Graphics Programming with Direct X 9 Part I

e-Institute Publishing, Inc.

©Copyright 2004 e-Institute, Inc. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system without prior written permission from e-Institute Inc., except for the inclusion of brief quotations in a review.

Editor: Susan Nguyen
Cover Design: Adam Hoult

E-INSTITUTE PUBLISHING INC
www.gameinstitute.com

Gary Simmons and Adam Hoult. *Graphics Programming with Direct X: Part I*

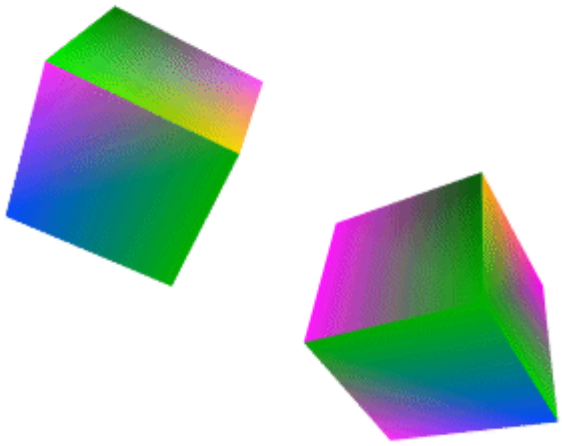
All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse of any kind of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

E-INSTITUTE PUBLISHING titles are available for site license or bulk purchase by institutions, user groups, corporations, etc. For additional information, please contact the Sales Department at sales@gameinstitute.com

Table of Contents

Chapter One

3D Graphics Fundamentals



Introduction

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

- user input
- resource management
- loading and rendering graphics
- interpreting and executing scripts
- playing sampled sound effects
- 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 series 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.

1.1 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 GILES™. 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).

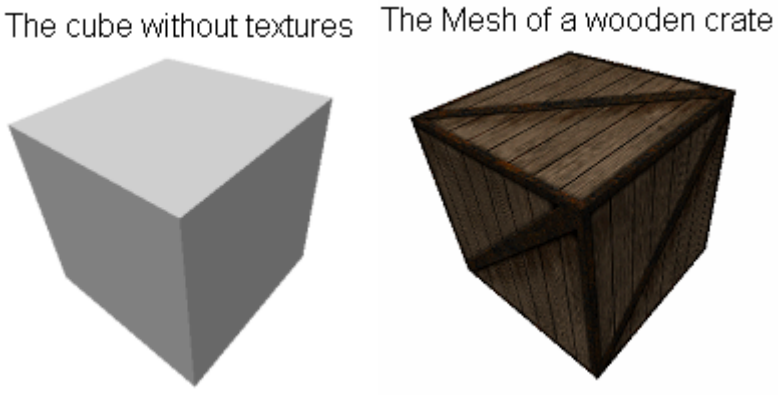


Figure 1.1

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:

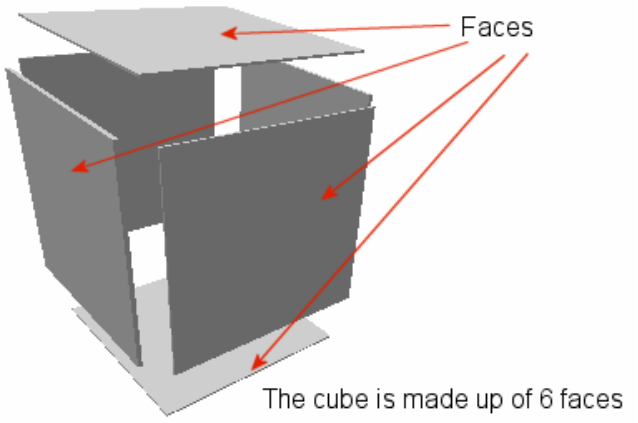


Figure 1.2

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).

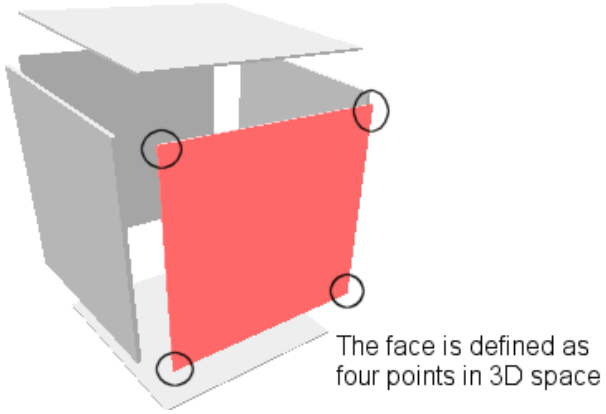


Figure 1.3

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.

1.1.1 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.

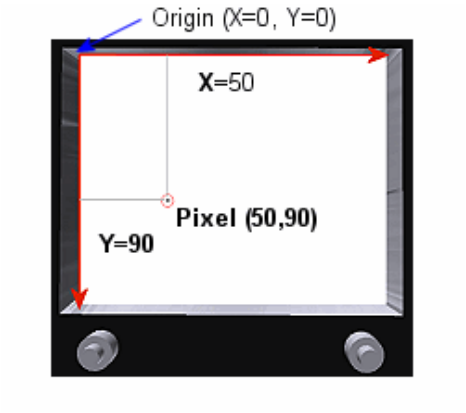


Figure 1.4

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.

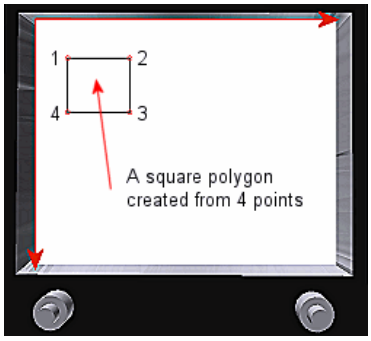


Figure 1.5

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).

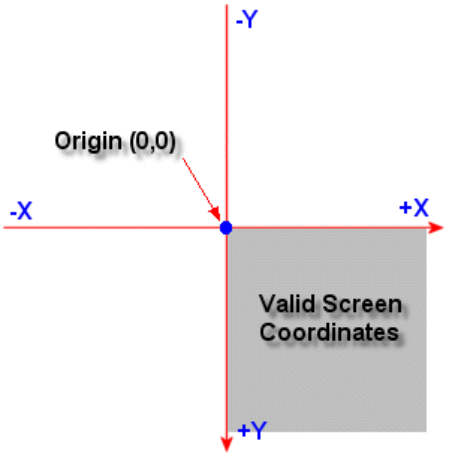


Figure 1.6

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:

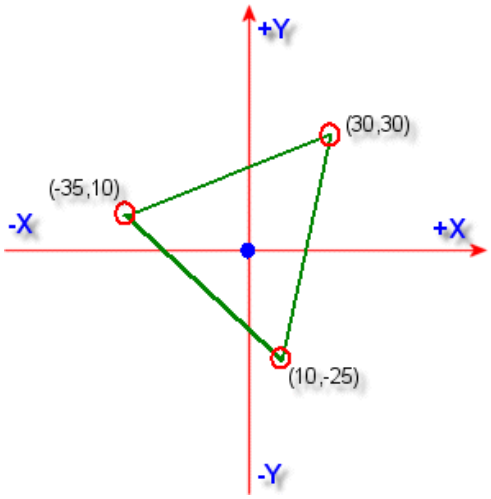


Figure 1.7

1.1.2 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.

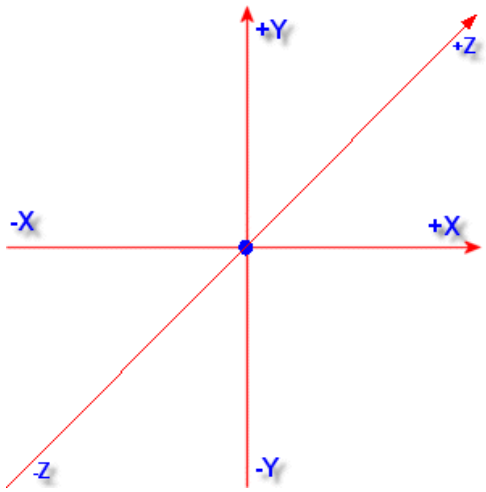


Figure 1.8

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).

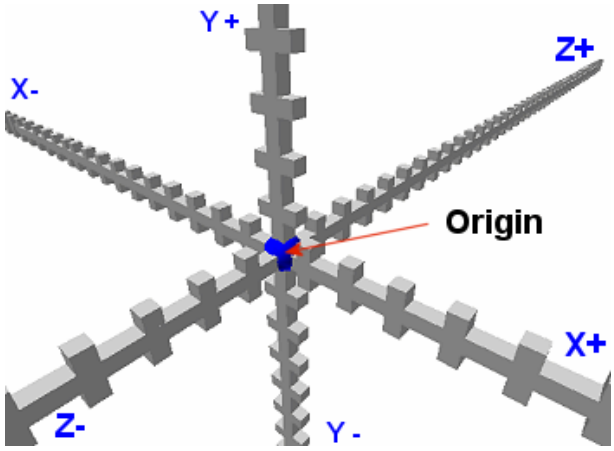


Figure 1.9

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.

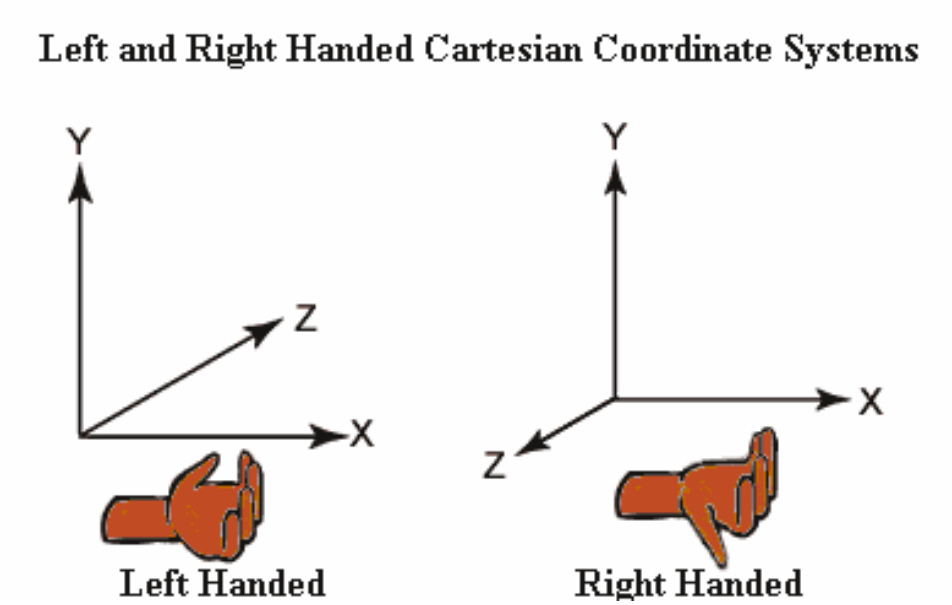


Figure 1.10

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.

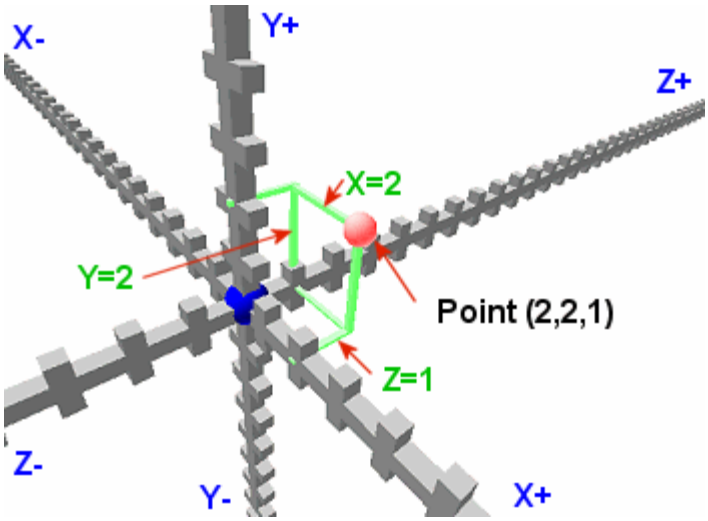


Figure 1.11

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;
};
```

1.1.3 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 4x4x4 where the extents of the cube on each axis range from –2 to +2.

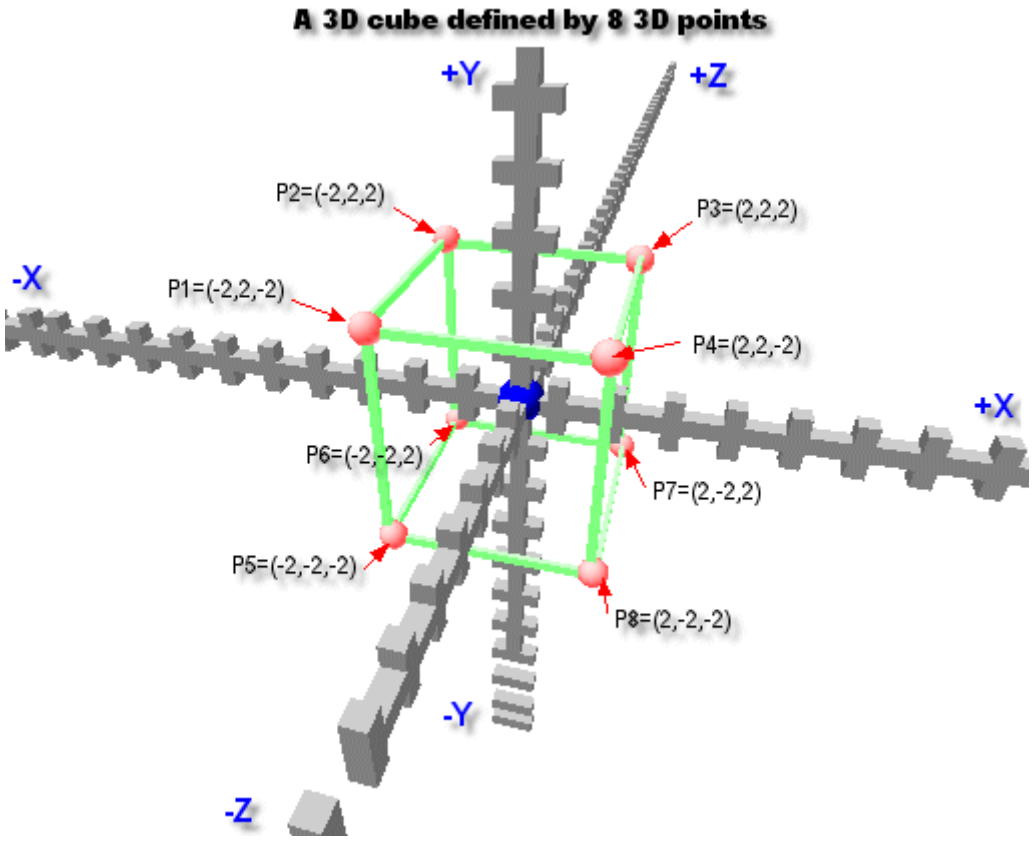


Figure 1.12

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.

1.1.4 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 x 4 = 24).

If you examine Lab Project 1.1, you will see that our vertex structure is defined as:

```
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
};
```

1.1.5 Polygon Winding Order

3D models will not usually be created programmatically but will be created within a modeling package such as GILES™ 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.

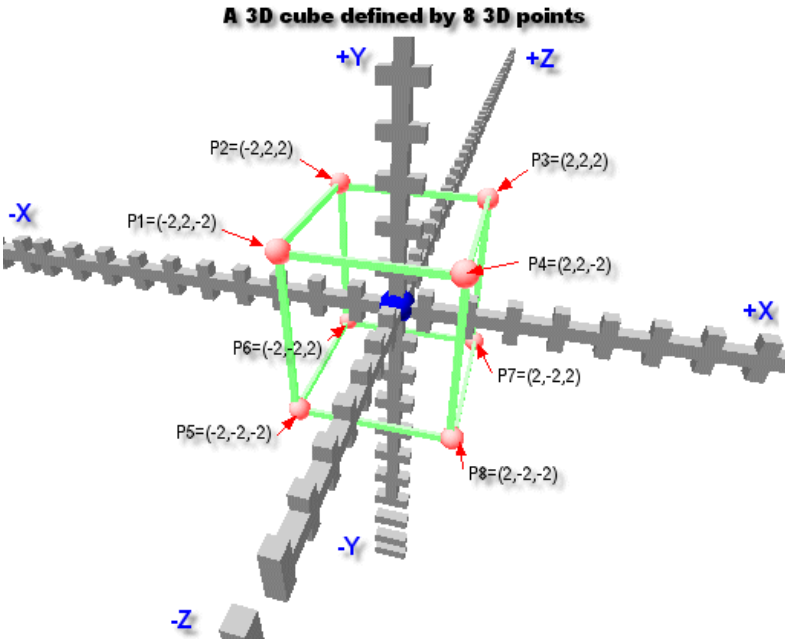


Figure 1.13

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**.

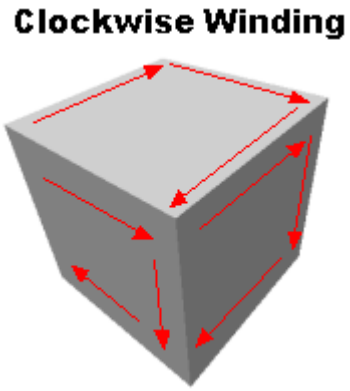


Figure 1.14

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. In this instance the back face would now be rendered and the old front face would not.

1.2 The Transformation Pipeline

1.2.1 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.

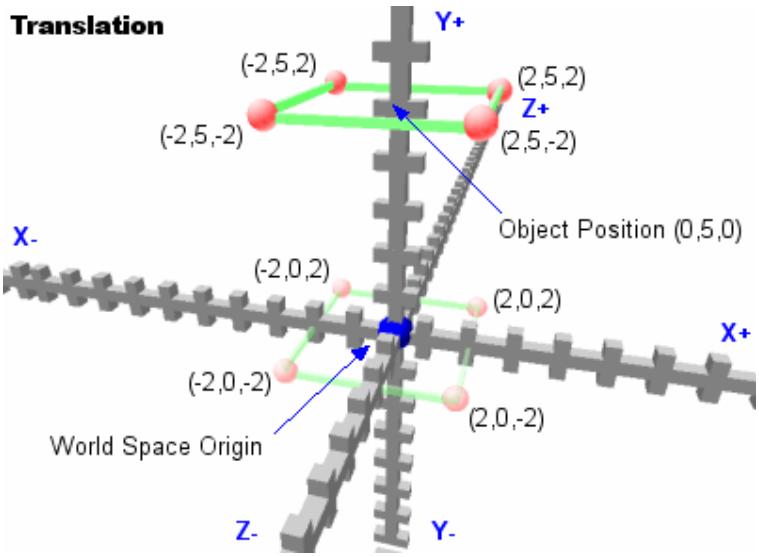


Figure 1.15

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.

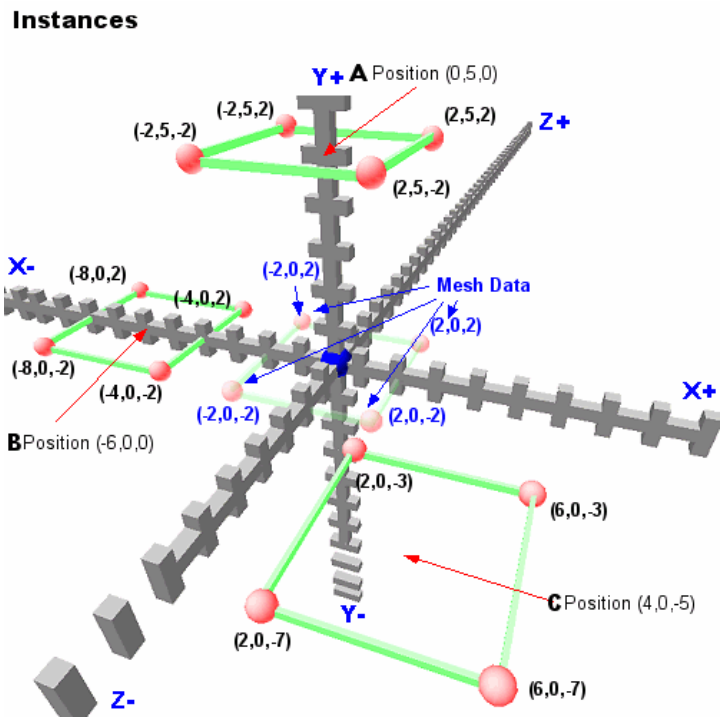


Figure 1.16

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.

1.2.2 Rotation

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

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

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 θ 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 (1.1570796 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 ) )
> ```

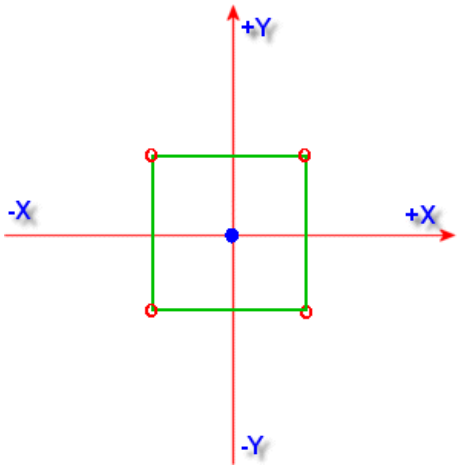


Figure 1.17

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:

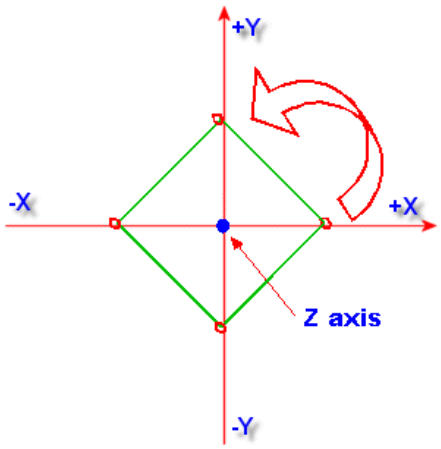


Figure 1.18

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

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

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.

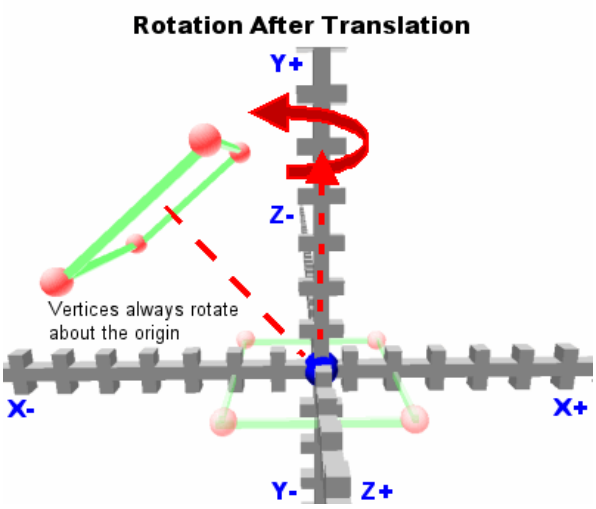


Figure 1.19

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).

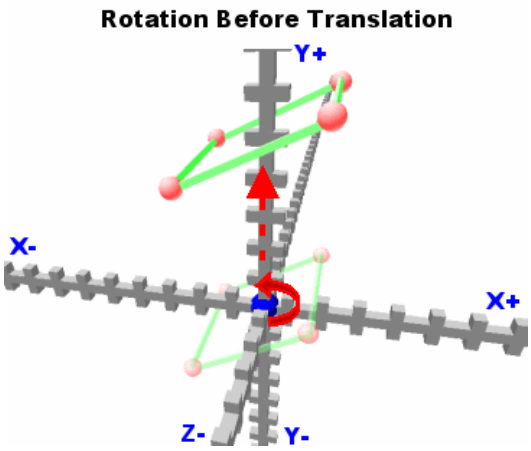


Figure 1.20

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 object 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.

1.2.3 Viewing Transformations

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. In effect, 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. 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
```

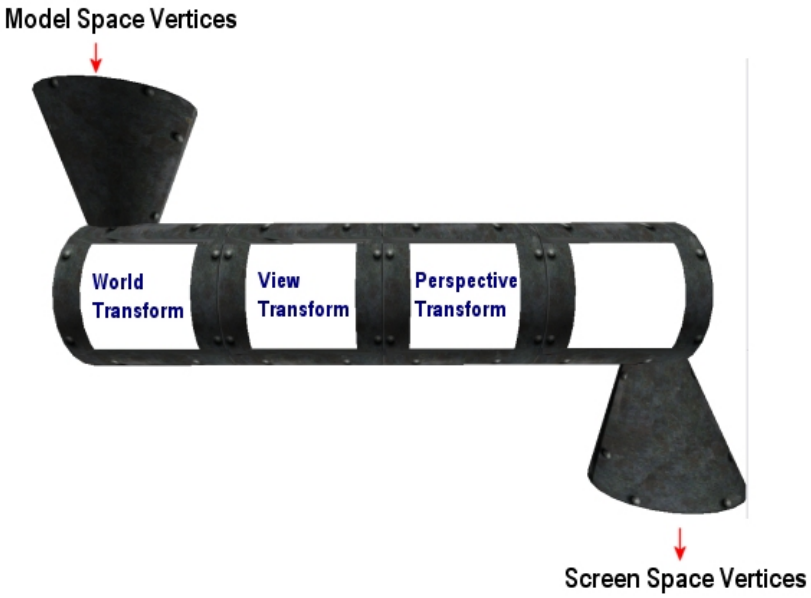
Y and Z axis rotations would follow along similar lines:

```
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.

1.2.4 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**.

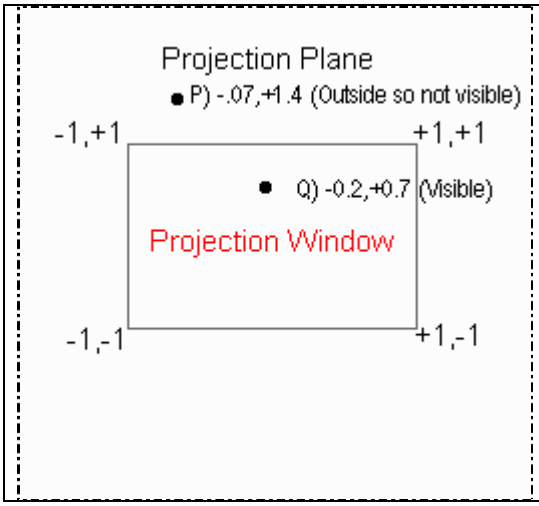
Imagine that we have a coordinate that is 5 units to the right of the camera, 20 units up from the camera and 100 units in front of the camera. This vertex would have a view space coordinate of (5, 20, 100). Performing the perspective projection:

$$projectedX = \frac{ViewSpaceX}{ViewSpaceZ} = \frac{5}{100} = 0.005$$

$$projectedY = \frac{ViewSpaceY}{ViewSpaceZ} = \frac{20}{100} = 0.2$$

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.



2D ***projection space*** coordinates have been mapped to the projection plane using the x/z and y/z technique.

Point ***P*** is not within the projection window because one of its coordinates is not within the -1 to +1 range.

Point ***Q*** will be visible from the camera position because its projected coordinates fall within the -1 to +1 range.

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 that the same logic would also apply to the X coordinate projections.

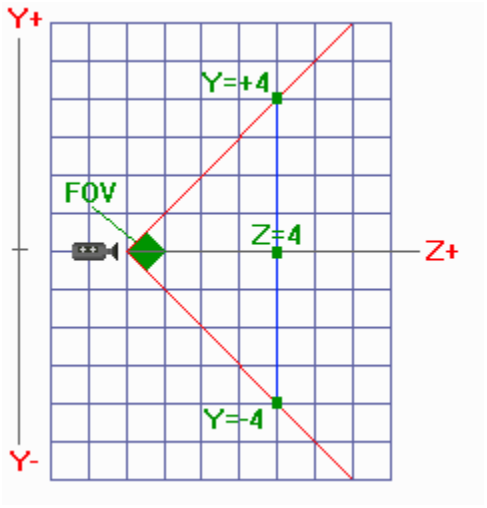


Figure 1.21

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:

$$projectedY = \frac{y}{z} = \frac{y}{y} = 1$$

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 in view space, 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 as shown in figure 1.21*. 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***.

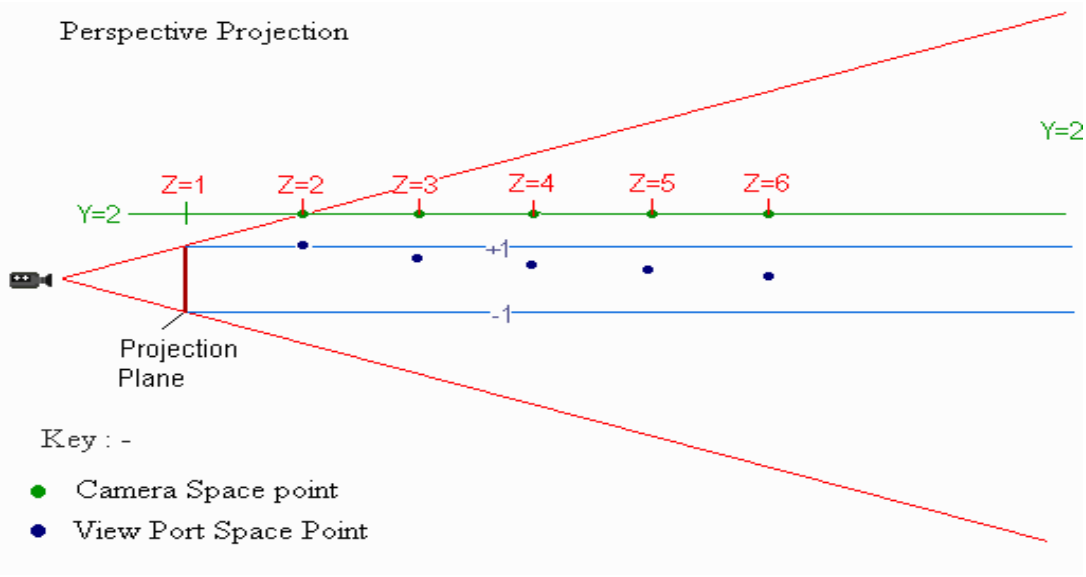


Diagram is not to scale.

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/projected 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:

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

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

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:

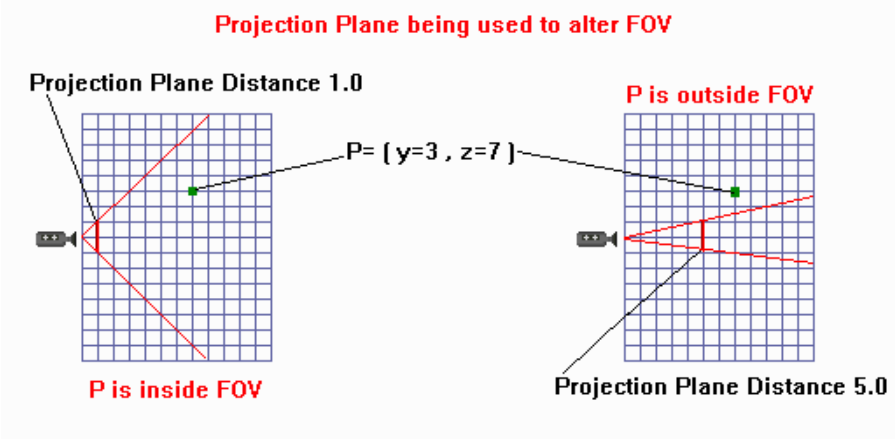


Figure 1.23

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:

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

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

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.

1.2.5 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;
```

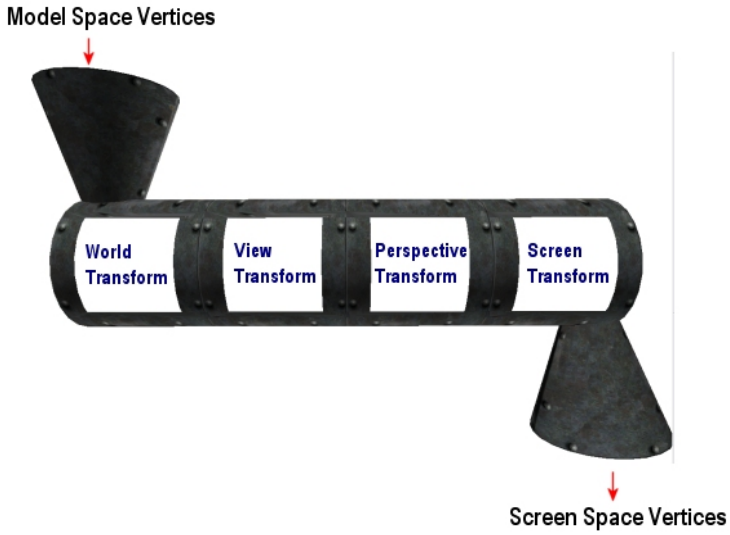
The Y value is projected into screen coordinates using the same approach but with one exception. In projection space (as with model space, world space and view space) the Y axis is positive running up and negative running down. In screen space (as we discussed earlier in this lesson) the Y axis would 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.

