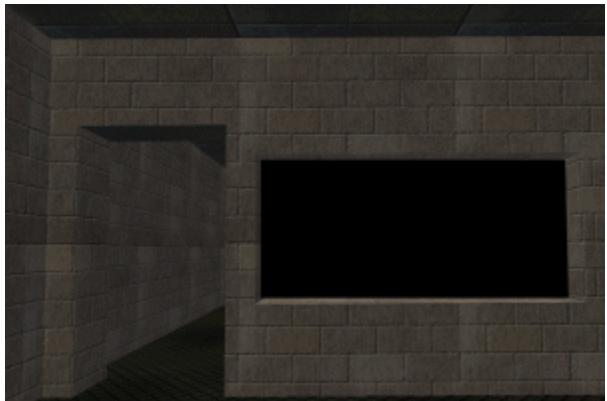
As we can see in the diagram, the color and alpha values are output from the texture stage cascade. If alpha testing is enabled then a comparison is made between the alpha value and some reference value that our application provides. We can specify the nature of the comparison function that is performed between the alpha value and the reference value. A common alpha testing comparison function is `D3DCMP_GREATEREQUAL`.

In this case, if the alpha value output from the texture stages is greater than or equal to the reference value, then the pixel passes the test and continues down the pipeline. Notice that Alpha Testing is completely separate from alpha blending. You do not need to have alpha blending enabled to use alpha testing and vice versa.



If a pixel is rejected by the alpha testing mechanism then its depth value will not be written to the Z-Buffer and it will not occlude any pixels behind it. Thus these pixels are not rendered or stored. This is useful when we have polygons where the alpha values are providing simple on/off transparency.

We can use alpha testing to mask out pixels that have an alpha value less than the reference value. For the sake of example, let us imagine that we have polygons that are black and have alpha values of 0 for each pixel (totally transparent).



The image on the left shows a scene rendered using no particular sorting order. In this image the black window polygon is totally transparent. Because it is rendered before all other polygons in the scene it is simply alpha blended with a black frame buffer. In this example, Z writing is not disabled so the window rendered to the frame buffer has its Z values written to the Z-Buffer and it occludes the polygons behind it.

Let us see what happens when we enable alpha testing so that any pixels with an alpha value of anything less than 255 (totally opaque) get rejected by the pipeline – they are never added to the frame buffer or the depth buffer.

We enable alpha testing using the **D3DRS\_ALPHATESTENABLE** render state.

```
pDevice->SetRenderState( D3DRS_ALPHATESTENABLE , TRUE );
```

The next step is to set the reference value. In this example we will reject pixels that are not totally opaque so we will set the alpha reference value to 255 and the alpha testing comparison function to **D3DCMP\_GREATEREQUAL**. The default alpha testing comparison function is **D3DCMP\_ALWAYS** which means that every pixel always passes the alpha test regardless of its alpha value.

```
pDevice->SetRenderState( D3DRS_ALPHAREF , (DWORD)0x000000FF );
pDevice->SetRenderState( D3DRS_ALPHAFUNC , D3DCMP_GREATEREQUAL );
```



The image on the left shows the result of using alpha testing. As you can see, all of the places where there are windows, nothing is rendered. The polygons that are rendered afterwards that occupy the same region of the frame buffer are still correctly rendered underneath. When using totally transparent surfaces or polygons that have totally transparent regions (for example, if it has a texture with an alpha channel where certain pixels within that texture have alpha values of 0), we do not have to sort these polygons.

Of course, alpha testing is not limited to masking out totally transparent pixels. We could set the reference value to any arbitrary value such that only pixels above (or below, or even equal to) the reference value pass the alpha test. Therefore, alpha testing provides us with a way to mask pixels regardless of the origin of the alpha value (texture alpha channel, interpolated diffuse or specular alpha, constant alphas from the **D3DRS\_TEXTUREFACTOR** render state, etc.).



We can control the alpha testing function using the **D3DRS\_ALPHAFUNC** render state. We pass in one of the members of the **D3DCMPFUNC** enumerated type shown below.

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

As you can see, there are a number of comparison options at our disposal. In Lab Project 7.2 we will use alpha testing with alpha values stored in a texture alpha channel.

There are some final points to keep in mind about alpha testing pixels that use on/off transparency (0x00 or 0xFF). First, we do not have to worry about disabling Z writes when rendering these polygons because the pixels will be rejected before they are written to the depth buffer. Second, if we are using simple on/off transparency, there will be no color blending for transparent pixels. As a result there is no need to render these polygons in a second pass after the opaque polygons. This in turn means there is also no need to sort them before they are rendered.

### Transparent Polygon Sorting

As discussed earlier, when we render scenes that include partially transparent polygons, we will need to use alpha blending and render all opaque polygons first. We will then render our alpha polygons in a second pass in back-to-front order with respect to the camera. When using alpha blending, the order we blend the polygons into the frame buffer is significant. For example, imagine that we have a blue frame buffer and we render an alpha blended green polygon followed by an alpha blended red polygon in the same location. We would not get the same result if we reversed the rendering order as the following demonstrates.

$$\text{Pixel Color} = \text{SourceColor} * \text{SrcAlpha} + \text{DestColor} * (1 - \text{SrcAlpha})$$

#### Example 1: Blue \* Green \* Red

*Blend Green Quad First*

$$\text{Pixel Color} = \text{GreenQuad} * 0.5 + \text{BlueFrameBuffer} * (1 - 0.5)$$

$$\text{Pixel Color} = (0, 1.0, 0) * 0.5 + (0, 0, 1.0) * 0.5$$

$$\text{Pixel Color} = (0, 0.5, 0) + (0, 0, 0.5)$$

$$\text{Pixel Color} = (0, 0.5, 0.5)$$

Pixel Color = Murky Blue Color



### ***Now Blend Red Quad***

```
PixelColor = RedQuad * 0.5 + MurkyBlue * (1 - 0.5)
PixelColor = ( 1.0 , 0 , 0 ) * 0.5 + ( 0.0 , 0.5 , 0.5 ) * 0.5
PixelColor = ( 0.5 , 0 , 0 ) + ( 0.0 , 0.25 , 0.25 )
Pixel Color = ( 0.5 , 0.25 , 0.25 )
Pixel Color = Brownish Color
```

### **Example 2: Blue \* Red \* Green**

#### ***Blend Red Quad First***

```
Pixel Color = RedQuad * 0.5 + BlueFrameBuffer * (1 - 0.5)
Pixel Color = ( 1.0 , 0 , 0 ) * 0.5 + ( 0 , 0 , 1 ) * 0.5
Pixel Color = ( 0.5 , 0 , 0 ) + ( 0 , 0 , 0.5 )
Pixel Color = ( 0.5 , 0 , 0.5 )
Pixel Color = Half Intensity Purple
```

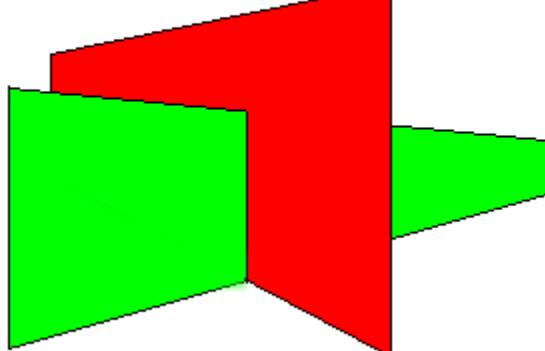
#### ***Now Blend Green Quad***

```
PixelColor = GreenQuad * 0.5 + Half Intensity Purple * (1 - 0.5)
PixelColor = ( 0 , 1 , 0 ) * 0.5 + ( 0.5 , 0.0 , 0.5 ) * 0.5
PixelColor = ( 0 , 0.5 , 0 ) + ( 0.25 , 0 , 0.25 )
Pixel Color = ( 0.25 , 0.5 , 0.25 )
Pixel Color = Grayish Green Color
```

The different colors generated provide clear evidence that we really do need to render our alpha polygons in back-to-front order if they are going to overlap.

In Chapter 2 we saw that without a depth buffer, we would need to use a back-to-front rendering technique like the Painters Algorithm. We also said that sorting all of our polygons every frame would

be slow and that even if we did sort our polygons before rendering, there would be cases where rendering order could not be fully resolved. The image to the left reminds us of one such case. Although the depth buffer allowed us to forget about these problems by providing depth testing at the pixel level, we now find ourselves in a similar situation once again, with the need to sort our alpha polygons back to front.



Although optional, we will usually want to disable Z writes when rendering alpha polygons in the second pass so that they do not occlude anything in the depth buffer. As transparent objects are supposed to be non-occluders by their very nature, it is often best not to write



their depth values. Disabling Z writes can speed up rendering a little as well because although the depth test is still done, the pipeline does not have to update the pixel in the depth buffer each time. In Lab Project 7.3 we will disable Z writes when rendering the alpha polygon list. This is done with via a render state call:

```
m_pD3DDevice->SetRenderState(D3DRS_ZWRITEENABLE, FALSE); //Z Writes Off  
m_pD3DDevice->SetRenderState(D3DRS_ZWRITEENABLE, TRUE); //Z Writes On
```

As it turns out, there is no correct order that can be determined in the above example. A common way to solve the problem is to compile the alpha polygons into a BSP tree, which splits polygons along the planes of other polygons. In the example above, the green polygon would actually be split into two new polygons – one on the front side of the red polygon, the other on the backside. Sorting by polygon then becomes straightforward again. However, compiling BSP trees is beyond the scope of this course and will not be discussed until the third course in this series. Besides, while a BSP tree is an elegant solution, it is not always necessary. Because we are rendering only the alpha polygons without the assistance of a depth buffer, the odds of such overlaps are pretty small. But even if there are overlaps, often the scene can be modified in the world editor to remove or rearrange them if the game engine does not want to address this issue. If such polygons are few and far between, it is sometimes preferable to live with the fact that the blending may be incorrect at times. Even in some commercial titles, if you know what you are looking for, you can see places where alpha polygons are sometimes blended in an ‘incorrect’ way. The errors produced are often quite subtle and generally go unnoticed unless you are specifically looking for them. Therefore, we will decide -- for the time being at least -- that we will proceed with a higher level technique like the Painter’s Algorithm and if such a situation should arise, we will accept any blending errors produced by intersecting alpha polygons.

Before moving on, it is also worth mentioning at this point that certain effects may benefit from disabling the Z buffer completely while they are being rendered. The water polygon in Lab Project 7.1 is a good example. In that case we are drawing a very large polygon, potentially over the entire viewport. Since we know in advance that all of its pixels will pass the depth buffer test anyway since nothing in the scene can occlude them (as is often the case with many such screen space effects – user interfaces, etc.) there is no point in running all of those per-pixel depth tests. We would prefer to simply forget about testing the polygon pixels and writing them to the depth buffer at all. To deactivate the depth buffer entirely (writes would be also be deactivated as a result), another render state is used:

```
m_pD3DDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_FALSE); //Z Buffer Deactivated  
m_pD3DDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE); //Z Buffer Activated
```

Whether you choose to deactivate depth buffer writing or the entire depth buffer testing/writing process is a function of the particular effect you are trying to achieve.

## Sorting Criteria

Now that we have decided to sort the alpha polygons and render them back-to-front using the Painter's Algorithm, the next question is, how will we determine if one polygon is behind or in front of another? This is actually another case where a BSP tree would help, but usually, we cheat a little and simply sort based on distance from the camera to the center of the polygon. This approach comes with some risks as a very large polygon may be overlapping a smaller polygon that is more distant. Because the large polygon is so large, its center point may be further away from the camera than the smaller polygon center point. But its nearest edge may be closer to the camera, or vice versa, as shown in Fig 7.10.

### The trouble with sorting by center points

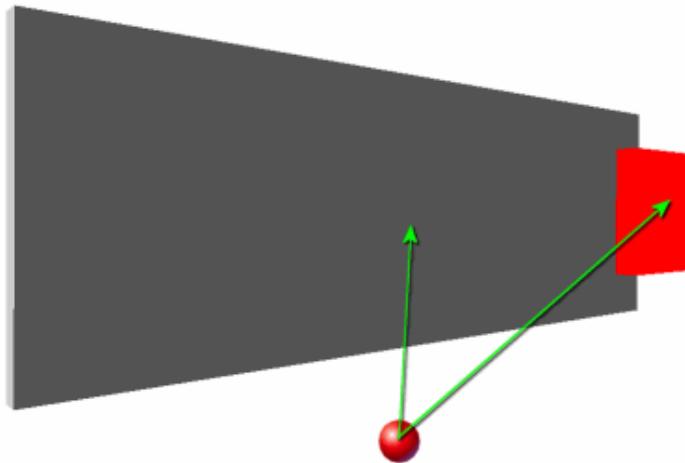


Figure 7.10

In Fig 7.10 we see a camera position (the red sphere) and a large gray polygon and a small red polygon. From the camera's perspective, the red polygon is overlapping the gray polygon such that if we were using alpha blending, we would want to render the gray polygon first and then alpha blend the red polygon on top of it. However, we can see that because the sizes of the polygons are so different, the distance from the camera position to the center point of the gray polygon is shorter than the distance from the camera position to the center point of the red polygon. When we render back to front such that polygons that have greater distances to their center points are rendered first, the red polygon would be rendered before the gray polygon. This produces incorrect blending results. Fortunately, since alpha polygons are relatively few in number this is another situation that does not typically arise very often and as such is something we can often live with.

## Calculating the Polygon Center

Calculating a polygon center point can be done using a simple averaging technique. Simply sum the polygon vertex positions and divide the result by the number of vertices. For example, in Lab Project



7.3, where we store geometry as triangle lists, we calculate the center point of a triangle using the following code:

```
D3DXVECTOR3 CalculateCenterPoint(D3DXVECTOR3 *v1, D3DXVECTOR3 *v2, D3DXVECTOR3 *v3)
{
    D3DXVECTOR3 CenterPoint;
    CenterPoint = *v1;
    CenterPoint += *v2;
    CenterPoint += *v3;
    CenterPoint = CenterPoint / 3;

    return CenterPoint;
}
```

If the function is passed the three world space vertex positions of a triangle, it will return a world space vector describing the position of the center point of the triangle. The center points are world space center points that will be used during rendering to subtract the position of the camera. This will return a vector whose magnitude describes the distance from the camera position to the center point of a polygon. The center points do not need to be updated when the camera position changes because the world space center points of the alpha polygons do not change unless the alpha polygons are animated or modified in some way. Usually we can calculate and store the center points of all alpha polygons as a pre-process. There is no need to store the center points for the opaque polygons as we do not need to sort them.

In our lab project, when we encounter alpha polygons during the first render pass we will simply store them for later use. After the opaque polygons have been rendered, we will sort the alpha polygons based on their distance from the camera. For each polygon, we will calculate its distance from the camera by subtracting the camera position vector from the alpha polygon center point. This builds a vector that describes the direction and magnitude of travel from the camera to the center point to the polygon. The length of this vector is the distance to the center point from the camera. We can store that alpha polygon in a list or an array along with its distance. The next step will be to sort that list and then render it in back-to-front order.

## Performance Concerns

Certainly we will want to do the distance calculations and the center point sorting as quickly as we possibly can. As discussed in Chapter 1, calculating the length of a vector is not a fast process. It involves first calculating the squared length of the vector by doing  $(X^*X) + (Y^*Y) + (Z^*Z)$  and then performing a square root on the result. Unfortunately, the square root operation is one of the slower math routines for a computer to perform and it would surely be preferable if we could avoid it.

If we need the true distance value, then this cost is something we simply have to accept. However, we do not need the true distance. We only need a number that we can use to accurately describe ordering information, such that the relationship between all polygon distances is maintained. So instead of calculating the actual distance, we will calculate the squared distance instead. This avoids the square root and simply returns  $(X^*X) + (Y^*Y) + (Z^*Z)$ . As it happens, just as D3DX contains a function to



return the true length of a vector (`D3DXVec3Length`), it also includes a function for calculating the less computationally expensive squared length of a vector.

```
FLOAT D3DXVec3LengthSq(CONST D3DXVECTOR3 *pV);
```

Like its counterpart, this function accepts the address of the vector you wish to know the length of. The function returns a single floating point value describing the squared length of the vector.

When we encounter an alpha polygon during our opaque polygon render pass, we will subtract the camera position from the polygon center point and call the above function to calculate the squared length of the resulting vector. This is the distance we will store with the polygon and use for sorting.

## Choosing a Sorting Algorithm

After we have created our unsorted list of alpha polygons, we must decide which sorting algorithm to use. We certainly want an efficient routine that scales well since we may have a scene with many hundreds of alpha polygons that need to be drawn.

There are many well-known sorting algorithms that programmers use to sort data. We will briefly discuss two of the most popular (and extreme): the bubble sort and the quick sort. Other sorting algorithms usually perform somewhere between the bubble sort and the quick sort. The bubble sort is just about the slowest choice and the quick sort is typically one of the fastest. Of course much of this depends on the data set being sorted, but in general, this is usually the case.

The quick sort is very fast for arbitrary data sets, although it can be difficult to understand due to its recursive nature. The bubble sort is generally much easier to understand and code because it uses a simpler algorithm.

One thing to keep in mind is that if the quick sort algorithm is implemented in a recursive fashion, large data sets may cause a stack overflow. However, if we did utilize a quick sort for our alpha polygons, we would probably not be in any danger due to the fact that we are typically only sorting a handful of alpha polygons at any one time. But even if we were rendering thousands of alpha polygons each frame, the quick sort would still be a fine choice.

## The Bubble Sort

The bubble sort is probably the simplest sorting algorithm to implement. It generally performs best when passed an already sorted or nearly sorted list, where its level of complexity can approach a constant  $O(n)$ . The more general case however is  $O(n^2)$ . While other sorting algorithms also have complexities of  $O(n^2)$ , the bubble sort is significantly less efficient than most of the alternatives.

The bubble sort works by visiting each item in the list, comparing its value to the item next to it, and swapping them if required. This depends on whether you want the list sorted in ascending or



descending order. This process is repeated until we pass through the list and find that no items need to be swapped.

If we think about smaller values sorted towards the beginning of the list and higher values pushed towards the end of the list, we can see how this sorting algorithm gets its name. A higher value slowly sinks towards the end of the list with each sorting pass while smaller values slowly bubble up towards the front of the list. The following code example shows how one might implement a function that performs a bubble sort on a passed array of integers.

```
void bubbleSort(int ListOfNumbers[], int ListSize)
{
    int a, b, temp;

    for (a = (ListSize - 1); a >= 0; a--)
    {
        for (b = 1; j <= a; j++)
        {
            if (ListOfNumbers[b-1] > ListOfNumbers[b])
            {
                temp = ListOfNumbers[b-1];
                ListOfNumbers[b-1] = ListOfNumbers[b];
                ListOfNumbers[b] = temp;
            }
        }
    }
}
```

Although easy to understand and implement, the bubble sort should definitely not be used for repetitive sorts or for sorting lists than contain more than a few hundred items. It certainly should not be used in time critical code (like our render loop).

## The Quick Sort

The quick sort is the fastest of the common sorting algorithms. While it is possible to write special purpose optimized sorting techniques that work with specific data sets to outperform quick sort, for the general case, the quick sort is the top choice when it comes to speed. The quick sort uses an approach that is relatively easy to understand in words but is harder to grasp in code. Essentially, it is a divide-and-conquer technique that is normally implemented using recursion. The algorithm implements the following basic steps to achieve the sort:

- If the list has only one (or zero) elements then exit -- the list is already sorted.
- Select a number in the list as the Pivot Point.
- Split the list into two lists (left and right).
- The right list contains numbers higher than the Pivot Point.
- The left list contains values lower than the Pivot Point.
- Recursively repeat the above steps on the two child lists.



So the quick sort repeatedly chooses a Pivot Point, reorganizes the numbers in the array such that values smaller than the Pivot Point end up on the left side, while higher values end up on the right side. We then recursively send the two sections of the list (to the left and right of the Pivot Point) into the quick sort function again and repeat the process until the list is completely sorted.