1.2.6 Draw Primitive Pseudo-code



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.

1.3 3D Mathematics Primer

1.3.1 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):

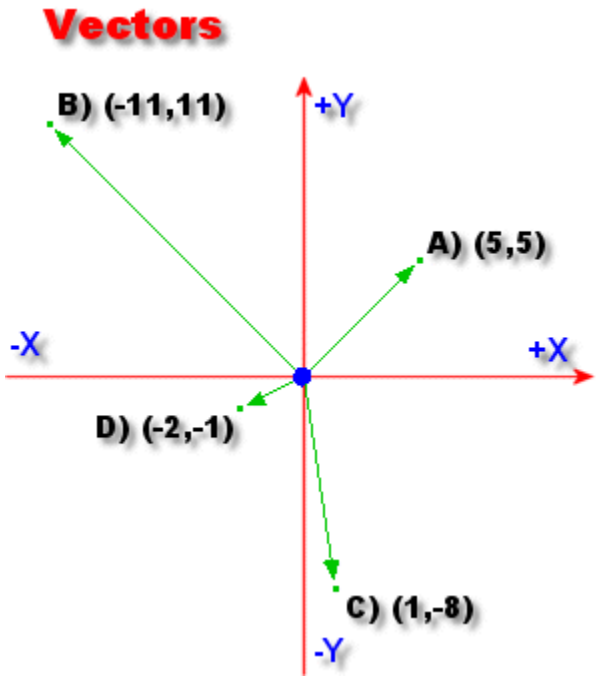


Figure 1.24

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 in Fig 1.24).

$$2D\,VectorMagnitude = \sqrt{X^2 + Y^2}$$

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:

$$3D\,VectorMagnitude = \sqrt{X^2 + Y^2 + Z^2}$$

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 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).

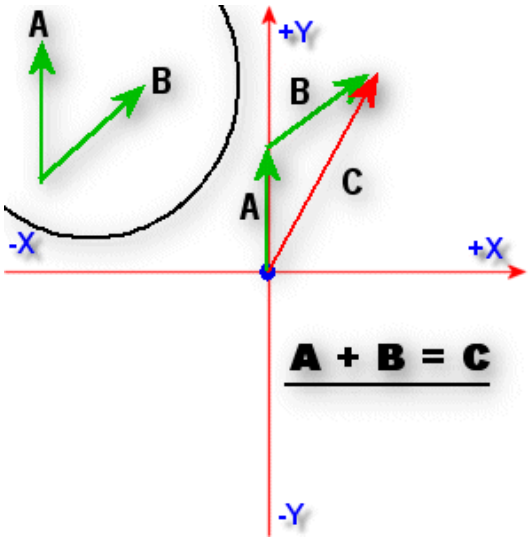


Figure 1.25

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:

C = A – B

OR (the negated version)

C = A + (-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.

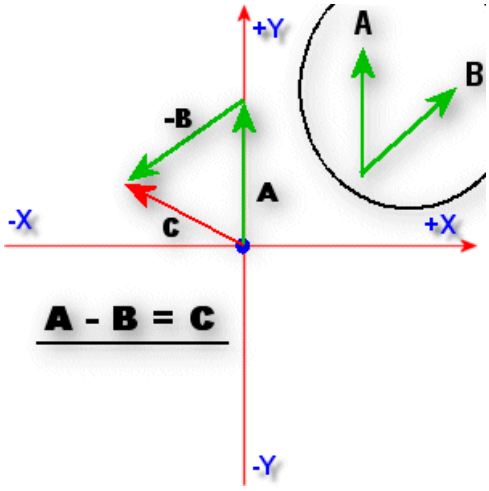


Figure 1.26

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).

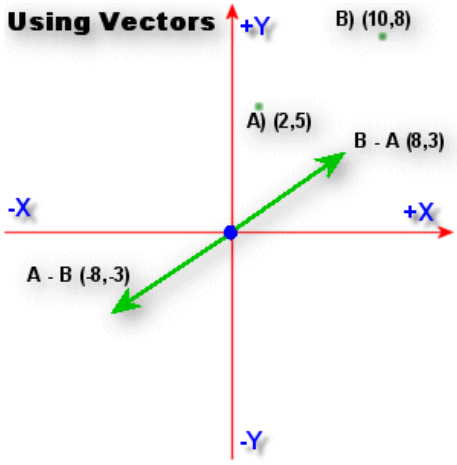


Figure 1.27

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

$$\text{MagnitudeB-A} = \sqrt{8^2 + 3^2}$$
$$= \sqrt{64 + 9}$$
$$= 8.544$$

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.

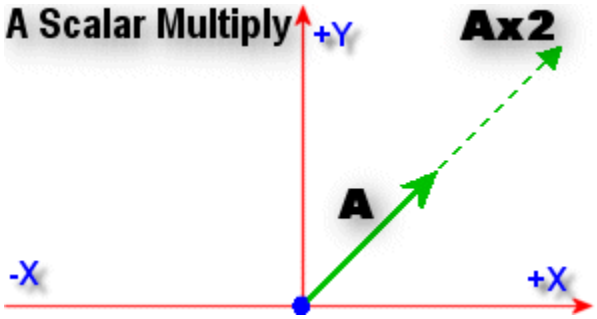


Figure 1.28

Vector Normalization

A special type of vector that is incredibly useful in 3D graphics programming is the **unit vector**. A unit vector is a vector with a magnitude of 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 length of the vector.

VectorC = (8,3,10)

$$|C| = \sqrt{8^2 + 3^2 + 10^2}$$

|C| = 13.152946

$$\text{UnitVectorC} = \left[\frac{C.x}{|C|} \quad \frac{C.y}{|C|} \quad \frac{C.z}{|C|} \right]$$

$$\text{UnitVectorC} = \left[\frac{8}{13.152946} \quad \frac{3}{13.152946} \quad \frac{10}{13.152946} \right]$$

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:

UnitVectorC = (0.60822, 0.2280, 0.7602859)

$$|\text{UnitVectorC}| = \sqrt{0.60822^2 + 0.2280^2 + 0.7602859^2}$$

|UnitVectorC| = 0.9999...

This is about 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:

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

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:

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

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:

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

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:

$$|Movement| = \sqrt{70.71068^2 + 0^2 + 70.71068^2}$$
$$|Movement| = 100$$

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.

Vector Cross Product

The ***cross product*** operation between two vectors results in a third vector perpendicular to the two input vectors. The '×' symbol is used to represent a cross product between two vectors.

Input vectors

```
A = (0, 1, 0)
B = (1, 0, 0)
```

Cross product calculation

```
C = A×B = ((A.y*B.z) - (A.z*B.y), (A.z*B.x) - (A.x*B.z), (A.x*B.y) - (A.y*B.x))
C = ((1*0) - (0*1), (0*1) - (0*0), (0*0) - (1*1))
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.

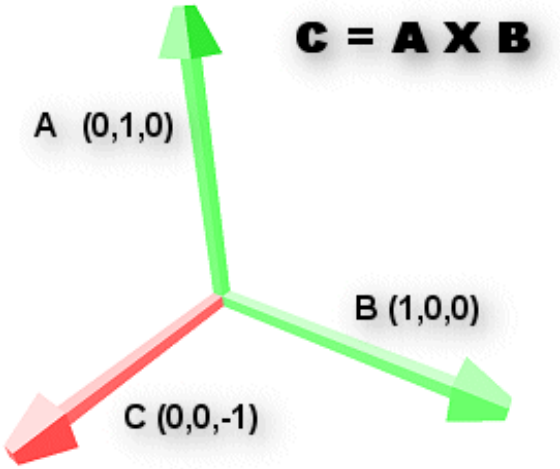


Figure 1.29

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×A instead of A×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.

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;
}
```

Vector Normalization

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

In Fig 1.30 we see a triangular polygon consisting of three vectors (v0, v1, v2). If we were to subtract v0 from v1 the result would be a vector which describes *Edge 1*. If we do the same again, this time subtracting v0 from v2 we get *Edge 3*. The cross product of these two edges yields a vector which, after normalization, is the polygon normal:

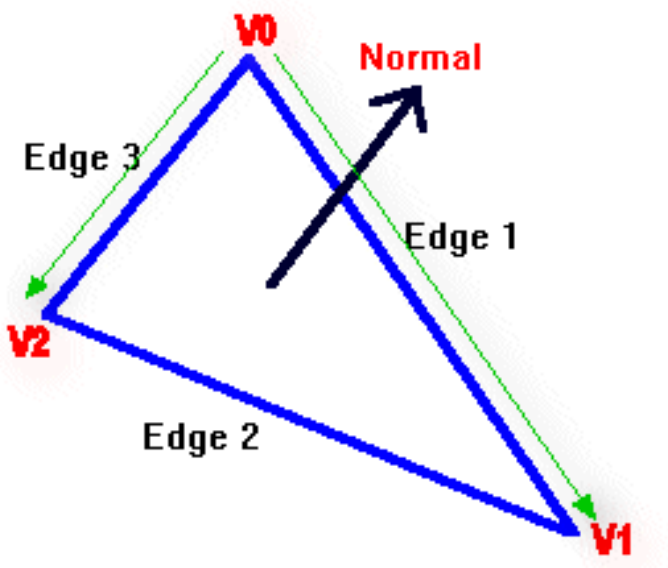


Figure 1.30

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, Edge3, Normal;

    Edge1 = SubtractVector3D(Polygon.Vertices[1], Polygon.Vertices[0]);
    Edge3 = SubtractVector3D(Polygon.Vertices[2], Polygon.Vertices[0]);

    Normal = VectorCrossProduct(Edge1, Edge3);
    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.

Vector 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:

A•B = (A.x * B.x) + (A.y * B.y) + (A.z * 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:

A•B = cos(θ)|A||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(θ) = A•B / |A||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(θ) = A•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 now 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 them.

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.

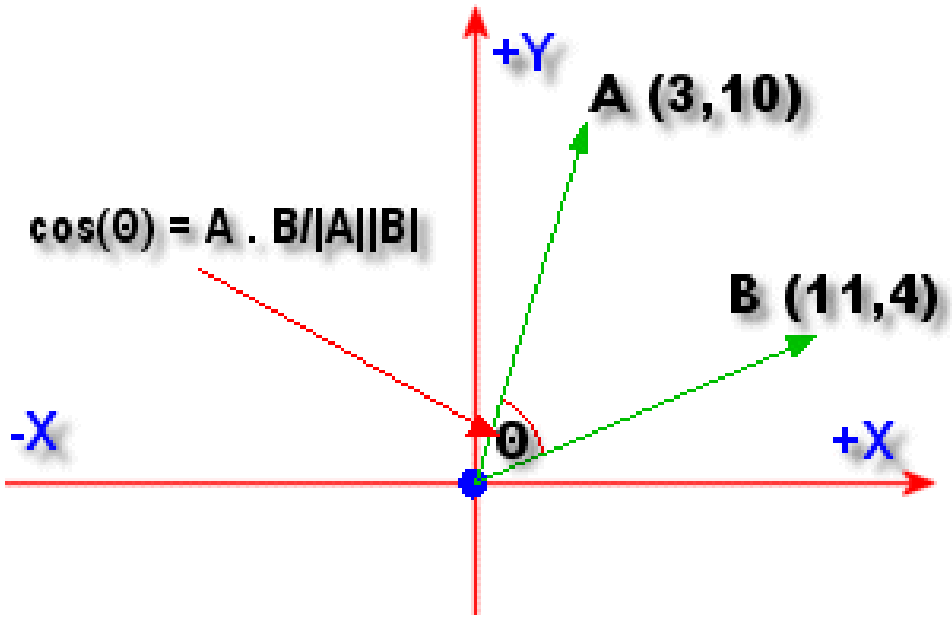


Figure 1.31

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

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

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

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

$\theta = \text{acos}(0.5973782)$

$\theta = 0.93056 \text{ radians}$

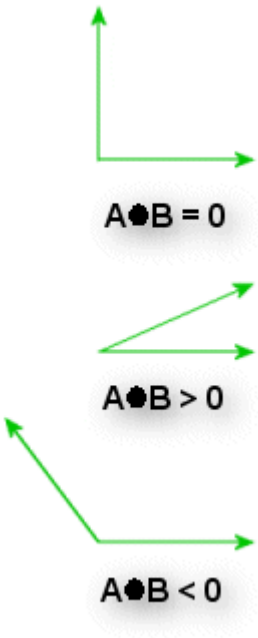
$\theta = 53.31 \text{ degrees}$

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



This diagram on the left outlines the different relationships between two vectors and the result returned by the dot product operation.

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.

Remember that both of the source vectors are moved to the origin during the dot product so that their tails share a single point.

1.3.2 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 shows 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 need to be rendered.

Fig 1.32 shows an infinite plane in the 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, the plane normal is also a unit length vector. It describes the orientation of the plane.

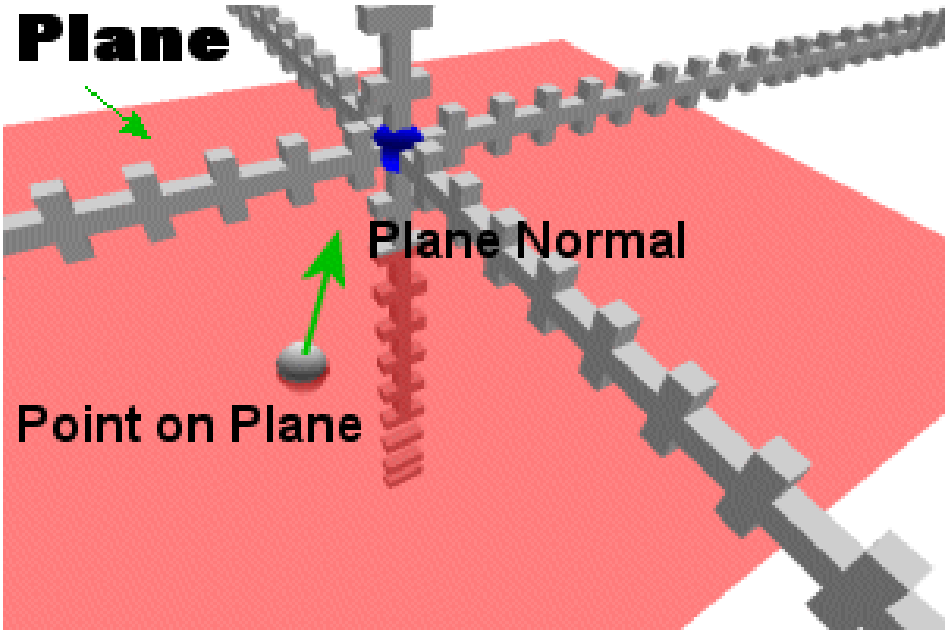


Figure 1.32

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 Fig 1.33). It can be determined by tracing a line from the origin to the plane in the direction of the plane normal (Fig 1.33).

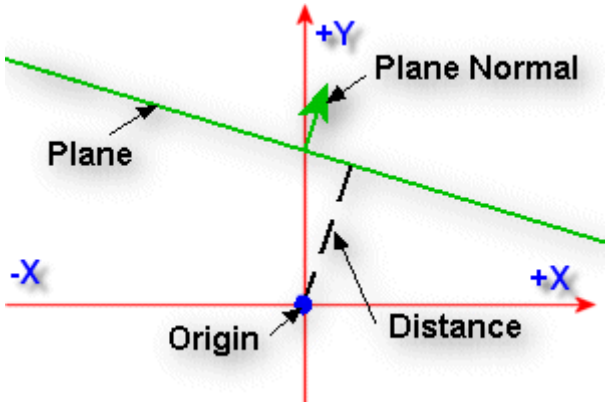


Figure 1.33

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 two 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.

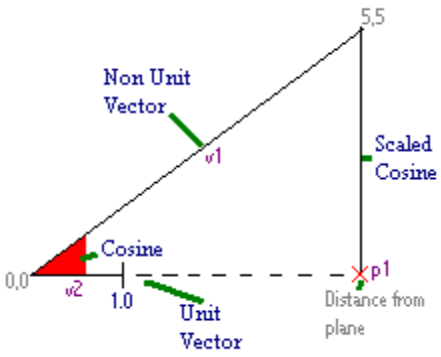


Figure 1.34

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.

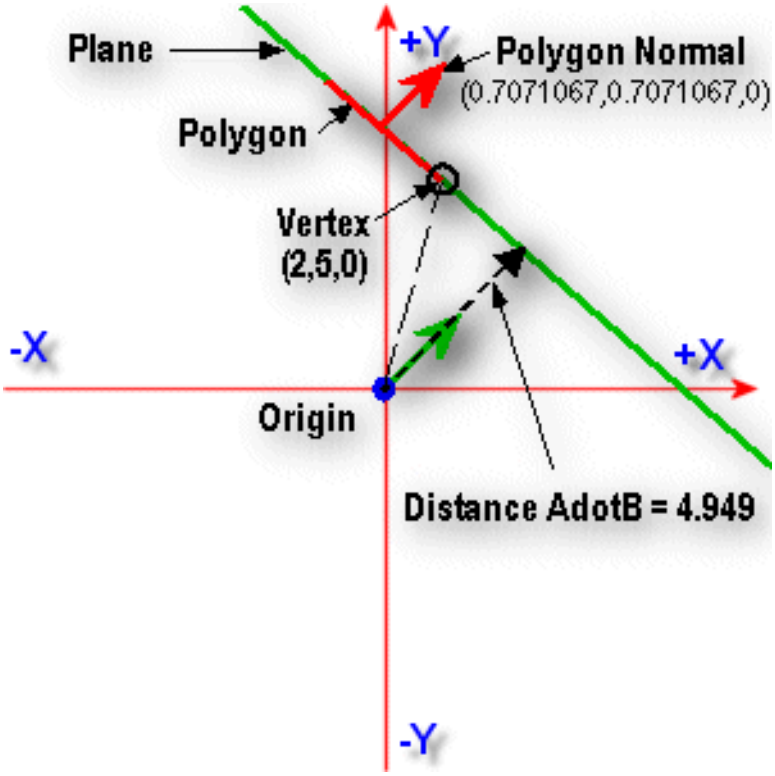


Figure 1.35

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
    // polygon's 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).

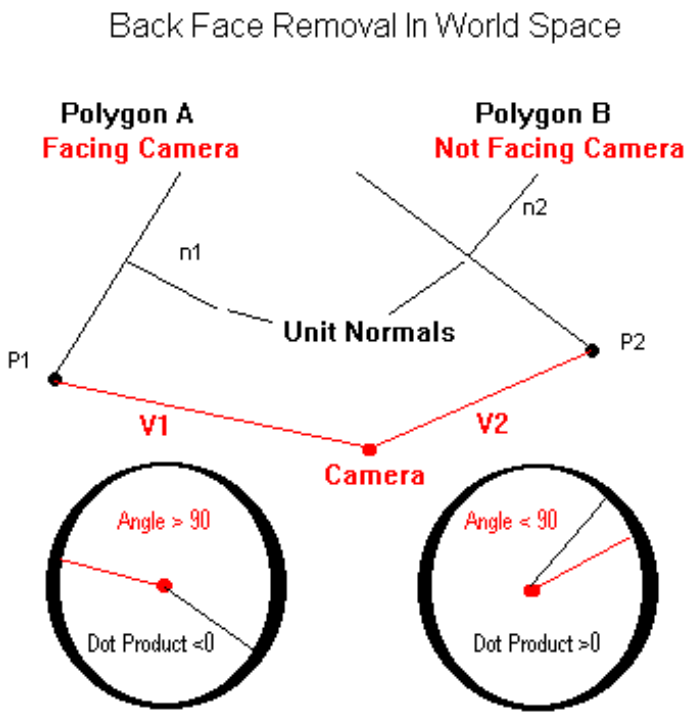


Figure 1.36

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 above 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 • 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 • V2 create an angle smaller than 90 degrees and the result is a positive number.

1.3.3 Matrices

Matrices are a fundamental mathematical concept in 3D graphics programming. Not only do they play a central role in the DirectX transformation pipeline, but they will be used in many other places outside the vertex pipeline as well. Throughout this course series we will encounter matrices for managing all types of transformation requirements and it is very important that you understand what matrices are and how they are used, so make sure careful attention is paid as you work your way through this section.

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 the 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. For example: 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 1x3.

A matrix having ***m*** rows and ***n*** columns is referred to as an ***m x n*** matrix (***order m x 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: a letter that describes the matrix, the first number indicating the row, and the second number indicating the column:

$$\begin{bmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{bmatrix}$$

> **Note:** In some textbooks and in some code implementations the labeling scheme is zero based for both rows and columns. This means that m11 would be referred to as m00 and m32 would be m21. If you are converting code from source that uses the m00 based convention, you will need to add '+1' to each label: m00 = m(0+1)(0+1) = m11.

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)

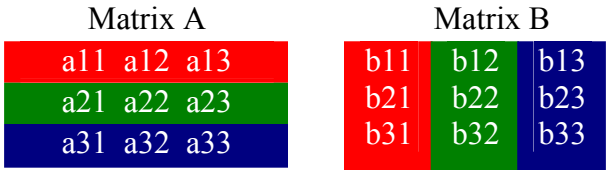
Above you see that when we line up the four matrix dimensions we get: 3x3 3x6. The inner dimensions of both matrices do indeed match and these matrices can be multiplied. The following matrices could not be multiplied together:

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

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

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:



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:

$$\boldsymbol{M} = \begin{bmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{bmatrix}$$

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 (1st row) in Matrix ***A*** with Vector 1 (1st 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.

$$\begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{bmatrix} \times \begin{bmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{bmatrix} = \begin{bmatrix} aRow1 \bullet bCol1 & aRow1 \bullet bCol2 & aRow1 \bullet bCol3 \\ aRow2 \bullet bCol1 & aRow2 \bullet bCol2 & aRow2 \bullet bCol3 \\ aRow3 \bullet bCol1 & aRow3 \bullet bCol2 & aRow3 \bullet bCol3 \end{bmatrix}$$

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 • bCol1

-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

$$\begin{bmatrix} x & y & z \end{bmatrix} \times \begin{bmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{bmatrix} = \begin{bmatrix} aRow1 \bullet bCol1 & aRow1 \bullet bCol2 & aRow1 \bullet bCol3 \end{bmatrix} = \begin{bmatrix} X & Y & Z \end{bmatrix}$$

Vector <x, y, z> is *transformed* into vector <X, Y, Z> 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, translate 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 <x, y> that we want to rotate by an arbitrary angle θ around the origin to get a transformed 2D vector <X, Y>. 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

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

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

$$\boldsymbol{V} = \begin{bmatrix} x & y \end{bmatrix}$$

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.

$$\boldsymbol{C} = [\mathrm{X} \;\; \mathrm{Y}]$$

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.

$$\mathrm{M} = \begin{bmatrix} m11 & m12 \\ m21 & m22 \end{bmatrix}$$

Then V * M = C using the above matrices:

$$\underset{\boldsymbol{V}}{\begin{bmatrix} x & y \end{bmatrix}} \times \underset{\boldsymbol{M}}{\begin{bmatrix} m11 & m12 \\ m21 & m22 \end{bmatrix}} = \underset{\boldsymbol{C}}{\begin{bmatrix} X = V \bullet mCol1 & Y = V \bullet mCol2 \end{bmatrix}}$$

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.

```
X = x * m11 + y * m21
Y = x * m12 + y * m22
```

Now look again at our rotation formula:

```
X = x * cos(θ) – y * sin(θ)
Y = x * sin(θ) + y * cos(θ)
```

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

```
Y = x * m12     + y * m22    // Matrix Calculation of Y
Y = x * sin(θ) + y * cos(θ) // Rotation Formula for Y
```

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

$$M=\begin{bmatrix} m11 & \sin(\theta) \\ m21 & \cos(\theta) \end{bmatrix}$$

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

```
X = x * m11 + y * m21
```

Compare this to our rotation formula of:

```
X = x * cos(θ) – y * sin(θ)
```

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

```
X = x * cos(θ) + y * -sin(θ)
```

Thus:

```
X = x * m11    + y * m21          // Matrix Calculation of X
X = x * cos(θ) + y * -sin(θ)     // Rotation Formula for X
```

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):

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

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.

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

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 3rd 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 did not need Z to change during the previous rotation transformation example that we could simply fill the 3rd 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:

```
Z = x * 0 + y * 0 + z * 1 = z ∴
Z = x * 0 + y * 0 + 10 * 1 = 10
```

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***:

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

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:

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

If you multiplied all of the vertices of a mesh by the above matrix, the object would become 10 times 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 as it significantly reduces the number of operations required to transform a vertex.

For example, imagine that we had a 10,000 vertex model that needed to be taken from model space to projection space. If we assume that we might apply a number of different transformations (e.g. local rotations and scaling in model space, perhaps translation in world space, our inverse translations and rotations in camera space, etc.) to these vertices along the way, it becomes clear that by concatenating all of these transformations into a single matrix, we reduce our mathematical operations down to a fixed cost. Once the concatenated matrix is created, all 10,000 vertices need only be multiplied by this one matrix to complete all of the transformations described. Contrast this with the manual transformation approach we studied earlier in the lesson and you will find that this is a major savings in terms of total number of multiplications and additions.

While the matrix concatenation step itself includes the overhead of matrix setup and potentially even some number of matrix/matrix multiplications, this is a one-time setup cost that takes place before the vertices are introduced to be transformed. Under most circumstances, even including this cost will still result in a massive gain in efficiency. However, it is also important to note that while it can theoretically be faster to do a one-off transformation (like a translation that requires only a few additions) directly to vertices, a matrix based system, which uses concatenation when multiple transforms are needed, provides us the opportunity to reduce our costs in the overall sense as well as keep our code clean and free of special cases that branch based on individual transform requirements. Matrices generally result in an overall net gain in efficiency in the system, even if there are some individual cases that may arise where an on-the-spot transform would be quicker. Of course, these can still be done in software to individual 3D points (like a world translation of a player position vector for example), but in the vertex transformation pipeline we are focused on in this lesson, matrices are the key to efficient operations (and not surprisingly, are required when using the DirectX Graphics transformation pipeline as we will see in Lesson Two).

Matrix multiplication is not **commutative** (A * B) != (B * A) and thus, multiplication order is significant. This should sound familiar since earlier in this chapter we saw the effects of rotating vertices before translating them and the very different results 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 followed with a rotation, the vertices were rotated around the 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.

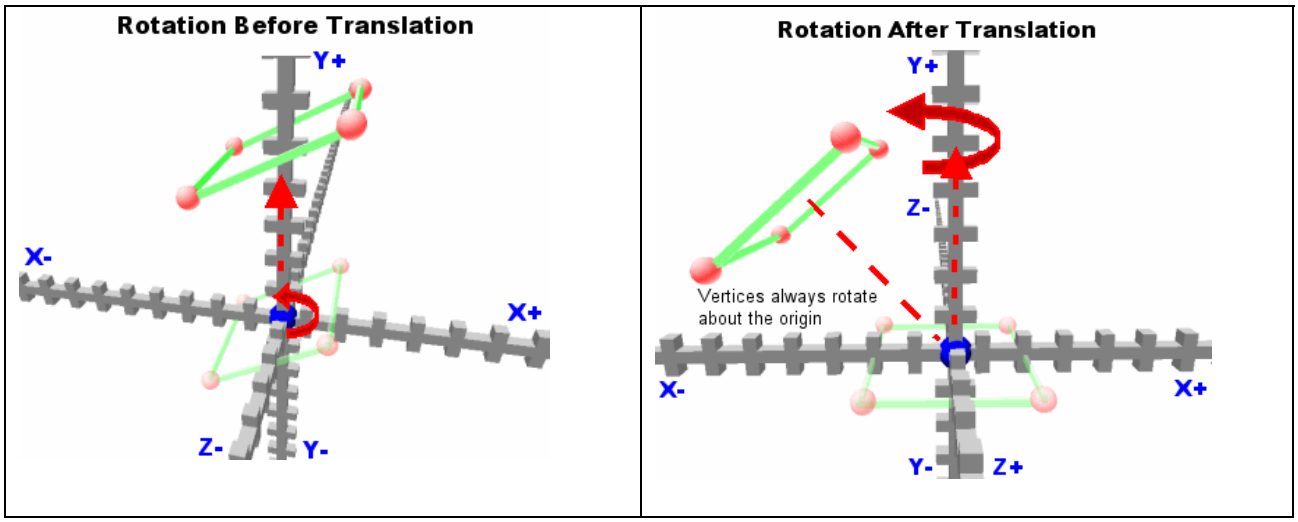


Figure 1.37

The Translation Problem

We know that a vector/matrix multiplication is essentially a series of row/column dot products. This of course means that the input vertex components are multiplied by the matrix components during the process. But what if we wanted to build a matrix that always produced an X component in the output vector with a value of 5 regardless of the value originally contained in the input vector?

$$\begin{bmatrix} x & y & z \end{bmatrix} \times \begin{bmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{bmatrix} = \begin{bmatrix} X & Y & Z \end{bmatrix}$$

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:

$$X = x*m11 + y*m21 + z*m31$$

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 coordinates (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 appear that the only solution would be to handle the other transformations first using a 3x3 matrix and then translate that result separately, like so:

V' = **MV** + **T**

Where V and V' represent the input and output vectors respectively, M is a 3x3 rotation/scaling matrix, and 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 ***m11***, ***m21*** or ***m31*** for any value that would simply add 50 to the x value for example.

Of course, there is a solution. It involves changing our perspective a bit and introducing a new dimension to the equation.

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:

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

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:

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

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

$$\begin{bmatrix} x/1 = x & y/1 = y & z/1 = z & w = 1 \end{bmatrix}$$

This type of vector is known as a ***homogeneous coordinate***. The operation of dividing ***x***, ***y*** and ***z*** by ***w*** 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 project a 3D vector ***V***:

```
V = (x , y , z)
```

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

```
V = (x/z , y/z , z/z)
```

This resulting vector ***R*** looks like:

```
R = (X , 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.

> **Note:** A vector of dimension ***N*** can be projected back into a space of dimension ***N-1*** by dividing its components by the **Nth** component of that vector.

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 will work with 4D coordinates where the ***w*** component does not equal 1, and we will discuss those cases later in the course.

So 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***.

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

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:

$$V\begin{bmatrix} x & y & z & 1 \end{bmatrix} \times M \begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix} = R\begin{bmatrix} X & Y & Z & 1 \end{bmatrix}$$

To calculate element X in output vector R:

$$X = Vx*m11 + Vy*m21 + Vz*m31 + 1*m41$$

We know that ***w=1*** so the last portion of the calculation for ***X*** will always be "***+ 1*m41***". In other words, whatever value we put into element ***m41*** will be used in 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 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 to or subtract from the values contained in the input vector.

With this knowledge, we can now create a matrix that would translate a vector ***TX*** along the X axis, ***TY*** along the Y 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).***

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

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)
Rz = 200 * 0 + 70 * 0 + 500 * 0 + 1 * 1 = 1* 1 =***1*** (Rw=1)

$$\therefore R = \begin{bmatrix} 400 & 70 & 450 & 1 \end{bmatrix}$$

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 (such as he one shown above) *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 therefore always adding the 4th row of a column to the result. 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 4th 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 4th 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.

1.4 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. Lab Project 1.1 will be using D3DX 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 the D3DX library for many other important tasks. This section will examine some of the D3DX data types and functions that we will use on a frequent basis.

1.4.1 Data Types

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 3rd row and the 2nd 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:

```
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 );
```

1.4.2 Matrix and Transformation Functions

D3DXMatrixMultiply

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.

D3DXMatrixRotation{XYZ}

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:

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

D3DXMatrixTranslation

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.

D3DXMatrixRotationYawPitchRoll

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.

D3DXVec3Transform{…}

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.

1.4.3. Vector Functions

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

```
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

```
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 );
```

Magnitude

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

Returns the length of the passed vector.

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

Normalization

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

This function 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, thus normalizing vector A and storing the result back in vector A.

1.5 The Transformation Pipeline II

In our first Lab Project, 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.

1.5.1 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:

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

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.

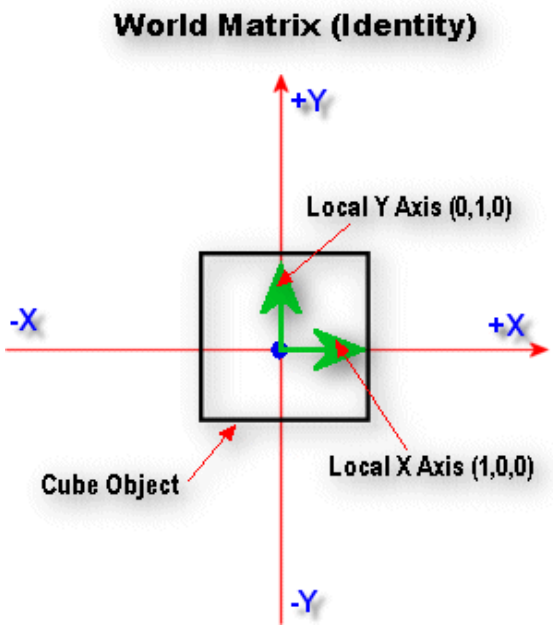


Figure 1.38

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).

$$\underset{\textbf{Identity}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} X \underset{\textbf{45 degree Z rotation}}{\begin{bmatrix} \cos(45) & \sin(45) & 0 & 0 \\ -\sin(45) & \cos(45) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} =$$

$$\underset{\textbf{Identity}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} X \underset{\textbf{45 degree Z rotation}}{\begin{bmatrix} 0.707106 & 0.707106 & 0 & 0 \\ -0.707106 & 0.707106 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} = \underset{\textbf{New Rotated World Matrix}}{\begin{bmatrix} 0.707106 & 0.707106 & 0 & 0 \\ -0.707106 & 0.707106 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \begin{matrix} \textit{LocalXAxis} \\ \textit{LocalYAxis} \\ \textit{LocalZAxis} \\ \textit{Position} \end{matrix}$$

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

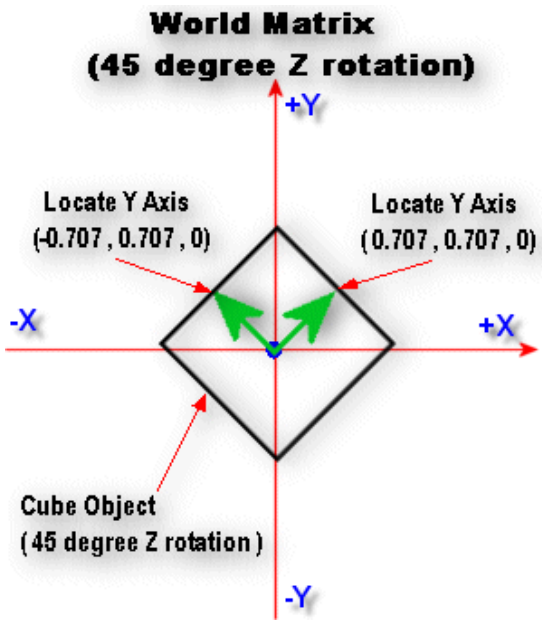


Figure 1.39

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 3rd 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 3rd row of the matrix to do this:

```
// move spaceship forward
void MoveForward( float distance )
{
   D3DXVECTOR 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 3rd 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 4th 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)
{
   D3DXVECTOR 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 += LookVector.x * distance;
   SpaceShip.WorldMatrix._42 += LookVector.y * distance;
   SpaceShip.WorldMatrix._43 += LookVector.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:

WorldSpaceVertex = ModelSpaceVertex * WorldMatrix

1.5.2 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 Vector
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:

View Matrix

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

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 the following:

```
class Camera
{
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.

1.5.3 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:

$$2D\,\mathrm{Pr}\,ojectionPo\,\mathrm{int}\,X = \frac{ViewSpaceX}{ViewSpaceZ}$$

$$2D\,\mathrm{Pr}\,ojectionPo\,\mathrm{int}\,Y = \frac{ViewSpaceY}{ViewSpaceZ}$$

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 looked 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 4x4 matrix and that it will output a 1x4 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

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

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

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

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

$$\boldsymbol{W = x*0 + y*0 + z*1 + 1*0 = z}$$

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:

$$2D\,\mathrm{Pr}\,ojectionSpacex = \frac{ViewSpace.x}{ViewSpace.z}$$

$$2D\,\mathrm{Pr}\,ojectionSpacey = \frac{ViewSpace.y}{ViewSpace.z}$$

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:

ViewSpaceVector = (x , y , z , 1)
HomogneousClipSpaceVector = (x , y , z , z)

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:

$$2D\,\mathrm{Pr}\,ojectionSpacex = \frac{HomogenousClipSpace.x}{HomogenousClipSpace.w}$$

$$2D\,\mathrm{Pr}\,ojectionSpacey = \frac{HomogenousClipSpace.y}{HomogeneousClipSpace.w}$$

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:

P(X , Y , Z=depth buffer value , W=z)

Ignore the depth buffer value for now since 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 Field of View

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*1 = x*** and ***y*1=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.

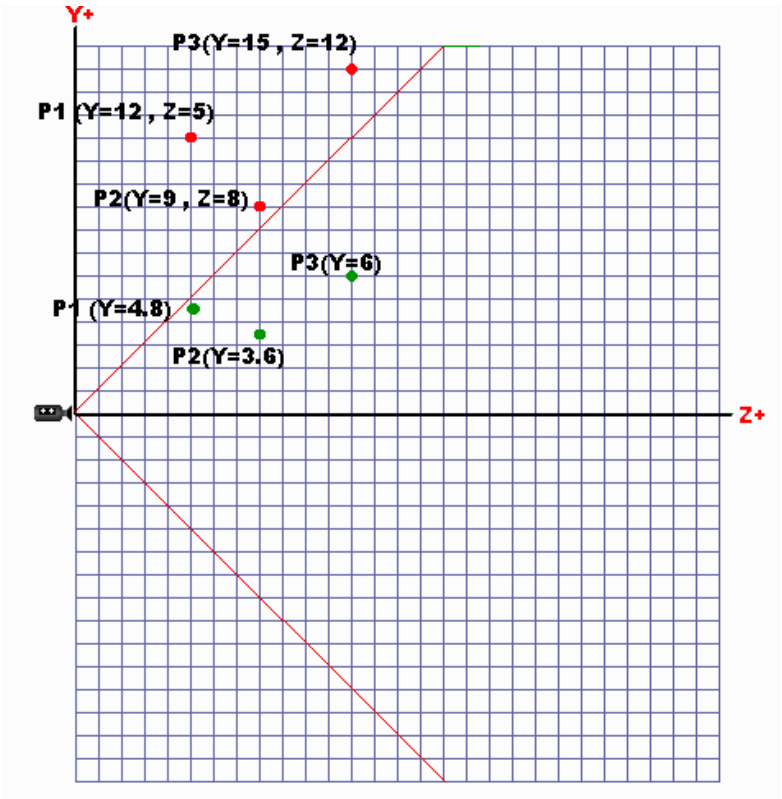


Figure 1.40

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:

$$-W < X < W$$
$$-W < Y < W$$

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:

$$\mathbf{Y = x*0 + y*0.4 + z*0 + w*0}$$

This simplifies to:

$$\mathbf{Y=y*0.4}.$$

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

$$\mathbf{X=x*0.4 + y*0 + z*0 + w*0}$$

This simplifies to:

$$\mathbf{X=x*0.4}$$

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.

The Co-Tangent

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.

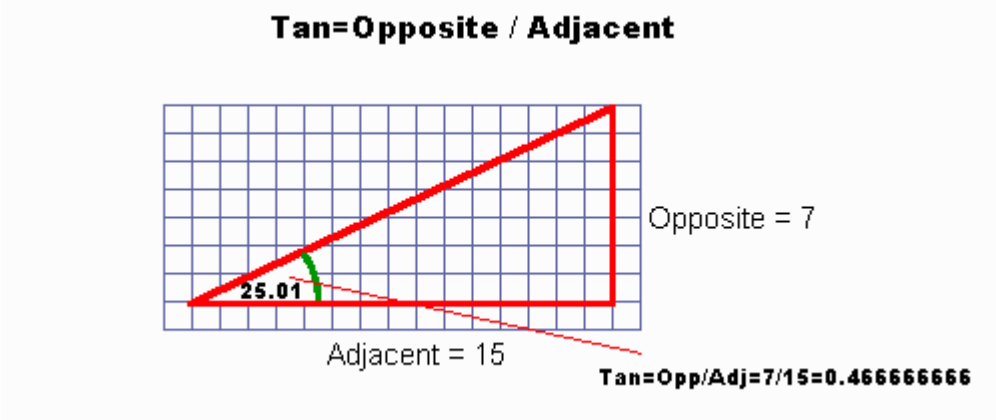


Figure 1.41

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:

Opposite= Tan (25.01689345) * 15 = 0.466666666*15 = 7 (approx, actually 6.999999999999)

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. Tan^{-1})***. First we calculate the tangent:

Tan = Opposite/Adjacent = 7/15 = 0.46666666

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.

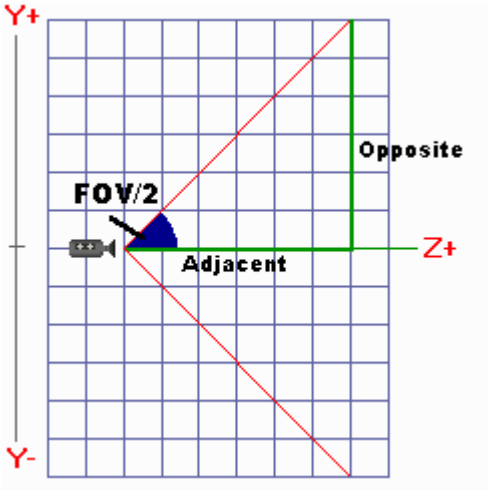


Figure 1.42

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.

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

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:

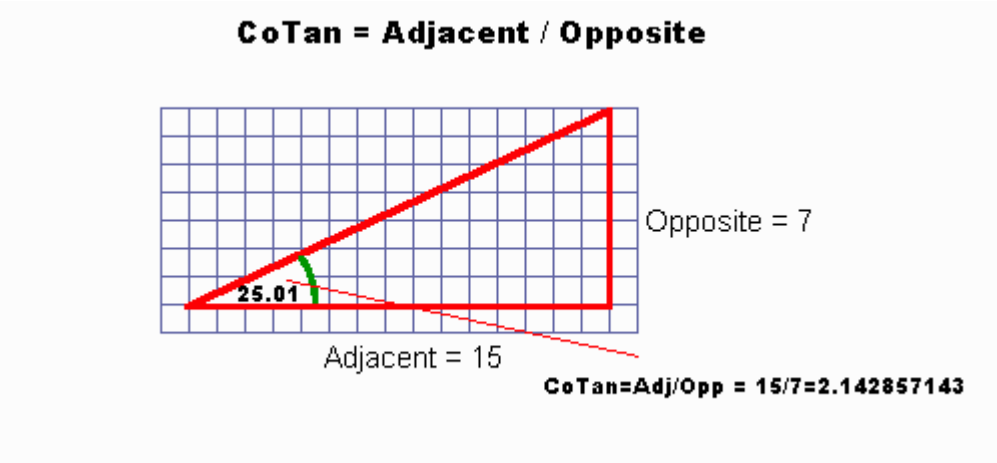


Figure 1.43

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:

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

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.

$\theta = 1.047197551$ radians (*60 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 4th column simply copies the input ***z*** value into ***W*** of the output vector for the divide by w. The 3rd 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 (2x2 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).

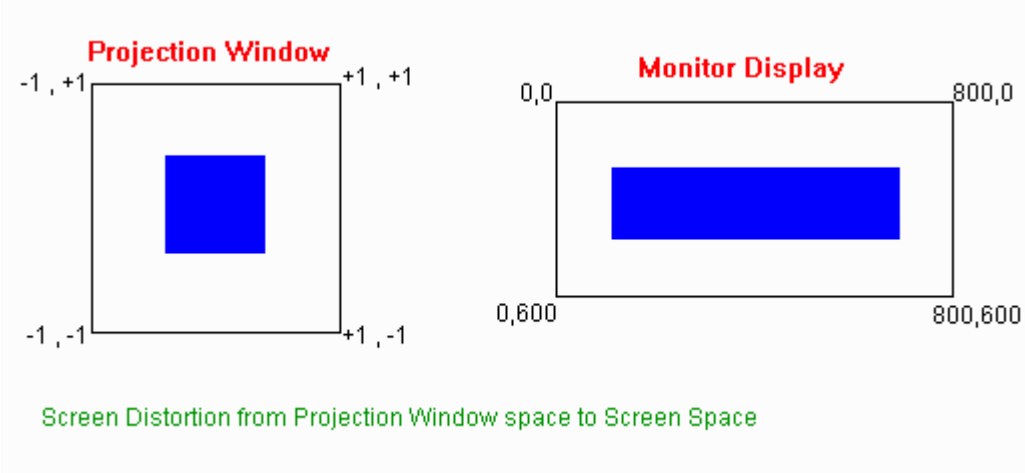


Figure 1.44

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).

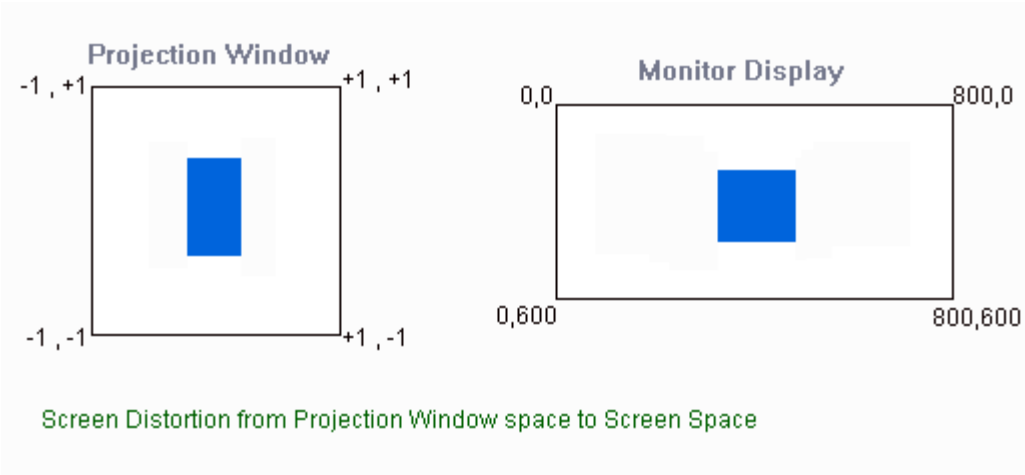


Figure 1.45

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:

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

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)`

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

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

> **Note:** In some source implementations you might see Aspect Ratio calculated as $\frac{Height}{Width} = 0.77777777$. 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.

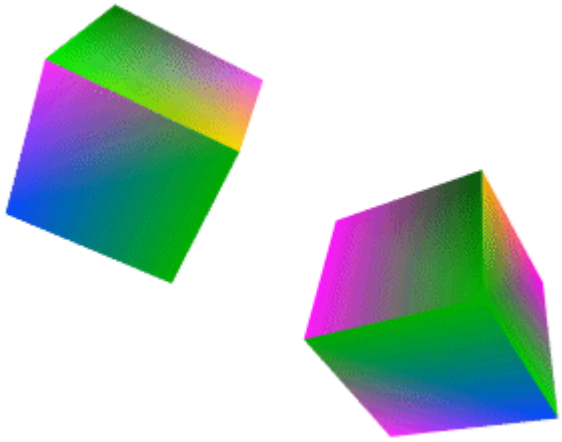
The two parameters at the end of the parameter list in the above function (zn and zf) are used to configure the 3rd column of the projection matrix to scale the Z value of the input vector into a range that can be used by the DirectX Graphics depth buffer. We will not be using a depth buffer in our first 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 reviewed the core mathematics techniques that will be invaluable as we progress through the course. At this point it is recommended that, if you have not already, you should enroll in the Game Mathematics course to continue to reinforce this mathematics knowledge as well as learn new concepts. The two courses can be taken in parallel since the foundation math you will need for this course has been covered.

Chapter Two:

DirectX Graphics Fundamentals



Introduction

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.

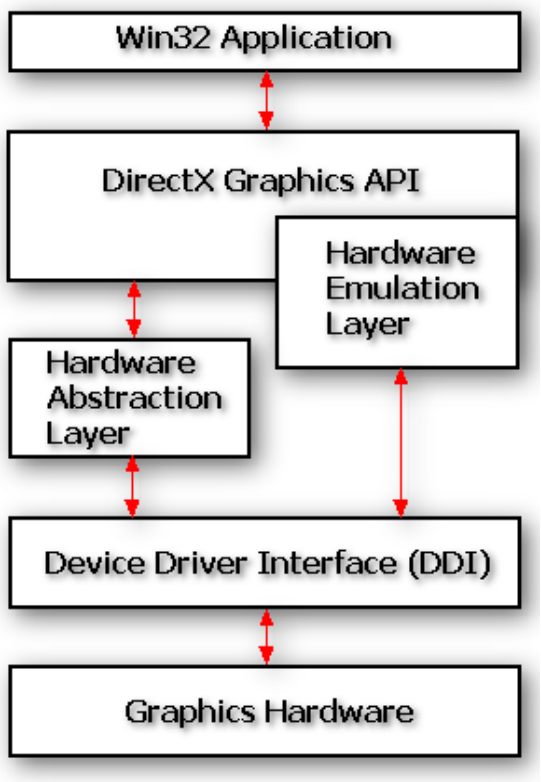
Most hardware manufacturers package a DirectX Graphics compliant device driver called the **Hardware Abstraction Layer (HAL)**. When a HAL is found on the current system, it indicates that the graphics hardware has hardware accelerated support for at least some DirectX Graphics functionality. DirectX 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 over 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.

Before we begin discussing DirectX Graphics initialization details, we will first take a short detour and discuss the Component Object Model (COM). This will set the stage for discussions later in the lesson that address proper DirectX Graphics initialization.

2.1 The Component Object Model (COM)

DirectX is implemented using an object oriented programming model called the *Component Object Model* (or COM for short). Although programming with COM and creating your own COM objects can be a complex task, we will not have to deal with such things in this course. We only have to learn how to use COM objects with our application, which is quite simple since it is similar to using C++ objects. Although the COM programming we will need to do will be very straightforward, we will still need to have an understanding of the underlying COM technology and how it works in order to interact with DirectX in a safe manner. Therefore, we will use this opportunity to take a high-level look at COM.

COM objects are code modules, more correctly called software components, most often although not always implemented as dynamic link libraries that can be installed on an end users' computer system. A COM object, like a conventional DLL, exposes functions that applications can call to request functionality. Therefore COM objects offer a nice way to extend the functionality of the operating system. When COM objects are installed they can be used by all applications that wish to use them – as is the case with DirectX. When DirectX is installed, a number of COM objects are installed and registered with the operating system, making the functionality provided by those COM objects available to any application running on the machine. They can be treated as "black boxes" from the application perspective and have their functions called to perform supported tasks.

The goal of COM is to provide a programming model where software components can be created and installed on systems with ease. Those same components need to be easily upgraded in the future without breaking backwards compatibility with software that used the older version of the object. COM objects are also completely portable.

The nice thing about COM objects is that the programming language the software component was written in does not limit its use. A visual basic application for example could access functions in a COM object that was written in C++. This means that COM can be used in a variety of software development environments. This is all achieved by exposing an object's functionality through an ***interface.***

2.1.1 COM Interfaces

A COM object's code is completely encapsulated and you will have no direct access to its data. You can think of this as a C++ class where all member variables are private. The only way you can communicate with a COM object is through one or more ***interfaces*** that it supports. An interface is a collection of functions that share a common purpose. You can think of an interface as being like an abstract C++ class with no code implementations of the functions, but with a class signature. Interfaces provide a way for objects to communicate with other objects, and also provide a means for those objects to expose their functionality to other objects seeking it. Below we see an example of an interface that has two member functions. Interface names are always preceded with an 'I' as in IMyIterface or IMyInterface2.

```
typedef struct Interface

Interface IMyInterface
{
    HRESULT  MyFunction1 ( int , int );
    HRESULT  MyFunction2 (bool);
};
```

This example interface has no code, but it does tell us that this interface supports two functions: MyFunction1 and MyFunction2. If any COM object supports this interface then it must support both of these functions. Interfaces can be inherited amongst objects and it is likely that one COM object may support many interfaces. Many COM objects may even support the same interface.

The developer of the COM object will typically provide you with header files that can be linked into your application. These headers define all of the interfaces supported by this object. If you open up the d3d9.h header file that ships with the DirectX9 SDK, you will see the definitions of the interfaces supported by the DirectX Graphics objects. This means that our applications will recognize the interface names (such as IMyInterface in the above example).

Below we see a code snippet from the DirectX Graphics header file d3d9.h which shows all of the interfaces supported by the main DirectX Graphics COM objects. These are the interfaces we will be dealing with throughout this course. By including this header file in our CPP files we can, for the most part, call the functions of any one of these interfaces just like they were C++ class member functions.

Except from the d3d9.h header file

```
interface IDirect3D9;
interface IDirect3DDevice9;
interface IDirect3DStateBlock9;
interface IDirect3DResource9;
interface IDirect3DVDecl9;
interface IDirect3DVShader9;
interface IDirect3DPShader9;
interface IDirect3DBaseTexture9;
interface IDirect3DTexture9;
interface IDirect3DVolumeTexture9;
interface IDirect3DCubeTexture9;
interface IDirect3DVertexBuffer9;
interface IDirect3DIndexBuffer9;
interface IDirect3DSurface9;
interface IDirect3DVolume9;
interface IDirect3DSwapChain9;
```

Once we have an interface to a COM object, we can call its member functions like we would if we were using a C++ class. The following example uses the IDirect3D9 interface to a valid COM object to call one of its member functions (GetAdapterCount). This call finds out how many graphics adapters are on the system on which the application is currently running.

```
IDirect3D9 pd3d;                        // Assume this links to a valid COM object
ULONG Count = pd3d->GetAdpaterCount(); // Call one of the interfaces functions
```

Although we have not covered how to get an interface to a COM object yet, you can see in the above example that once we have an interface, we can call its member functions just like C++ class member functions.