The efficiency can depend very much on the value chosen as the Pivot Point for a given data set. Often the leftmost value in the list will simply be selected as the Pivot Point by default. In the absolute worst case, complexity can approach  $O(n^2)$ . This occurs when the leftmost value is selected as the Pivot Point and the list passed into the quick sort is already perfectly sorted. If you suspect that the data passed into the quick sort will be mostly sorted, you are better off choosing a random Pivot Point from the list. As long as the Pivot Point is chosen randomly, the quick sort has a complexity of  $O(n \log n)$ . Below, we see a simple quick sort implementation:

```
void QuickSort ( int ListOfNumbers[], int ListSize )
{
    QSort( ListOfNumbers, 0 , ListSize - 1);
}

void QSort ( int ListOfNumbers[], int Left , int Right )
{
    int PivotPoint, LeftStart, RightStart;

    LeftStart = Left;
    RightStart = Right;
    PivotPoint = ListOfNumbers[Left];

    while ( Left < Right )
    {
        while ( (Left < right) && ( ListOfNumbers[Right] >= PivotPoint ) ) Right--;
        if ( Left != Right)
        {
            ListOfNumbers[Left] = ListOfNumbers[Right];
            Left++;
        }
        while ( ( Left < Right ) && (ListOfNumbers[Left] <= pivot) ) Left++;
        if ( Left != Right )
        {
            ListOfNumbers[Right] = ListOfNumbers[Left]; Right--;
        }
    }

    ListOfNumbers[Left] = PivotPoint;
    PivotPoint = Left;
    Left = StartLeft;
    Right = StartRight;
    if ( Left < PivotPoint) QSort ( ListOfNumbers, Left , PivotPoint -1 );
    if ( Right > PivotPoint) QSort ( ListOfNumbers, PivotPoint+1, Right );
}
```



If you do not immediately understand the code above, do not panic. Most people find that they have to sketch it out with a pen and paper to understand what is happening the first time around. Fortunately, we do not have to implement our own quick sort function since it has been a part of the C standard for quite some time. The function is called **qsort** and to use it all you have to do is include stdlib.h. The **qsort** function accepts four parameters as shown below:

```
void qsort
(
    void *base,
    size_t num,
    size_t width,
    int (*__cdecl *compare )(const void *elem1, const void *elem2 )
);
```

The first parameter should be a pointer to the array of elements that you wish to sort. The second parameter describes the number of elements in the array. The third parameter describes the size in bytes of a single element in the array. The fourth parameter is a pointer to an application defined callback function that returns an **int** and accepts two **void** pointers. For every element comparison that needs to take place during the quick sort, the callback is called to determine if element 1 is larger, smaller, or equal to element 2. This is necessary because the **qsort** function has no idea about the data you are sorting (it has only a void pointer). You can use the **qsort** function to sort arrays of arbitrary structures because you are responsible for writing the callback function that you register with **qsort**. The callback function should return the following integer values to describe the relationship between element 1 and element 2.

Return Value	Description
< 0	<i>elem1</i> less than <i>elem2</i>
0	<i>elem1</i> equal to <i>elem2</i>
> 0	<i>elem1</i> greater than <i>elem2</i>

Here is a bit of code to demonstrate the use of the **qsort** function -- just in case you decide to use it to sort your own game objects.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int compare(const void *arg1, const void *arg2); // Our compare call back function
int List[5] = { 9 , 3 , 100 , 1 , 5 };
```



```
void main( void )
{
    int i;

    /* Sort the list using our 'compare' function for comparisons */
    qsort( (void *) List, 5, sizeof( int ), compare );

    /* Output sorted list: */
    for( i = 0; i < 5; ++i )    printf( "%d ", List[i] );
    printf( "\n" );
}

int compare( const void *arg1, const void *arg2 )
{
    if ( (int)*arg1 > (int) *arg2 ) return 1;      // Arg 1 greater than Arg 2
    if ( (int)*arg1 < (int) *arg2 ) return -1;     // Arg 2 greater than Arg 1
    return 0;                                       // Arg1 = Arg2
}
```

## Hash Table Sorting

Lab Project 7.3 will not use a bubble sort or quick sort to sort alpha polygons. It will use a data structure called a **hash table**. Hash table sorting is really a rather simple and elegant idea and it works quite well in this particular situation. Hash tables provide fast random access to data. They do this through the use of what is known as a **hash function**. This function accepts some data element as input and then converts it into a number that indicates where in the table that data should reside. This number is called a **hash key**. Depending on the nature of the data to be stored, along with any number of other factors, hashing functions can range from the very simple to the very complex. Fortunately for us, our hashing function will turn out to be quite straightforward.

At the end of our first render loop, we have an unsorted list of alpha polygons. Each polygon is stored with its squared distance from the camera. Let us imagine that we had an array and that we could use the distance value as the index of the element in the array to which a given polygon should be assigned. By taking the distance value of each alpha polygon, mapping it into an integer range of between zero and the size of the array, and then inserting the polygon into the array at that location, polygons will be added to the array automatically sorted by distance. This is essentially the core of our hashing function. Polygons with higher distance values will be at the back of the list and polygons with smaller distance values will be at the front of the list. Then all we have to do is enable alpha blending and render the list, starting at the back and working towards the front -- rendering each alpha polygon stored at each index. There may of course be many elements in the array with no polygons assigned (a sparse array) and we would skip these because the element would be set to NULL. In effect, this gives us nearly free sorting.

Let us start off with an example that uses small values to better illustrate the process. Imagine that we have our far plane set such that it is a distance of 100 units away from the camera in view space. Also



imagine that our camera is at position <0,0,10> in world space and that there is an alpha polygon with a center point at <0,0,20> in world space. In this example we will use a hash table of 1000 pointers. Each element in the array is a pointer to a polygon that will (potentially) eventually be stored there. We can easily increase or decrease the size of our hash table to improve sorting speed and accuracy.

```
FarPlane          = 100
CameraPos         = (0, 0, 10);
AlphaPoly->CenterPoint = (0, 0, 40);
HashTableSize    = 1000;
Polygon **HashTable[ HashTableSize ];
```

For each alpha polygon encountered, we calculate the squared distance:

```
D3DXVECTOR3 DistanceVector = AlphaPoly->CenterPoint - CameraPos; // (0, 0, 30)
float PolyDistanceSq = D3DXVec3LengthSq(&DistanceVector);
// (0*0)+(0*0)+(30*30) = 900
```

Now we will map the distance value into the far plane range so that we end up with a value in the range [0.0, 1.0]. Thus a value of 1.0 would describe a polygon that has its center point right on the far plane. If we were not dealing with a squared distance we could just divide the polygon distance value by the far plane distance value. However the polygon distance is a squared distance and the far plane distance is not. So we will square the far plane distance and use the result to divide the polygon center point squared distance.

```
float fIndex = PolyDistanceSq / (FarPlane * FarPlane); // 900 / 10,000 = 0.09
```

Now, we simply map this floating point value to the size of our hash table array to generate our hash key. Our hash table in this example is 1000 elements in size, so we multiply the value by 1000 to take it into the integer range of [0, 999] and add the polygon to the hash table array at that index.

```
ULONG Index = fIndex * (HashTableSize-1); // 0.09 * 999 = 89;
HashTable[ Index ] = Poly;
```

Note that this approach does not map the distance values linearly to hash table indices. In fact, one might say that it suffers from the opposite problem of that of the depth buffer. Because we are using squared distances, polygons nearer the back of the scene will map to a larger range of hash table keys. In the above example, we had a non-squared distance of 30 from the camera to the polygon and a far plane distance of 100. We are using values that make it easy to see that the polygon center point is 30% of the distance between the camera and far plane. Therefore, with a hash table size of 1000 you would expect the index generated for this polygon to be 30% of the hash table array size  $1000/100*30 = 300$ . However, we can see that because we are using squared distances, we actually get a value of 89. Therefore, the first 30% of the scene in this example is mapped to the first 9% of the hash table array. We could of course decide to use non-squared distances but that would involve a square root per-polygon. To solve this problem we could increase the size of the hash table. This reduces polygon/index ambiguities and it will be less likely that polygons in the near distance will be mapped to the same key value in the hash table.



In reality, it is not so terrible if we end up with multiple polygon distances mapped to the same hash key (a hashing phenomenon known as a **collision**). After all, we are not limited to storing only a single polygon at each element in the hash table. The pointers themselves can simply be used as the head of a sorted linked list. This is exactly the technique that we will use in Lab Project 7.3.

Once we have calculated the index of an alpha polygon, we check the element in the array. If it is set to NULL, then we store the polygon pointer at that array element because no other polygon has previously been stored there. If the pointer is not NULL then we step through the linked list and compare the squared distances. Once we locate the exact place in the linked list where the alpha polygon belongs, we add it there. Polygons with smaller distance values are now at the tail of the linked list.

In summary, our render function loops through each polygon in the scene. If the polygon is opaque it is rendered. If the polygon is an alpha polygon, we calculate a hash table index (a hash key) based on the distance from the camera formula and add it to the hash table using the technique just discussed. Once we have finished looping through each polygon, the frame buffer will contain all opaque polygons and the hash table will contain all alpha polygons. Closer polygons will be at the top of the hash table and distant polygons will be towards the bottom of the table. At this point we loop through the hash table, from bottom to top, and render each polygon -- or linked list of polygons -- with alpha blending enabled. At the end of this process, our alpha polygons have been rendered and blended with the frame buffer correctly.

Fig 7.11 depicts a ten element hash table. It just so happens that in this example, ten polygons have been added to the table using some particular hash function.

### 10 Element Hash Table Example

0	Polygon 0 Distance = 5 →NULL
1	Polygon 1 Distance = 8 →NULL
2	Polygon 2 Distance = 10 →NULL
3	Polygon 5 → Polygon 4 → Polygon 3 →NULL Distance = 20      Distance = 17      Distance = 13
4	NULL
5	Polygon 6 →NULL Distance = 25
6	NULL
7	Polygon 7 →NULL Distance = 60
8	Polygon 9 → Polygon 8 →NULL Distance = 80      Distance = 70
9	NULL

Figure 7.11



Notice how each polygon in Fig 7.11 is stored in the list ordered by distance, and that some elements are NULL when no polygon distance maps to that index. Also notice that array elements 3 and 8 in this example have more than one polygon mapped to their hash key (i.e. there was a collision). For these cases we use a linked list sorted so that larger distance values are stored at the head of the list and smaller values are stored towards the tail.

The alpha polygons can be rendered in the correct back-to-front order using the following pseudo code. It loops through the hash table array in reverse order and traverses the linked lists stored at each index in the array.

```
POLYGON *Poly = NULL;
for ( ULONG i = HashTableSize-1 ; i >= 0 ; i -- )
{
    for ( Poly = ppHashTable[i]; Poly; Poly = Poly->Next )
    {
        Poly->Render();
    }
}
```

Note that while the hash table approach is useful, it is not a panacea. To be sure, you will wind up using other sorting algorithms (especially quick sort) many times in your coding projects. Please spend some time now looking over the code for Lab Project 7.3. Make sure that you understand the sorting algorithm and try to think about how you might use an alternative algorithm like the quick sort as well. Perhaps take the time to test both and compare performance.

## Alpha Surfaces

One very useful technique involves extracting alpha values in the texture stages from one or more textures that are not actually going to be used for color rendering. For example, you could have a terrain texture assigned to stage 0 that is modulated with the vertex color and rendered with the terrain polygons as normal. However, you could also have a second set of texture coordinates in your vertex used to sample the alpha value from a texture assigned to the second stage. This second texture might not actually be mapped to the polygon from a rendering perspective but could be used to describe how the sampled colors from the texture in the first texture stage blend with the frame buffer. We can do this using the alpha value sampled from the texture in the second texture stage.

In the following code snippet, we see an example of this technique. We have a normal color texture assigned to the first texture stage, and we assign a texture with an alpha channel to the second stage. We set the first stage to modulate the color sampled from the texture in stage 0 with the interpolated vertex color. The result is passed on to the next stage. In stage 1, we set the color operation to simply accept the result of the previous color operation and output it from the texture cascade unaltered. The texture assigned to stage 1 does not have its RGB values sampled at all, and it will not contribute to the color output from the texture stage cascade.



In stage 0, we set the alpha operation to sample the alpha from the assigned texture. This is done simply to enable the alpha flow through the pipeline. In fact, this alpha value will not be used at all and will be replaced in stage 1.

```
// Stage 0 Coloring : Modulate vertex color and texture color
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// stage 0 alpha : Just set up flow to next stage
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOOP, D3DTOP_SELECTARG1 );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_CURRENT );

// Bind the render texture (the color texture) to stage 0
m_pD3DDevice->SetTexture( 0 , pMyColorTexture );
```

In stage 1, the alpha value is sampled from the assigned texture using the second set of texture coordinates by default. The alpha value sampled from this second texture will be output from the texture cascade as the final alpha value along with the color value calculated in the first texture stage.

```
// stage 1 coloring : Output stage 0 texture color unaltered by this stage
m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_CURRENT );

// stage 1 alpha : Output alpha sample from texture assigned to this stage.
m_pD3DDevice->SetTextureStageState( 1, D3DTSS_ALPHAOOP, D3DTOP_SELECTARG1 );
m_pD3DDevice->SetTextureStageState( 1, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );

// Bind the alpha texture to stage 1
m_pD3DDevice->SetTexture( 1 , pMyAlphaTexture );
```

Assuming alpha blending is enabled, the blending equation will color the face of each polygon using the color resulting from modulating the first texture with the interpolated vertex alpha, and an alpha value sampled from the texture assigned to stage 1. The alpha operation in stage 0 is only significant in that it ensures that the alpha pipeline is enabled.

DirectX Graphics includes a pure alpha surface format. In a pure alpha surface, each pixel is an 8-bit alpha value and it does not contain red, green, or blue components. This is an ideal surface to use for the above example, where we only need the second texture for alpha information and would be otherwise wasting memory. The **D3DFORMAT** enumerated type that describes an alpha surface is:

```
D3DFMT_A8; //Used to create Textures/Surfaces that just contain alpha information
```



**Note:** At the time of this writing, pure alpha surfaces (D3DFMT\_A8) have only minimal support on some graphics cards and no support on many others. Therefore, our demos will continue to use normal ARGB textures for the 'alpha texture', but will only store meaningful values in the alpha component at each pixel.

Note that while the RGB pixel components of our alpha texture will not be used by the color pipeline in our demos, you could theoretically come up with some interesting additional uses for them (storing per-pixel lighting data for example). The only time our alpha textures will be used is when sampling alpha in the 2<sup>nd</sup> texture stage.

Regardless of whether we are using a pure alpha surface or just using the alpha channel of an ARGB surface, the techniques are the same.

Lab Project 7.4 uses this technique to implement an interesting and relatively advanced concept called **texture splatting**. Although a demo of this complexity is technically not necessary to show us how to use a separate texture as an alpha-only source in the texture cascade, it is a technique that you will find adds significant enhancements to your terrain application (and can be useful in other areas as well). Please refer to your course workbook for further details on texture splatting and its implementation before continuing with the next section.



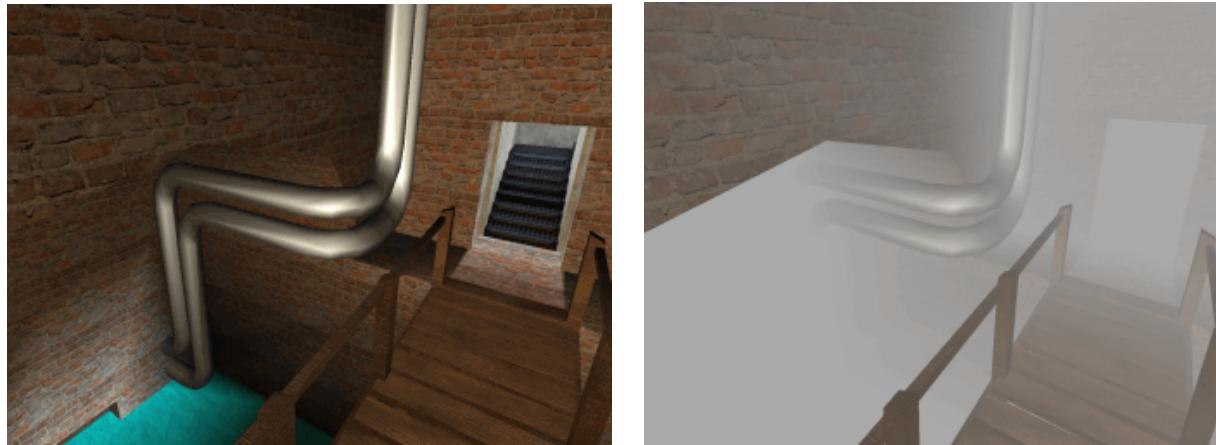
## Fog

Let us now turn our attention to another form of blending used to create the effect of fog in our scenes. If we were to describe fog technically we could say that it is the phenomenon produced when water and dust particles suspended in the air are dense enough to scatter light. If we were to give fog a non-technical description, we might say it was a misty substance that obscures our viewing of distant objects. Fog is definitely a key ingredient for providing a sense of atmosphere and mood and is often a must-have in many computer games.

In addition to aesthetic appeal, fog also provides a way to optimize our rendering engine. As you know, we have a projection matrix that defines a view frustum with a near and a far plane. Objects on the opposite side of the far plane are considered out of viewing range and are not rendered. Even if we use very large viewing distances, we can often see objects popping in and out of view as our camera is navigated about the game world. If we had a far plane of 5000 units for example and an object was placed at a distance of 5100 units away from our camera along the Z axis, the object would be rejected by the pipeline and would not be rendered because it is considered outside the view frustum. If we were to move our camera forward a couple of hundred units along Z, the object would suddenly appear since its position would then fall between the near and far planes of the view frustum. Although we can push the far plane out to a very far distance so that objects that pop in and out are relatively small, this is usually in direct contrast to what we want to do to make the rest of our application run as smoothly as possible. Usually, we want to render as little as we can get away with. One way to optimize our rendering is to bring the far plane closer to the camera so that objects in the distance do not get rendered. The closer we bring the far plane, the fewer objects we have to render. Also recall that we discussed the various depth buffer artifacts that can occur if the distance between the near and far planes of our frustum is too great (Chapter 2).

Fog provides a solution for both of these problems. We can set the fog such that as an object's distance with respect to the camera increases -- and it approaches the far plane -- the object begins to become more and more obscured by fog. At the point where the object is about to be clipped by the far plane, it is no longer visible and therefore we do not see it suddenly disappear from the scene. As objects first enter the frustum at the far plane, they are initially obscured by fog, so we do not see them suddenly appear. Instead, they slowly fade into view, appearing out of the fog. This hides the fact that we have a far clip plane at all.

Fig 7.11 shows two views of the same scene. The image on the left shows our demo level from previous chapters rendered normally. The image on the right shows the same scene rendered using fog supplied by the DirectX Graphics pipeline.



**Figure 7.11**

The image on the right uses a very dense fog. It does not look very good, but it was done to prove a point. We have to be careful not to overuse fog at the expense of severely limiting the player's ability to see more than a few meters. This can make the game frustrating to play and quite claustrophobic. For example, the original *Turok: The Dinosaur Hunter™* on the *Nintendo 64™* was quite graphically impressive. But in order to achieve high frame rates on a relatively low powered console (by today's standards) the developers used very dense fog settings and the player could usually only see a short distance in front of himself. This was done to minimize the number of polygons that had to be rendered each frame, but unfortunately it marred the game experience because players could not see what they were firing at until it emerged from the fog (often only a few meters away and with deadly intent).

## DirectX Fog

In DirectX Graphics, fog is implemented as a form of color blending. We typically set a fog color along with other properties like density and range, and then enable the fog mechanism with a single render state. When objects are rendered, the color of the object is blended with the fog color based on the distance between the object and the camera and how we currently have the fog set to change over distance. When objects are close to the viewer, their colors are only lightly blended with the fog color while objects far away (towards the back of the fog range) will have their colors heavily blended with the fog color.

The pipeline calculates a value called the **fog factor**. This is a scalar between 0.0 and 1.0. Far away objects that are completely obscured by fog will have a fog factor of 0.0. Objects very close to the camera and thus not obscured by fog will have a fog factor of 1.0. We can configure the pipeline to use one of three different equations to calculate the fog factor. These equations will change the way fog affects our scene. We can choose the Linear Fog Model, the Exponential Fog Model, or the Squared Exponential Model. We will look at how we set the fog model in a moment but for now, just keep in mind that each of these fog models produces a fog factor in the range of [0.0, 1.0].



The pipeline alters the original color of an object by blending its color with the fog color using the following color blending equation:

$$C_{final} = F_{factor} \times C_{original} + (1 - F_{factor}) \times C_{fog}$$

$C_{final}$  = Final color after fog has been applied.

$C_{original}$  = Original color prior to fogging.

$F_{factor}$  = Fog Factor value between 0.0 and 1.0.

$C_{fog}$  = Application defined fog color.

You might have noticed that this is essentially the same structure as the alpha blending equation we discussed earlier. In this case, the fog factor replaces the alpha value, the original color is the source color, and the fog color is the destination color. In fact, when performing vertex fog, the pipeline internally stores the fog factor in the alpha component of the specular color calculated for each vertex. This means that even if we are not using the DirectX transformation pipeline and we are rendering using transformed and lit vertices, we can calculate the fog factor for each vertex ourselves and store this value in the alpha component of our vertex specular color. If fog has been enabled and a fog color has been set, the pipeline will still apply vertex fog to our scene. When using the transformation pipeline however, DirectX will calculate the fog factors automatically for each vertex/pixel, using one of the three fog models mentioned above. If we are not using the transformation pipeline and we calculate the fog factors ourselves, we can use whatever fog model we choose to produce fog factors. We will discuss these models momentarily.

## Enabling Fog

If the application is to use fog, then it must enable the fogging module on the Direct3DDevice. Just as we turned on lighting using a render state, we will do the same with fog. To enable or disable DirectX fog, we use the **D3DRS\_FOGENABLE** render state and pass in either **TRUE** or **FALSE** to enable to disable the fog calculations in the pipeline respectively.

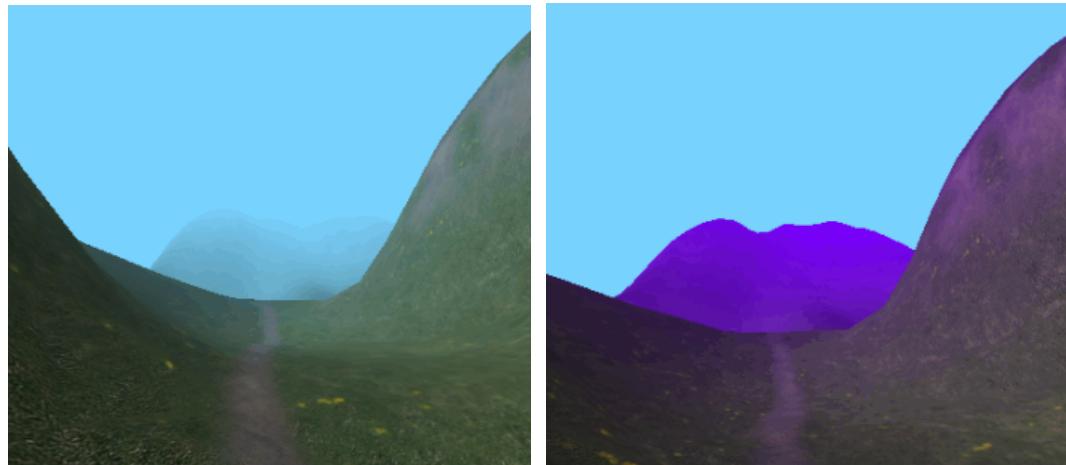
```
// Enable Fog Calculations
pDevice->SetRenderState( D3DRS_FOGENABLE , TRUE );