2.1.2 GUIDS

Every interface and COM object has an ID that is unique so that there can be no confusion as to which interface or object your application wants to use. The question then becomes, if you created your own COM object, how could you be sure that the ID you gave your COM object was not already be in use by other COM objects being developed elsewhere in the world?

In order to assign each COM object or interface a unique ID, globally unique identifiers (GUIDs) are used. These GUIDs are 128-bit (16 byte) structures. Although it may at first seem that there is not enough scope in a 128-bit structure for guaranteed uniqueness, there actually is plenty. GUIDs can be created in such a way that they are guaranteed to be unique. They use an extremely high precision time stamp as well as things like the IDs of the specific hardware components installed on the system for which the GUID is being created (the COM object developer's machine for example). GUIDs can be created very easily using a tool that ships with Microsoft Visual C++ called GUIDGEN.exe.

GUIDs are used for both COM objects and interfaces. When a COM object is installed on your computer system, its GUID is entered into the registry. When an application wants to interact with this COM object, it passes in the GUID of the COM object it wants to use and the operating system looks it up in the registry. This will tell the OS which DLL module to load (if it is not already loaded).

The GUID assigned to a COM object is called a class identifier (CLSID) and it is this GUID you must specify when you want to create an instance of a COM object. If the GUIDs of either interfaces or objects are not made publicly available, then applications will not be able to use them. You will usually find the GUIDs of an object or an interface in either the reference manual or appropriate header files that shipped with the component.

A GUID assigned to an interface is called an interface identifier (IID). Whenever you wish to find out if a COM object supports a particular interface, you can pass in the GUID of the interface to validate its support for that object.

GUIDs are typically pretty unfriendly things to work with. And although they are technically structures, they are often expressed as a string of 32 hexadecimal numbers in the format 8-4-4-4-12. Below we see the GUID for a DirectX Graphics COM object interface (the IDirect3D9 interface).

{1DD9E8DA-1c77-4d40-b0cf-98fefdff9512}

Imagine having to type in numbers like that every time you wanted to ask a COM object for a specific interface, or worse, trying to remember all of the GUIDs for the many interfaces DirectX Graphics supports. Fortunately, when a COM object is installed, its GUID is registered and assigned an alias (an

alternative name) which is typically pretty intuitive. It is common practice for the name assigned to the GUID of an interface to be the same as the name of the interface with IID_ preceding the name. So instead of typing in the above GUID to request the support of the IDirect3D9 interface, we could instead use its counterpart: **IID_IDirect3D9.**

When referring directly to a COM object (versus an interface), you need to precede the name of the COM object with CLSID_ .

If we look at the d3d9.h header file, we see the GUIDs listed for each interface. Below we see how the GUID for IDirect3DDevice9 has the alias IID_IDirect3DDevice9 assigned to it. The first parameter to the DEFINE_GUID macro is the IID_ name we wish to assign to the GUID. The numbers that follow are the GUID itself.

```
DEFINE_GUID(IID_IDirect3DDevice9, 0xd0223b96, 0xbf7a, 0x43fd,
            0x92, 0xbd, 0xa4, 0x3b, 0xd, 0x82, 0xb9, 0xeb);
```

Because this is included in our application source files, whenever we wish to specify this interface to work with, we can call it by its intuitive name (IID_IDirect3DDevice9) instead of its GUID.

2.1.3 The IUnknown Interface

All COM interfaces must derive from an interface called IUnknown. This is an interface that provides the core COM object usability. The result is that the first three entries in the vtable of all interfaces are the same three functions.

```
typedef struct Interface

Interface IUnknown
{
   HRESULT   QueryInterface (REFIID *idd , void ** ppbObject );
   ULONG     AddRef ( void );
   ULONG     Release ( void );
};
```

These three functions are available through all COM interfaces and together they provide two very important concepts that fundamentally define how COM works.

Before we look at how these functions work, let us first examine how one might go about creating an instance of a COM object. Imagine for now that you had installed a software component called 'Aeroplane' (a single COM object) and that this object grouped functions into several interfaces depending on there relationship to the plane.

The first thing our application must do before it can work with COM is initialize itself as a COM user with the operating system. We use the following CoInitialize function to initialize the application COM layer. This would typically be called at the start of an application:

```
CoInitialize (NULL);
```

Our application is now ready to invoke the services of any COM object that is installed on the system. When the application closes, we must remember to match this call with a call to CoUninitialize to clean up all resources used by the COM library. This would typically be in your application's ShutDown() function.

```
CoUninitialize ();
```

Continuing our example, imagine that the developers of this COM object decided to group the functionality of the entire Aeroplane object into three interfaces:

- ICockpit
- IPassengers
- ICargo

So this imaginary COM object exposes three interfaces. The ICockpit interface might have functions relating to the actual flying of the plane:

```
Interface ICockpit
{
    HRESULT SetAirSpeed  (ULONG Speed);
    HRESULT SetAltimeter (ULONG Altitude);
    HRESULT UnderCarriage(BOOL Down);
};
```

We could have another interface that configures the passenger compartment:

```
Interface IPassengers
{
    HRESULT SeatBeltsOn    (BOOL Belts);
    HRESULT SmokingAllowed (BOOL Smoking);
};
```

The third interface allows for loading and unloading of cargo:

```
Interface ICargo
{
    HRESULT AddCargo     (ULONG Units);
    HRESULT RemoveCargo  (ULONG Units);
};
```

Note that we are being very abstract here. As you can see, the one object supports three interfaces. Its functions are divided among these three interfaces by relevance to a particular category. Remember that we are not worried about how these functions are implemented inside the COM object itself; we only need access to the interfaces so that we can request that the COM object perform tasks.

Once the COM layer has been initialized in our application by calling CoInitialize, our next task is to create an instance of the object wherever we need it. If we wanted to create an instance of the 'Aeroplane' object, we need to do this by calling the following function:

```
HRESULT hr;
ICockpit *pCP;

Hr = CoCreateInstance(CLSID_Aeroplane, NULL, CLSCTX_INPROC_SERVER,
                      IID_ICockpit, (LPVOID*)&pCP);
```

The first parameter to the above function is the class id (CLSID) of the COM object we wish to create. Remember that it is common practice for COM objects to have a CLSID_ as an alternative to specifying the actual GUID string. This function will search the registry for the passed class ID and if found (in other words, if the 'Aeroplane' software component has been installed on the system), it will retrieve the dll/module that this object is located in and make sure it is loaded for use.

The second parameter can safely be set as NULL for our purposes.

The third parameter specifies that the object will run as part of the application's process space.

The fourth parameter requires some explaining. When the object is created, we will never receive a pointer to the object itself. This violates the rules of COM. Instead, we get a pointer to an interface that is used to communicate with the underlying COM object. As we have discussed, an object may support more than one interface, so when the object is first created, you pass in the interface id (IID_) of the interface you want returned. In the above code we have asked to initially have an ICockpit interface returned; we specified IID_ICockpit and for the final parameter passed in an address of a pointer to an ICockpit interface. This interface will hold a valid pointer to an ICockpit interface if the function is successful. We know that our Aeroplane object supports the ICockpit interface, so this function should be successfully executed.

Now that we have a pointer to the ICockpit interface, we can call the functions of the ICockpit interface to instruct our COM object to perform some bit of work. The following code shows how we could use our newly returned interface to instruct the COM object to set the plane's air speed to 325km, its altitude to 800 meters, and retract its wheels.

```
pCP->SetAirSpeed (325);
pCP->SetAltimeter (800);
pCP->UnderCarriage(FALSE);
```

Now this is all well and good, but we only have the ICockpit interface to hand. We know that the object supports two more interfaces (IPassenger and ICargo) which allow us to instruct the Aeroplane object to perform other tasks that we may wish to use. However, when we created the instance of the Aeroplane object, we only received a pointer to one of its interfaces. We could have requested one of the other interfaces just as easily, but that does not help us understand how to access the other interfaces after the fact. As it happens, the IUnknown interface (from which all COM interfaces derive) has a member function called QueryInterface that will assist us in this task.

```
HRESULT QueryInterface (REFIID *idd , void ** ppbObject );
```

All COM interfaces support this function as a means of querying and obtaining pointers to other interfaces that are supported by the parent COM object. Once we have set up the cockpit parameters, we could request a pointer to the ICargo or IPassenger interfaces by calling the ICockpit::QueryInterface member function:

```
IPassenger *pPassenger;
ICargo *pCargo ;

pCP->QueryInterface (IID_IPassenger , (LPVOID *)&pPassenger);
pCP->QueryInterface (IID_ICargo , (LPVOID*)&pCargo);

// Use our two new interfaces
pPassenger->SeatBeltsOn (TRUE);
pPassenger->SmokingAllowed (FALSE);
pCargo->AddCargo(25);
```

There are some very important concepts to be noted in the previous code snippet. Most importantly, if we have any interface to an object, we can query and obtain a pointer to any other interface the object supports by calling QueryInterface. For example, we initially only have one valid interface (ICockpit) linked to its underlying COM object. By calling ICockpit::QueryInterface we can request another supported interface. If this interface is not supported by the underlying COM object then the function will return a failure message. But if the requested interface is supported, then a pointer to an interface of that type will be returned. In the code we first asked for an IPassenger object, but we could have first asked for the ICargo interface instead. Order is not significant here so long as the interface is supported by the object. Note that we called ICockpit::QueryInterface both times to query for new interfaces. However, once the IPassenger interface was created we could have called IPassenger::QueryInterface instead to query the ICargo interface. The result would have been the same because both of these interfaces are attached to the same object. These concepts demonstrate why the IUnknown interface must serve as the base in all COM interfaces. Without this core functionality, the application would have no way to request additional interfaces supported by a COM object.

So we now know how to create an instance of a COM object and also how to get access to any of its interfaces in order to call their functions. The next important thing that the IUnknown interface provides COM objects is the ability to handle ***lifetime encapsulation***.

2.1.4 Lifetime Encapsulation

One interesting point about COM objects is that the application does not control their creation or destruction. A COM object keeps an internal *reference count* of how many interface requests it has dispatched to an application. This count is set to 1 when the object is first created and the first interface is returned via CoCreateInstance. Every time we call QueryInterface from one of the object's supported interfaces to retrieve a different interface, the internal reference count is incremented.

When we no longer need to use an interface, we call the Interface::Release() method. This is another IUnknown member function inherited by all interfaces. Release() signals to the COM object that we no longer need this interface to be valid and it can therefore be detached from the object. The COM object will decrement its internal reference count by one to indicate that one less interface is now actively linked to the object. An interface pointer should not be used after it has been released, so it is good programming practice to set the pointer to NULL immediately afterwards. This makes it easier to track a bug that might be caused by your application trying to use an interface pointer which was already

released. Once the object's reference count drops to zero, it means that there are no interfaces to this object in use by any application. The object interprets this as indicating that its services are no longer required, and at this point the object destroys itself. This means that our application does not have to release the memory for the COM object since it controls its own lifetime through the tracking of active interfaces.

This is actually just as well, because we have no idea how the memory for the COM object was allocated to begin with. Remember that we did not create it manually and we would hate to mismatch a call to ***free*** to release memory allocated with ***new*** and vice versa. Allowing the object to control its own construction and destruction is an elegant solution. Another vital reason for this reference counting mechanism is to allow for other applications that may be using the same COM object. If our application was allowed to simply destroy the COM object when it was no longer needed, other applications still using it -- unaware of its destruction -- would most likely crash.

Let us have a look at this concept in action. In this next example we create an instance of an Aeroplane object as before and initially request an ICargo interface. This goes to show that we can request any of its supported interfaces to be returned from the CoCreateInstance function. Please note that this code would, in a real situation, check error codes returned from each function, but we have not covered the values returned from COM member functions yet.

```
// global interface pointer
ICockpit *pCockpit;

void ApplicationSetup(){
    // Initialize COM layer for application
    CoInitialize(NULL);

    // Local Interface pointer
    ICargo * pCargo;

    // Create an instance of IAeroplane with an initial ICargo interface
    CoCreateInstance( CLSID_Aeroplane , NULL , CLSCTX_INPROC_SERVER,
                      IID_ICargo , (LPVOID*)& pCargo);

    // We have a valid object and one interface linked to it.
    // The objects internal reference count is 1
    // Now we could use the ICargo interface
    pCargo->AddCargo(35);

    // Now get a pointer to a ICockpit interface that the object
    // also supports and store in global pointer
    pCargo->QueryInterface( IID_ICockpit , (LPVOID*)&pCockpit)

    // Objects reference count is now 2 because it has dispatched 2 interface
    // requests(ICargo and ICockpit) We have no further use for the ICargo
    // interface here so release it.
    pCargo->Release()
    pCargo = NULL;

    // pCargo pointer is no longer valid. the com object has its reference count
    // decremented. the reference count is now 1
}
```

In the above function we initialize the COM layer, create an instance of our Aeroplane COM object, and retrieve a pointer to an ICargo interface. Behind the scenes, the object's internal reference counter will be incremented to 1 because it has dispatched an interface to the application (ICargo). Next we use the ICargo interface member functions to set some states within the object (in this case we set the cargo of the object to 35 units) and then call the QueryInterface function to request the COM object to dispatch an ICockpit interface. At this point the object's internal reference count is incremented again. It now equals 2 because there are currently two interfaces that the object has dispatched to the application that are still active.

We release the pCargo interface because we no longer need it anywhere else. If we did need it somewhere else we could always request the interface again so long as we had a surviving interface from which to call QueryInterface. We could also store it in a global pointer if we needed its lifetime to extend past the scope of the function. At this point the COM object decrements its internal counter from 2 back to 1 and detaches the interface pointer so that it is no longer valid. The pCargo interface pointer should not be used from this point forward.

There is now only one active interface still referencing the object and this brings up an important point. In this example we purposely made the ICockpit pointer a global variable. This means of course, that we can access it and eventually release this interface from its underlying COM object anywhere in the program. If pCockpit was a local pointer we still have to make sure that we Release the interface before the pointer itself goes out of scope. This would cause the object to be destroyed because the reference count would fall to 0.

In the following function you can see that because we still have our global pointer to an ICockpit interface, the COM object remains alive and can be called elsewhere. The reference count of the COM object at this point would be 1.

```
void SomeOtherFunction ()
{
       // The COM object is still alive because the ICockpit interface has not bee released.
       // Because this is a global pointer we can continue to use it else where.
       // The Object's reference count is 1 because the
       // ICockpit interface has not yet been released from the object
       pCockpit->SetAirSpeed(400);
       pCockpit->SetAltimeter(1000);
}
```

Finally, at application shutdown we release the ICockpit interface. This reduces the reference count of the COM object to 0 and causes the COM object to destroy itself and release any resources back to the system. As our final act, we uninitialize the COM layer by calling CoUninitialize.

```
void ApplicationShutdown()
{
    //We should release any interfaces here from their objects.
    //We still have the ICockpit interface attached so we should release it.
    //This will drop the objects reference count from one to zero causing the
    //object to unload itself from memory
    pCockpit->Release();
    pCockpit=NULL;

    //... Other cleanup stuff here
```

```
    // finally uninitialize the COM layer
    CoUninitialize()
}
```

There is one more key function from the IUnknown interface that all interfaces inherit. It is called IUnknown::AddRef() and it allows the application to manually increase the reference count of the COM object.

```
ULONG AddRef (void);
```

Now why would we want to do this? Does this not render the auto-reference counting system useless?

This function is provided because you will often wish to make a copy of a pointer to an interface and it is cleaner to always call Release for every pointer. If we were to make a copy of an interface pointer and not increase the reference count, the outcome could be fatal. The following code shows such an error.

In the following example, assume that the Aeroplane COM object currently has one active interface (ICockpit) and a pointer to that interface is passed into the function. The reference count of the COM object is 1.

```
void BADfunction( ICockpit *pCockpit)   // COM Object counter = 1
{
       ICockpit * pCockpitCopy;

       pCockpitCopy = pCockpit;          // Make copy of pointer
       pCockpit->Release();              // Counter = 0 so object is destroyed
       pCockpit=NULL;

       pCockpitCopy->SetAirSpeed(400);   // ERROR! -- object no longer alive
       pCockpitCopy->Release();
       pCockpitCopy=NULL:
}
```

In the code we made a copy of the pCockpit pointer (into pCockpitCopy) but we forget to call AddRef to increase the reference count of the object. This is dangerous because the reference count is still 1 but we now have two pointers using the object. The problem occurs when we call pCockpit::Release(). This causes the COM object to decrement its internal reference count back to 0 -- causing the object to be destroyed. However, we now have another pointer to the interface (pCockpitCopy) hanging around which we theoretically think is still usable because we have not yet released it. Of course, this causes the application to fail because it is a dangling pointer to an interface for an object that is no longer alive.

Therefore, whenever you make an explicit copy of an interface pointer you should call AddRef. Make sure that you always call Release() for each interface pointer as well. This makes it much easier and tidier to see if the object is still valid. Here is the altered code:

```
void GOODfunction( ICockpit * pCockpit)  // COM Object counter =1
```

```
{
       ICockpit * pCockpitCopy;

       pCockpitCopy = pCockpit;          // Make copy of pointer
       pCockpitCopy->AddRef();           // Increment counter. Now =2
       pCockpit->Release();              // Counter now = 1
       pCockpit=NULL;

       pCockpitCopy->SetAirSpeed(400);   // Com object still alive all is well
       pCockpitCopy->Release();          // counter =0 so object is destroyed
       pCockpitCopy=NULL:
}
```

This is a habit you should adopt immediately since it is a crucial part of good COM programming. It is easy to see when debugging if a pointer is used and never had Release called. If you do not properly manage the reference counting of your COM object, then your application will have memory leaks or perhaps fail if an object is unloaded earlier than it should have been.

2.1.5 Return Values

All COM interface member functions return a 32-bit integer called an HRESULT. For most functions this is a structure that contains three bits of vital information about whether or not the function was successful. The 32-bit integer is actually divided up into 4 separate fields as shown below.

Severity - Bit 31 of the integer is set to 1 if an error has occurred or 0 if the function was a success.

Reserved – There are four bits of the HRESULT structure which are reserved

Facility – There are eight bits reserved to hold a facility code. This is a number that can be used to indicate the responsibility for the error. As an example, all DirectX Graphics functions return an HRESULT with a facility code of hex value 0x876. This number makes it easier to track down which code module the error was created in for debugging purposes.

Error Code – This is a 16-bit field that the COM object can use to return a meaningful error code. For example, if you open up the header file d3d9.h and scroll to the bottom of the file, you will see a list of many of the defined error codes being used alongside a macro to create a valid HRESULT.

Excerpt from D3D9.h

```
#define D3DERR_WRONGTEXTUREFORMAT                MAKE_D3DHRESULT(2072)
#define D3DERR_UNSUPPORTEDCOLOROPERATION         MAKE_D3DHRESULT(2073)
#define D3DERR_UNSUPPORTEDCOLORARG               MAKE_D3DHRESULT(2074)
#define D3DERR_UNSUPPORTEDALPHAOPERATION         MAKE_D3DHRESULT(2075)
#define D3DERR_UNSUPPORTEDALPHAARG               MAKE_D3DHRESULT(2076)
#define D3DERR_TOOMANYOPERATIONS                 MAKE_D3DHRESULT(2077)
#define D3DERR_CONFLICTINGTEXTUREFILTER          MAKE_D3DHRESULT(2078)
#define D3DERR_UNSUPPORTEDFACTORVALUE            MAKE_D3DHRESULT(2079)
#define D3DERR_CONFLICTINGRENDERSTATE            MAKE_D3DHRESULT(2081)
```

```
#define D3DERR_UNSUPPORTEDTEXTUREFILTER        MAKE_D3DHRESULT(2082)
#define D3DERR_CONFLICTINGTEXTUREPALETTE       MAKE_D3DHRESULT(2086)
#define D3DERR_DRIVERINTERNALERROR             MAKE_D3DHRESULT(2087)
#define D3DERR_NOTFOUND                        MAKE_D3DHRESULT(2150)
#define D3DERR_MOREDATA                        MAKE_D3DHRESULT(2151)
#define D3DERR_DEVICELOST                      MAKE_D3DHRESULT(2152)
#define D3DERR_DEVICENOTRESET                  MAKE_D3DHRESULT(2153)
#define D3DERR_NOTAVAILABLE                    MAKE_D3DHRESULT(2154)
#define D3DERR_OUTOFVIDEOMEMORY                MAKE_D3DHRESULT(380)
#define D3DERR_INVALIDDEVICE                   MAKE_D3DHRESULT(2155)
#define D3DERR_INVALIDCALL                     MAKE_D3DHRESULT(2156)
#define D3DERR_DRIVERINVALIDCALL               MAKE_D3DHRESULT(2157)
#define D3DERR_WASSTILLDRAWING                 MAKE_D3DHRESULT(540)
```

What you see in the parentheses above are the actual error codes for each error. The MAKE_D3DHRESULT macro takes this error code and embeds it into an HRESULT with the DirectX Graphics Severity code.

```
#define MAKE_D3DHRESULT( code )  MAKE_HRESULT( 1, 0x876, code )
```

The first parameter indicates that bit 31 should be set to 1. All HRESULTs have bit 31 set if they are reporting errors. All of the codes listed above are error codes, so they all have the 31st bit set. The second parameter is the facility code which is used to identify that this is a DirectX Graphics error, as opposed to some other COM object being used which is not related to DirectX Graphics. This other COM object would have its own facility code to allow us to distinguish between them. Finally, the last parameter is the actual error code itself. With it we can identify which of the DirectX Graphics errors actually occurred.

HRESULTS are also used to return success, or sometimes just the status of some object state. For example, in the same header file you can see that another HRESULT is created which is not an error but instead indicates success:

```
#define D3DOK_NOAUTOGEN    MAKE_D3DSTATUS(2159)
```

Here we use a macro which is almost the same as MAKE_D3DHRESULT with the exception that the 31st bit is cleared to indicate that this is not an error.

```
 #define MAKE_D3DSTATUS( code )  MAKE_HRESULT( 0, 0x876, code )
```

The D3DOK_NOAUTOGEN success value is one example where the function itself was successful BUT the information that was returned could be vital in terms of how you proceed. In this case it is successfully reporting that Auto Texture Generation is not supported by the hardware (do not worry about what that means for now).

Because an HRESULT can indicate any number of success or failure values, it can make testing the result of a function tedious and error prone. For example, imagine that we have called a COM member function that returns a number of error codes. We cannot simply test for success or failure as we do with a Boolean because we may miss something important:

```
HRESULT hr= SomeInterface->SomeFunction()
```

```
if (hr == E_FAIL)
{
    Report and error and handle the error
}
else
{
    Report success and continue
}
```

This approach may seem fine, but what happens if the function can return many different error codes and not just an E_FAIL error? The function could also return E_OUTOFMEM or E_INVALIDARGUMENT for example, and the above code would not catch these two errors. Although you could add a large collection of tests in the above code to test all possible return values, it is often the case that you are only interested in whether or not an error occurred. What the error actually was may not be as important to you. Certainly if an error has occurred and you have detected it, you could always take additional steps to narrow down which error was reported.

To do easy testing of HRESULTS there are two macros we can use: SUCCEEDED and FAILED. With them we can test HRESULTs in a Boolean style as shown below. These macros can generically signal success or failure simply by testing the severity bit of the HRESULT.

```
HRESULT hr = SomeInterface->SomeFunction();

if (FAILED(hr))
{
        report and handle error
}
else
{
        report success
}
```

The above code would catch all errors returned by the function. Whether you use SUCCEEDED or FAILED or both is totally up to you. The code could instead be written as:

```
HRESULT hr = SomeInterface->SomeFunction();

if (SUCCEEDED(hr))
{
        report success
}
else
{
        report and handle error
}
```

Note as well that the '!' operator works here. !SUCCEEDED is the same as FAILED and !FAILED is the same as SUCCEEDED:

```
if (!SUCCEEDED(hr))
{
        report and handle error
}
else
{
        report success
}
```

Another handy DirectX function is included in the header file dxerr9.h and lib file dxerr9.lib. If you include and link these with your application then you can call two functions that accept an HRESULT and return a string of text explaining the error. For example, you could do the following once an error has occurred:

```
if (FAILED(hr)
{
    Char *Error     = GetErrorString9(hr);
    Char *ErrDesc = GetErrorDescription9(hr);
    PrintMyString (Error);
    PrintMyString (ErrDesc);
};
```

After trapping the error, we retrieve a meaningful error code and a description of the error. These results can be output to the screen or to a log file. This is very handy for debugging.

There are alternatives for handling HRESULT errors if you do not wish to include dxerr9.h and dxerr9.lib. You could write down the hex value of the entire HRESULT and use the D3DXErr.exe that ships with the DX9 SDK to input the value and have the actual error returned. This application is located in the BIN folder inside the folder where you installed the SDK. You can also extract the error code from the HRESULT in code by using the following Windows macro, checking the value against the definitions in the header files. Windows defines three macros that can be used to extract all 3 bits of information from the 32-bit HRESULT.

```
#define HRESULT_CODE(hr) ((hr) & 0xFFFF)
#define HRESULT_FACILITY(hr) (((hr) >> 16) & 0x1fff)
#define HRESULT_SEVERITY(hr) (((hr) >> 31) & 0x1)
```

So to extract the actual error code we could just do:

```
int ErrorCode = HRESULT_CODE(hr);
```

Non-HRESULT Return Values

While an HRESULT can be handled trivially thanks to the FAILED and SUCCEEDED macros, there are exceptions to the rule. Some COM functions return an HRESULT as a simple 32-bit integer and this is the case with two of the IUnknown interfaces functions:

```
ULONG     AddRef (void);
ULONG     Release(void);
```

These functions return the current reference count of the object after the function call. The value returned by these functions are mostly for debugging purposes and should not be relied on as means of COM object management.

2.1.6 Backwards Compatibility

The Component Object Model does have important restrictions. We saw earlier that every interface has a unique GUID for identification purposes. Because of this, once an interface has been released to the public it must never change. It would seriously undermine COM's design for backwards compatibility if a single GUID was the same for many different versions of an interface -- each with different functions sets. Even fairly trivial interface changes, such as adding an additional function, is strictly forbidden because this could break applications using an older version of the interface.

So the question is, how does one upgrade COM object functionality? Refer back to our Aeroplane COM object and its three interfaces. Let us say that we wanted to release a new version of our Aeroplane object with extra functionality that allowed the user to set the autopilot. This new COM upgrade would replace the old COM object when installed on the user's machine. Since older applications will want to use this COM object as if it was still the older version, we must make sure that the old interfaces are supported and implemented within the new object.

Let us say that the logical place for our new function SetAutopilot() would be as a member of the ICockpit interface. Although this is true, the ICockpit interface has already been made public and cannot be changed. So it is common practice to create a new interface which adds the extra functionality like so:

```
HRESULT Hr;
ICockpit   *pCP;
ICockpit2  *pCP2;

CoCreateInstance( CLSID_Aeroplane , NULL , CLSCTX_INPROC_SERVER,
                  IID_ICockpit , (LPVOID*) &pCP);

// Use old interface as normal, old app don't break cause its still here
pCP->SetAirSpeed(325);
```

```
// New apps can request the addition interface to access extra function
pCP2->QueryInterface ( IID_ICockpit2 , (LPVOID*)&pCP2);
pCP2->SetAutopilot (true);
```

As you can see, the object still supports the older interface, so older applications can request this interface and carry on using the newer COM object without incident. Applications that are aware of the COM object's extended features can request the new interface to access the new functionality. In this example the ICockpit2 interface would look like this.

```
Interface ICockpit2
{
    HRESULT SetAutopilot (BOOL on);
};
```

It is customary for newer versions of an interface to have the number appended to the end as shown above. This is one way that new interfaces can be created.

Another way is for the developer to support all of the older interface functions in the new interface through the use of interface inheritance. This is a much nicer solution in most circumstances because the newer applications do not have to request the older interface first and then query again to access the new functionality. In this case the two ICockpit interfaces would look like so:

```
Interface ICockpit
{
    HRESULT SetAirSpeed   (ULONG Speed);
    HRESULT SetAltimeter  (ULONG Altitude);
    HRESULT UnderCarriage (BOOL Down);
};

Interface ICockpit2
{
    HRESULT SetAirSpeed   (ULONG Speed);
    HRESULT SetAltimeter  (ULONG Altitude);
    HRESULT UnderCarriage (BOOL Down);
    HRESULT SetAutopilot  (BOOL on)
};
```

Newer applications can now forget about the ICockpit interface and simply request an ICockpit2 interface when the object is initially created. All of the functions of the original interface are supported in the new interface so this solution works well:

```
HRESULT Hr;
ICockpit2  *pCP2;

CoCreateInstance(CLSID_Aeroplane , NULL , CLSCTX_INPROC_SERVER,
                 IID_ICockpit2 , (LPVOID*) &pCP);

// Use old interface as normal, old app don't break cause its still here
pCP2->SetAirSpeed(325);
pCP2->SetAutopilot(true);
```

This is not something we have to concern ourselves too much with in the latest versions of DirectX Graphics however because it uses all new interfaces, negating the need to query later ones. In the above example, older applications still have access to the totally unchanged ICockpit interface and newer applications can just forget all about the ICockpit interface and use an ICockpit2 interface for everything.

2.1.7 COM and DirectX Graphics

You might be starting to feel that COM requires too many complex procedures for seemingly simple tasks. Fortunately, with each new release of DirectX, the interaction between your application and the underlying COM layer becomes more and more encapsulated. In DX9 we no longer have to call CoInitialize or CoUninitialize because it is all handled for us by DirectX. Also, we hardly ever have to manually create a COM object using CoCreateInstance since this is something DirectX has wrapped for us as well. But this is not the case with all modules in DirectX, so the information learned here will surely come in handy in the future. When using DirectX Graphics, we rarely need to manually create a COM object because DirectX Graphics has a global function to create the main COM object and return the top level interface to that object. Once we have an interface to the top level object, we can use its member functions to create other DirectX Graphics objects and interfaces automatically. Thus, we will hardly ever need to call the QueryInterface function either, as there are helper functions within the DirectX Graphics interfaces to wrap the querying of interfaces. These wrappers also increment the reference count before returning the interface to the application. However, what is vitally important is that you still need to manage your reference count correctly and remember to do the following:

- Always release your interfaces when you have finished using them.
- If you make an explicit copy of an interface pointer, call AddRef to increase the reference count.

Make a note of this because if you do not release your interfaces after use, the objects will not be destroyed. This can lead to significant memory leaks. Also, if you do not call AddRef as you copy interface pointers, you could end up with dangling pointers to objects that have already been destroyed.

To better show the COM encapsulation of DirectX Graphics, the following code shows everything we need to do to create the top level DirectX Graphics COM object and access its interface. We can then use this interface to query the graphics capabilities of the system (among other things) on which the application is running, and create other DirectX Graphics objects and interfaces.

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

Believe it or not, that is it. As you can see, there is no need to initialize the COM layer via CoInitialize and no need to call CoCreateInstance to create the Direct3D9 object. We simply call the DirectX Graphics global function Direct3DCreate9. It will initiate the COM layer transaction, create the COM

object, and return a pointer to an IDirect3D9 interface. We can then use the pointer to this interface to call member functions.

> **NOTE:** The variable type LPDIRECT3D9 is defined in the d3d9.h header file as a pointer to an IDirect3D9 interface. Many people mistakenly call this a pointer to the Direct3D9 object but it is not. We never get a pointer to a COM object, only to an interface that the object supports. Therefore, this is actually a pointer to an IDirect3D9 interface which is attached to our Direct3D9 object.
>
> typedef struct IDirect3D9 *LPDIRECT3D9
>
> You will find that all interfaces have been typedef'd in a similar way. Another interface that DirectX Graphics uses is called IDirect3DDevice9. It has a pointer that is typedef'd in the same way:
>
> typedef struct IDirect3DDevice9 *LPDIRECT3DDEVICE9
>
> Whether you prefer to use LPDIRECT3DDEVICE9 or IDirect3DDevice9 * is up to you.

The next example shows that the Direct3D9 object exposes functions to create other COM objects and return interfaces for them. First we call the IDirect3D9::CreateDevice member function. This function creates the new COM object (the rendering device) and returns an interface to it (IDirect3DDevice9). The only parameter of importance for this discussion is the final one -- the address of a pointer to an IDirect3DDevice9 interface. If the function is successful, on function return this will point to a valid interface for the device object that was created. This behavior is typical of most of the DirectX Graphics objects. Using this approach we can create all of the objects DirectX Graphics provides and gain access to the full range of the DirectX Graphics functionality.

```
LPDIRECT3D9             pD3D;
LPDIRECT3DDEVICE9       pD3DDevice;
pD3D = Direct3DCreate9( D3D_SDK_VERSION );
pD3D->CreateDevice( blah , blah , blah , blah , blah , (LPVOID*)&pD3DDevice);
pD3DDevice->SomeFunction();
```

This encapsulation reduces most of what we have just learned (managing COM objects and interfaces) to nothing more than making sure that you release interfaces when you are finished using them and calling AddRef if you make an explicit copy of an interface pointer. That is nice to know.

> **NOTE:** The Direct3DCreate9 function is a global function used to kick-start the COM interaction with DirectX Graphics. This function is not a COM method, but is used to initialize the COM layer and create the initial COM object. Once we have a Direct3D9 object we can either directly or indirectly create all other DirectX Graphics COM objects through COM interface member functions.

2.2 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.

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, D3DFORMAT BackBufferFormat,
                        BOOL Windowed );
```

```
HRESULT EnumAdapterModes(UINT 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 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                 Adapter = 0;
D3DFORMAT            Format = D3DFMT_R5G6B5;
LPDIRECT3D9          pD3D;

pD3D = Direct3DCreate9( D3D_SDK_VERSION );
if (!pD3D) return FALSE;

UINT NumberOfModes = pD3D->GetAdpaterModeCount (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_X1R5G5B5 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.

2.3 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 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.

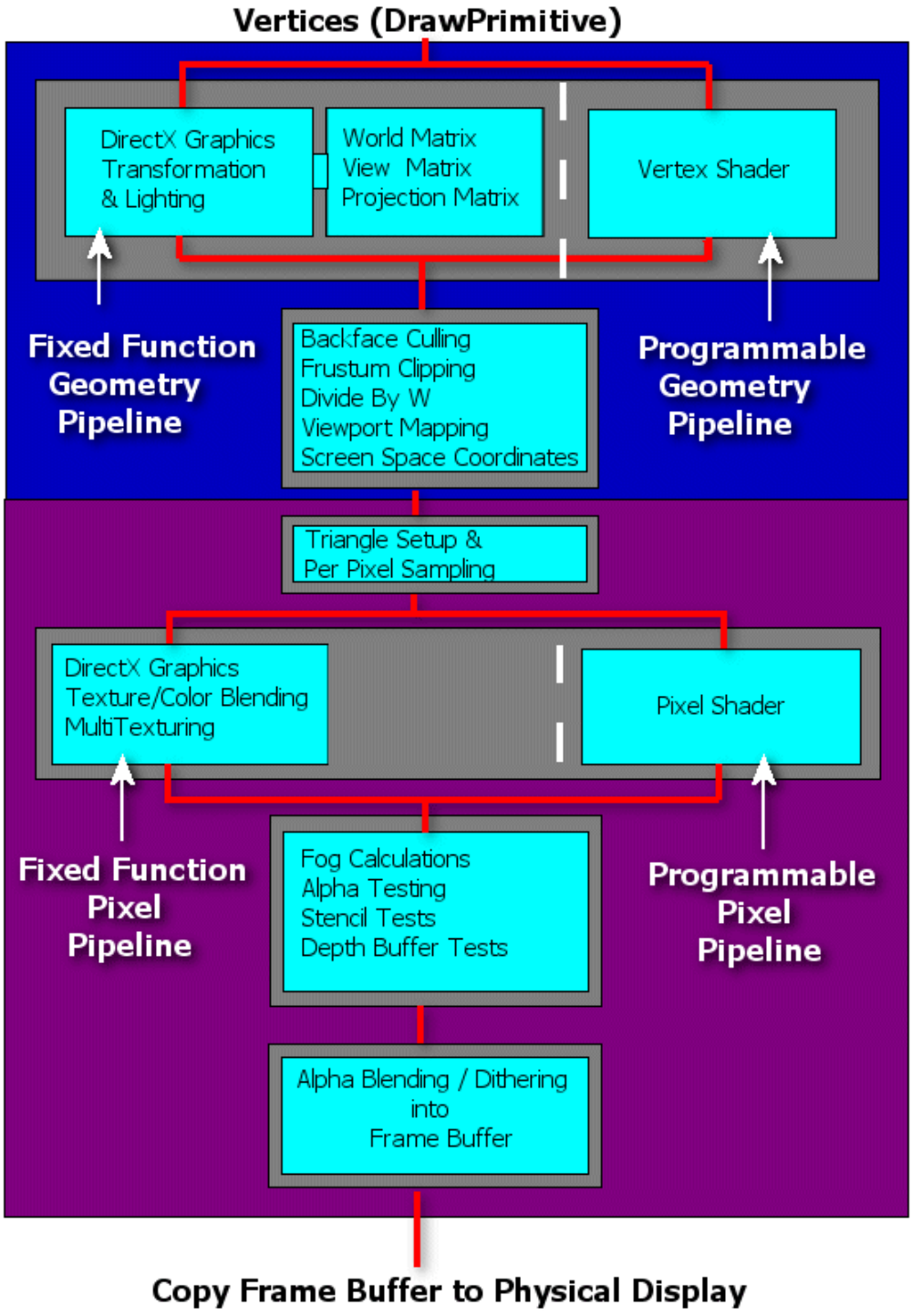


Figure 2.2

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).

2.3.1 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 `D3DTRANFORMSTATETYPE` 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, if 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.

2.3.2 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.

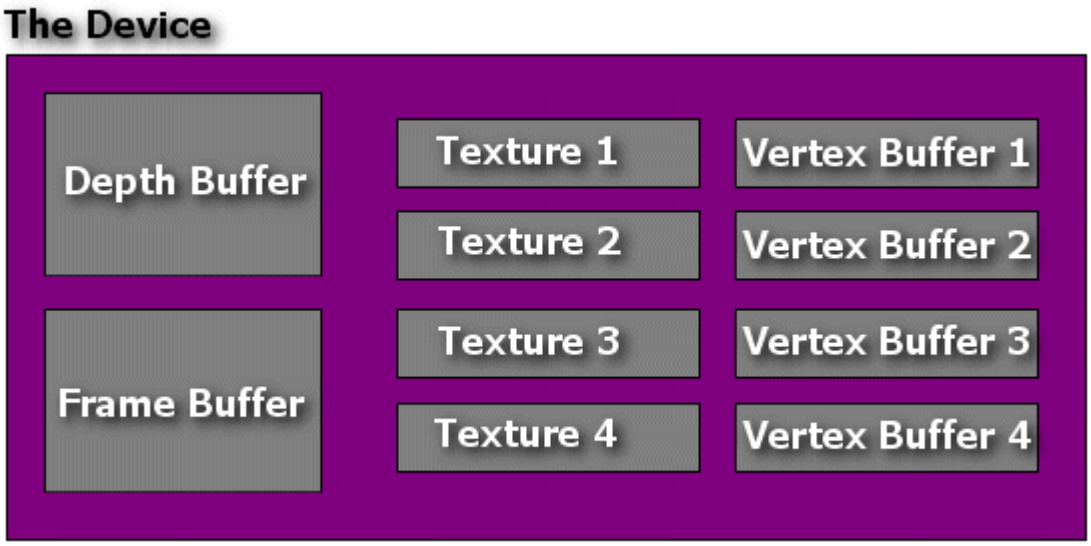


Figure 2.3

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 from 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.

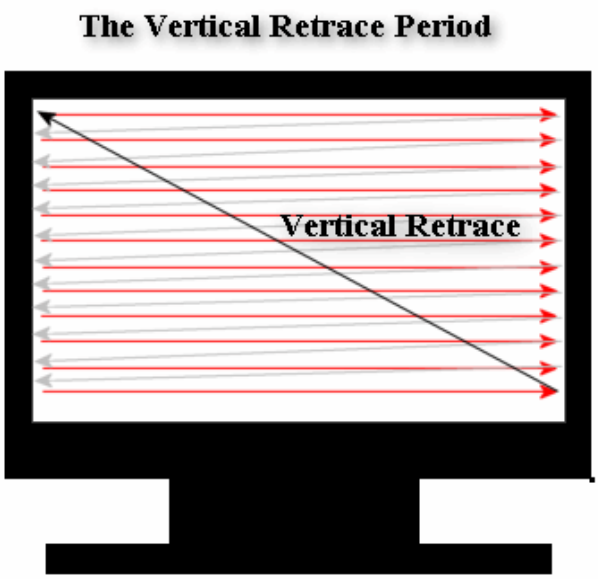


Figure 2.4

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:

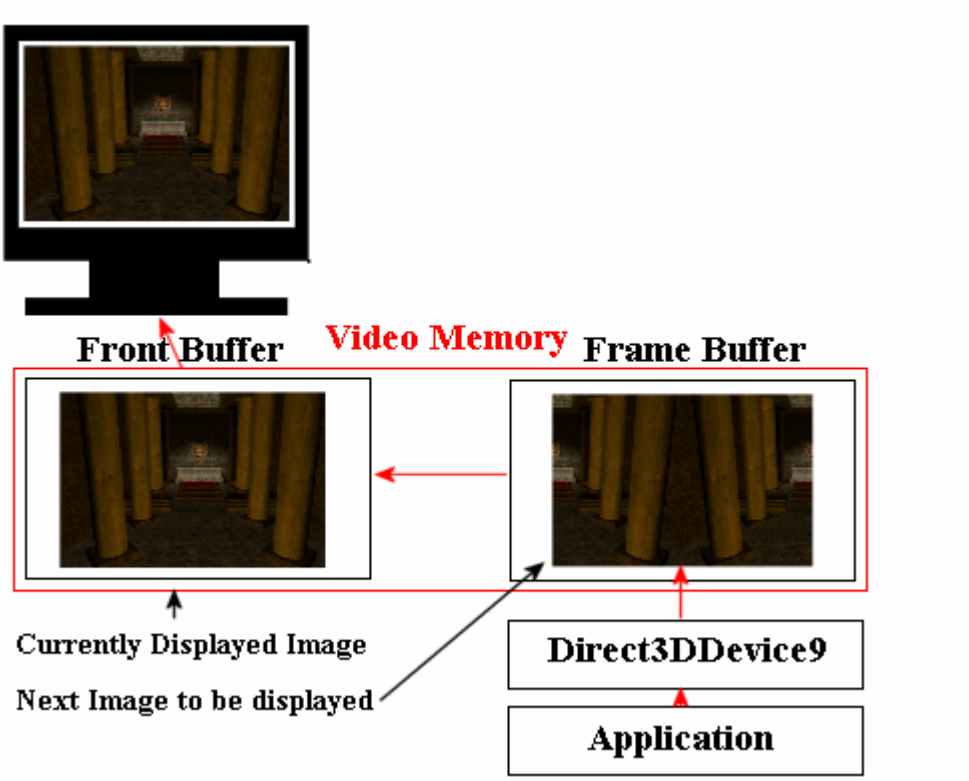


Figure 2.6

2.3.3 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:

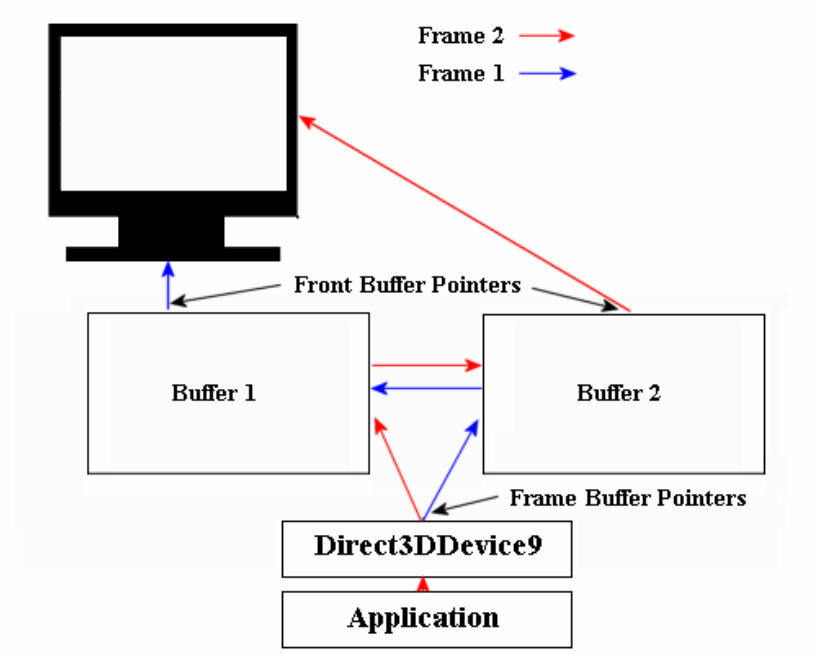


Figure 2.7

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.

2.3.4 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:

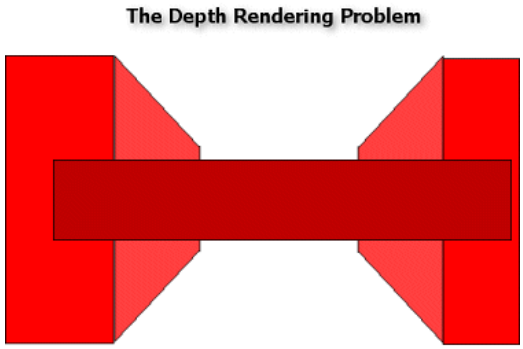


Figure 2.8

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.

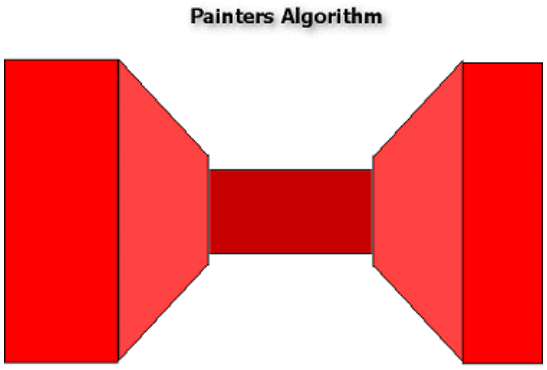


Figure 2.9

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).

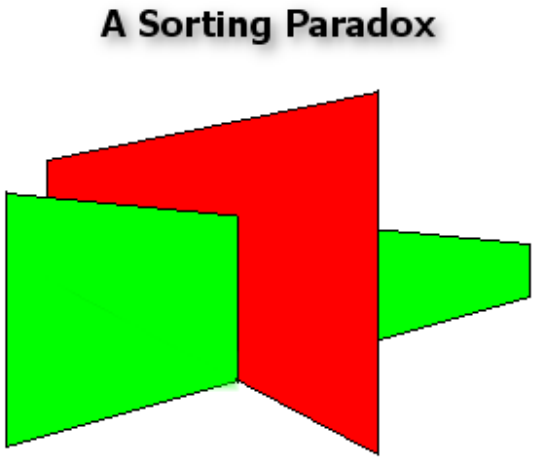


Figure 2.10

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:

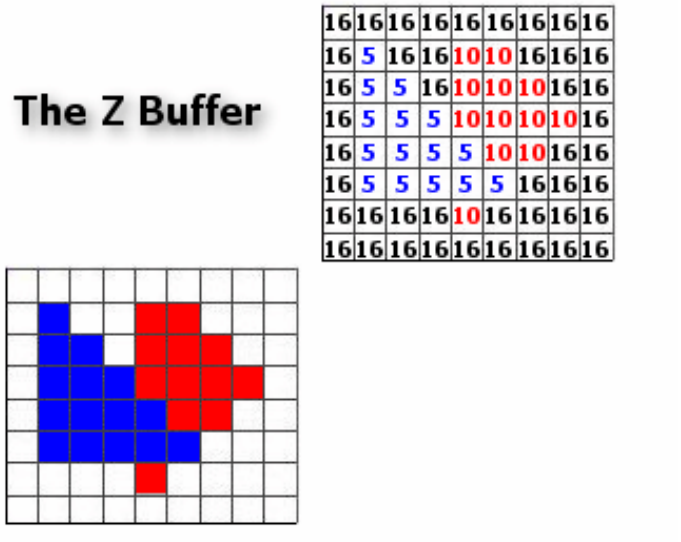


Figure 2.11

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 3rd 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:

Depth Z = z / w
Depth Z = z / z
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:

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.

2.4 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

D3DFMT_R8G8B8	24-bit RGB pixel format with 8 bits per channel.
D3DFMT_A8R8G8B8	32-bit ARGB pixel format with alpha, using 8 bits per channel.
D3DFMT_X8R8G8B8	32-bit RGB pixel format, where 8 bits are reserved for each color.
D3DFMT_R5G6B5	16-bit RGB pixel format with 5 bits for red, 6 bits for green, and 5 bits for blue.
D3DFMT_X1R5G5B5	16-bit pixel format where 5 bits are reserved for each color.
D3DFMT_A1R5G5B5	16-bit pixel format where 5 bits are reserved for each color and 1 bit is reserved for alpha.
D3DFMT_A4R4G4B4	16-bit ARGB pixel format with 4 bits for each channel.
D3DFMT_X4R4G4B4	16-bit RGB pixel format using 4 bits for each color.
D3DFMT_A2B10G10R10	32-bit pixel format using 10 bits for each color and 2 bits for alpha.
D3DFMT_A8B8G8R8	32-bit ARGB pixel format with alpha, using 8 bits per channel.
D3DFMT_X8B8G8R8	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.

2.4.1 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.

2.4.2 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`.

2.5 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 _D3DDEVTYPE {
    D3DDEVTYPE_HAL = 1,
    D3DDEVTYPE_REF = 2,
    D3DDEVTYPE_SW = 3,
    D3DDEVTYPE_FORCE_DWORD = 0xffffffff
} D3DDEVTYPE;
```

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 (D3DADPATER_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 previously 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_DEFAULT, 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_DEFAULT,
                                              D3DDEVICETYPE_HAL, hWnd,
                                              D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                              &d3dpp, &m_pDevice)))
                {

                     // last resort is the reference rasterizer
                     if (FAILED(pD3D->CreateDevice(D3DADAPTER_DEFAULT,
                                                   D3DDEVICETYPE_REF,hWnd,
                                                   D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                                   &d3dpp , &m_pDevice)))
                     {
                           // We couldn't even create a HEL device
                           // something is wrong and app will not run on machine
                           return FatalError;
                     } // End Reference Rasterizer
                }// End Hal - Software VP
          }// End Hal - Hardware VP
    }// End Hal - Pure Device
```

2.5.1 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:

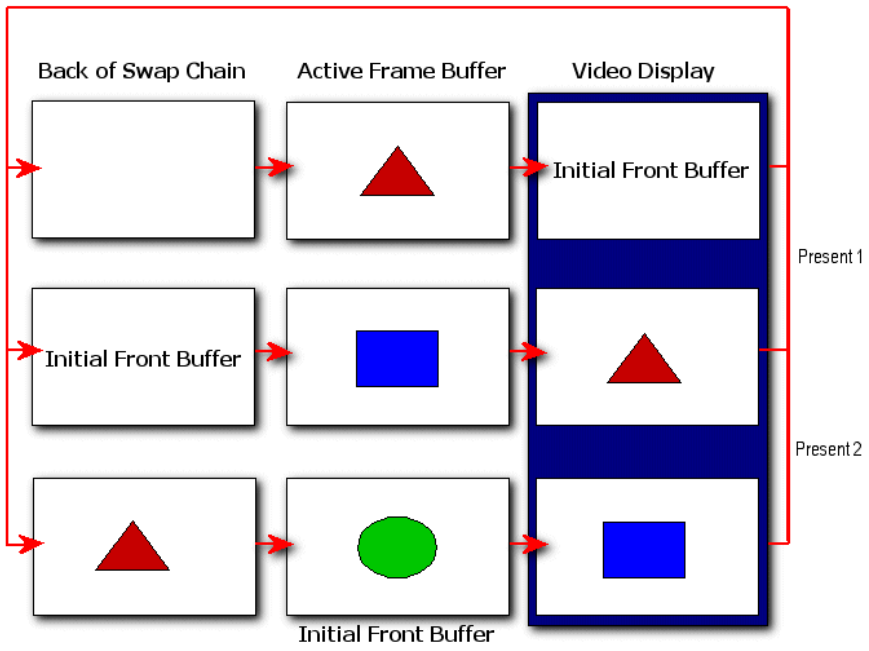


Figure 2.12

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:

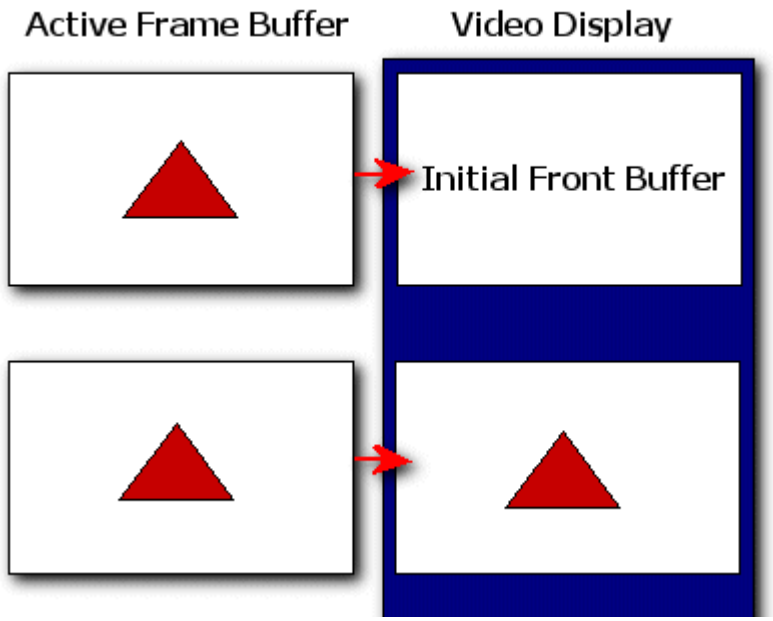


Figure 2.13

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.

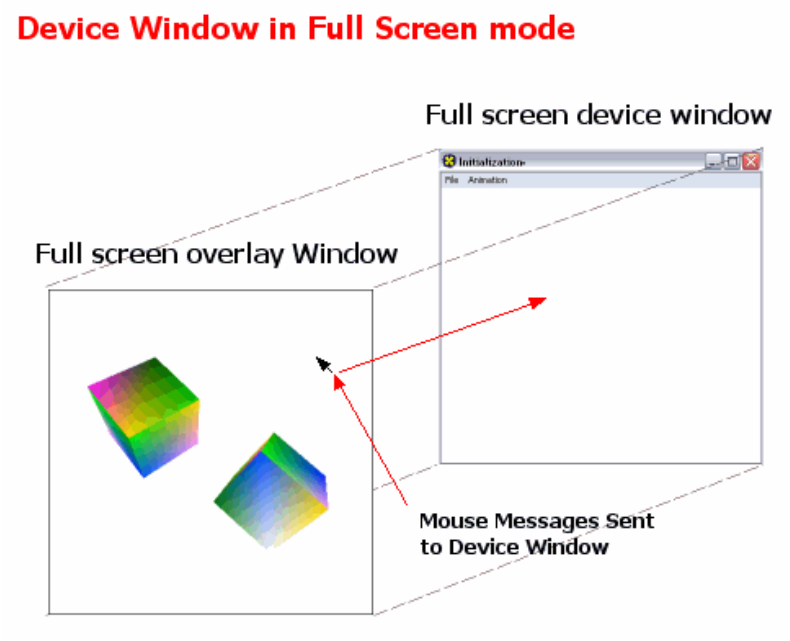


Figure 2.14

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.

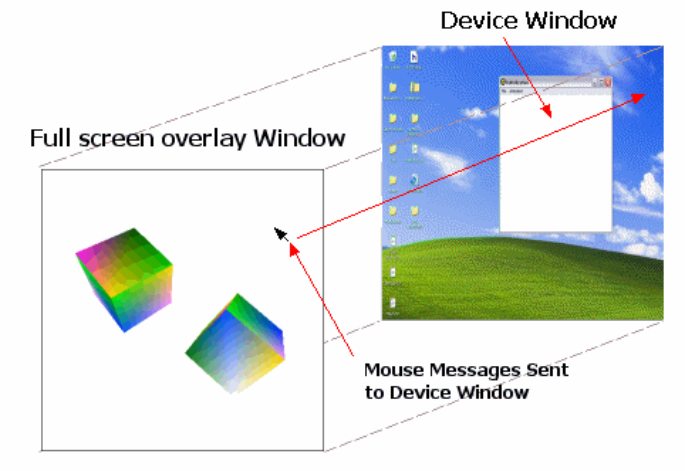


Figure 2.15

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:

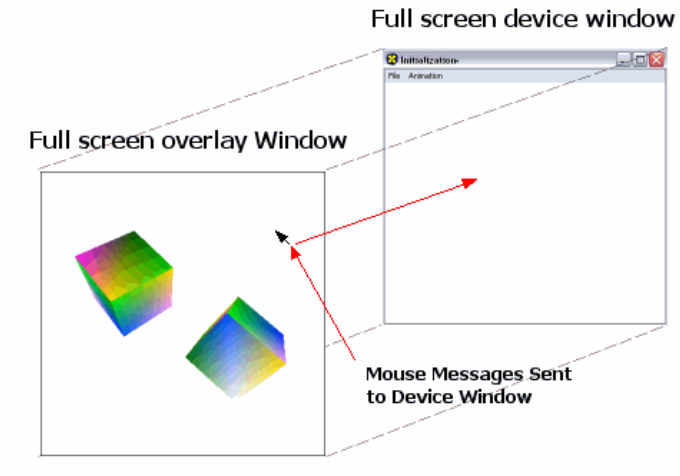


Figure 2.16

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.

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.

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.