// Disable Fog Calculation
pDevice->SetRenderState( D3DRS_FOGENABLE, FALSE );
```

## Setting the Fog Color

Although we often think of fog as being white or perhaps grayish white, our application can set the fog color to any value we choose. However, when choosing the fog color we must be cognizant of our backgrounds – especially skies when we are outdoors. We will usually want to match the background color with the fog color so that objects fade into the horizon appropriately. For example, consider the two images in Fig 7.13. The image on the left shows a terrain being rendered with fog that is equal in

color to the color of the sky. We see that the terrain fades slowly into the sky color in the distance until it slowly disappears from view. The image on the right however uses a purple fog color. When the terrain is rendered into the blue frame buffer it does not allow the terrain to fade smoothly with distance. Instead it simply renders the distant terrain as fog colored polygons.



**Figure 7.13**

To set the fog color we use the `D3DRS_FOGCOLOR` render state and pass in a DWORD containing the 32-bit ARGB color. Only the RGB components are used. To specify the color red, we would pass `0x00FF0000`. In the following example we set a green fog color:

```
// Setting the Fog Color
pDevice->SetRenderState( D3DRS_FOGCOLOR , 0x0000FF00 );
```

### Fog Types

DirectX can provide fog at the per-vertex level or the per-pixel level. We will usually use either one or the other although if the hardware supports both types, we could enable both simultaneously (although there is little point in doing so).

### Vertex Fog

Vertex fog is typically the least desirable fog type because it is calculated at the vertex level -- much like vertex lighting. This means that it suffers from many of the same problems as vertex lighting (Chapter 5). Vertex fog is performed in the DirectX transformation pipeline, so if you are not using the pipeline, you will need to calculate the fog factors for each vertex yourself and store them in the alpha component of the vertex specular color.

When vertex fog is enabled, the pipeline calculates the fog factor for each vertex. This is typically quite expensive, especially when using one of the exponential fog models. By default, this fog factor is



calculated using the view space Z component of a vertex as the distance variable in its equations. Note that this is not the true distance from the vertex to the camera; it is only the Z displacement. However the view space Z component is convenient because it is already available to the pipeline because we have a ‘W-friendly’ projection matrix. This means that our projection matrix has a W column of (0, 0, 1, 0) -- which is a Z identity column (Chapter 1). Recall that a view space vertex multiplied with our projection matrix will have its Z component copied into the W component of the output vector where it can be used by the fog calculations (and W buffers incidentally). If for some reason the projection matrix you are using does not have a normalized W column of (0, 0, 1, 0) you will need to make your projection matrix W-friendly by scaling every other element of your matrix by the inverse of element m34 as shown below.

Non-W-Friendly matrix where ‘e’ does not equal 1:

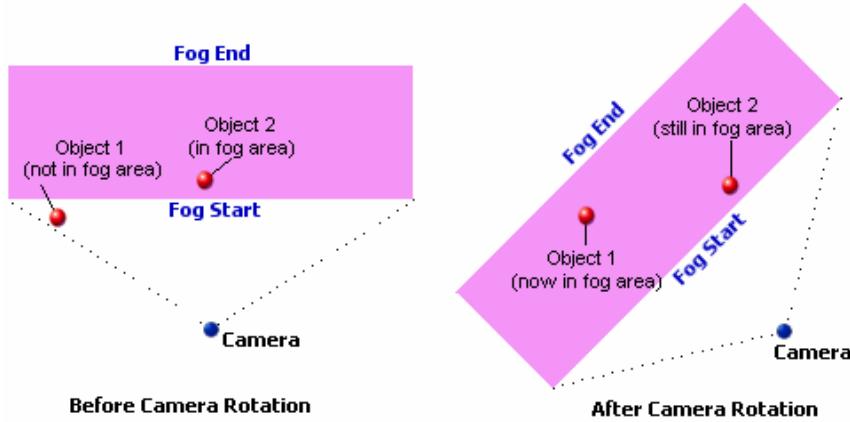
$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & e \\ 0 & 0 & d & 0 \end{bmatrix}$$

To force the matrix to be W friendly, divide all elements by element m34 (‘e’) and set m34 to 1:

$$\begin{bmatrix} a/e & 0 & 0 & 0 \\ 0 & b/e & 0 & 0 \\ 0 & 0 & c/e & 1 \\ 0 & 0 & d/e & 0 \end{bmatrix}$$

All of the projection matrices we have used in our applications, and the projection matrices returned by the D3DXMatrixPerspectiveLH function, are W-friendly matrices. They will not need to have this process applied to them.

When using the view space Z component as the distance for fog factor calculations, rotational artifacts may result. This is because the view space Z component only tells us the depth of the vertex along the Z axis and not the true distance. When the camera rotates, a vertex Z component can move in and out of the fog zone. Fig 7.14 demonstrates the problem.



**Figure 7.14**

In Fig 7.14, the light purple boxes show where the fog starts and ends with respect to the camera. Objects inside the purple boxes will receive fogging to some degree. The image on the left shows our camera before it has been rotated. Using the view space Z component we can see that object 1 is not considered to be inside the fog zone. This is because we have set the fog to start at a certain distance from the camera and the view space Z component of object 1 is not large enough to be considered inside the fog zone. The problem here is that we are only considering the Z component of the object and not its true distance (which would account for the X and Y offsets in view space also). If you draw a line from the camera to object 1 in the left diagram and then rotate object 1 about that line such that it was now directly in front of the camera, you should be able to see that it does indeed have a distance from the camera which is larger than the fog start distance. Thus it should be contained inside the fog zone. However, since we are only using the view space Z component as the distance, it is not. Now perhaps we could accept this flaw, but note what happens if the camera was to rotate left 45 degrees. It is now facing object 1 as shown in the image on the right. The Z depth of object 1 (tilt your head to the left when looking at the diagram to see view space) is now greater than our fog start distance and the object would be fogged. The unfortunate result is that as we rotate the camera, objects seem to suddenly pop in and out of the fog.

Some hardware supports an additional render state that instructs the pipeline to calculate the true distance from the camera to the vertex rather than the view space Z depth. This eliminates the rotational artifacts at the cost of performance. This is called **range based fog**. Range based fog is only supported for the vertex fog state.

After the pipeline has calculated the fog factor for each vertex and stored it inside the alpha component of the specular color, if Gouraud shading is enabled, the fog factor is interpolated across the face of the polygon just like any other color/alpha value stored in a vertex. This is later used to generate the fogged color for each pixel on the surface.



## Enabling Vertex Fog

To enable vertex fog we will set a render state and specify the fog model we would like to use. Before we do this, we should check the `RasterCaps` member of the `D3DCAPS9` structure returned by the `IDirect3DDevice::GetDeviceCaps` function for the `D3DPRASTERCAPS_FOGVERTEX` flag. If it is set, then the device supports vertex fog.

```
D3DCAPS9 Caps;  
pDevice->GetDeviceCaps( &Caps );  
  
if ( Caps.RasterCaps & D3DPRASTERCAPS_FOGVERTEX )  
{  
    // Vertex Fog is Supported  
}
```

Once we know that vertex fog is supported, we enable it by setting the `D3DRS_FOGVERTEXMODE` render state and specifying a member of the `D3DFOGMODE` enumerated type as the second parameter to the function. The `D3DFOGMODE` has four members which allow us choose the fog model used to calculate the fog factor. These members are shown below.

```
typedef enum _D3DFOGMODE  
{  
    D3DFOG_NONE = 0,  
    D3DFOG_EXP = 1,  
    D3DFOG_EXP2 = 2,  
    D3DFOG_LINEAR = 3,  
    D3DFOG_FORCE_DWORD = 0x7fffffff  
} D3DFOGMODE;
```

`D3DFOG_NONE` is the default setting when the device is created. Essentially it means that vertex fog is disabled. We can enable vertex fog by choosing one of the three fog models: `D3DFOG_EXP`, `D3DFOG_EXP2` or `D3DFOG_LINEAR`. We will discuss the three fog models in detail shortly.

The following code snippet enables vertex fog using the linear fog model. Note that we specify two view space distances. The first specifies the minimum distance a vertex would need to be from the camera in order to be influenced by fog (`D3DRS_FOGSTART`). The second specifies the distance at which a vertex would be fully fogged (`D3DRS_FOGEND`). Notice that we also check for range fog support and enable it when available. This means that each vertex will have its true distance from the camera used to calculate the fog factor rather than the view space Z component (at a higher performance cost).



```
D3DCAPS9 Caps;
pDevice->GetDeviceCaps( &Caps );

if ( Caps.RasterCaps & D3DPRASTERCAPS_FOGVERTEX )
{
    pDevice->SetRenderState (D3DRS_FOGENABLE , TRUE );
    pDevice->SetRenderState (D3DRS_FOGCOLOR , 0x00AAAAAA );
    pDevice->SetRenderState (D3DRS_FOGVERTEXMODE , D3DFOG_LINEAR );

    if ( Caps.RasterCaps & D3DPRASTERCAPS_FOGRANGE )
    {
        pDevice->SetRenderState (D3DRS_RANGEFOGENABLE , TRUE )
    }

    float fStart = 100;
    float fEnd = 1000;

    pDevice->SetRenderState (D3DRS_FOGSTART , *(DWORD*) (&fStart) );
    pDevice->SetRenderState (D3DRS_FOGEND , *(DWORD*) (&fEnd) );
}
```

There are a few noteworthy items regarding the above code. First of all, we are checking the **D3DPRASTERCAPS\_FOGRANGE** flag against the RasterCaps member of the D3DCAPS9 structure to see if the hardware supports range-based vertex fog. If it does, we enable it using the **D3DRS\_RANGEFOGENABLE** render state. Next, because we are using the linear fog model, we must also specify the fog start and fog end values described above. The linear fog model applies fog to objects such that fog intensity increases linearly between these two distances. We will discuss these two values in more detail when we cover the fog models, but it is worth taking note of the way the values are passed into `SetRenderState`. The `SetRenderState` function expects a `DWORD` as its second parameter, but our values need to be floats when we use fog. Therefore, we cast the address of the float to a `DWORD` pointer and then de-reference the result.

## Pixel Fog (Table Fog)

Pixel fog calculates the fog intensity for each pixel rather than each vertex and provides much better looking fog effects. Thus, if pixel fog is available on the current hardware we will usually want to use it instead of vertex fog. Pixel fog uses many of the same settings as vertex fog. For example, we still set the fog color using the **D3DRS\_FOGCOLOR** render state and we can also use the same three fog models: linear, exponential and squared exponential. The fog factor calculations are exactly the same, only they are performed on a per-pixel basis in the driver instead of a per-vertex basis in the transformation pipeline.

The idea of calculating a per-pixel fog factor raises obvious concerns about performance. To address such concerns, the driver builds a lookup table containing fog factors for a number of fixed distances. When the fog factor for a pixel is needed, the view space Z depth of the pixel is used to look up a fog



factor that has been pre-calculated for that depth and stored in the table. This concept has led to pixel fog commonly being referred to as **table fog**.

## Enabling Pixel Fog

To determine whether or not a device supports pixel fog, we check the **RasterCaps** member of the **D3DCAPS9** structure for the **D3DPRASTERCAP\_FOGTABLE** flag. If it is set, then we can enable pixel fog by setting the **D3DRS\_FOGTABLEMODE** render state to a member of the **D3DFOGMODE** enumerated type, describing which of the three fog models we would like to use. Also, remember that whether we are using vertex or pixel fog, we must always have the **D3DRS\_FOGENABLE** render state set to TRUE in order for fog calculations to be performed.

We have one more thing to consider if we intend to use the linear fog model with pixel fog. To alleviate fog-related graphic artifacts caused by uneven distribution of z values in a depth buffer, most hardware devices will use the view space Z component of a vertex to produce an eye relative depth value that can be used to lookup the fog factor for that depth. You will recall that the view space Z component is preserved in the W component of the vector output from the projection matrix (see Chapter 1). If eye relative depth calculations are supported by the device, we can describe the FogStart and FogEnd parameters for the linear fog model using view space units (just as we did when setting up the linear fog model for vertex fog). However, if the hardware does not support the use of W-based fog, then we will need to specify the FogStart and FogEnd parameters in device coordinates (depth buffer coordinates) in the range of [0.0, 1.0]. This will only need to be done if we are using the linear fog model because the exponential and squared exponential fog models do not use the FogStart and FogEnd render states. We can check the **RasterCaps** member of the **D3DCAPS9** structure to see if the **D3DPRASTERCAPS\_WFOG** flag is set. If so, then the preferred W-based fog will be supported and used automatically -- provided we are using a W-friendly projection matrix.

There is also a **D3DPRASTERCAPS\_ZFOG** flag that can be set in the **RasterCaps** member. One might assume that if W-based fog was not available, then Z-based fog would be supported. But the relationship is a little more complex. Although a W-based fog capable device will always use W-based fog by default, this will only be possible if we have a W-friendly projection matrix. If we do not, then Z fog will be used instead as a next best option for pixel fog. In this case, we must use the [0, 1] distance range for FogStart and FogEnd.

It is possible that a device may support W-based fog, but not support Z-based fog as a backup. In such a case, we must be absolutely sure that we set a W-friendly projection matrix in order for fog to work correctly. Note that if pixel fog is supported, at least one of these two fog types will also be supported.

The following code enables pixel fog using the linear model for the fog factor calculations. We also see how to convert our fog start and fog end distances into the [0, 1] range if W-based fog is not supported (where Z-based fog will be used automatically). We assume that the projection matrix is W-friendly.



```
D3DCAPS9 Caps;
pDevice->GetDeviceCaps( &Caps );

if ( Caps.RasterCaps & D3DPRASTERCAPS_FOGTABLE )
{
    float fStart = 100; //Fog starts at 100 from the camera
    float fEnd   = 1000; //Fog reaches full intensity at a distance of 1000 units

    pDevice->SetRenderState (D3DRS_FOGENABLE      , TRUE );
    pDevice->SetRenderState (D3DRS_FOGCOLOR       , 0xAAAAAA );
    pDevice->SetRenderState (D3DRS_FOGTABLEMODE, D3DFOG_LINEAR );

    // if W fog is not supported we need start/end distances in device units
    if ( !(Caps.RasterCaps & D3DPRASTERCAPS_WFOG) )
    {
        fStart /= fFarPlaneDistance ;
        fEnd   /= fFarPlaneDistance ;
    }

    pDevice->SetRenderState (D3DRS_FOGSTART, * (DWORD*) (&fStart) );
    pDevice->SetRenderState (D3DRS_FOGEND,   * (DWORD*) (&fEnd) );
}
```

In the above code we enable fog, set the fog model to linear, set the fog type as table (pixel) fog, and check to see if the device is going to use W-based fog or not. If it is not, then we need to divide our fog distances by the far plane distance specified when we built our projection matrix. This only needs to be done if we are using a linear fog model.

In the next example, we will look at some code that sets up fog using the exponential model. The code will try to use pixel fog if it is supported on the current device and fall back to vertex fog if it is not. Notice that when we use either the exponential or squared exponential model, we no longer specify start and end distances. Instead we specify a **fog density** value between 0.0 and 1.0. This is used as a weight in the exponential fog formulas used to calculate the fog factors. We will look at these equations in the next section.

```
D3DCAPS9 Caps;
pDevice->GetDeviceCaps( &Caps );
float fDensity = 0.6;

if ( Caps.RasterCaps & D3DPRASTERCAPS_FOGTABLE )
{
    pDevice->SetRenderState (D3DRS_FOGENABLE      , TRUE );
    pDevice->SetRenderState (D3DRS_FOGCOLOR       , 0xAAAAAA );
    pDevice->SetRenderState (D3DRS_FOGTABLEMODE, D3DFOG_EXP );
    pDevice->SetRenderState (D3DRS_FOGDENSITY   , * (DWORD*) (&fDensity) );
}