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 the 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 ½ 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 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 ¼ 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.`

2.5.2 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;
}
```

2.5.3 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.

2.6 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.

2.6.1 Fill Modes

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.

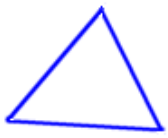
Point

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. Here 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. To the right we see 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. To the right 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);
```

2.6.2 Shading Modes

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);
```

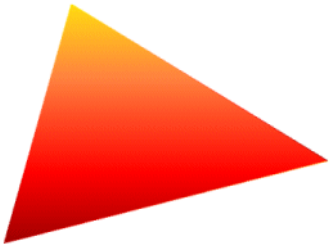
Flat Shade Mode

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 image on the right 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.

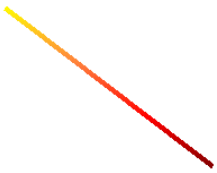
Gouraud Shade Mode

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 triangle to the right 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:



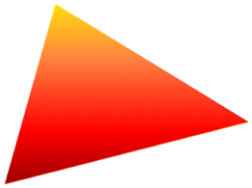
The line above consists of two vertices. The top vertex contains a yellow color and the bottom vertex contains a red color. We thickened the line so that it is several pixels wide for ease of viewing but keep in mind that in DirectX Graphics, line thickness in wireframe rendering mode is always one pixel.

2.6.3 Vertex Data

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.

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 (FVF)

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.
D3DFVF_NORMAL	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.

D3DFVF_DIFFUSE	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.
D3DFVF_SPECULAR	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.
D3DFVF_TEX0 through D3DFVF_TEX8	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:

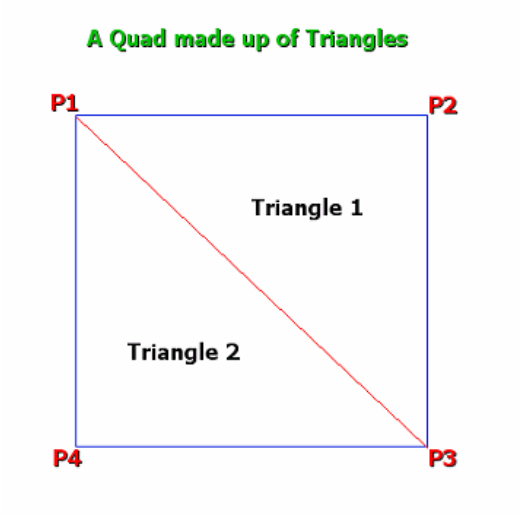


Figure 2.19

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.

2.6.4 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:

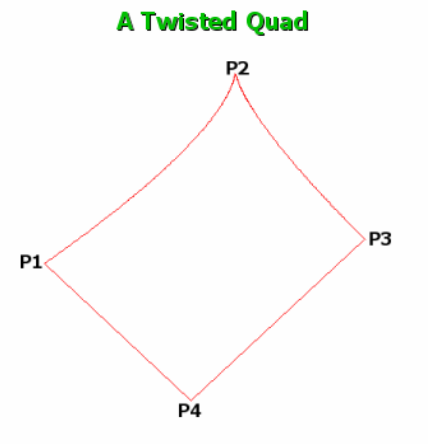


Figure 2.20

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-planar 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.

2.6.5 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 DrawPrimtiveUP 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 D3DPRIMITIVETYPE 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

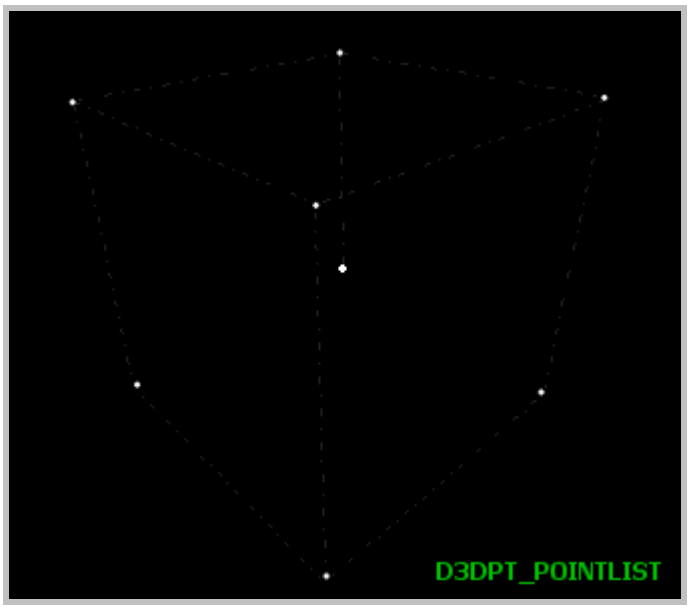
The D3DPT_POINTLIST primitive type informs the device that the vertex data should be treated as a list of points to be rendered, not as a list of triangles. The vertices pass through the transformation and lighting pipeline and have their vertex colors calculated just like normal vertices, but at rendering 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.

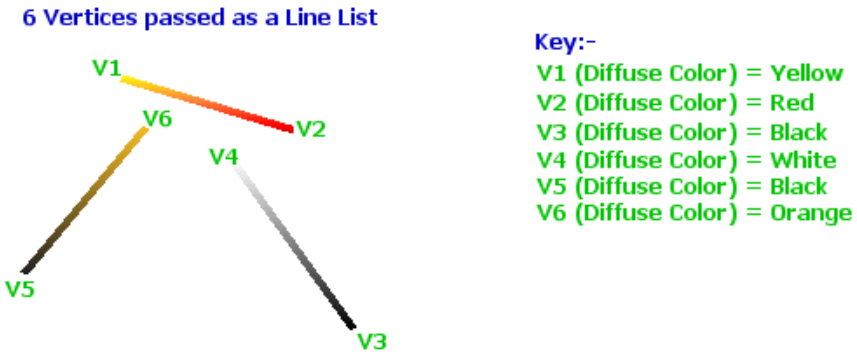


Figure 2.22

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 (NumberOfLines) * 2 (VerticesPerLine) = 10):

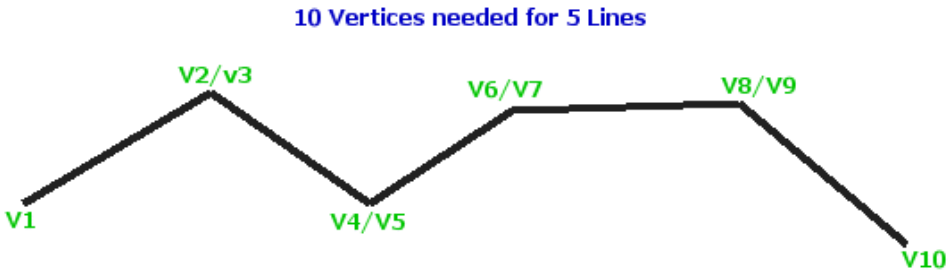


Figure 2.23

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:

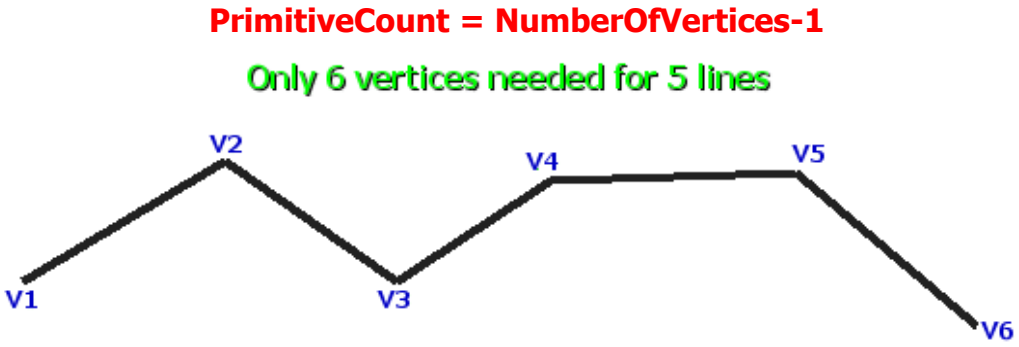


Figure 2.24

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):

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:

PrimitiveCount = 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:

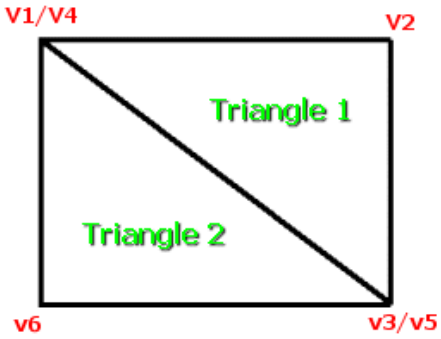


Figure 2.26

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 DrawPrimtiveUP 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.

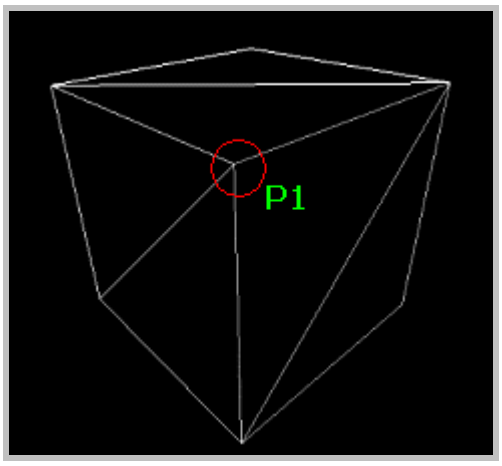


Figure 2.27

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 DrawPrimtiveUP 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 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:

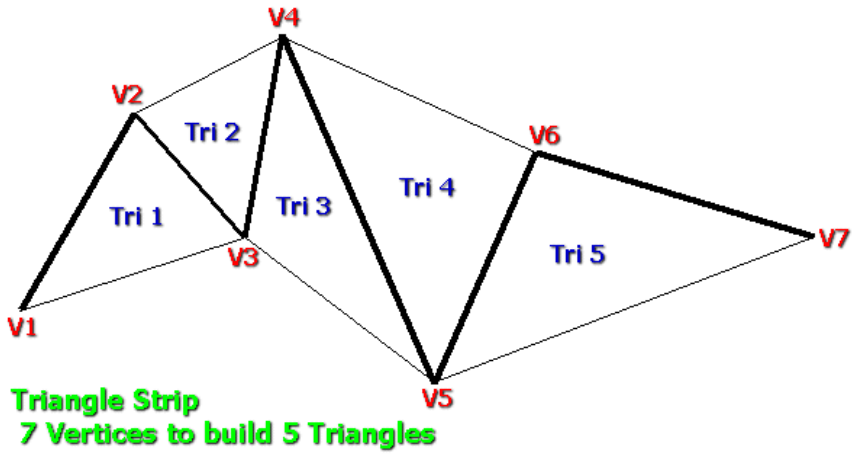


Figure 2.28

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:

PrimitiveCount = 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.

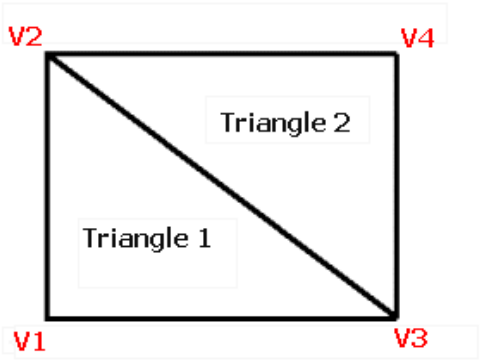


Figure 2.29

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 rendering 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.

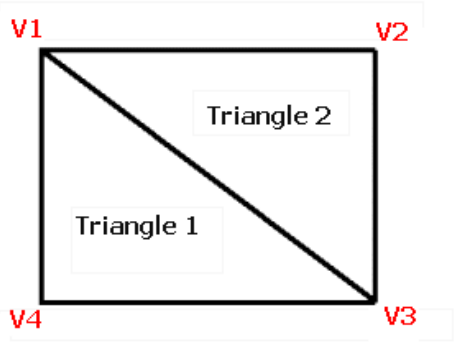


Figure 2.30

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.

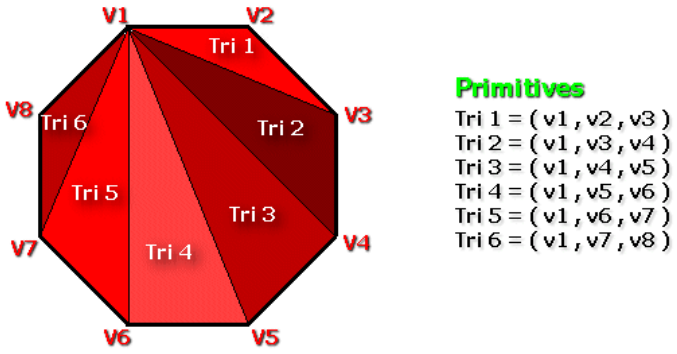


Figure 2.31

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.

2.6.6 The Rendering Pipeline

We now understand how to define vertices and how to render them using the DrawPrimtiveUP 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);
```

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.

2.7 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.

2.7.1 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);
```

2.7.2 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.

> **Note:** 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);
```

> 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.

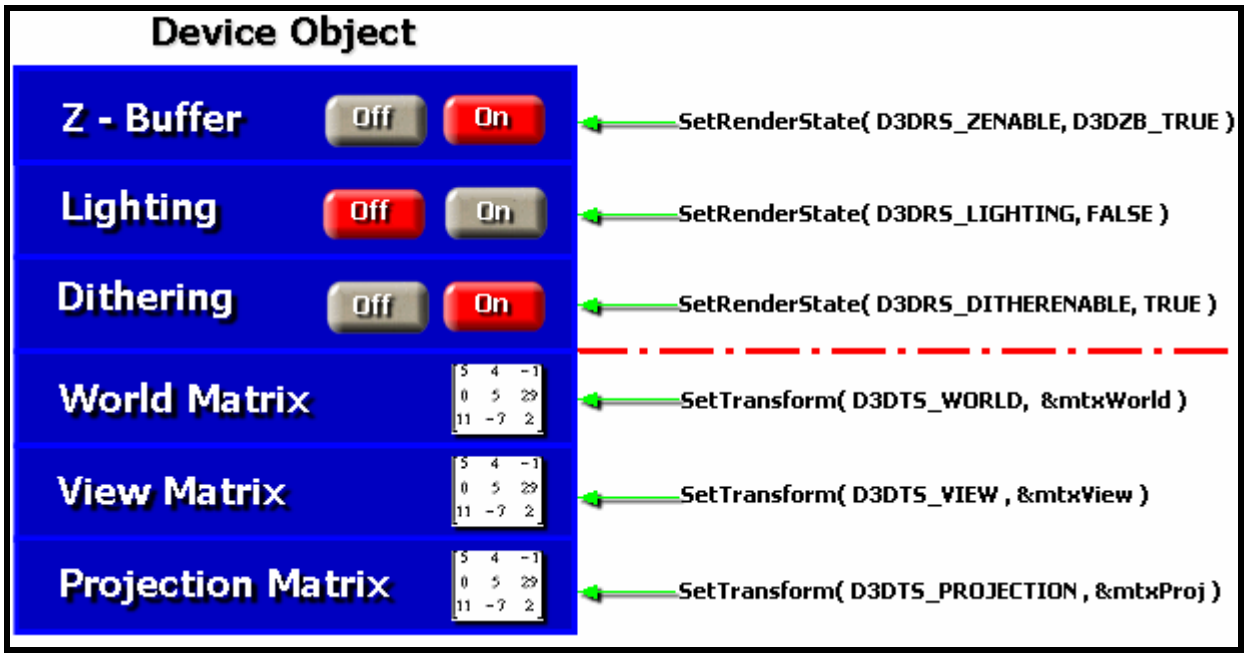


Figure 2.33

2.7.3 Scene Rendering

Frame/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.

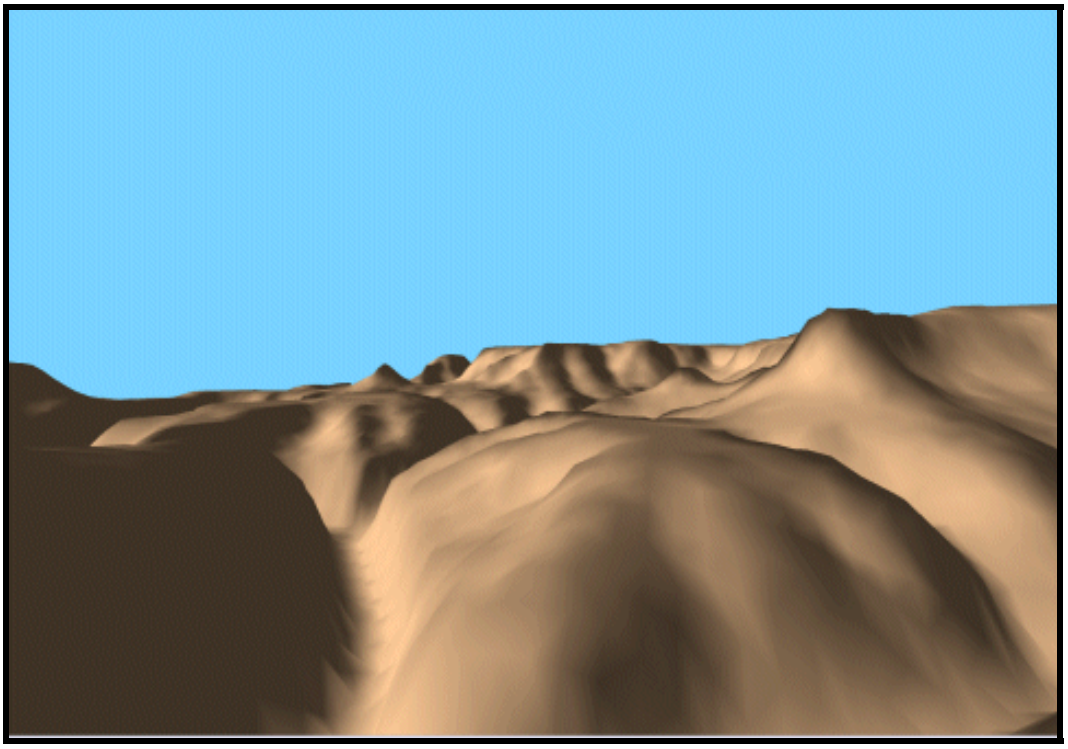
Conclusion

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.

Chapter Three:

Vertex and Index Buffers



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.

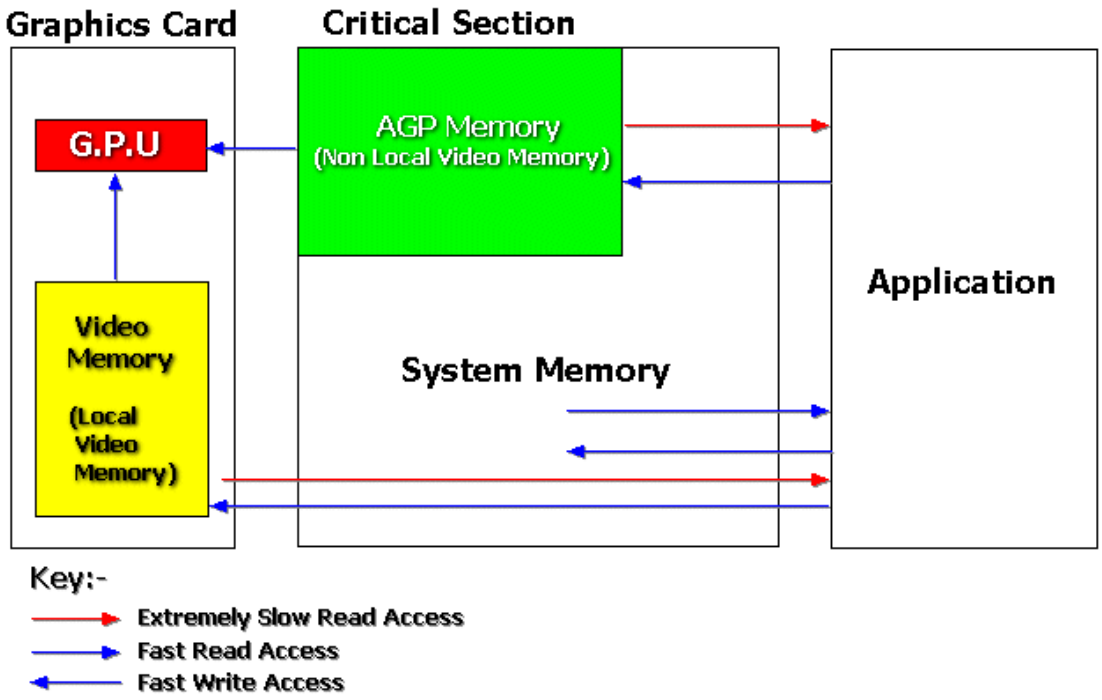
When the graphics adapter does not have hardware vertex processing capability, the situation is different. In such a case, the transformation and lighting of our vertices is done by the DirectX Graphics device in software. In this situation, using DrawPrimitiveUP would not degrade performance quite as 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.

3.1 Working with Device Memory

Vertex buffer behavior 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).



3.1.1 Memory Types

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 video cards 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.

3.1.2 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.

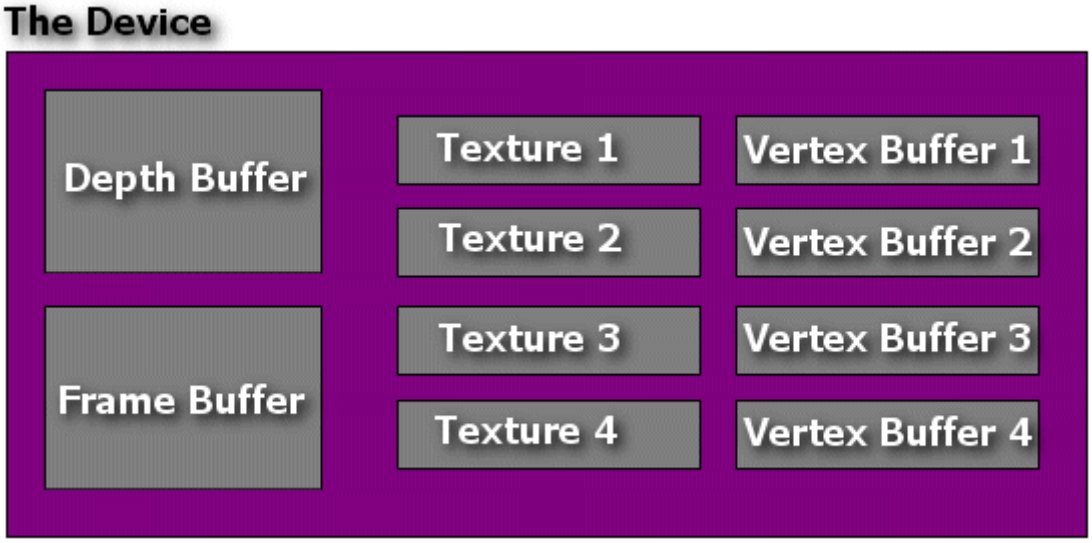
If your application requires read access to a vertex buffer then your best solution is to create it in system memory (even if a GPU is available). Of course, there are often solutions to help you work around the performance penalties associated with reading operations. One of the most obvious is keeping 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.

3.1.3 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.



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.

3.2 Vertex Buffers

3.2.1 Creating Vertex Buffers

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 * 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 D3DUAGE_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 behavior, 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 place into. When combined with the D3DUSAGE flags, it directly governs the performance and behavior 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;
```

3.2.2 Vertex Buffer Memory Pools

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 **D3DUAGE_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 behavior 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 behavior.

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 `D3DUAGE_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.

3.2.3 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 behavior.

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 behavior.

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 behavior.

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 behavior 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:

D3DPOOL_DEFAULT	(Hardware Vertex Processing)	23 s 80.1601 ms
D3DPOOL_MANAGED	(Hardware Vertex Processing)	42.7314 ms
D3DPOOL_SYSTEMMEM	(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 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 obscure. Lock 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.

3.2.4 Filling 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 optimized 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^{th} vertex in the vertex buffer. If you pass in zero, the pointer returned will point to the start of the vertex data.

UINT 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^{th}, 12^{th}, 13^{th}, 14^{th}, and 15^{th} vertices in the buffer you would use:

OffsetToLock = 10 * sizeof(CVertex)
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 uncompressed 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_NO_DIRTY_UPDATE
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.

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.

3.2.5 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.

3.2.6 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 **D3DPRIMTIVETYPE** 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.

3.3 Index Buffers

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.

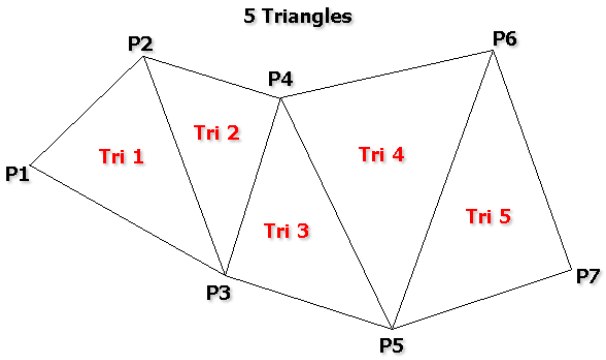


Figure 3.1

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
                    ------------   ------------   ------------   ------------   -------------
VertexBuffer =   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:

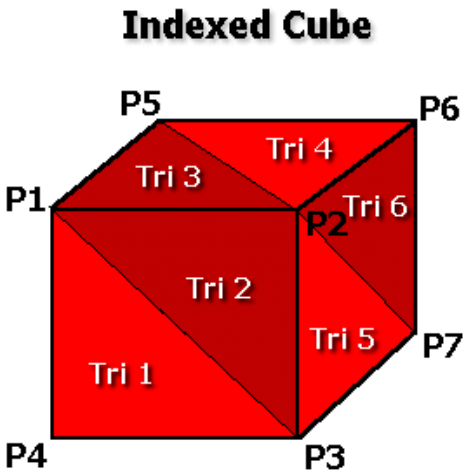


Figure 3.2

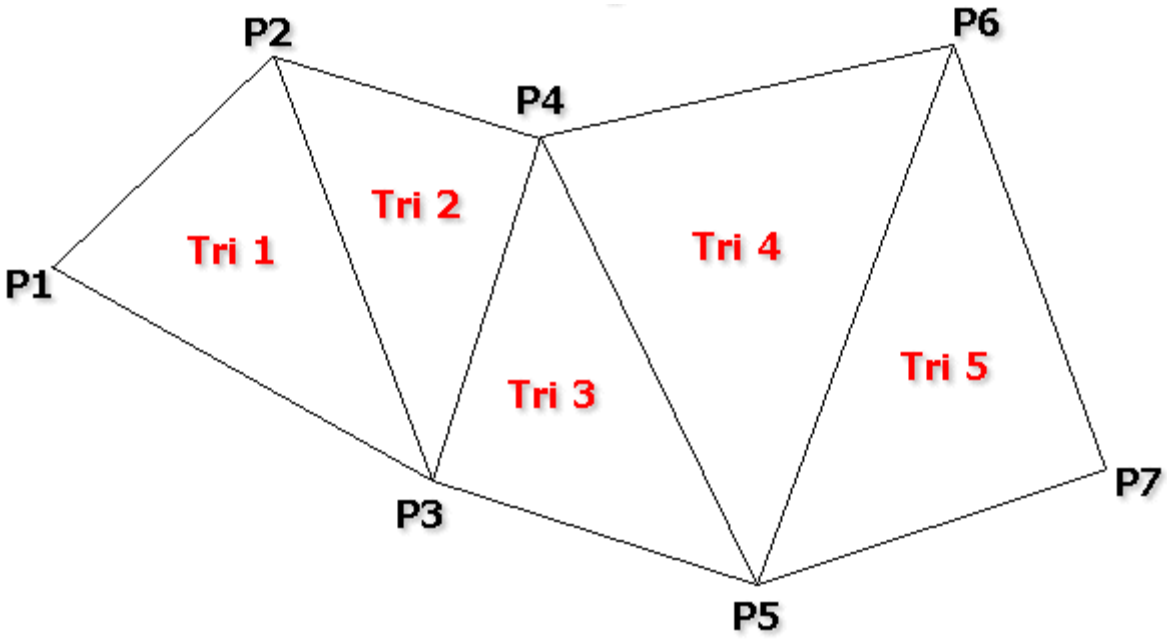
In Fig 3.2 we have labeled only the seven vertices that are used by the six visible triangles making up the three cube faces. Ignoring the back faces for now, we could create a vertex buffer for the six visible triangles as:

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	Tri5	Tri6
	------	------	------	------	-------	-------
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 behavior carries over when using indexed triangle strips.

We can calculate the number of indices needed to render an indexed triangle strip as NumTriangles * 2. This is identical to the way we calculated the vertices needed for a non-indexed triangle strip in Chapter Two.

3.3.1 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();
```

3.3.2 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.

D3DPRIMTIVETYPE 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.