else
if ( Caps.RasterCaps & D3DPRASTERCAPS_FOGVERTEX )
{
    pDevice->SetRenderState (D3DRS_FOGENABLE      , TRUE );
    pDevice->SetRenderState (D3DRS_FOGCOLOR       , 0xAAAAAA );
    pDevice->SetRenderState (D3DRS_FOGVERTEXMODE, D3DFOG_EXP );
    pDevice->SetRenderState (D3DRS_FOGDENSITY   , * (DWORD*) (&fDensity) );
}
```



The density value is used to scale the vertex or pixel distance values when either the EXP or EXP2 fog models are enabled. A value of 0.0 equates to no fog being applied to the scene at all. A value of 1.0 means everything in the scene will be completely fogged. Values in between allow us to control where the fog starts (at least at a significant intensity). It is often necessary to experiment with a few different values to find the density level that best suits your scene. When the device is first created, the default density is 1.0 and everything will be fogged.

## Fog Factor Formulas

When using either fog type (vertex or pixel) we will specify the model that is used to calculate fog factors. Each model calculates the way fog intensity increases with distance in a different fashion. There are three equations that we can choose from and they are selected by passing in a member of the **D3DFOGMODE** enumerated type when setting the **D3DRS\_FOGVERTEXMODE** or **D3DRS\_FOGTABLEMODE**. The default mode for both is **D3DFOG\_NONE** which means that no fog model is being used. Our options are:

### Linear Fog (D3DFOG\_LINEAR)

$$f = \frac{end_f - d}{end_f - start_f}$$

$f$  = Resulting Fog Factor.

$start_f$  = FogStart value.

$end_f$  = FogEnd value.

$d$  = Distance.

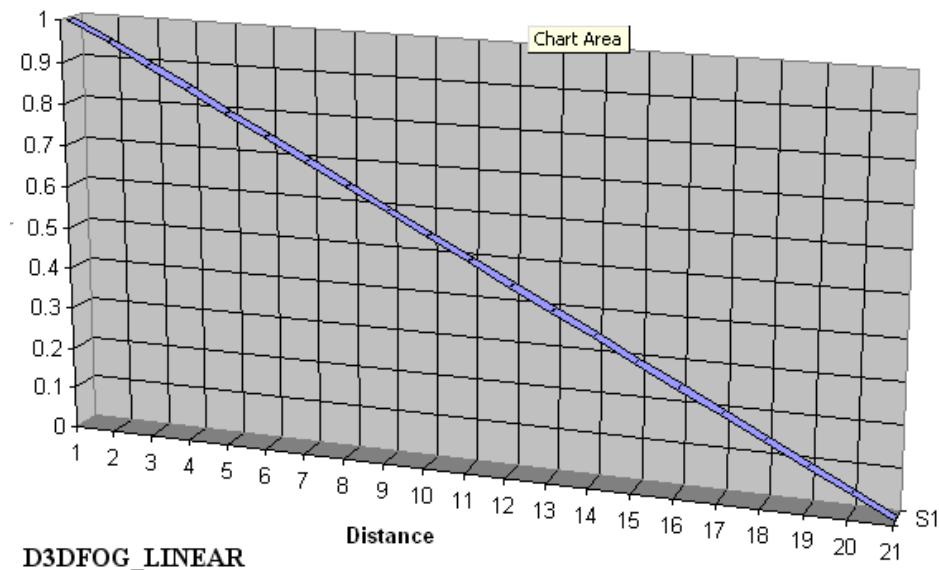
Linear fog is the fastest but least visually appealing model available. In the formula,  $f$  is the fog factor that is used by the pipeline to control the blending of the vertex/pixel color with the fog color. The calculation also uses the start and end values set by the application to linearly distribute fog to objects between those distances.

If this equation is used for table fog and W-based fog is active, then the start and end values will be view space distances and the pixel distance  $d$  will be the view space z component of the pixel (interpolated). If W-based fog is not being used, then the start and end values will be device coordinates (depth buffer coordinates) in the range of 0.0 to 1.0 and the distance  $d$  will be the depth of the pixel in depth buffer coordinates.

If this equation is used for vertex fog, then the start and end distances will be specified by the application in view space units. If range-based vertex fog is enabled, then the vertex distance  $d$  will be the actual distance between the camera and the vertex. Otherwise  $d$  will be the view space Z component of the vertex.

The linear model does not accurately represent how we perceive fog density to change over distance. In real life, as the distance between an object and the viewer increases, the density of the fog appears to increase exponentially. Therefore, for more realistic fog we would use the more performance heavy exponential or the squared exponential fog model.

When we plot this function as a graph, we see a diagonal line demonstrating that fog density increases linearly and predictably with distance. Fig 7.15 uses a fog start value of 0.0 and a fog end value of 200. We plot for distances along the X axis in 10 unit increments from 0 to 210.



**Figure 7.15**

Note that the fog factor is 0 when the object is far away (totally fogged) and 1.0 when the object is very close to the camera (no fog). If you read the line from right to left instead of from left to right, you can more easily visualize how fog increases with distance under the linear fog model.

At first it may seem counterintuitive to think of the fog factor as being 0 when an object needs to have total fog applied and 1.0 to have no fog applied. However, recall that the fog factor as used in the blending equation is an inverse. When the fog factor is zero, then none of the original color is used and all of the fog color is used; if the fog factor is 1 then all of the original color is used and none of the fog color is used. If the fog factor is between 0.0 and 1.0 then some degree of color blending takes place.

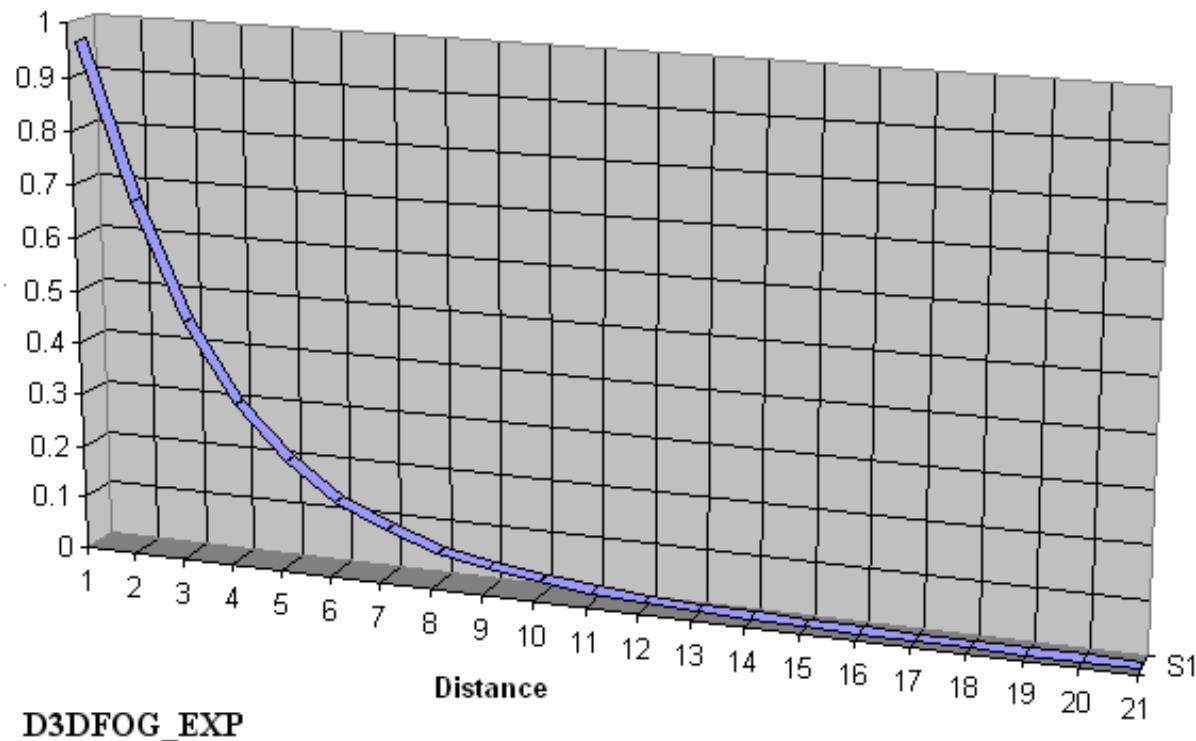
### Exponential Fog (D3DFOG\_EXP)

Unlike linear functions, where the graph increases or decreases uniformly with respect to distance, exponential function graphs have abrupt and steep changes. Exponential functions generally take the form  $f(x) = a^x$ . In this formula,  $a$  is a real number that is referred to as the base of the exponential

function. This can be any number that fulfills the requirements of the function.  $x$  is the number we pass into the function which will be used as the exponent.

When discussing exponential functions, there is one very important number that can be used as a base. This special number is called the **natural base** and is represented by the symbol  $e$ . When an exponential function uses the natural base, it is referred to as the **natural exponential function** or sometimes just **the exponential function** -- which just goes to show just how significant it is. The natural base is approximately 2.7182818284590452354.

The reason the natural base and the natural exponential function are so important is that they help to accurately model rates of change. The natural exponential function has been used to predict the rate at which populations expand, the effect of interest rate increases and decreases on investment, and more importantly to this discussion, the rate at which fog density changes over distance. When using the exponential fog model, we no longer pass in fog start and end distances, but instead pass a fog density value in the range [0.0, 1.0]. Smaller density values lead to a less dense fog and vice versa.



**Figure 7.16**

By altering the density value we can shape the intensity curve. We see in Fig 7.16 that although fog intensity varies over the view space range of 200 units, the fog intensity increases rapidly in the first 80 view space units. Objects situated at view space distances greater than 80 will receive almost total fog.



This provides much greater control over modeling the way distance affects fog intensity. Contrast this with the **D3DFOG\_LINEAR** mode where intensity increases rigidly in direct proportion to distance.

When the **D3DFOG\_EXP** model is used for vertex or table fog mode, the following formula is used. This calculation is more expensive than the linear model fog factor calculation.

$$f = \frac{1}{e^{d * density}}$$

$f$  = Resulting Fog Factor

$e$  = Natural Base

$d$  = Distance

$density$  = Application specified density in the range [0, 1]

### **Exponential Squared (D3DFOG\_EXP2)**

Our final fog model is very similar to the previous model. It is also the most expensive performance-wise. The difference is that it squares the exponent in the denominator of the equation. Once again we will want to experiment with the density value in order to find a fog curve that suits our application. The fog factor is calculated using the following equation:

$$f = \frac{1}{e^{(d * density)^2}}$$

$f$  = Resulting Fog Factor

$e$  = Natural Base

$d$  = Distance

$density$  = Application specified density in the range [0, 1]

If you spend some time examining your workbook for this lesson, you will have an opportunity to examine these various fog models in action. Try to experiment with some of the different fog models and associated settings in the various lab projects. Look at the results of fog in your indoor scenes as well as your outdoor scenes. Experiment with modifying the position of your far plane and see what effects this has on frame rate.



## Conclusion

In this lesson we have examined a number of important new techniques. With our discussions in Chapter 6, and again in this lesson, we now have a very strong understanding of the various blending options available in DirectX Graphics. We know how to use color and alpha data in vertices, materials, constants, and textures.

We also know how to configure our pipeline to produce any number of transparency based effects. These are the bedrock concepts upon which all of our important special effects will be based. From magic spells, to fire, to explosions, we will be ready to immediately tackle all sorts of special effects programming techniques with little effort.

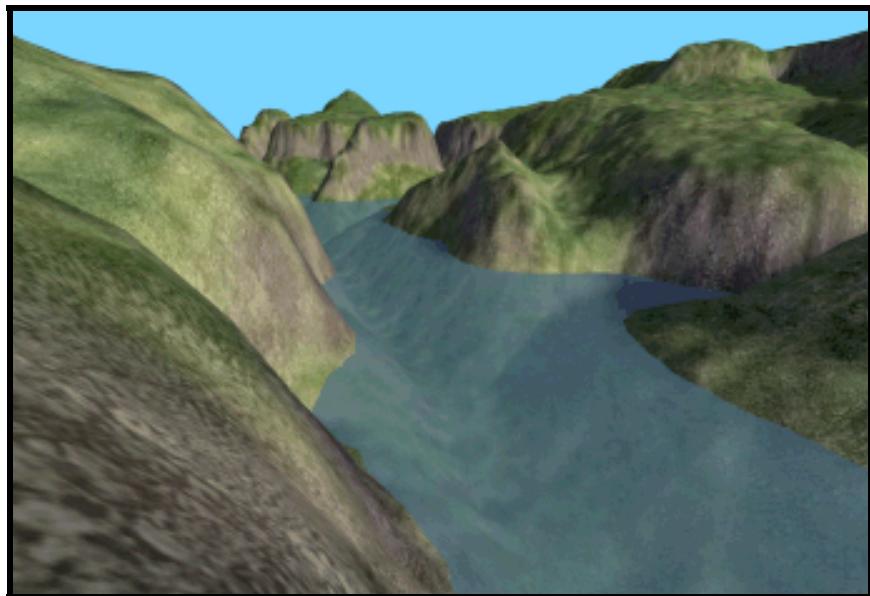
To wrap things up, we took a look at adding fog to our scenes to add mood and atmosphere and to optimize engine performance. Configuring the pipeline to add fog was a relatively simple process, so you should have no trouble adding it to your scenes in the specified workbook exercises.

Please make sure that you spend time examining the workbook projects for this week. In addition to the implementations of the concepts discussed here, as usual, you will also find a good deal of discussion on other relevant topics in 3D graphics programming. We will look at geometry manipulation, polygon splitting, ray intersections, screen space polygons, and much more. You will find this to be a very exciting lesson, so take your time, and enjoy the new coding techniques you learn. They will prove to be very valuable as we move forward into new subject areas as we finish up this course and prepare for the next one.

# **Workbook Chapter Seven:**

## **Alpha Blending and Fog**

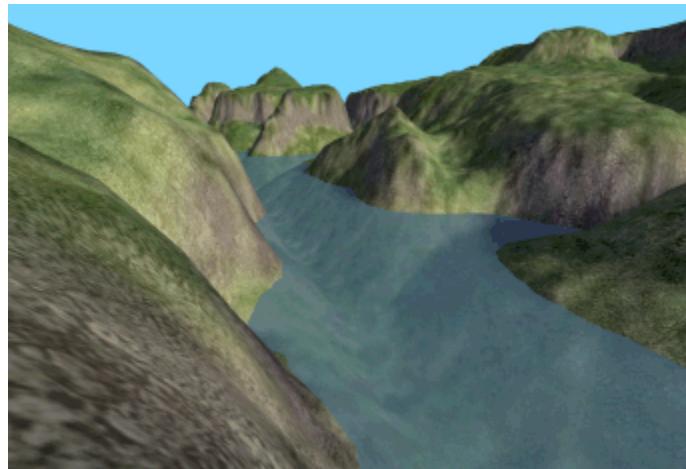
---



© 2003, eInstitute, Inc.

You may print one copy of this document for your own personal use. You agree to destroy any worn copy prior to printing another. You may not distribute this document in paper, fax, magnetic, electronic or other telecommunications format to anyone else.

## Lab Project 7.1: Vertex Alpha

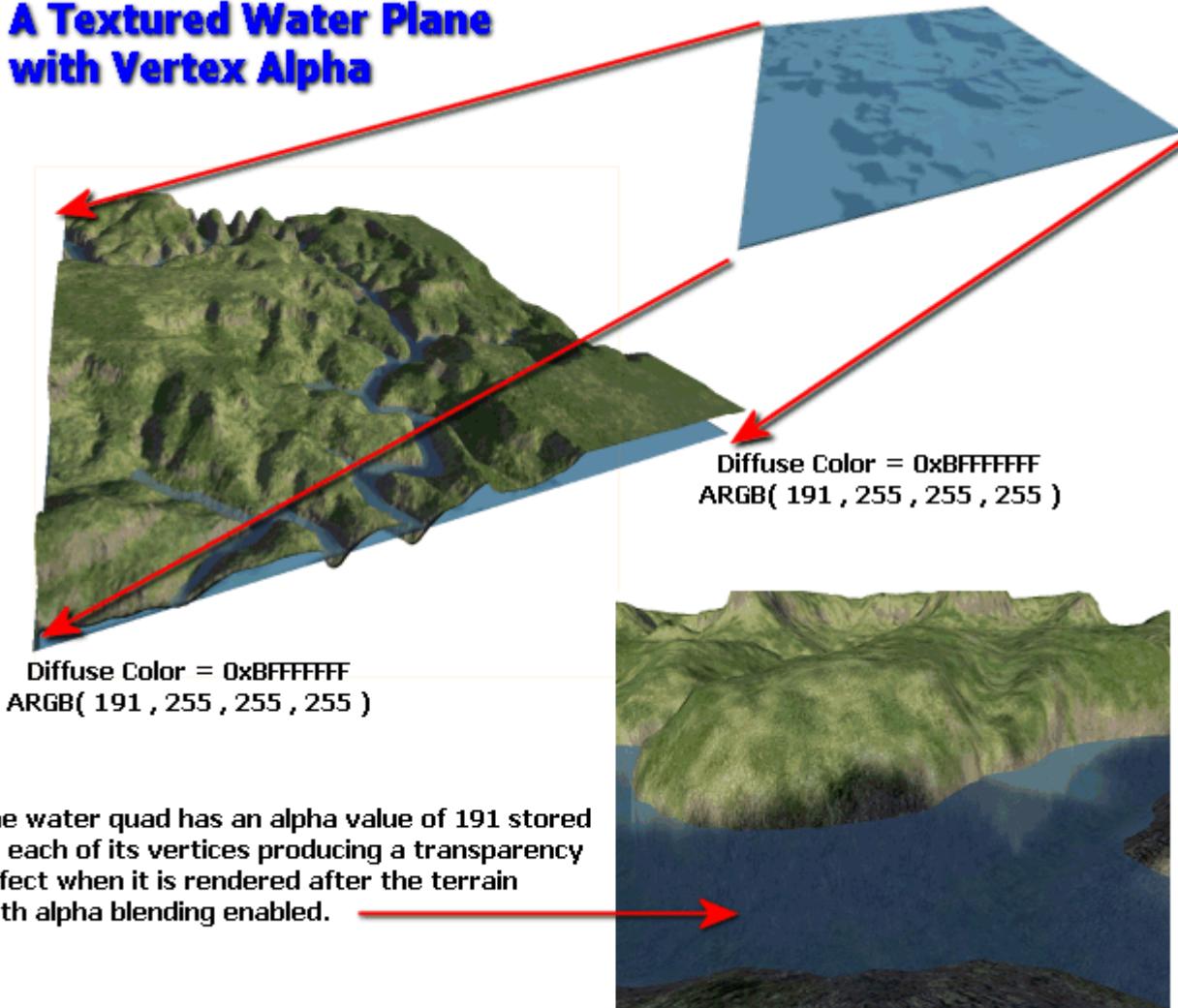


One of the simplest ways to perform alpha blending is by specifying the alpha component in the diffuse color of the vertex. This color is interpolated across the surface of the polygon at render time to generate a per-pixel color value that is fed into the texture stage cascade. The alpha value is part 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.

## A Textured Water Plane with Vertex Alpha



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 ( $\frac{1}{4}$  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.

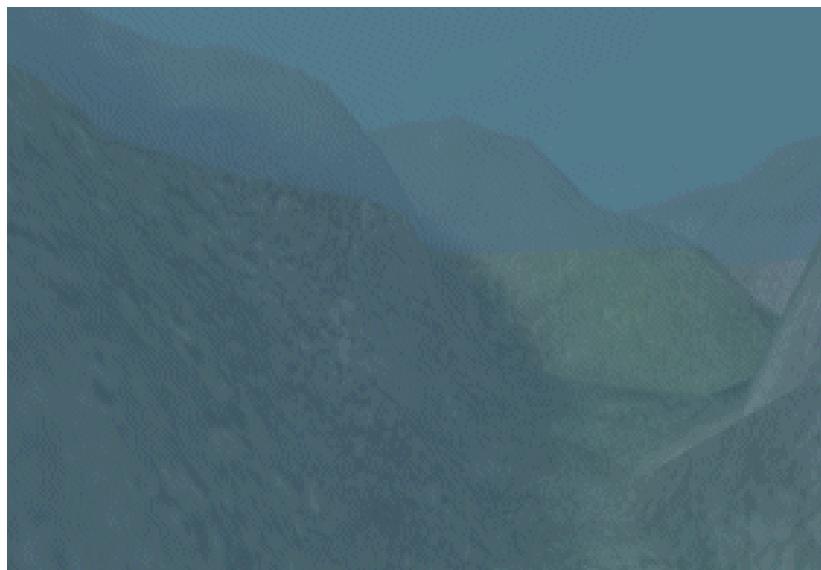


Certainly the terrain should not look the same when we are under the water. At the very least, it should be tinted (blue) since we are viewing it through a volume of (blue) water. There are other effects we can use to enhance our water, but those will have to wait until the next course in this series. 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 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 );
CLitVertex TopLeftVertex ( 0.0 , ViewPortHeight , 0.0 , 1.0 , 0x800000FF );
```

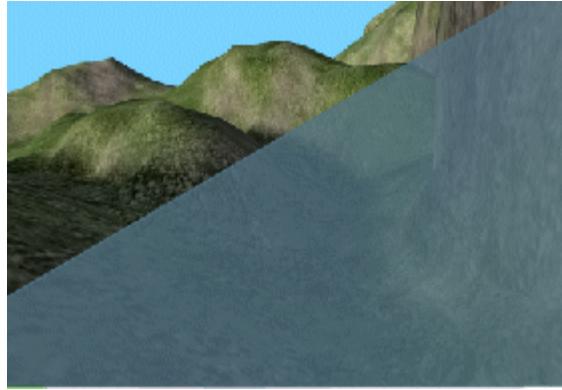
**Note:** Remember that in screen space increasing Y moves downwards towards the bottom of the screen.

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  $\frac{1}{2}$  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.

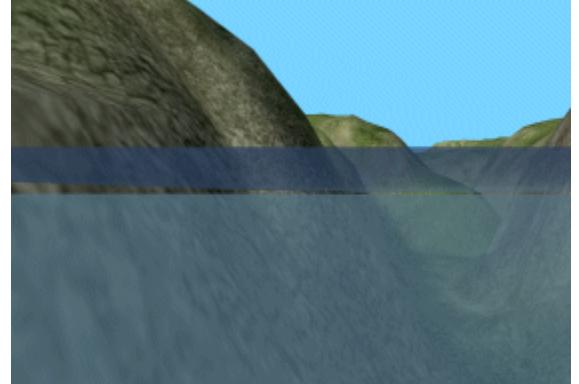


We will need to calculate the slope of the water line with relation to our camera and build a screen space polygon with the same slope so that it connects properly to the water plane itself.

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.

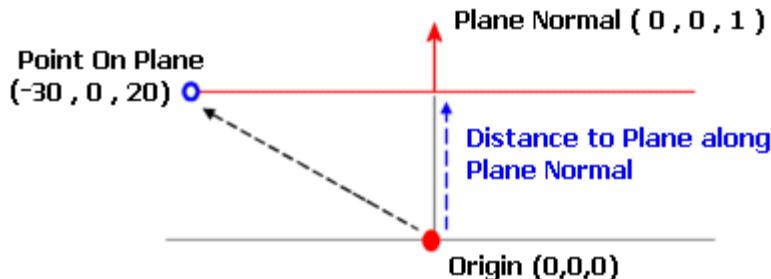
To calculate the plane distance we perform a dot product between the plane normal and a point known to be on the plane. This returns a value that is negative if the origin is in front of the plane and positive when behind. When using the distance in this manner, we need to use Plane Equation 2 to classify 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:

### Calculating the Plane Distance for form $Ax+By+Cz=D$



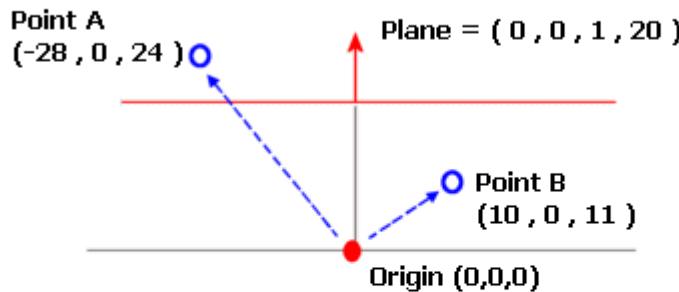
$$\begin{aligned} \text{Plane Distance} &= \text{PointInPlane} \bullet \text{Plane Normal} \\ \text{Plane Distance} &= (-30 * 0) + (0 * 0) + (20 * 1) \\ \text{Plane Distance} &= 20 \end{aligned}$$

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.

$$\begin{aligned} (0,0,0) + (0,0,1) * \text{Distance} &= \text{Point on Plane nearest to origin} \\ (0,0,0) + (0,0,1) * 20 &= (0, 0, 20) \end{aligned}$$

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**

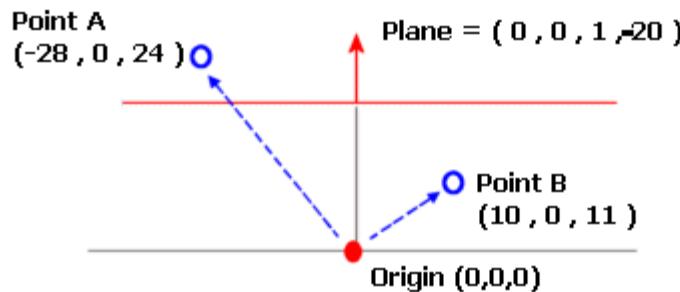
**Classifying Point A**  
 $(-28*0) + (0*0) + (24*1) - 20 = 4$  (Positive Distance)

**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.

## Classification of points using the form Ax+By+Cz+D



**Classifying Point A**

$$(-28 \cdot 0) + (0 \cdot 0) + (24 \cdot 1) + -20 = 4$$

**Classifying Point B**

$$(10 \cdot 0) + (0 \cdot 0) + (11 \cdot 1) + -20 = -9$$

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.