3.3.3 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
);
```

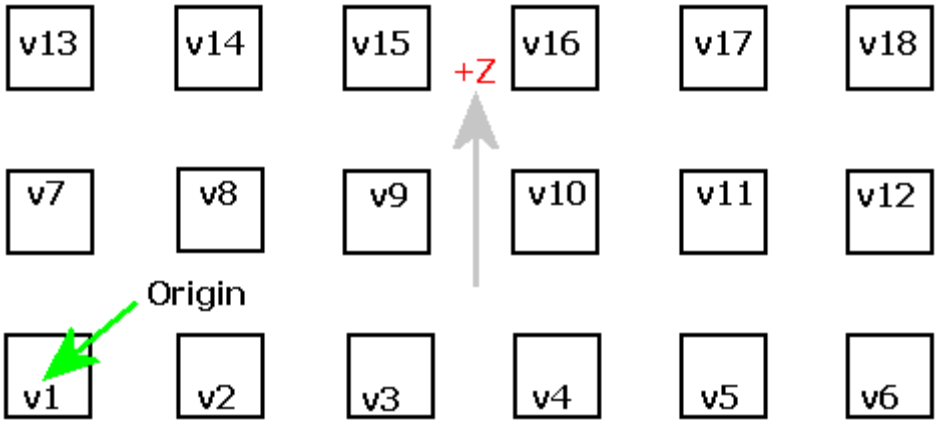
3.3.4 Indexed Triangle Strips

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.

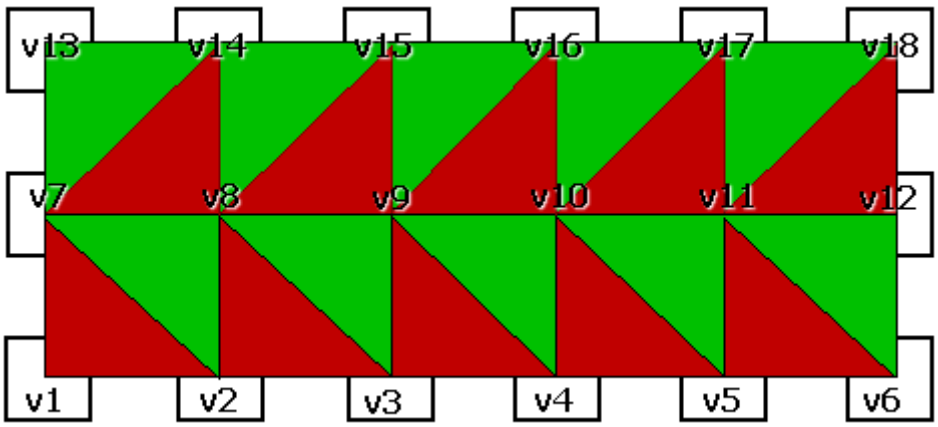
Let us first look at how the vertices will be arranged in the vertex buffer. The following diagram shows the vertices in world space, with the origin of the coordinate system at the bottom left vertex. We are

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.



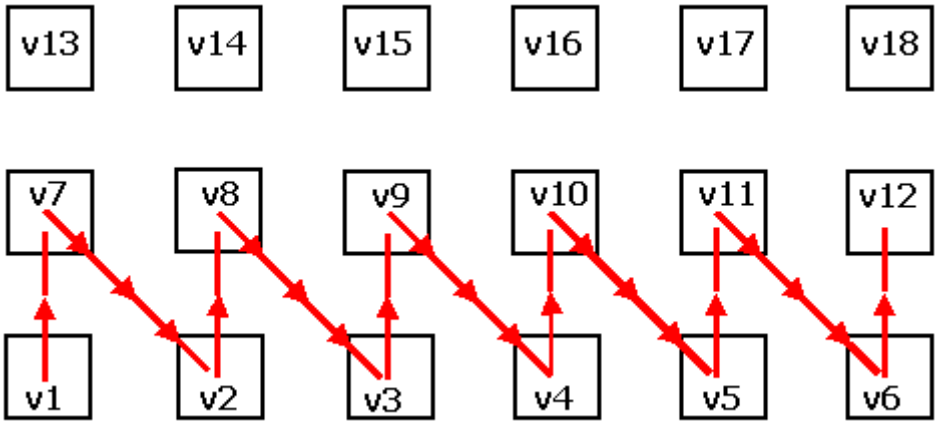
As we read in each row of the image, the vertices are arranged in rows stretching out from the origin along both the positive X- and Z-axes. The vertex buffer really is as simple as that. Each mesh vertex buffer will be a 17x17 pool of vertices arranged in rows.

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.

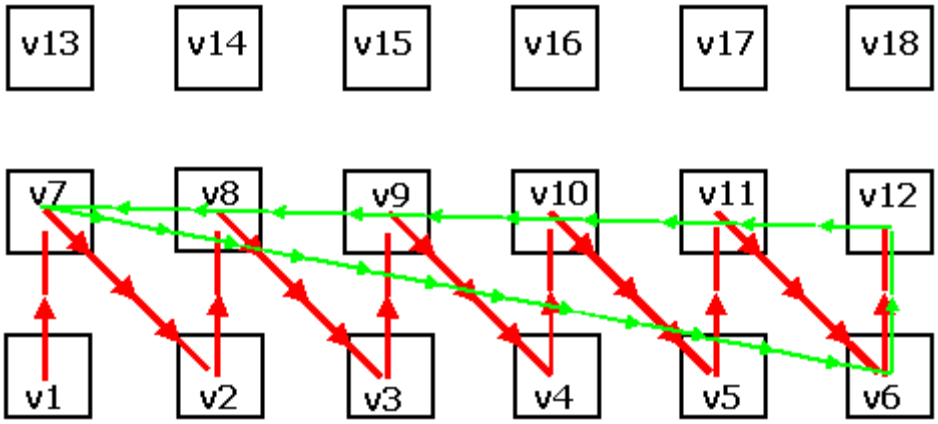
Index Buffer = 0 , 6 , 1 , 7 , 2 , 8 , 3 , 9 , 4 , 10 , 5 , 12

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

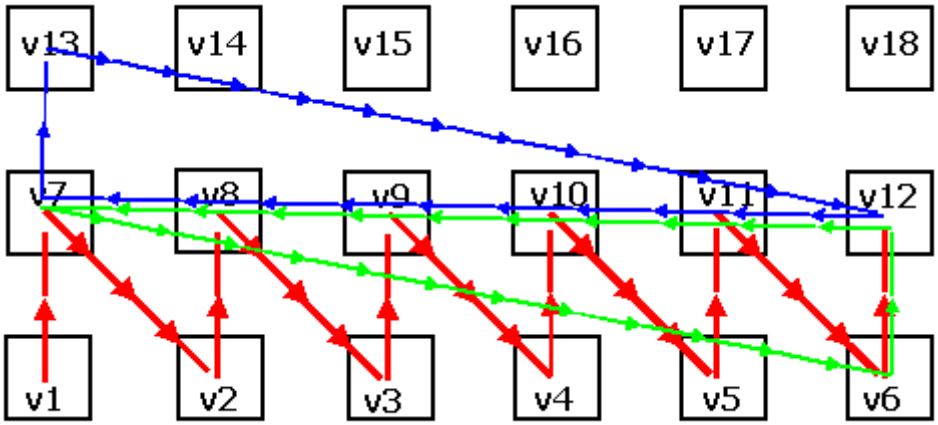
Indices after v7 has its index is added

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).

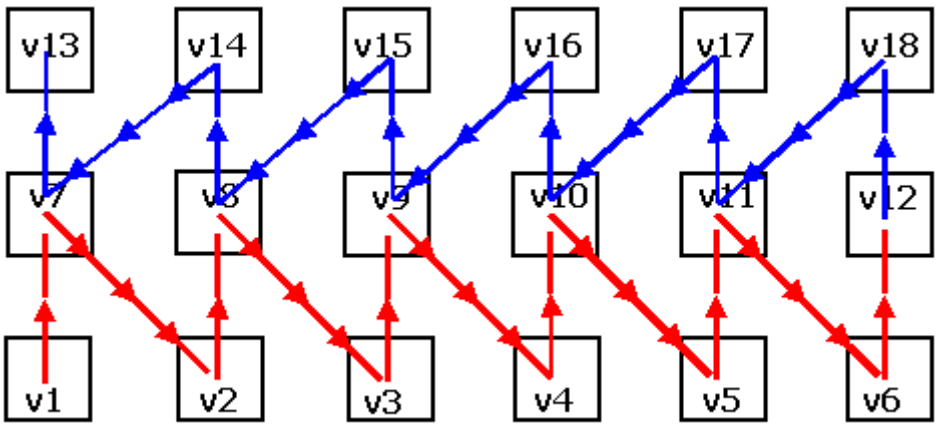
Indices after vertex v13 had its index added:

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.

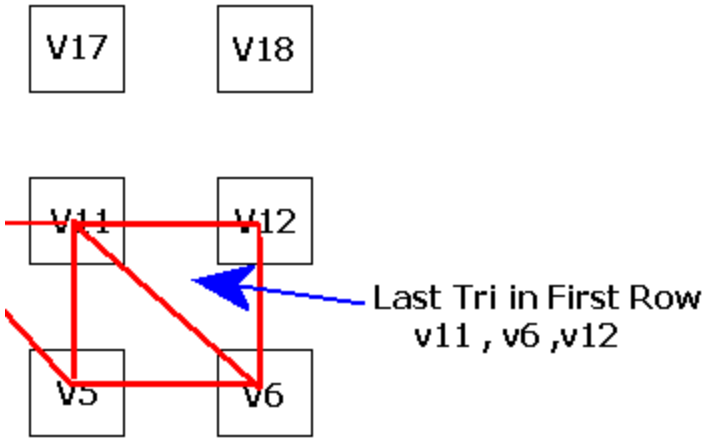
Degenerate Triangles

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.

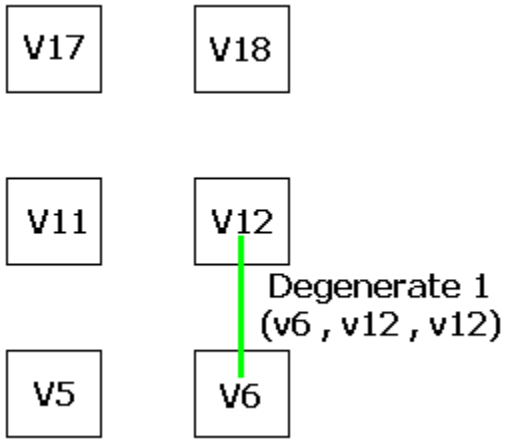
A classic example of a degenerate triangle is one where all of the indices reference the same vertex (or a vertex in the same position). The triangle in this case would be infinitely small and would be rejected by the pipeline. Another example would be when a triangle has two vertices that are the same. 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.

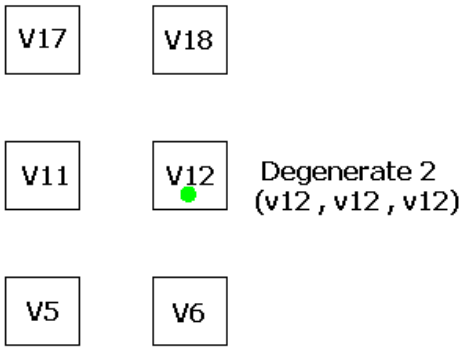


At this point we have reached the end of the row and can move up and start adding the pairs of vertices for the next row of quads (consisting of vertex pairs in rows 2 and 3). For example, v12 and v18 followed by v11 and v17, and so on. However, before we start the second row, we add the first index 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.

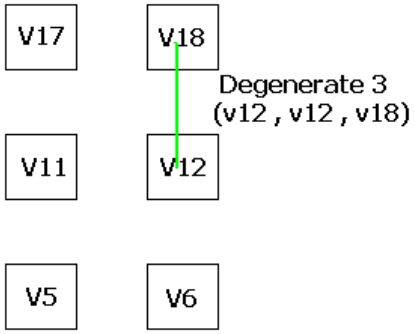


Because the last triangle was (v11, v6, v12), adding an extra v12 at the start of the next row creates a triangle where two to of its indices reference the same vertex. This is rejected by the pipeline and it is not rendered.

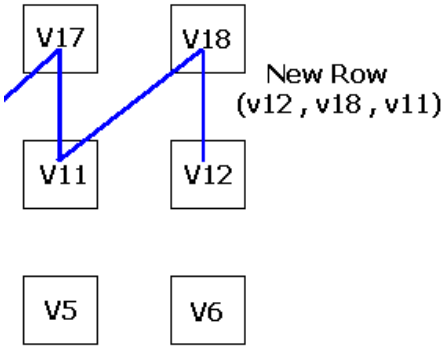
Now we can start constructing the second row. First we add the first index of the first vertex (v12). We now have a situation where the last three indices in the index buffer are now all the same. This creates another degenerate triangle as shown below.



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 looks 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

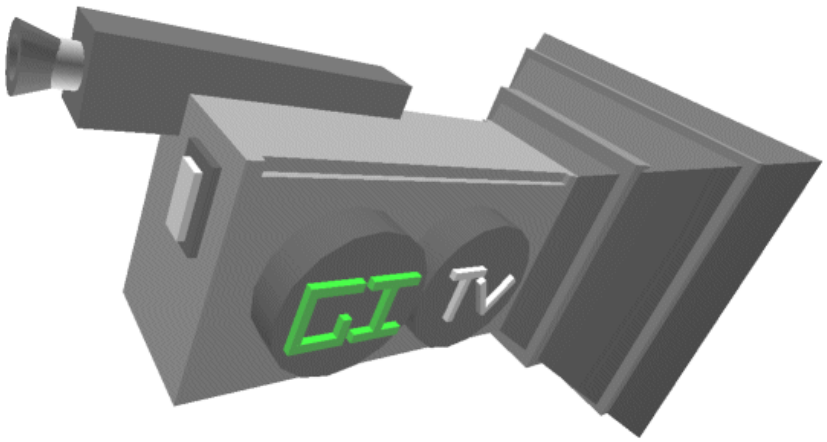
We have just studied some of the most important aspects of 3D graphics programming with DirectX. We learned how to render using vertex buffers as well as indexed primitives. We learned about efficiently creating device resources and proper use of memory pools. We also learned how to index and render 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.

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.

Chapter Four:

Camera Systems



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.

4.1 The View Matrix

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 **P** from model space and produce a new vertex **P'** in world space by applying a series of transformations using matrix multiplication. The relationship between these two vertices is represented as:

$$\mathbf{P'} = \mathbf{P} * \mathbf{M}_{world}$$

Note that the algebraic inverse of this equation describes the reverse relationship. To solve for **P**:

$$(\mathbf{1} / \mathbf{M}_{world})\ \mathbf{P'} = \mathbf{P} * (\mathbf{1} / \mathbf{M}_{world})\ \mathbf{M}_{world}$$

$$\mathbf{P} = \mathbf{M}_{world}^{-1}\ \mathbf{P'}$$

$\mathbf{M}_{world}^{-1}$ is the inverse of matrix $\mathbf{M}_{world}$. When we multiply the world space point **P'** by this matrix, we get back the original local space point **P** as expected. So we can say that $\mathbf{M}_{world}^{-1}$ undoes the effect that $\mathbf{M}_{world}$ had on **P**. Again this makes sense since we used one to cancel out the other in the equation above so that **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:

$$\mathbf{I = M * M^{-1}}$$

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.

In Chapter 1 we learned 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).

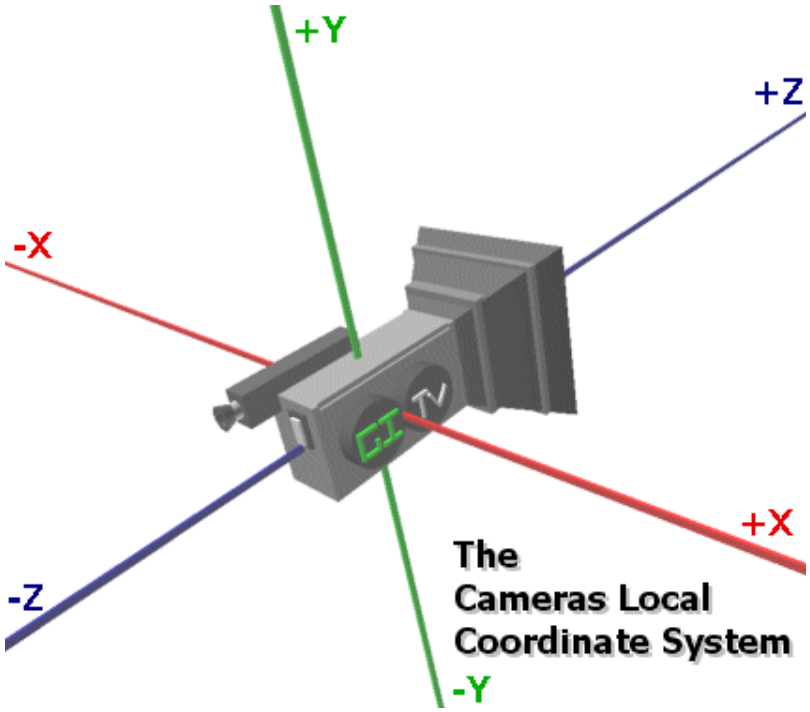


Figure 4.1

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).

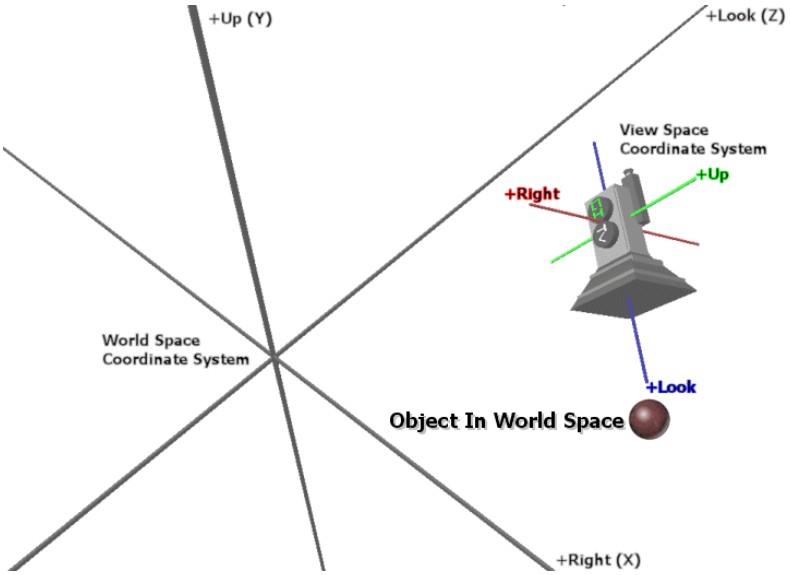


Figure 4.2

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.

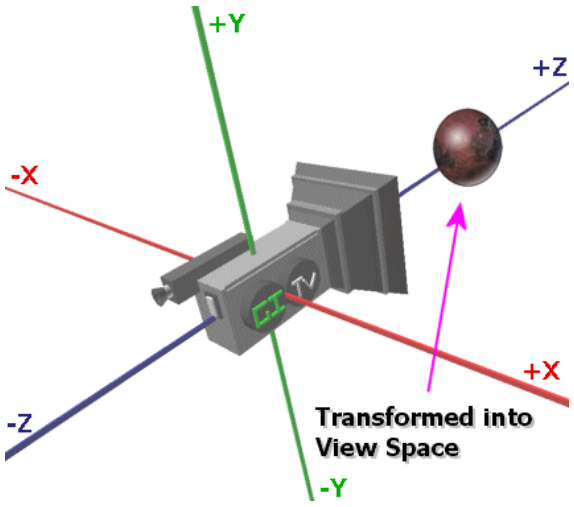


Figure 4.3

4.1.1 Vectors, Matrices, and Planes 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,

traveling 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.

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.

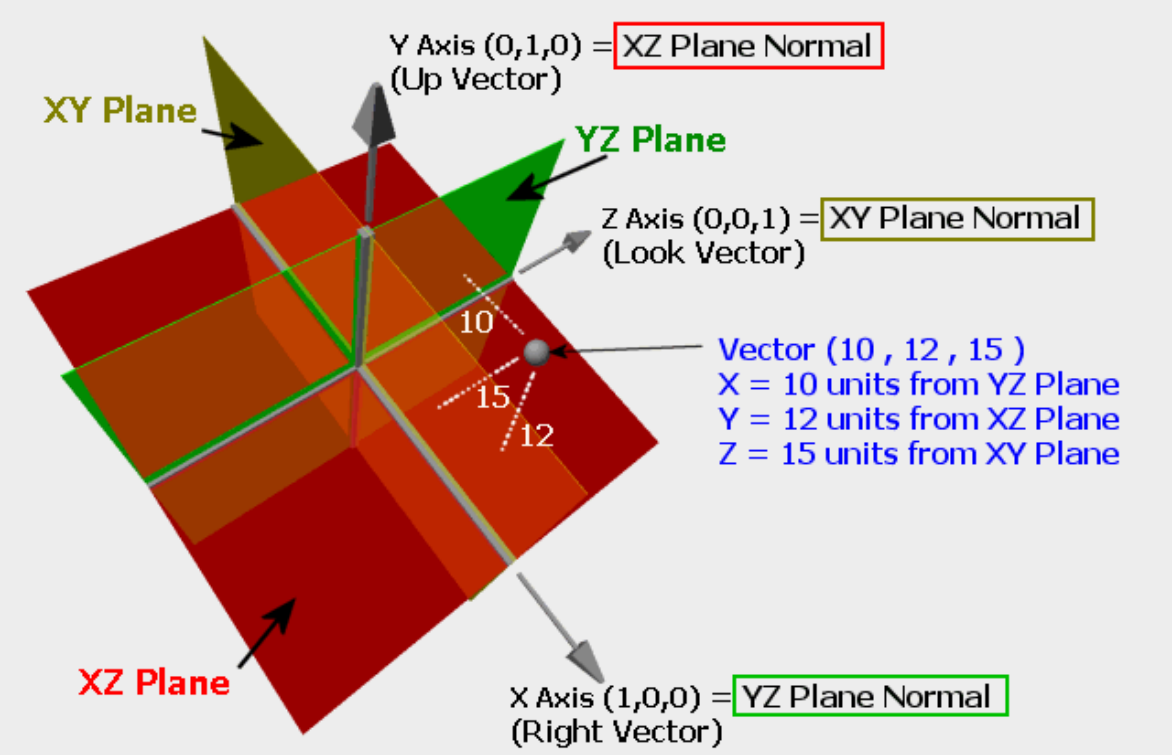


Figure 4.4

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.

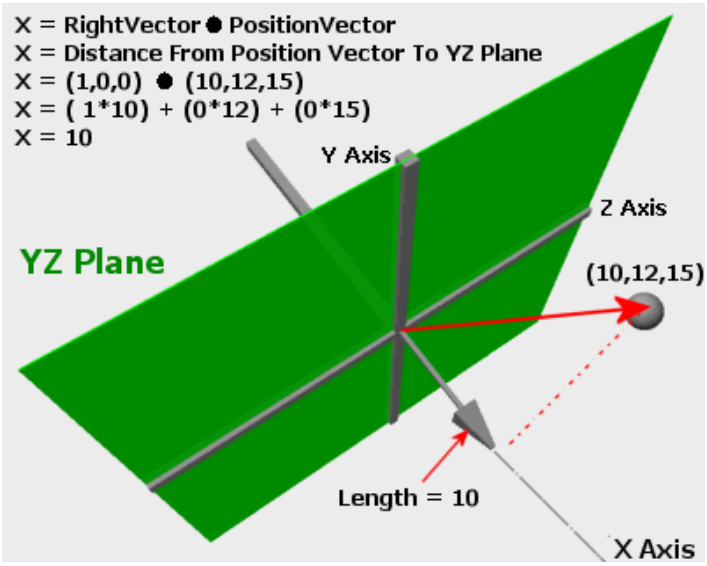


Figure 4.5

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.

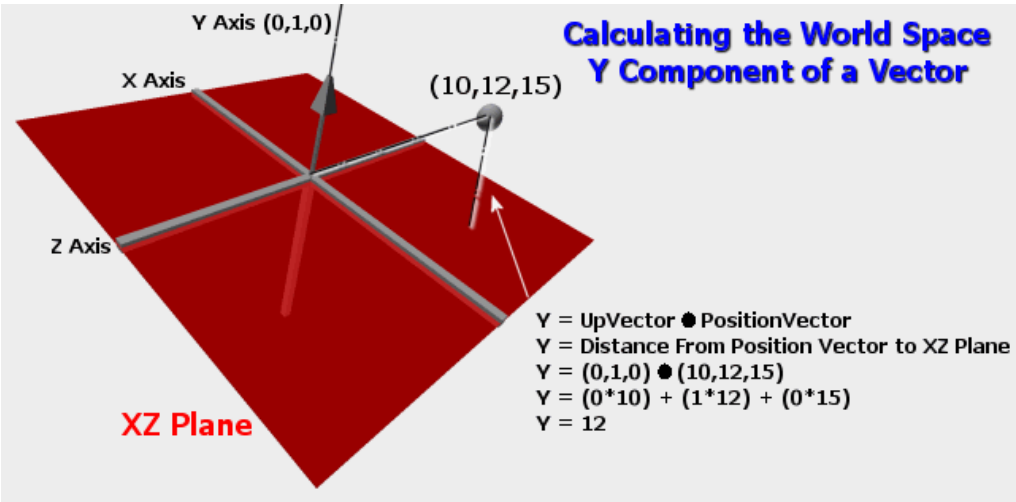


Figure 4.6

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.

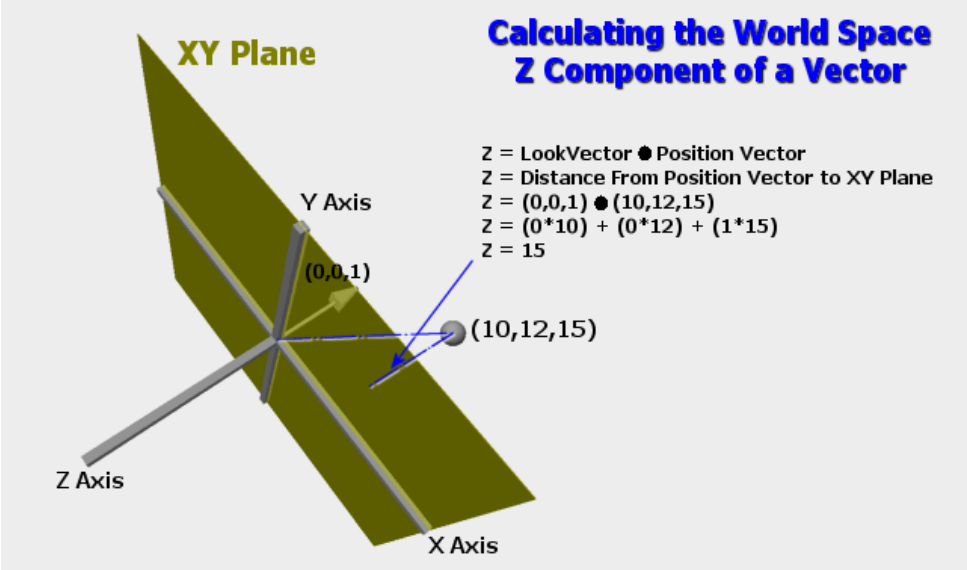


Figure 4.7

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.

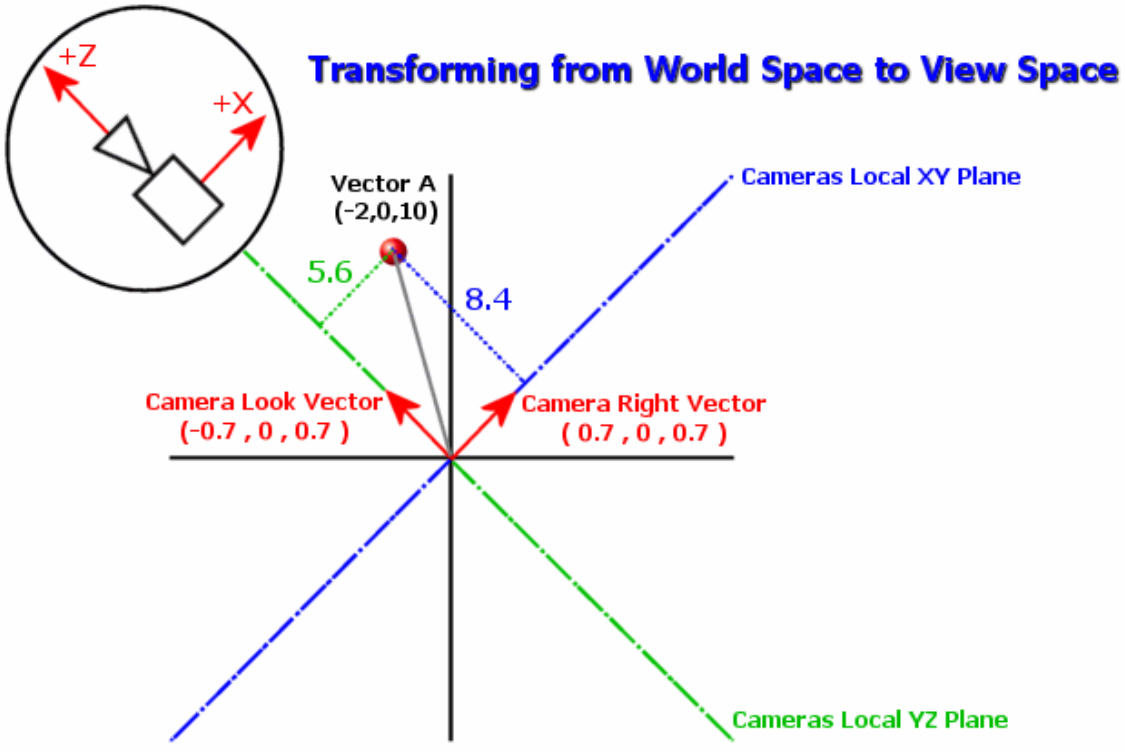


Figure 4.8

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 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.

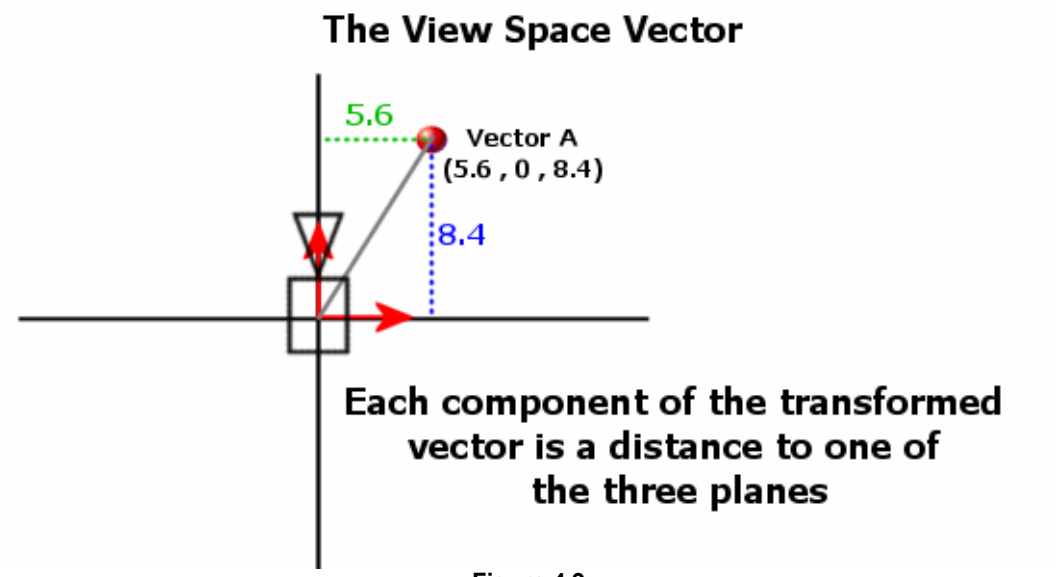


Figure 4.9

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.

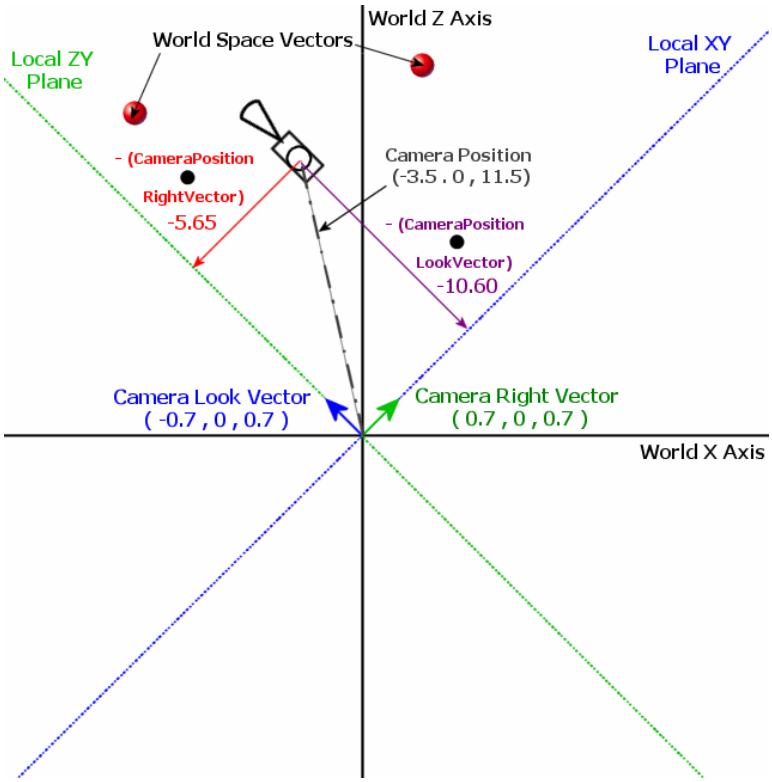
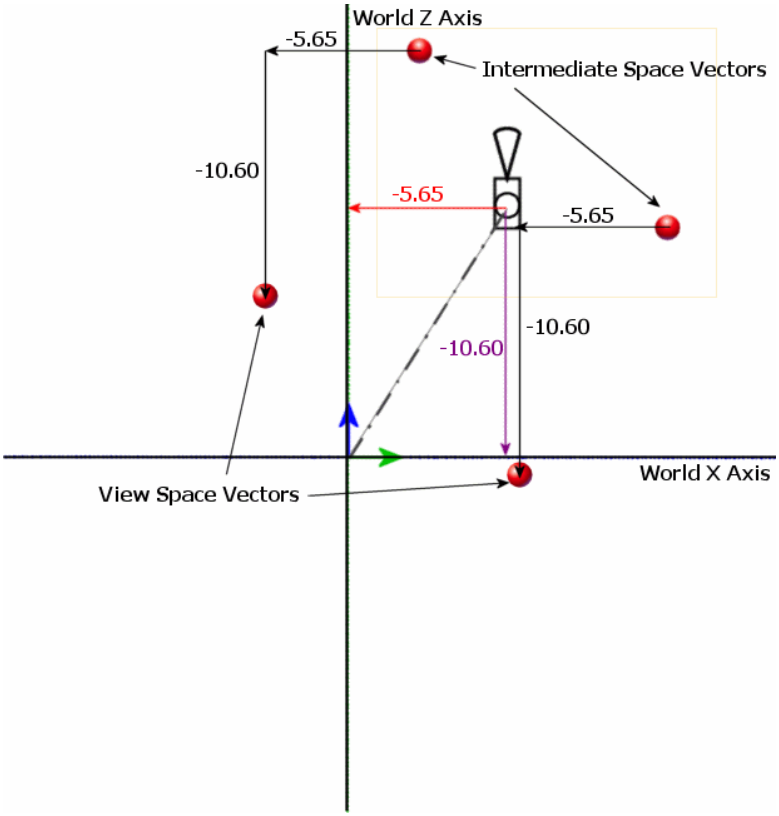


Figure 4.10

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.



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.

Figure 4.11

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.

4.2 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.

4.2.1 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:

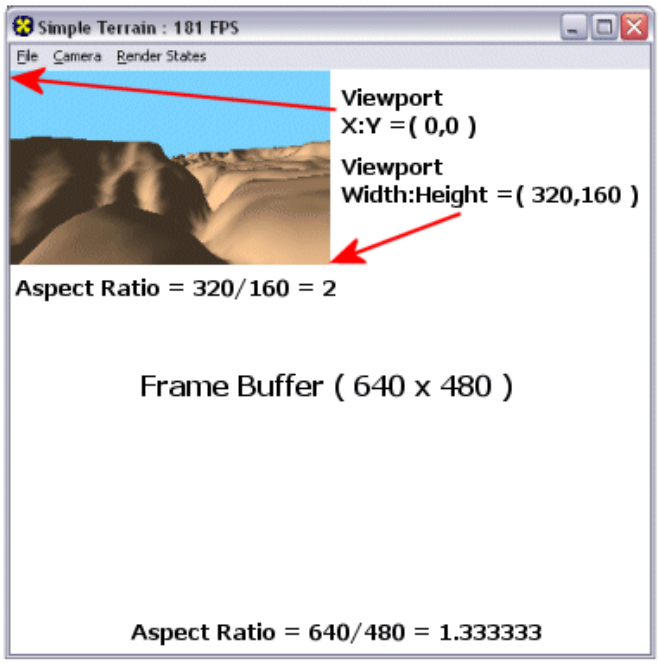


Figure 4.12

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.

4.2.2 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 1.3333333, 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.

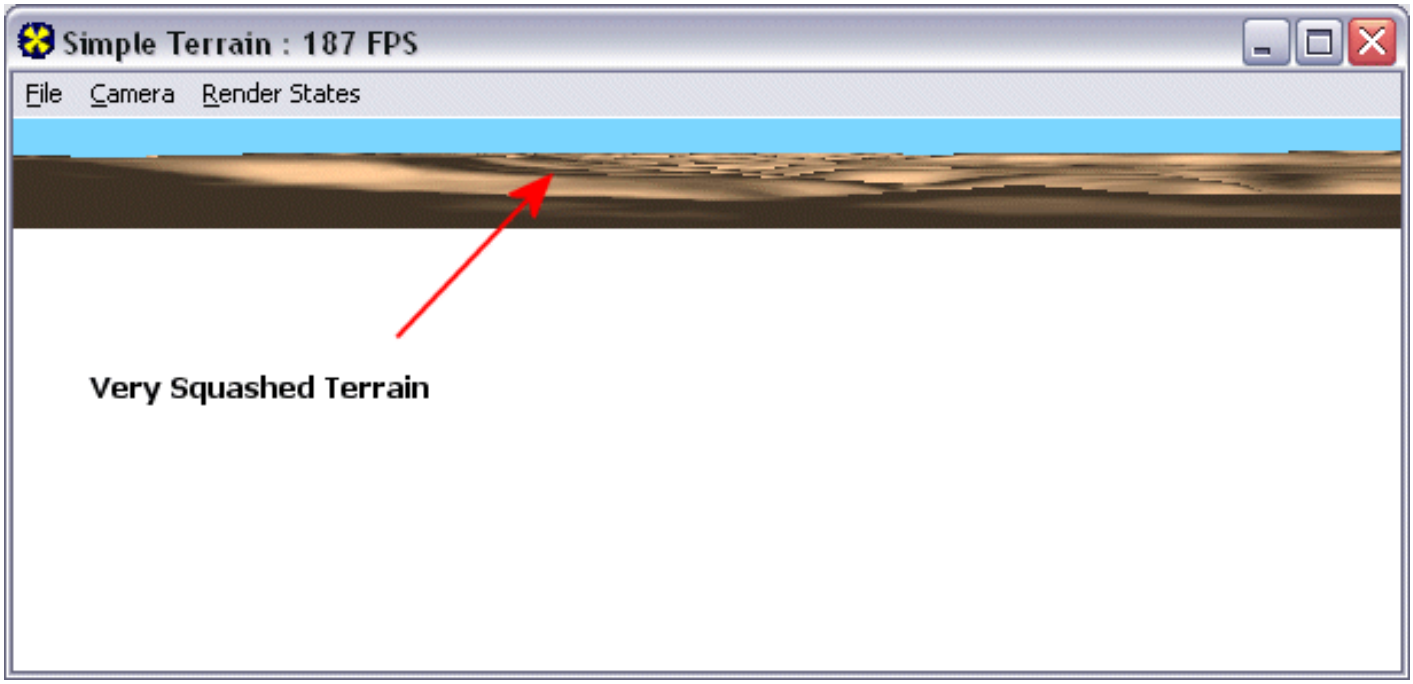


Figure 4.13

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.

4.3 Camera Systems

4.3.1 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 the 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.

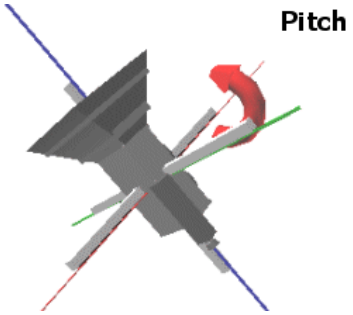


Figure 4.14

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

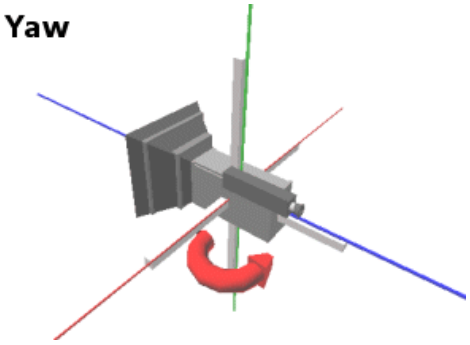


Figure 4.15

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);
```