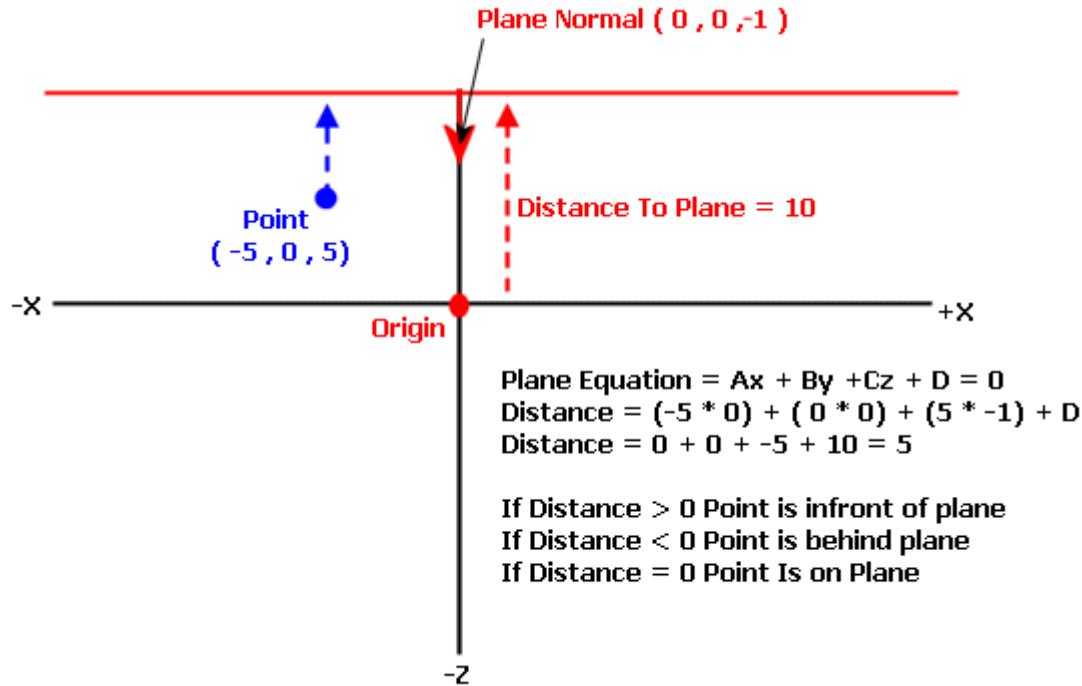
The D3DXPLaneDotCoord function is shown below:

```
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 4<sup>th</sup> component of the point will be treated as a 1.0 (making it a homogeneous coordinate) so that a 4D dot product can be performed:

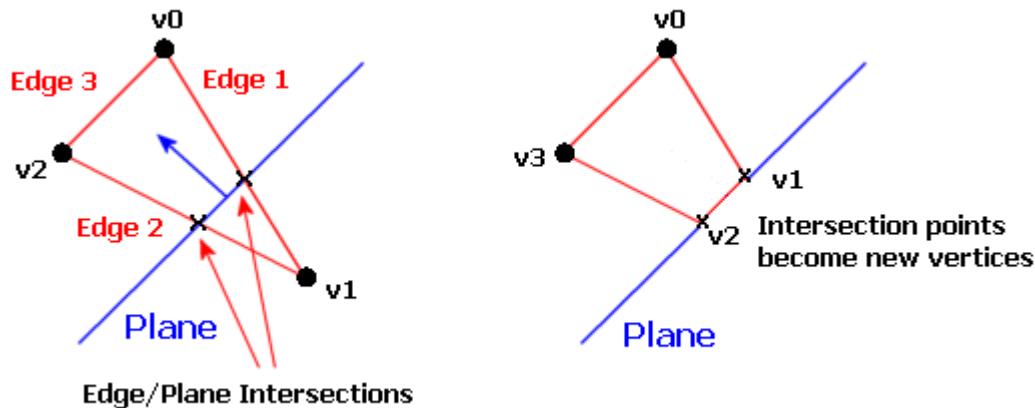
```
FLOAT D3DXPlaneDotCoord ( D3DXPLANE * pP , D3DXVECTOR3 *pV)
{
    return ( pP->x * pV->x ) + ( pP->y * pV->y ) + ( pP->z * pV->z ) + (pP->d * 1.0)
}
```

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:

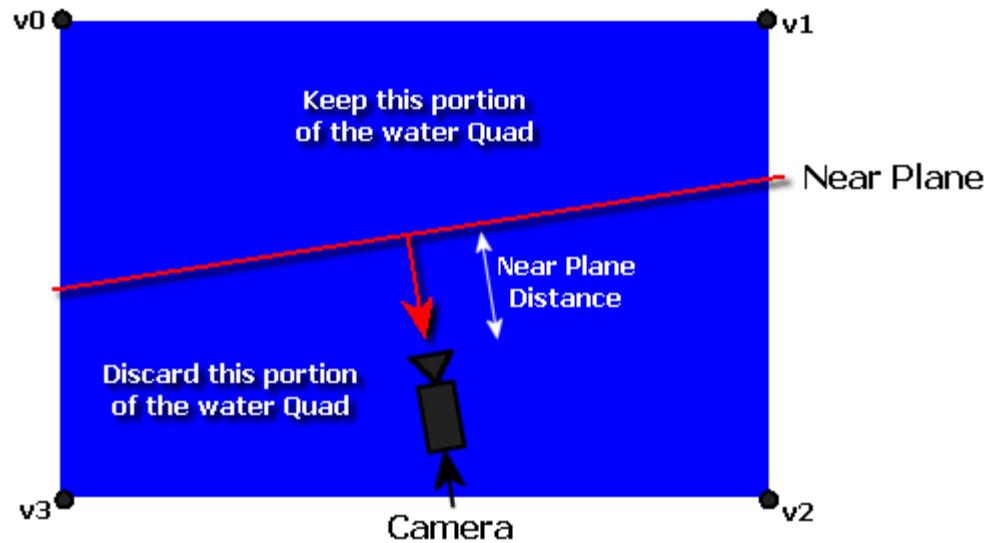


Thus the procedure for each polygon to be clipped would be:

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);

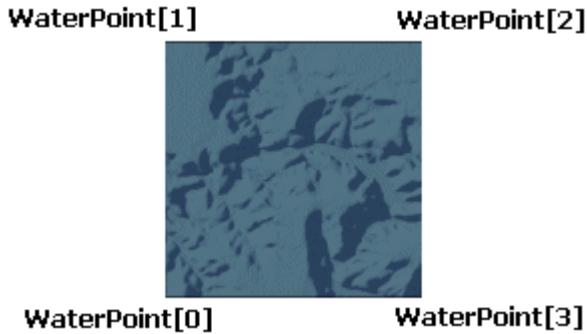
```

```

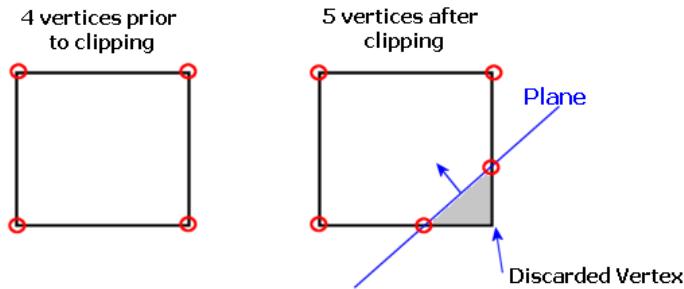
vecWaterPoints[2] = CLitVertex( m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                               m_nHeightMapHeight * m_vecScale.z, 0xFFFFFFFF,
                               1.0f, 1.0f );

vecWaterPoints[3] = CLitVertex( m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                               0.0f, 0xFFFFFFFF, 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).

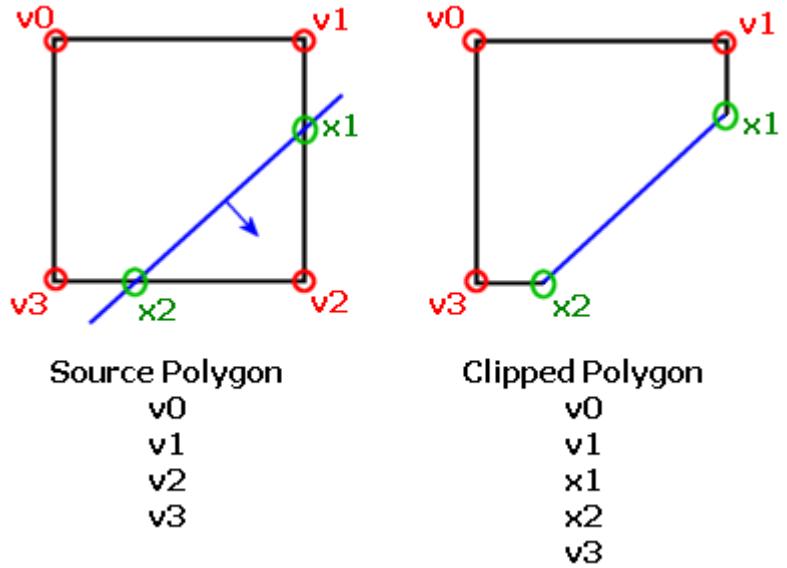


The image on the left shows the quad before clipping and the image on the right shows the resulting polygon. Clipping a quad to a single plane can only introduce one new vertex (at most) in the resulting polygon.

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 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
    if ( Location1 == -1 )
        Location2 = 0;
    else
        Location2 = 1;
}
```

```

// 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 its 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 D3DPLANE *pP,
    CONST D3DXVECTOR3 *pV1,
    CONST D3DXVECTOR3 *pV2
);

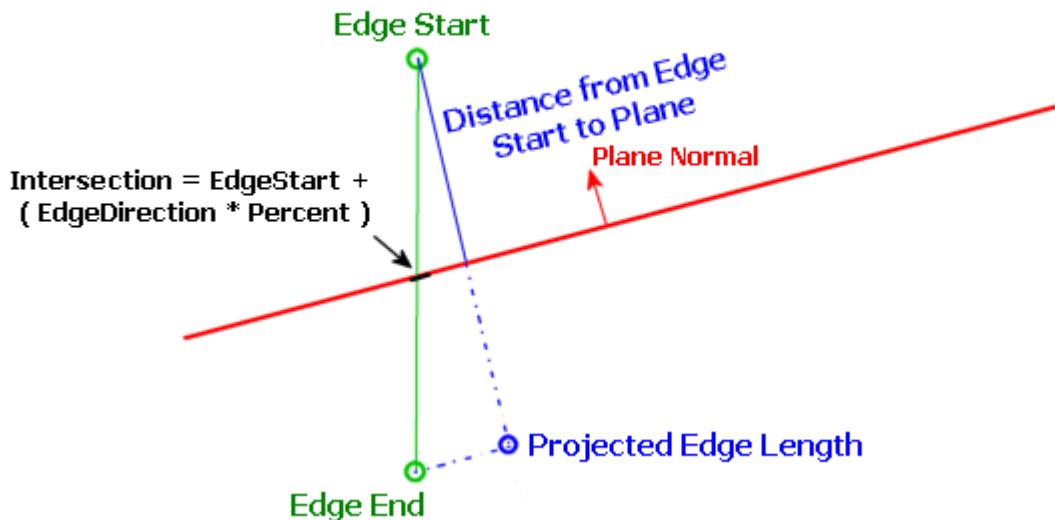
```

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.

## Intersecting a Ray with a Plane



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:

$$\begin{aligned}\mathbf{EdgeDirection} &= \mathbf{EdgeEnd} - \mathbf{EdgeStart} \\ \mathbf{EdgeEnd} &= \mathbf{EdgeStart} + \mathbf{EdgeDirection}\end{aligned}$$

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:

$$\mathbf{Intersection} = \mathbf{EdgeStart} + (\mathbf{EdgeDirection} * \mathbf{Percent})$$

*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       = D3DXVec3Dot ( 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:

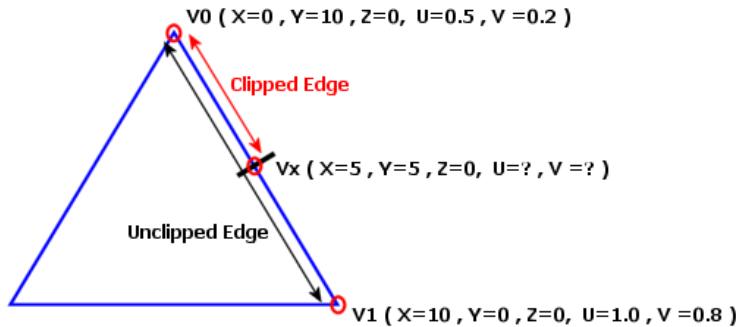
```
// Calculate the intersection point  
D3DXVECTOR3 vecIntersection;  
D3DXPlaneIntersectLine( &vecIntersection, &NearPlane, (D3DXVECTOR3*)&vecWaterPoints[v1],  
                         (D3DXVECTOR3*)&vecWaterPoints[v2] );
```

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 = 0xFFFFFFFF;
```

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.

$$\begin{aligned}\text{UnclippedEdgeVector} &= \mathbf{V1} - \mathbf{V0} = (10, -10, 0) \\ \text{ClippedEdgeVector} &= \mathbf{Vx} - \mathbf{V0} = (5, -5, 0)\end{aligned}$$

Next we calculate the length of each vector. In this case the results are:

$$\begin{aligned}\text{UnclippedEdgeLength} &= 14.142136 \\ \text{ClippedEdgeLength} &= 7.071068\end{aligned}$$

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.

$$\begin{aligned}\text{Percent} &= \text{ClippedEdgeLength} / \text{UnclippedEdgeLength} \\ \text{Percent} &= 7.071068 / 14.142136 = 0.5\end{aligned}$$

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:

$$\begin{aligned}\mathbf{Vx.tu} &= \mathbf{v0.tu} + ((\mathbf{v1.tu} - \mathbf{v0.tu}) * \text{Percent}) \\ \mathbf{Vx.tv} &= \mathbf{v0.tv} + ((\mathbf{v1.tv} - \mathbf{v0.tv}) * \text{Percent})\end{aligned}$$

Using the values from the above diagram:

$$\begin{aligned}\mathbf{Vx.tu} &= 0.5 + ((1.0 - 0.5) * 0.5) = 0.5 + 0.25 = 0.75 \\ \mathbf{Vx.tv} &= 0.2 + ((0.8 - 0.2) * 0.5) = 0.2 + 0.3 = 0.5\end{aligned}$$

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 by 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, 0xFFFFFFFF, 0.0f, 0.0f);
        vecWaterPoints[1] = CLitVertex(0.0f, WaterHeight,
                                      m_nHeightMapHeight*m_vecScale.z,
                                      0xFFFFFFFF, 0.0f, 1.0f );
        vecWaterPoints[2] = CLitVertex(m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                      m_nHeightMapHeight * m_vecScale.z,
                                      0xFFFFFFFF, 1.0f, 1.0f );
        vecWaterPoints[3] = CLitVertex(m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                      0.0f, 0xFFFFFFFF, 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 = 0xFFFFFFFF;

        // 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, 0xFFFFFFFF, 0.0f, 0.0f);
    Points[PointCount++] = CLitVertex(0.0f, WaterHeight,
                                    m_nHeightMapHeight * m_vecScale.z,
                                    0xFFFFFFFF, 0.0f, 1.0f );
    Points[PointCount++] = CLitVertex(m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                    m_nHeightMapHeight*m_vecScale.z,
                                    0xFFFFFFFF, 1.0f, 1.0f );
    Points[PointCount++] = CLitVertex(m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                    0.0f, 0xFFFFFFFF, 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 to 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.

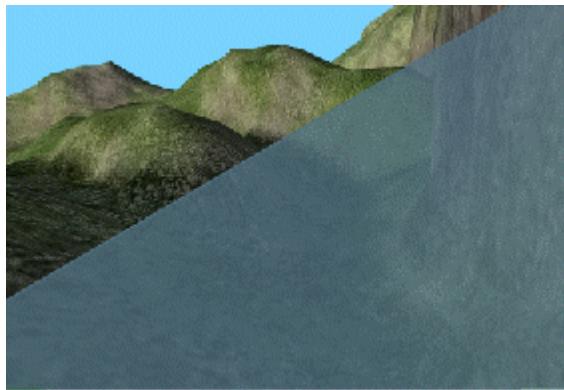
Note that many games forbid you from doing having a partially submerged camera by automatically pushing the player all the way under as soon as they enter the water. If you decided to employ such a

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 )
{
    int          PointCount = 0;
    3DRS_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 );
    }
}

```

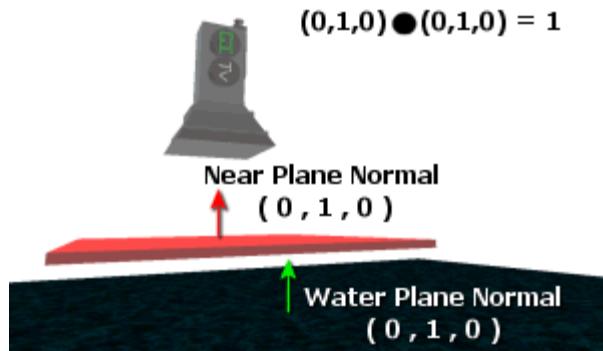
It is now time to create our screen effect polygon which may or may not need to be clipped depending on the position and orientation of the camera. If the near plane is totally or partially in the water then a screen effect polygon will be needed.

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

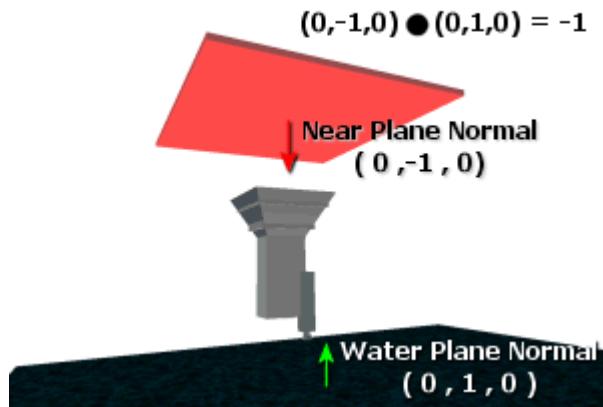
In the first case where the camera is looking down at the water, the two normals will be the same and the dot product will return 1.0 (approximately) as shown next:

### Checking for CoPlanar



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.

### Checking for CoPlanar



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++]=CLTLitVertex((float)m_Viewport.X,(float)m_Viewport.Y,
                                    0.0f, 1.0f, 0xBF547686);

Points[PointCount++]=CLTLitVertex((float)m_Viewport.X + m_Viewport.Width,
                                    (float)m_Viewport.Y,
                                    0.0f, 1.0f, 0xBF547686);

Points[PointCount++]=CLTLitVertex((float)m_Viewport.X+m_Viewport.Width,
                                    (float)m_Viewport.Y+m_Viewport.Height,
                                    0.0f,1.0f, 0xBF547686);

Points[PointCount++]=CLTLitVertex((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 );
```

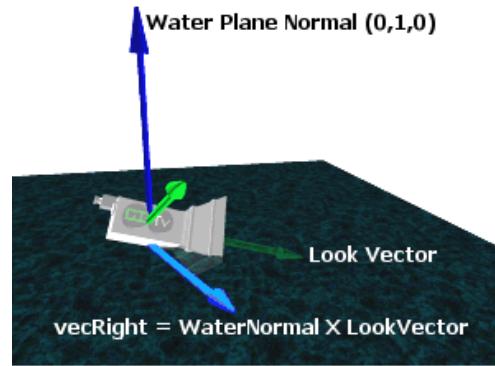
The image below shows a camera that is half in and half out of the water, intersecting the water at a complex angle. The diagram also depicts the orientation of the camera look and right vectors at this point.

### Camera Partially in Water



The next image shows what this new flattened right vector (vecRight) would look like.

### Flattening the Right Vector

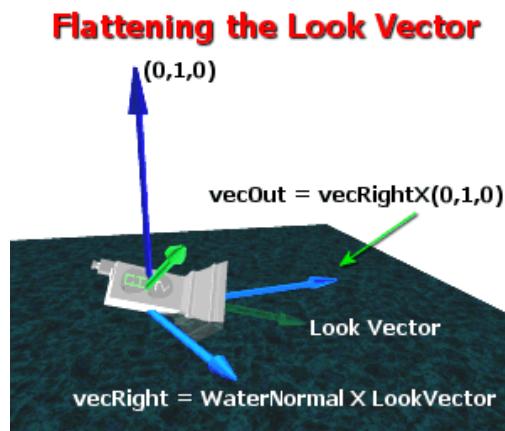


If we converted this vector into camera space, it would describe the slope of the water on the screen. Try it for yourself in your head. Imagine rotating the camera in the image above so that it is facing perfectly to the right on the page. Also imagine that you rotated all the vectors and the water 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 ( $\text{vecOut}$ ).

```
D3DXVec3Cross( &vecOut, &m_vecRight, &D3DXVECTOR3( 0, 1, 0 ) );
```

The image below shows us what this new vector would look like:

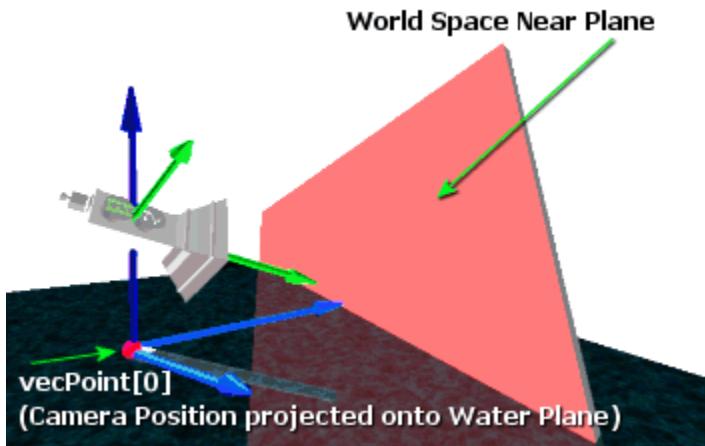


We now have new right and look vectors projected onto the water polygon plane. These vectors remain (partially) aligned to the camera look and up vectors. We will see why we need these vectors in a moment.

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 );
```

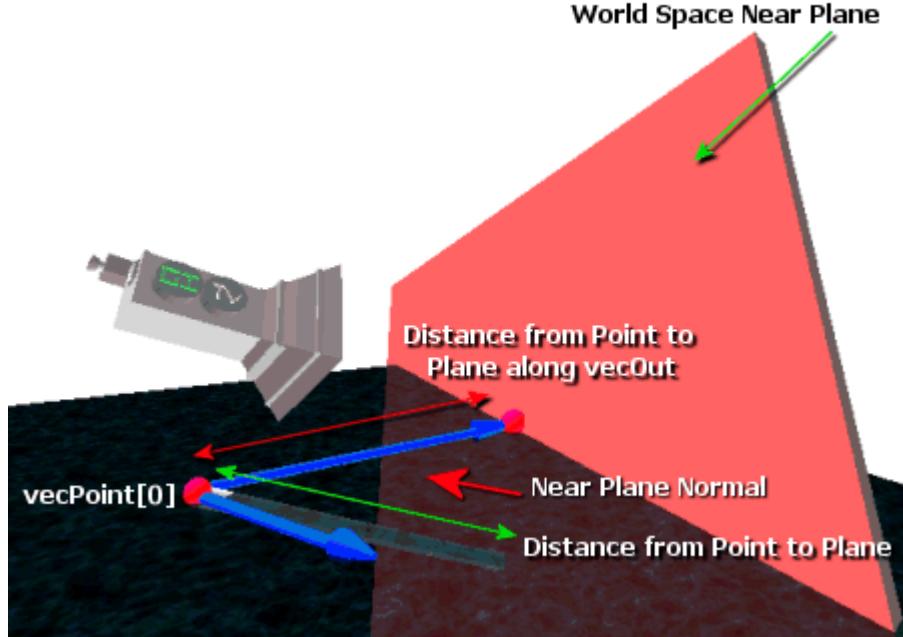
If this is hard to imagine, the following image should help. We have taken some liberties with the diagram so that we can see things more clearly. The camera is nudged up just out of the water purely for demonstration purposes. In reality, the camera would be much closer to the water plane.



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.

### Projecting vecPoint[0] onto Near Plane



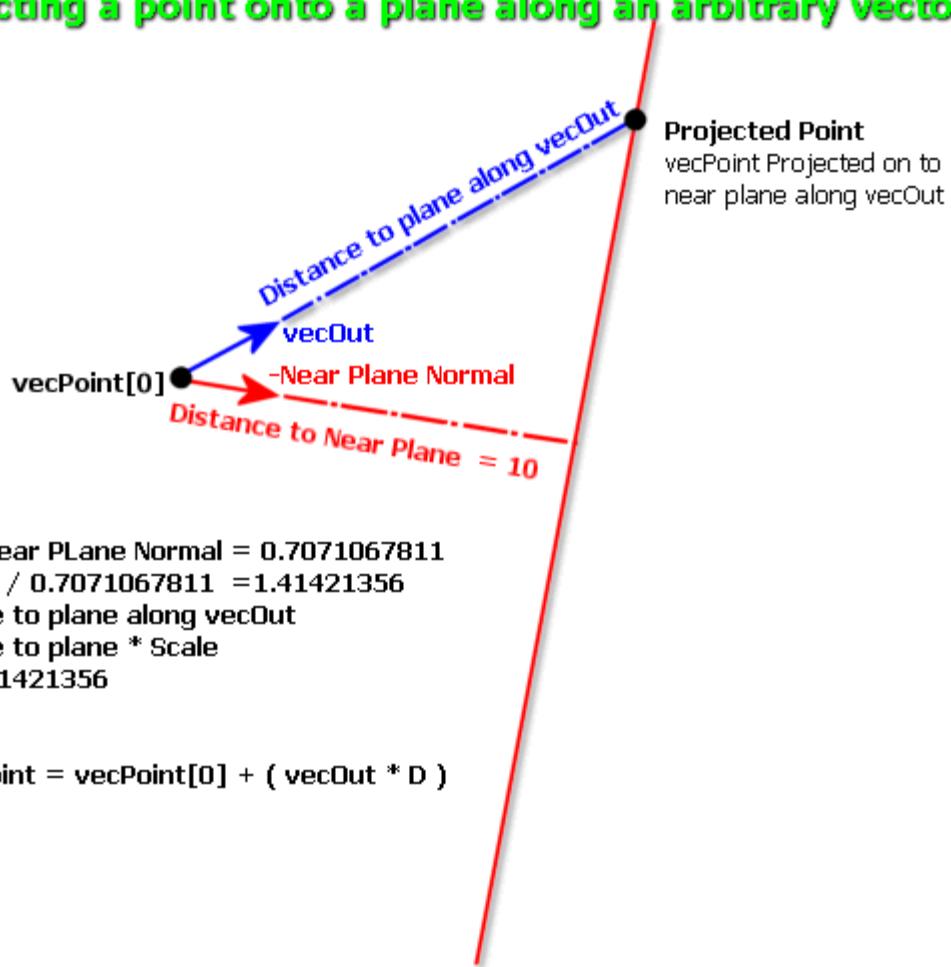
The above image shows us that in order to do this we need to find the distance from our point to the near plane along the flattened look vector. We know that to get the distance from a point to a plane we can use the D3DXPlaneDotCoord function. However the problem with this function is that it provides the shortest distance from the point to the plane along the plane normal as shown by the green arrow in

The 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 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.

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.

### Projecting a point onto a plane along an arbitrary vector



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 ( $1/\text{CosineOfAngle}$ ). 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.

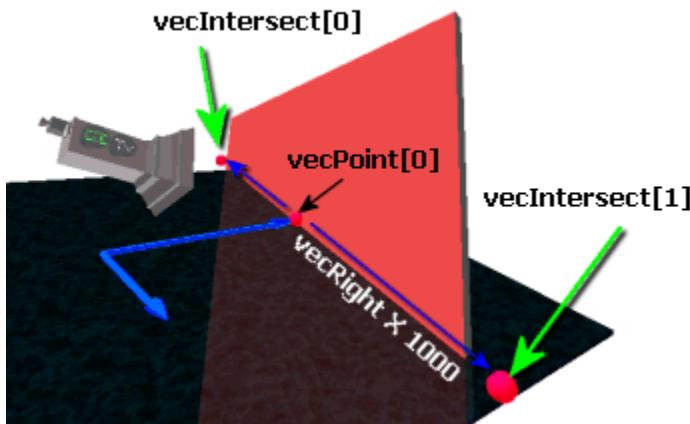
```
// Project the water point onto the plane along the vecOut
DistFromPlane = D3DXPlaneDotCoord( &NearPlane, &vecPoint[0] );
DistFromPlane *= (1 / D3DXVec3Dot( &(-PlaneNormal), &vecOut ));

// 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);
```

### Sliding `vecPoint[0]` along `vecRight`



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

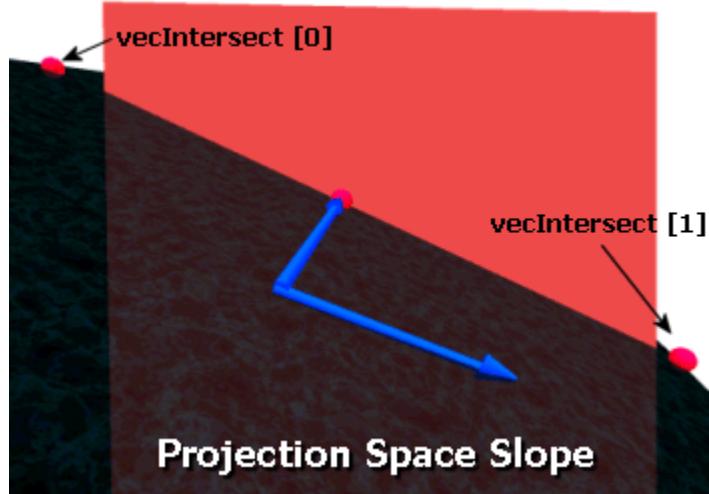
a combined view/projection matrix available which we used to extract the world space view plane, so let us multiply the two end points (`vecIntersect[0]` and `vecIntersect[1]`) by this combined matrix. This will convert the vertices into projection space. Points in the  $-1$  to  $+1$  range on both 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 =  $v_0 - v_1$**       (this line is described by our two projection space points)

**EdgeNormal.x =  $-(v_1.y - v_0.y)$**

**EdgeNormal.y =  $(v_1.x - v_0.x)$**

Let us imagine that we have a straight horizontal line consisting of  $v_1 (0, 0)$  and  $v_2 (10, 0)$ . 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 =  $-(0 - 0) = 0$**

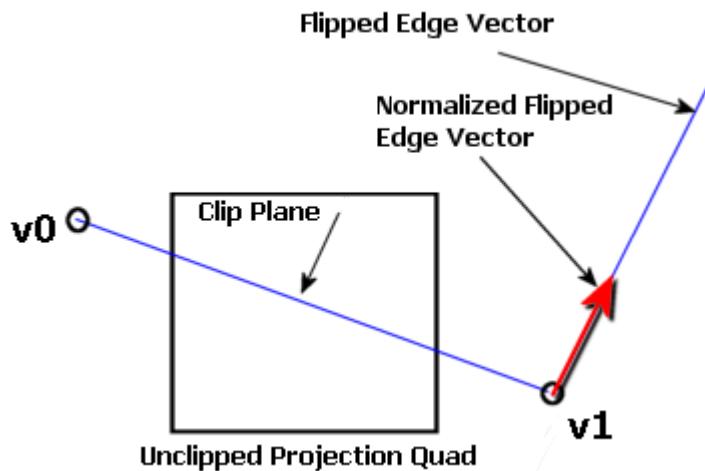
$$2DNormal.y = (10 - 0) = 10$$

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 );

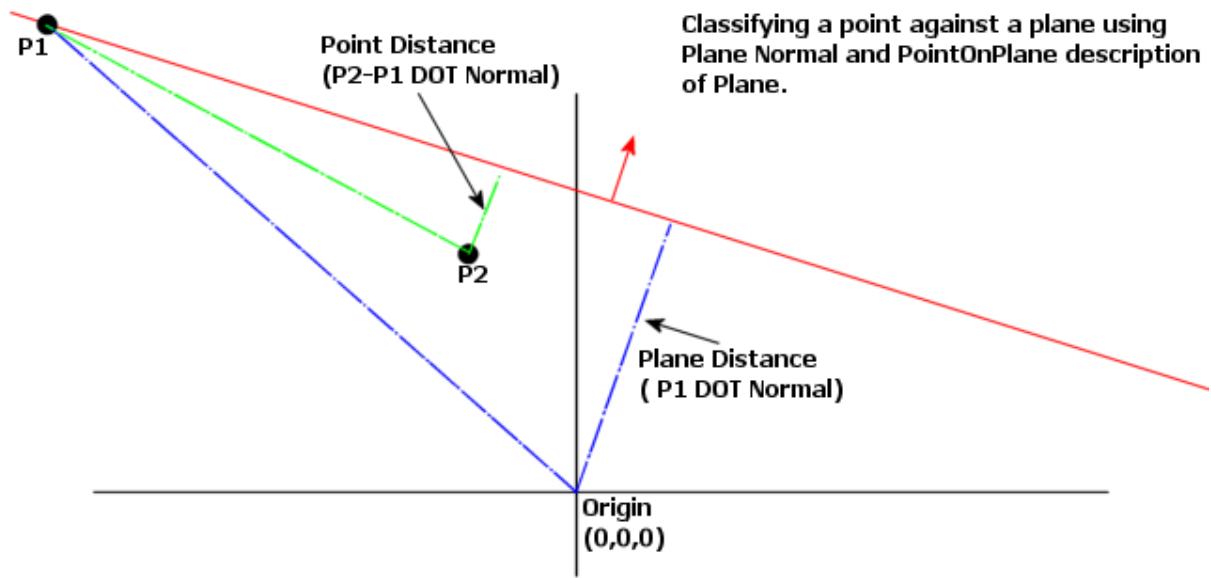
// Build initial 4 corner vectors
D3DXVECTOR2 vecScreenPoints[4];
vecScreenPoints[0] = D3DXVECTOR2( -1.0f, 1.0f );
vecScreenPoints[1] = D3DXVECTOR2( 1.0f, 1.0f );
vecScreenPoints[2] = D3DXVECTOR2( 1.0f, -1.0f );
vecScreenPoints[3] = D3DXVECTOR2( -1.0f, -1.0f );
```

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 known to be on the plane (any of our line edge points would do) this does present us with a nice opportunity to look at 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;

    float result=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 its 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));

    // Reeset 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.

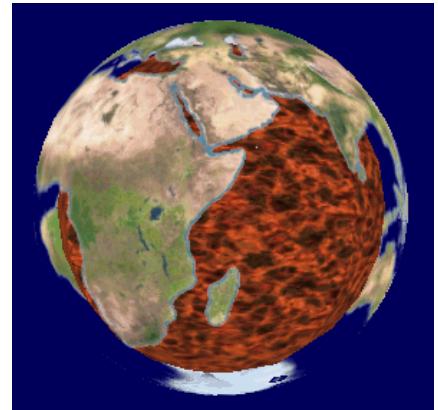
Next we set the flexible vertex flag informing the device that we are using pre-transformed and pre-lit vertices. We will not be requiring the transformation or lighting pipeline here since our vertices are already in screen space and already colored. We then enabled alpha blending and rendered our array 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 than 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 GILEST™. GILEST™ 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 rendered 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.

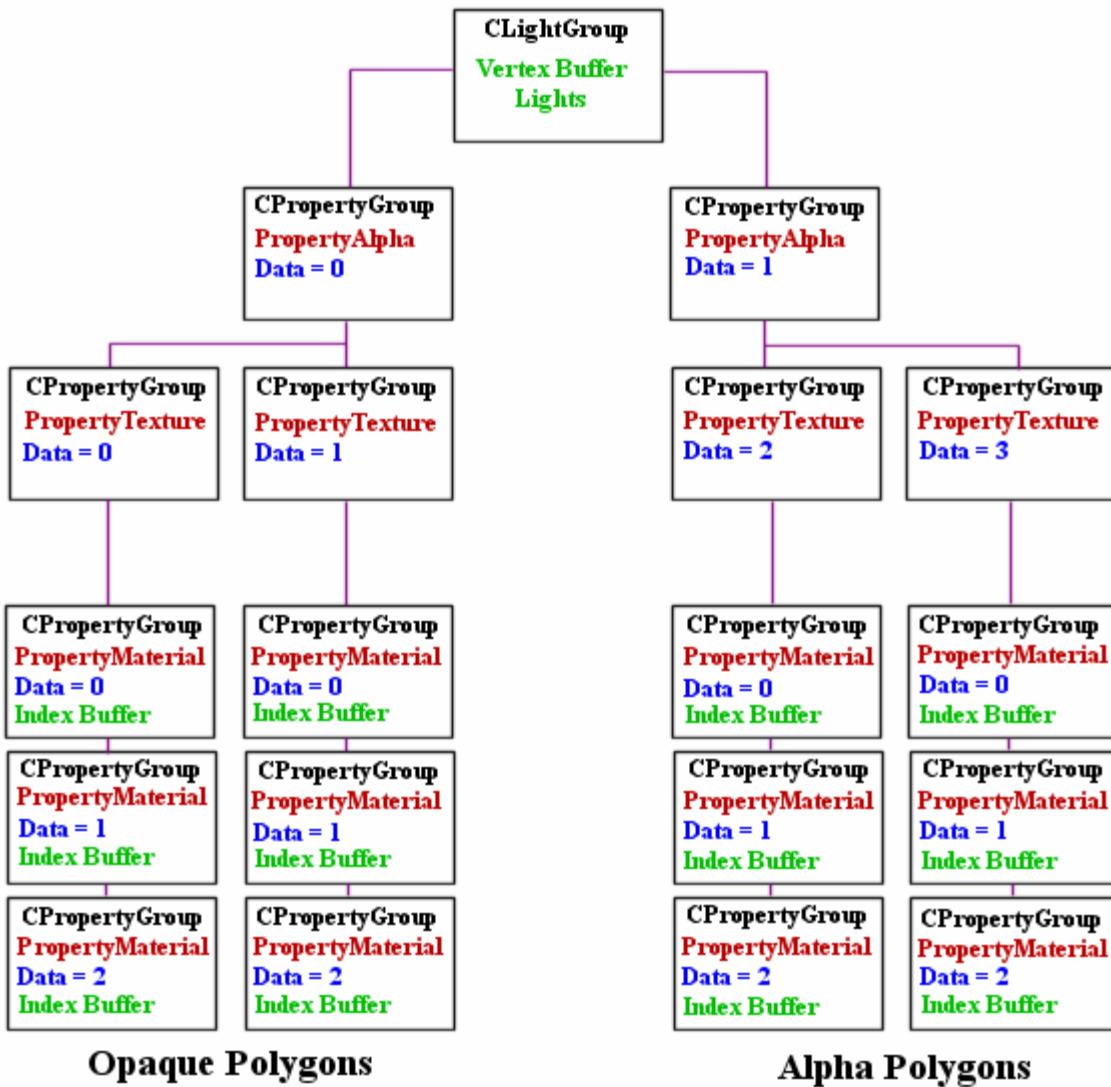
Alpha polygons will need to be sorted using the distance from the polygon to the camera as the sort heuristic. Since the camera can move about from frame to frame, we know that this will constantly change the relationship between the camera and the alpha polygons. Thus we cannot sort the polygons as 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:

## Property Group Hierarchy ( per Light Group )



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 be 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 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 != 1 || 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 IWFSSurface 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 be compiled.

```
bool CPropertyGroup::BuildBuffers(LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL,
                                  bool ReleaseOriginals, 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 );

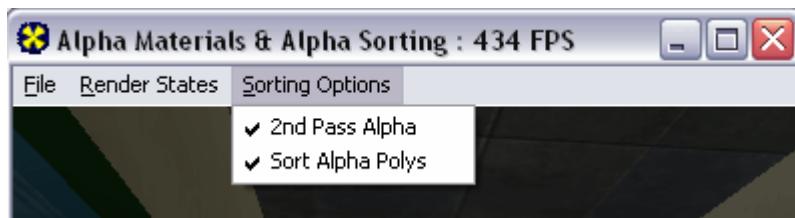
// Select alpha blending states
m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
```

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 2<sup>nd</sup> 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 2<sup>nd</sup> 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 2<sup>nd</sup> 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).

If 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;
pSortItem->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 );

} // 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_ZWRITENABLE, 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
A dark brown, textured concrete sample.	A solid green grass sample.	A green grass sample with scattered yellow flowers.

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 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.

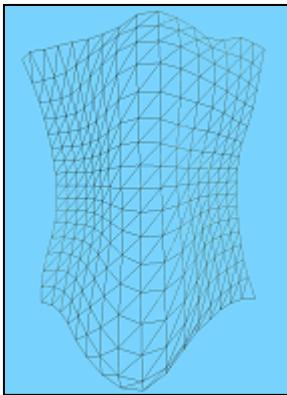
So clearly, using tileable textures and repeating them across the terrain affords us texture resolution that would otherwise not be possible. We will see later on that although the mesh is constructed such that its texture coordinates map a texture per quad, we can assign each layer a texture matrix 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 simple 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.

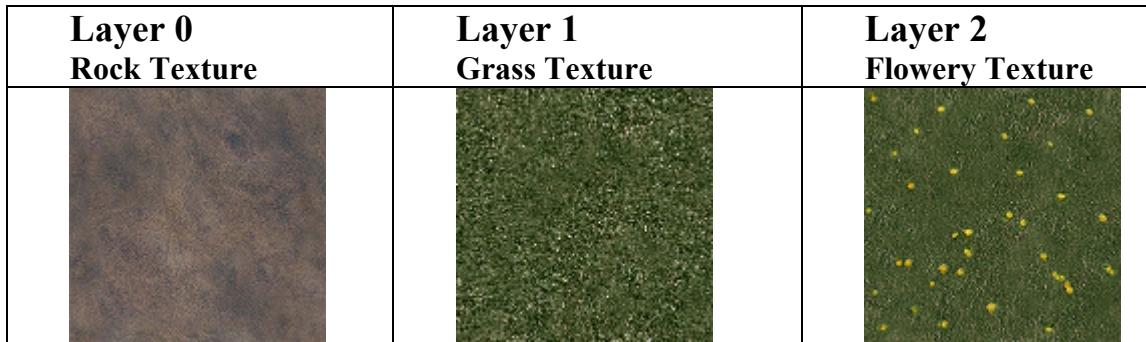
## What is a Texture Splat?

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.

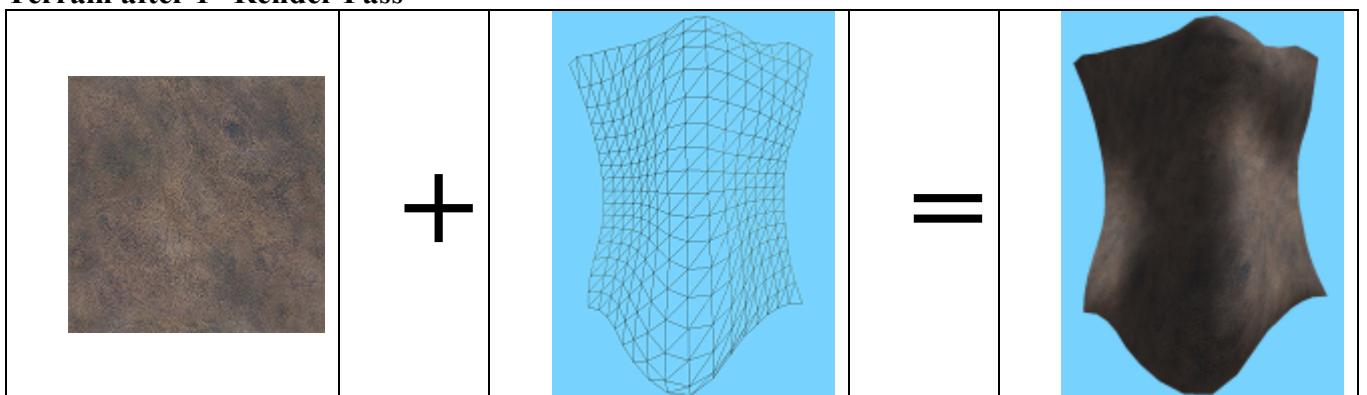


In our simple example, we will have an array of three CTerrainSplat objects, one for each texture layer depicted above. For simplicity, let us imagine that each splat object has an identical index buffer that contains all of the indices required to completely render every quad of the terrain. If we wanted 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 );
}
```

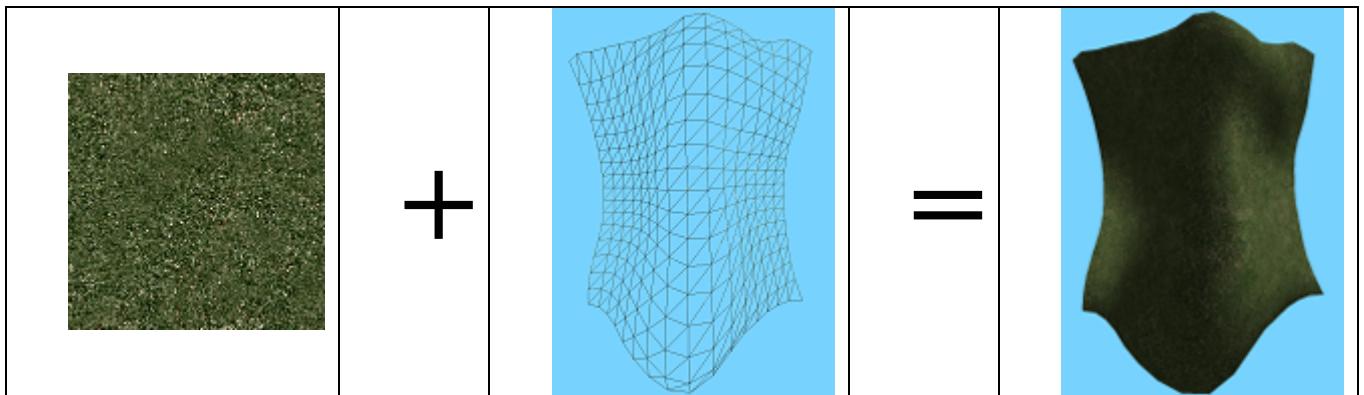
Assuming that the depth comparison function has been left at its default, where pixels are not rejected and are rendered if the Z value is less than or equal to a value already stored in the Z-Buffer, we can see that after the first iteration of the above loop we will have rendered the first layer (the rock layer). The frame buffer would contain the following image:

**Terrain after 1<sup>st</sup> 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:

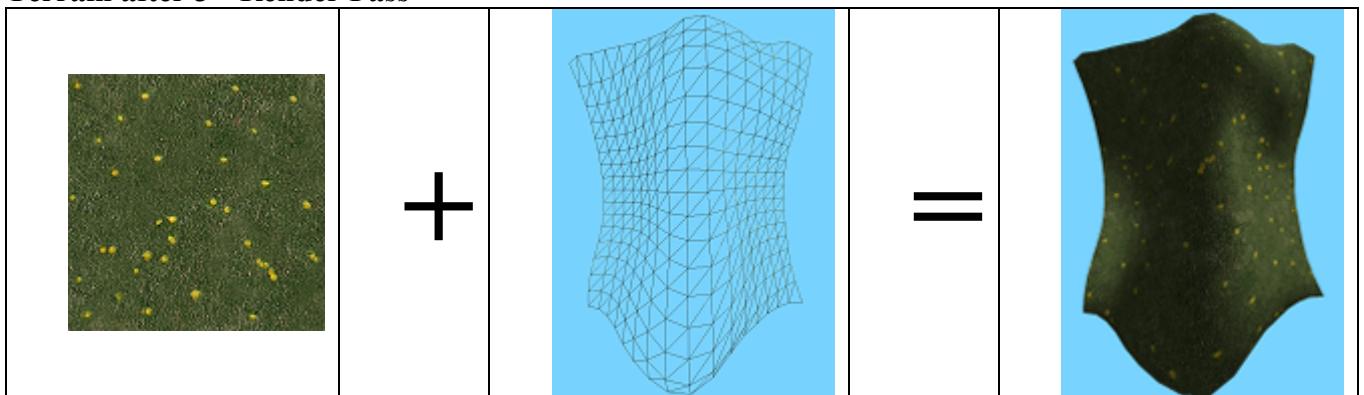
**Terrain after 2<sup>nd</sup> Render Pass**



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 3<sup>rd</sup> Render Pass**



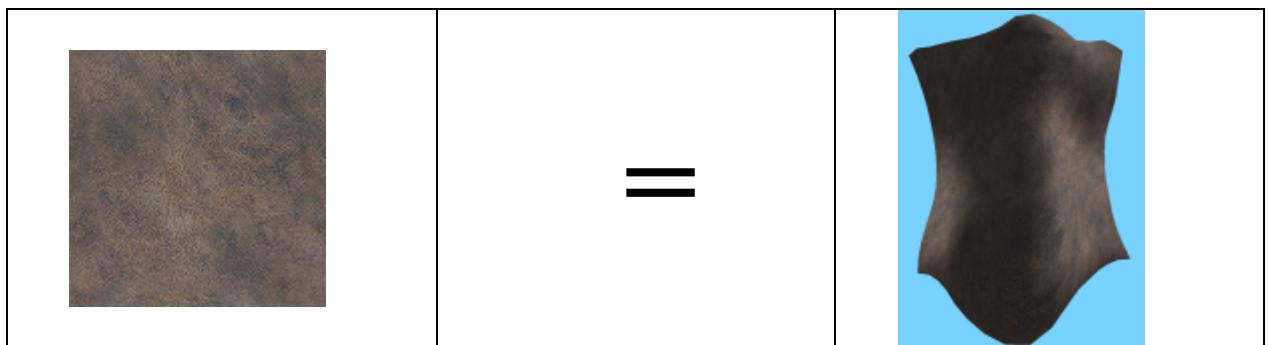
## The Alpha Texture

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 at 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

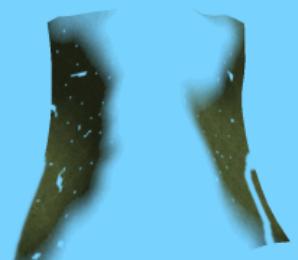
the vertex at the top left corner of the terrain has a UV coordinate of (0,0) and the vertex at the bottom right corner of the terrain has a UV coordinate of (1,1).

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.

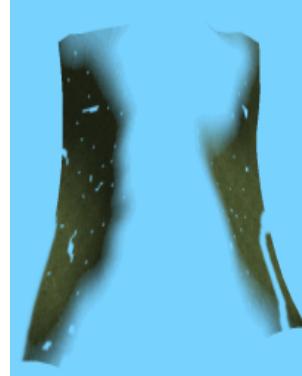
For example, as you will see later in our application, we load the image into memory, create a DirectX texture with an alpha channel, and copy only the blue component of each pixel in the source image into the destination texture's corresponding pixel alpha component. After we have filled the alpha components 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.

<b>Layer 1</b> Base Texture		<b>Layer 1</b> Alpha Map		<b>Rendered</b> <b>Terrain</b> Stage 0 – Base Texture Stage 1 – Alpha Map	<b>Blended</b>
	+		=		

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:

<b>Render Layer 0</b> No Alpha Map		<b>Render Layer 1</b> Using Alpha Map Layer 1		<b>Frame Buffer</b> Blended Result
	+		=	

### The 3<sup>rd</sup> Layer – The Flower Texture

The 3<sup>rd</sup> 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 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)**

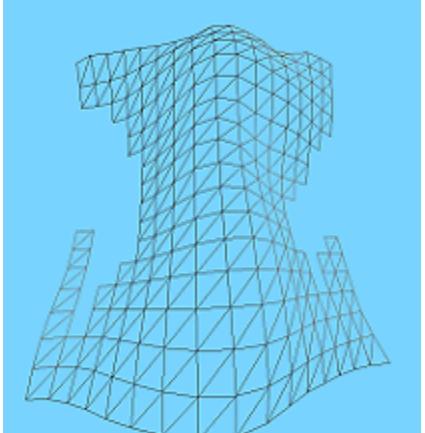
**If the matching pixel in any of the above layers blend maps is 255 (completely opaque)  
set the pixel in the current blend map to zero because it is completely obscured**

**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 contain 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.

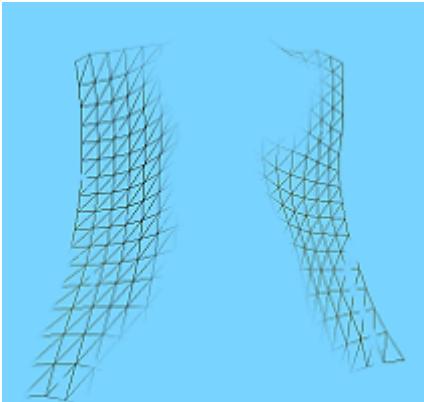
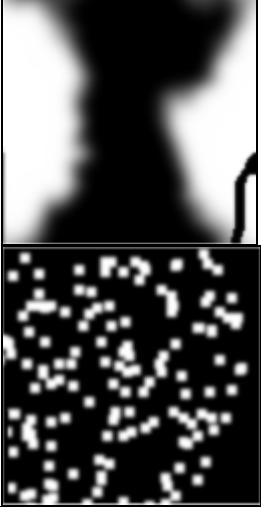
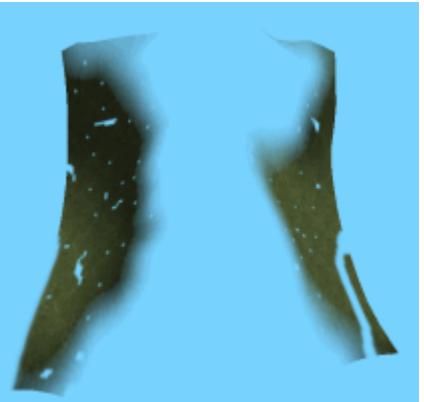
Below we can see what the splat for the base layer would look like after either completely transparent quads or quads completely obscured by higher level splats have been removed from consideration when building the splats index buffer.

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 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.

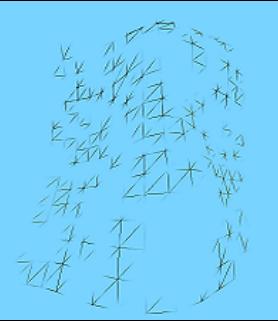
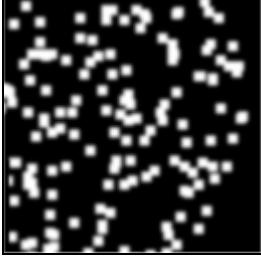
You can also see that where the layer above has black pixels in the alpha map, the current layer will not be overdrawn and will need to show through, so we leave the quads in place. We only remove quads from a splat if the corresponding pixels in the occluding layer's alpha map are totally white (full intensity). You will notice by looking at the alpha map of the 2<sup>nd</sup> 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.

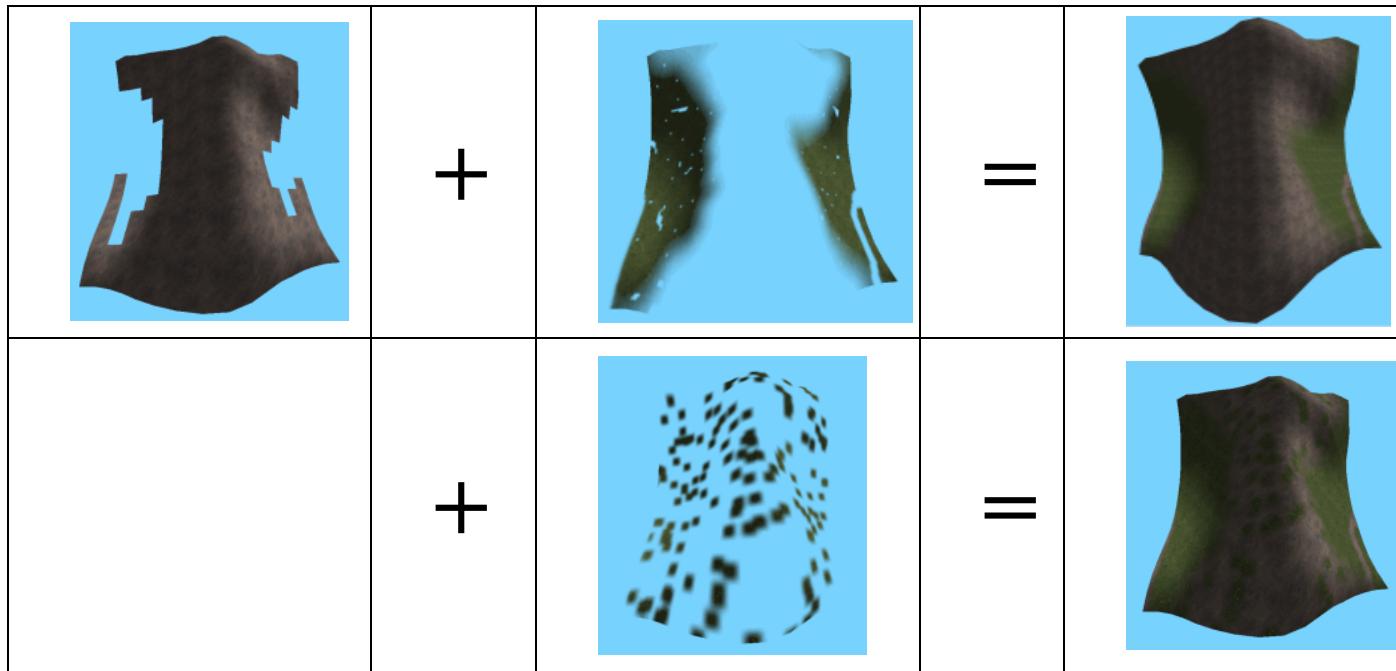
<b>Layer 1 Splat</b>	<b>Layer 1 Alpha Map Layer 2 Alpha Map</b>	<b>Rendered Result</b>
		

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.

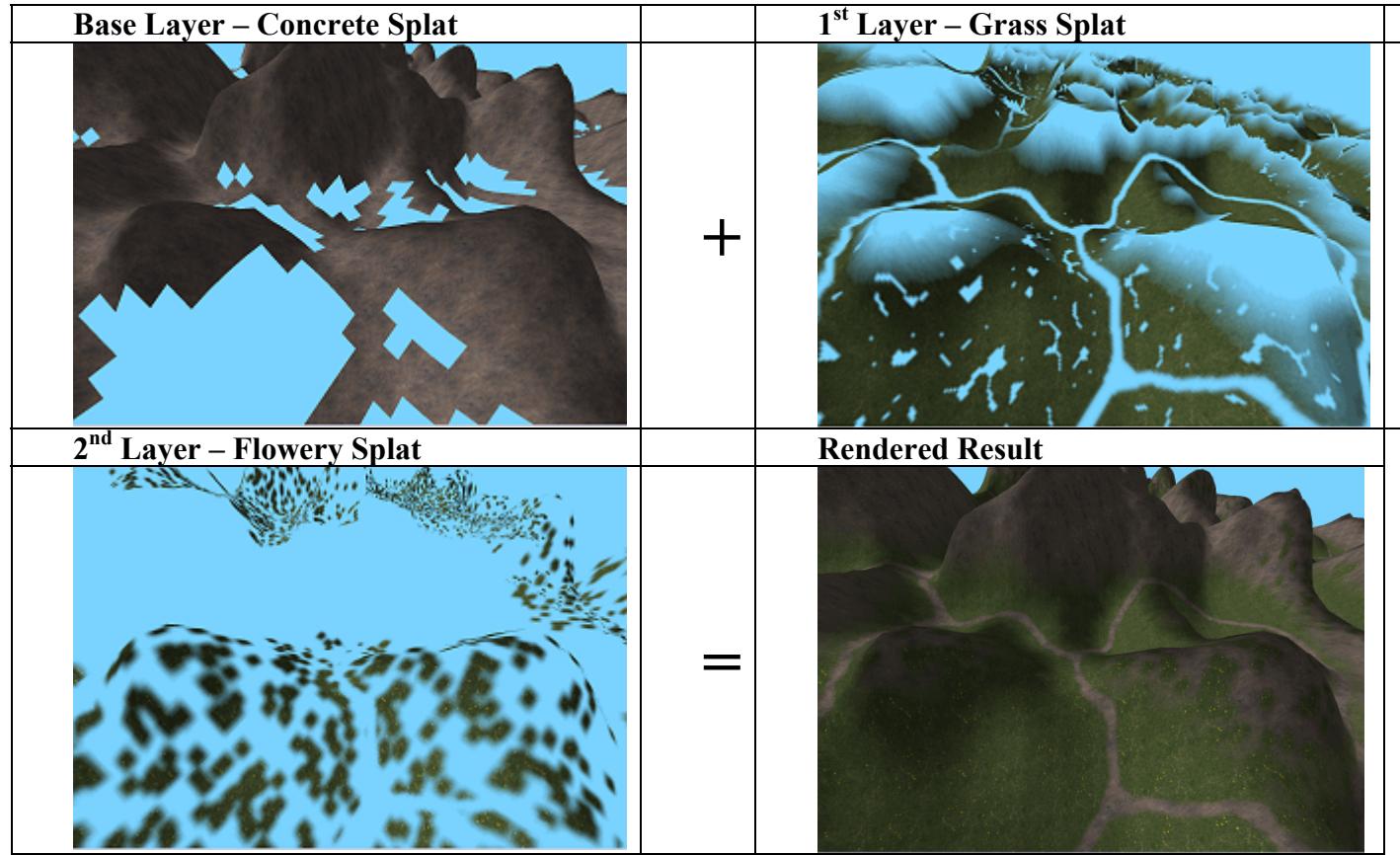
The following images show the splat created for the third and topmost layer, rendered using our flower texture to dot some random looking foliage about the terrain. Notice how the spaces in the splat where there are missing quads match up exactly with the larger dark regions of the layer's alpha map. 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.

<b>Layer 2 Splat</b>	<b>Layer 2 Alpha Map</b>	<b>Rendered Result</b>
		

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.



## 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.

```

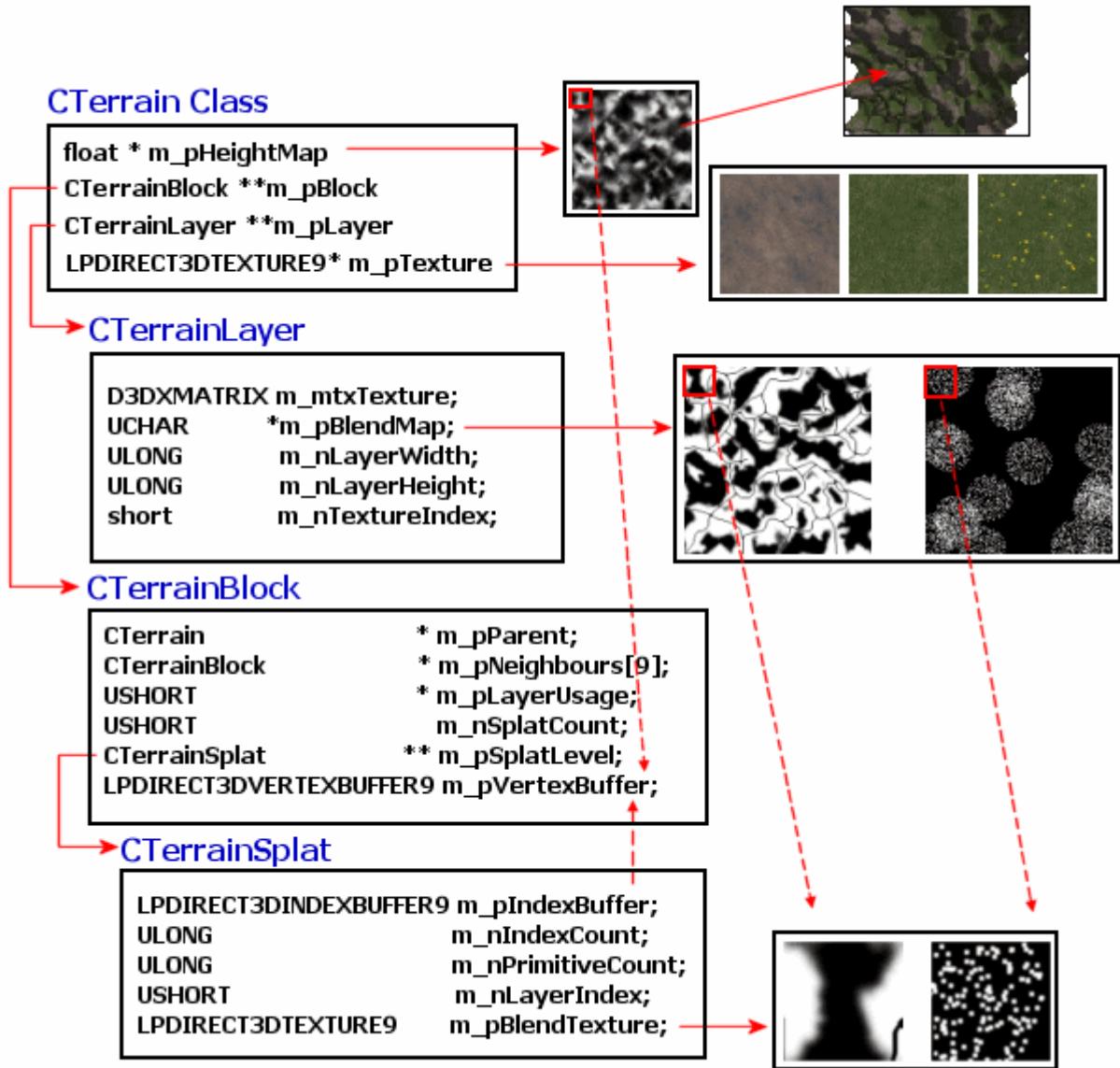
For Each Block of the Terrain
  For Each Layer
    If (Block.SplatLevel[layer] exists) Render ( Block.Splat(Layer) )
  End for Layer
End for Block

```

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 th 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.

<b>[General]</b>
<b>Name</b> = <b>Test Terrain</b>
<b>Desc</b> = <b>This terrain is designed to test the texture splatting technique.</b>
<b>Heightmap</b> = <b>Heightmap.raw</b>
<b>Scale</b> = <b>190.0, 10.0, 190.0</b>
<b>TerrainSize</b> = <b>129, 129</b>
<b>BlendTexRatio</b> = <b>8</b>
<b>BlockSize</b> = <b>17, 17</b>
<b>LayerCount</b> = <b>3</b>

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 is 128x128 quads, the alpha map files for each layer will be expected to be  $(128*8) \times (128*8) = 1024 \times 1024$  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 neighbouring 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; }
}

// 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.

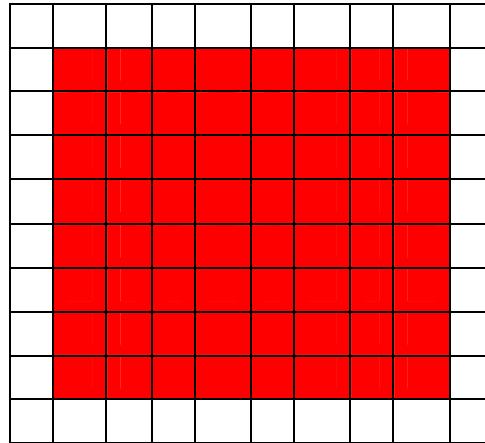
Our filter will loop through each pixel in the height map and will sum the color of a 9x9 block of pixels with the pixel in the middle. Once we have summed the nine colors together we will divide the final color by nine to produce an average pixel color for that region of the terrain as shown below.

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:

$$200+0+100+9+87+99+128+238+45 \quad / \quad 9 = 100.666$$

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 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 overwritten 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 neighbouring

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;

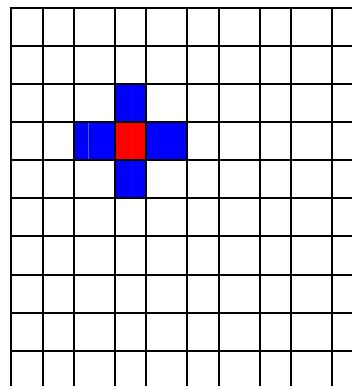
        } // End if Test Occlusion
    } // Next Column
} // Next Row
} // Next Layer

// 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 neighbouring 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 neighbour
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  $8 \times 8 = 64$  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_nBlocksWide; 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 colour 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 colour 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;

// Success!
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.

```

// Loop through each layer
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;
}

// Success!!
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, D3DTFF_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?

$$\text{SourceColor} * \text{SrcBlendMode} + \text{DestColor} * \text{DestBlendMode}$$

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 <b>D3DBLEND</b> 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 <b>D3DBLEND</b> 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 <b>D3DCOLOR</b> 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 <code>D3DTA_TFACTOR</code> this color will be used. If the state is blending two colors using the <code>D3DTOP_BLENDFACTORALPHA</code> color operation, the alpha component of this color is used to blend the two input colors.
<code>D3DRS_ALPHATESTENABLE</code>	True or False	If set to true, before a pixel is rendered its alpha value is tested against a reference value (set by the <code>D3DRS_ALPHAREF</code> ) render state using a comparison function selected by the <code>D3DRS_ALPHAFUNC</code> renderstate. If the alpha value for a pixel fails the comparison test then it is rejected and will not be rendered.
<code>D3DRS_ALPHAFUNC</code>	<p>A member of the <code>D3DCMPFUNC</code> enumerated type. This can be one of the following:</p> <p><code>D3DCMP_NEVER</code>,  <code>D3DCMP_LESS</code>,  <code>D3DCMP_EQUAL</code>,  <code>D3DCMP_LESSEQUAL</code>,  <code>D3DCMP_GREATER</code>,  <code>D3DCMP_NOTEQUAL</code>,  <code>D3DCMP_GREATEREQUAL</code>,  <code>D3DCMP_ALWAYS</code>.</p> <p>The default is <code>D3DCMP_ALWAYS</code> in which a pixel is never rejected based on its alpha value because it always passes the comparison test</p>	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 <code>D3DCMP_LESS</code> 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 <code>D3DRS_ALPHAREF</code> function. This is useful for rejecting completely transparent pixels so that they do not have their depth values written to the depth buffer.
<code>D3DRS_ALPHAREF</code>	<code>DWORD</code>	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	<b>D3DFOG_NONE</b> <b>D3DFOG_LINEAR</b> <b>D3DFOG_EXP</b> <b>D3DFOG_EXP2</b>	Sets the fog model used for vertex fog mode, or disables vertex fog if set to <b>D3DFOG_NONE</b> .
D3DRS_FOGTABLEMODE	<b>D3DFOG_NONE</b> <b>D3DFOG_LINEAR</b> <b>D3DFOG_EXP</b> <b>D3DFOG_EXP2</b>	Sets the fog model used for pixel fog mode, or disables vertex fog if set to <b>D3DFOG_NONE</b> .
D3DRS_FOGSTART	Float (must be past 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 past 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 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 past 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 artefacts. 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 .

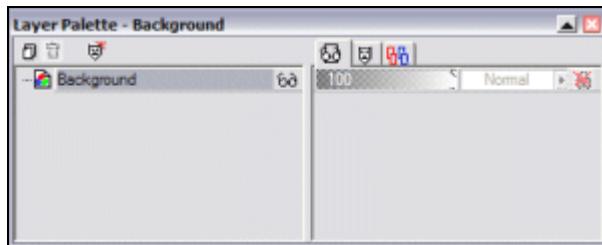
## Appendix B: A Quick Guide to Creating Alpha Channels in Paint Shop Pro™

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).



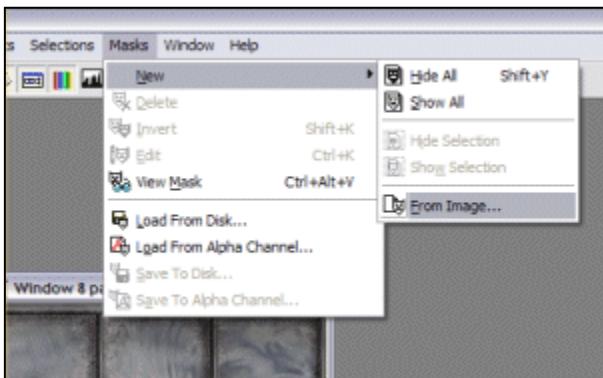
Before we begin, we need to pick an image that we wish to generate an alpha channel for. In this example we have chosen a window pane with 9 separate segments (shown to the left). We will create individual areas of translucency relatively easily by masking off the separated areas, and filling them in 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 (assuming we simply loaded up a single layer file such as a bitmap, etc.) you can pop open the Layer Palette window. At this point, you should notice that we have a single Background layer, as shown below.



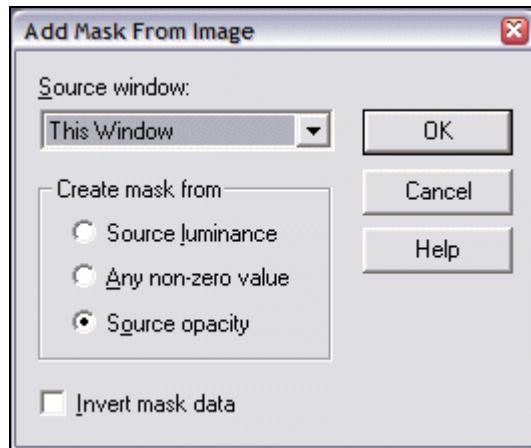
### Paint Shop Pro Masks

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 in order to apply alpha information to our image is to create a **mask**.

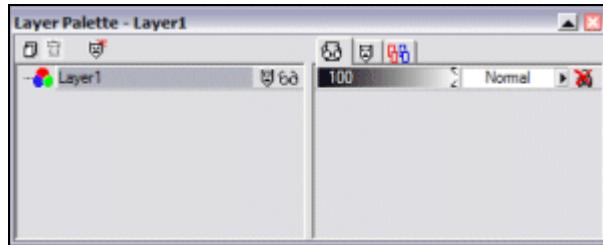


To create a mask, we need to select the 'From Image' item from within the 'Masks / New' menu item. After selecting this item, we are presented with an options dialog as shown below. This dialog allows us to specify how we would like our alpha mask to be set up initially. For now, we just want a completely 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:



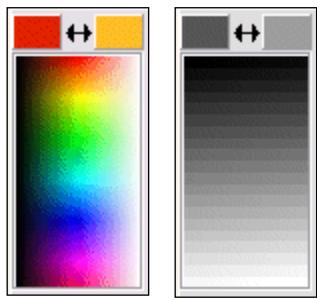
Once you have decided how your mask will be defaulted, select OK, and the mask for the currently selected layer will be generated. In this case, we had the single layer named 'Background' selected. These background layers are special types of raster layers which cannot be made translucent and are, as 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:



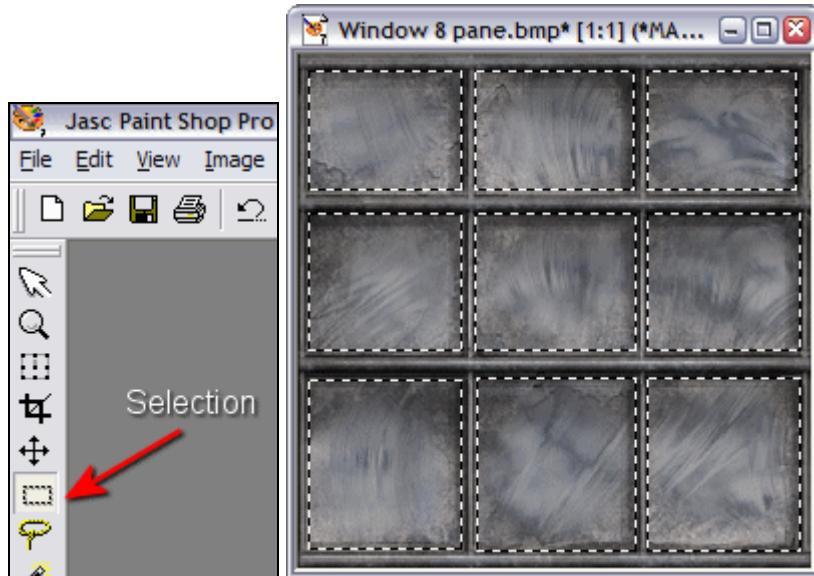
The palette shown to the left (the full color palette) is the traditional layer editing palette. The one on the right is used for editing the layer mask, and depicts the 256 levels of translucency as black (0 = Fully Translucent) through white (255 = Fully Opaque). It is worth noting however that it 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 before we do that we must make sure we are in solid color mode. To can click on the black arrow contained within the ‘Styles’ color underneath the palette. A small selection box will pop up allowing between ‘Solid’, ‘Gradient’, ‘Pattern’ or ‘Null’ modes. For now we 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 centre 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 window pane segment when it is rendered.



value, but do this you block you to choose want to select

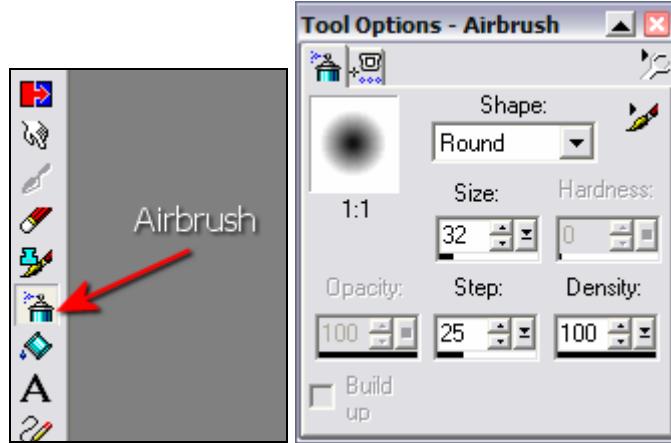
As we know, we want to leave the horizontal / vertical bars of our window pane totally opaque. This may be a little difficult to achieve, or at least a little laborious, if we were to avoid / adjust these areas by hand. We can solve this problem by using the selection tool:



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.

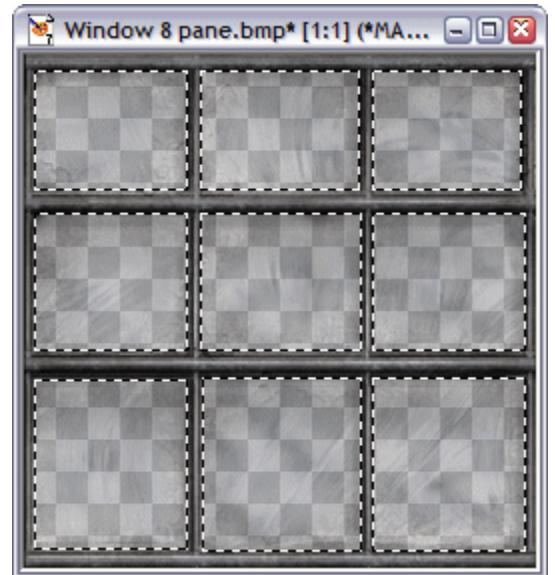
We are now ready to modify our mask. When in mask edit mode we can treat it just as if it was a simple palletized image, and can perform many of the same color based operations with it (ex. brightness, gamma, noise, etc.) in exactly the same way. With our alpha level “color” chosen, we can now pick an 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.

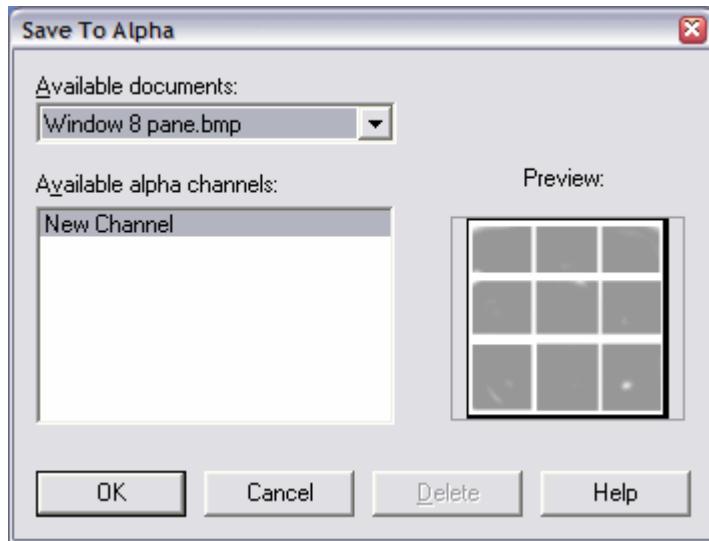
Now that we have our general alpha values set up, with our selection still intact (and still in the mask editing mode) we can touch up this image a little bit. We could for instance use the ‘Noise’ effect to add a little uniform noise (say 15%) which gives us a little variation in the alpha values. This 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).

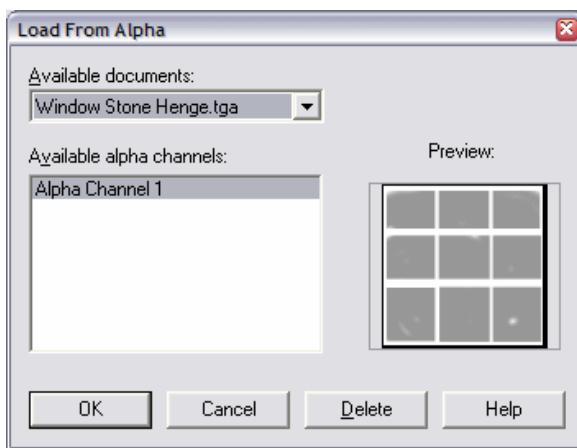
**Note:** Alpha channels are stored separately from the masks themselves. Therefore, if at any point after saving the mask to an alpha channel, you modify that mask, you will need to save it once again to the alpha channel in the same way, overwriting the original which will be displayed in this dialog.

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 ☺