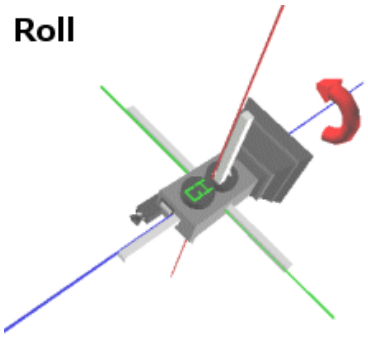


Figure 4.16

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.

4.3.2 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 in 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:

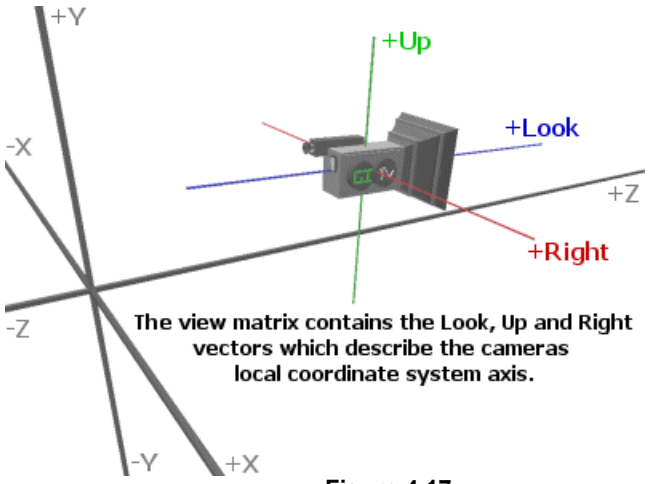


Figure 4.17

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:

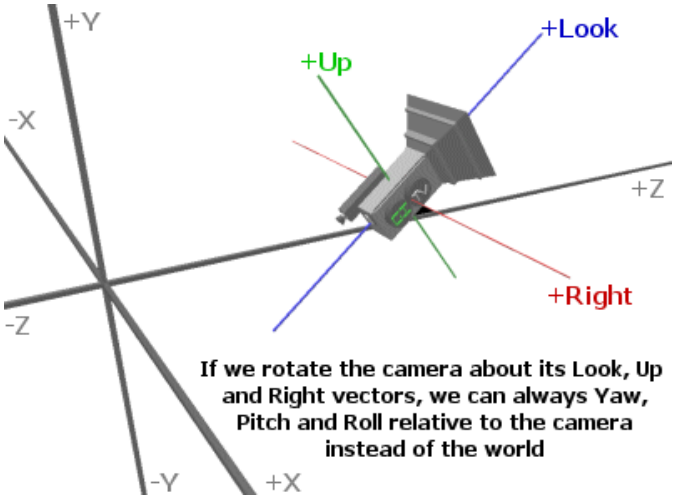


Figure 4.18

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.

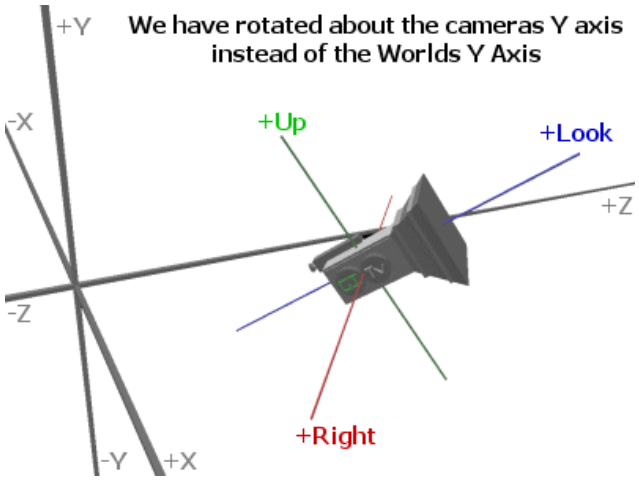


Figure 4.19

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.

4.3.3 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 3.141594444444444, 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:

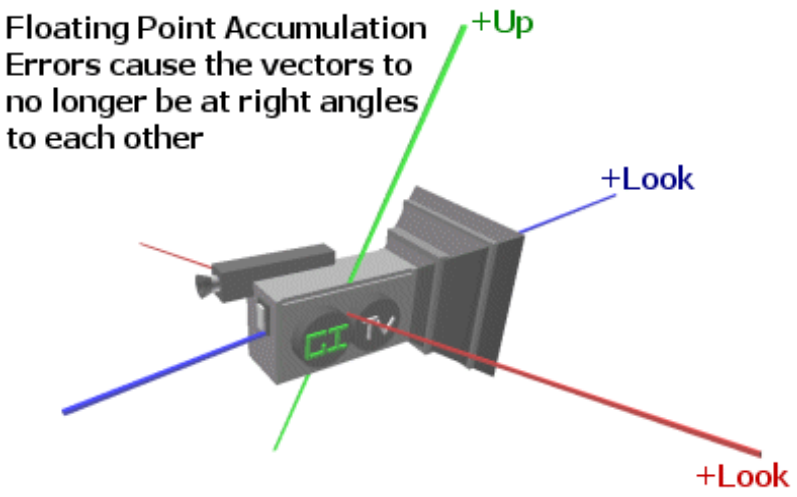


Figure 4.20

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 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.

4.3.4 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 to yaw around the *world* Y axis instead of the camera Y axis. Fig 4.21 demonstrates this concept.

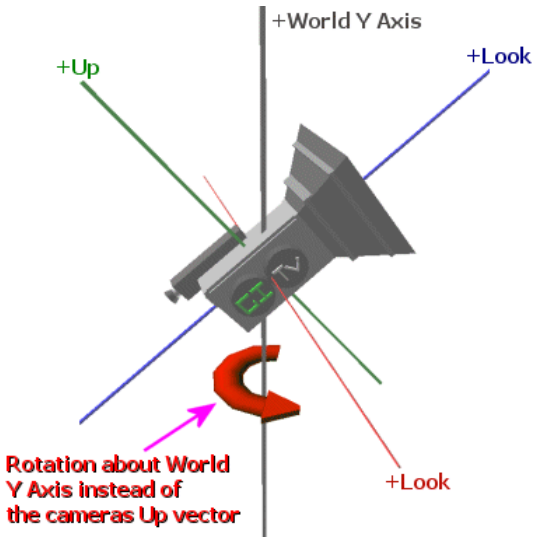


Figure 4.21

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.

4.3.5 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 ¼ 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.

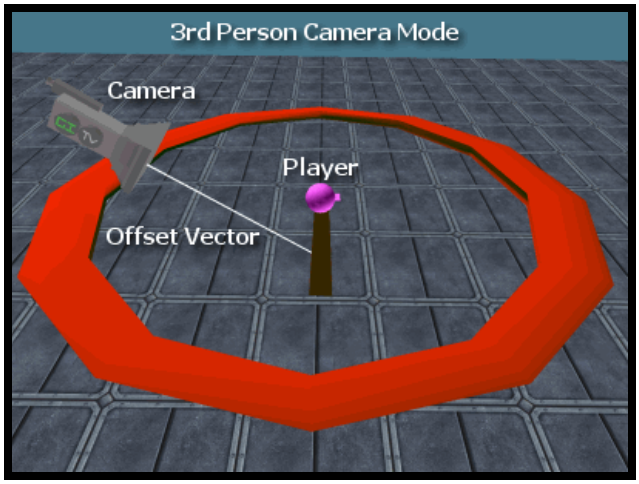


Figure 4.22

4.4 The View Frustum

As discussed earlier in the course, the projection matrix defines a camera view volume. Recall that the 1st and 2nd columns of the projection matrix store values that define the angle of the horizontal and vertical field of view respectively. We also use the 3rd 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).

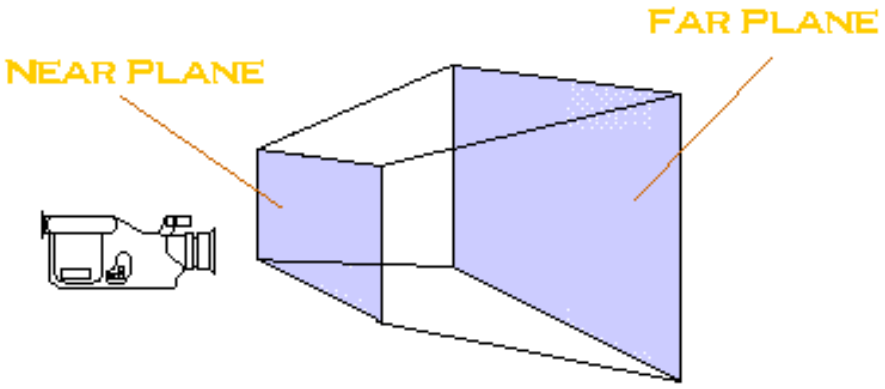


Figure 4.23

It could be said that the 1st 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 2nd 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 3rd 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.

4.4.1 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.

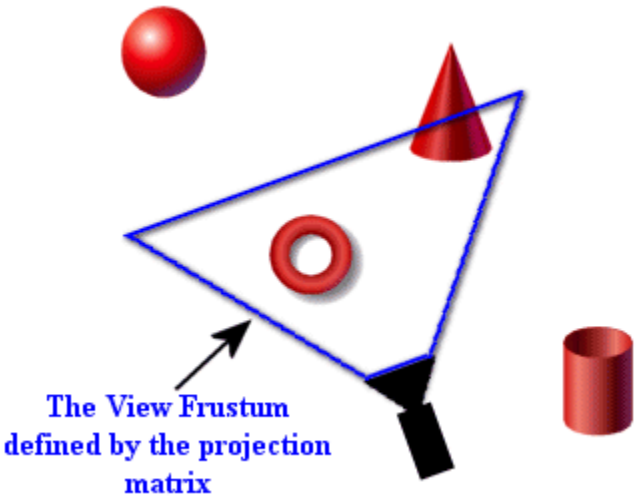


Figure 4.24

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 defense 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.

4.4.2 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.

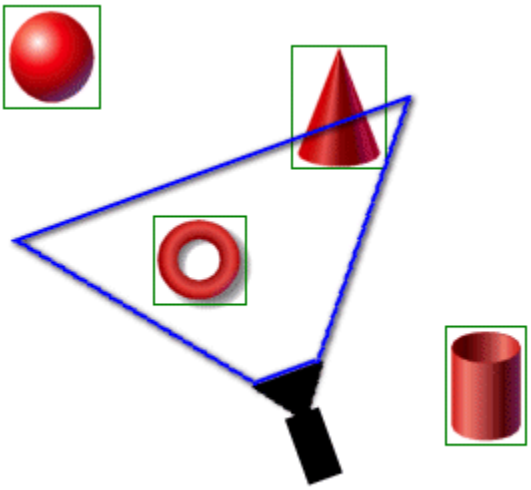


Figure 4.25

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.

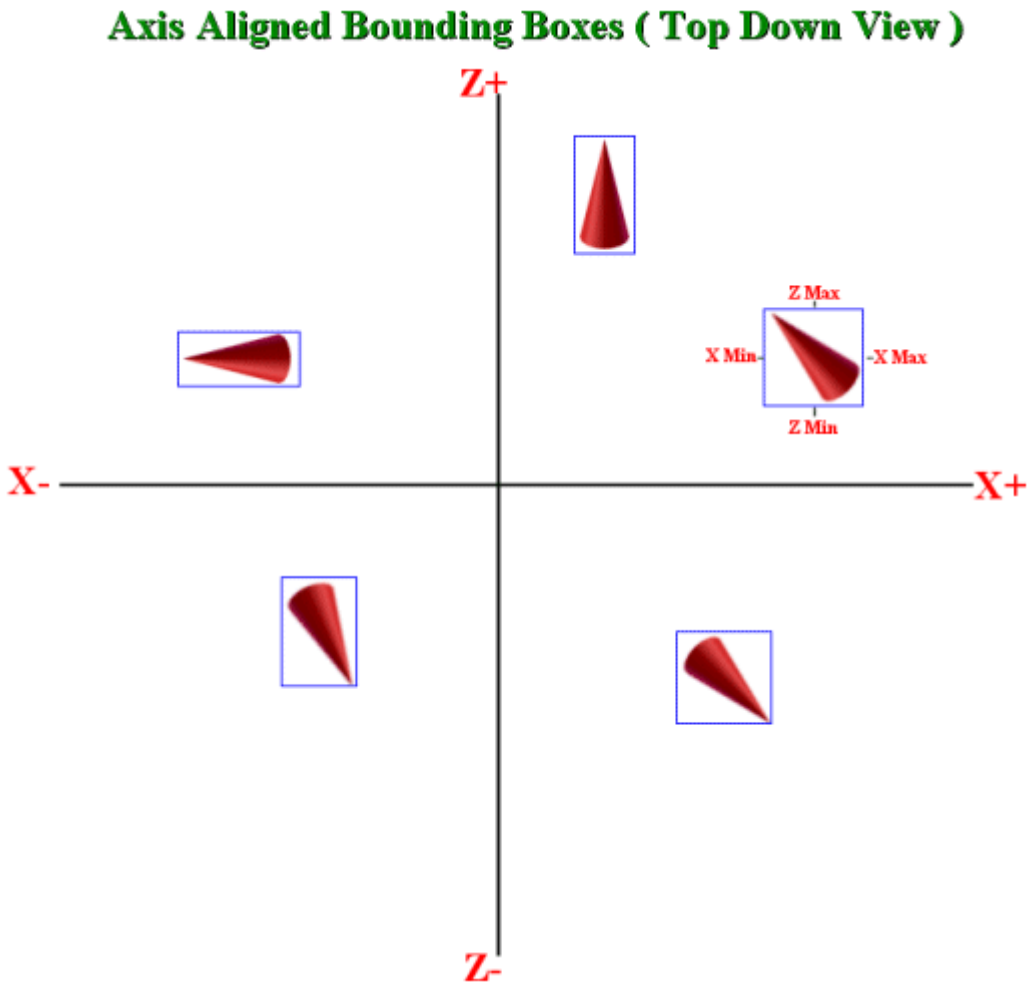


Figure 4.26

Recalculating an AABB can be costly if the mesh consists of thousands of vertices. This is because calculating an AABB requires our 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 vector components to very large negative 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.

4.4.3 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.

$$\text{Ratio} = \frac{FarPlane}{(FarPlane - NearPlane)} = \frac{100.0}{90} = 1.11111111$$

-Ratio * NearPlane = -(10 * 1.11111111) = -11.11111111

$$AspectRatio = \frac{ScreenWidth}{ScreenHeight} = \frac{800}{600} = 1.33333333$$

$$m11 = \frac{\frac{1}{\tan(60/2)}}{AspectRatio} = 1.299038109$$

$$m22 = \frac{1}{\tan(60/2)} = 1.732050808$$

$$M = \begin{array}{c} \begin{matrix} \boldsymbol{X} & \boldsymbol{Y} & \boldsymbol{Z} & \boldsymbol{W} \end{matrix} \\ \begin{bmatrix} 1.299038109 & 0 & 0 & 0 \\ 0 & 1.732050808 & 0 & 0 \\ 0 & 0 & 1.11111111 & 1 \\ 0 & 0 & -11.11111111 & 0 \end{bmatrix} \end{array}$$

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.

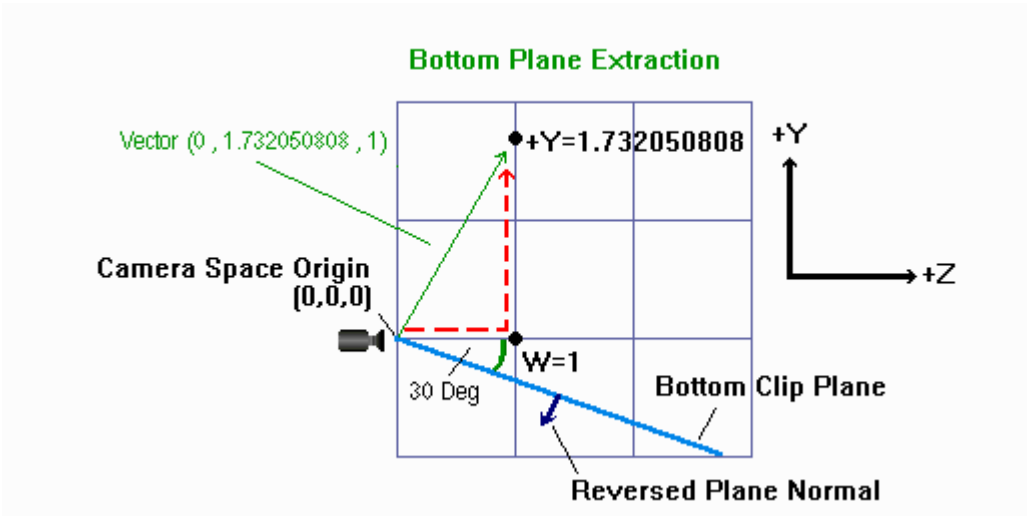


Figure 4.27

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:

X Y W

Column_4 + 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 labeled '*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:

	X	Y	W
Column_4 – 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.

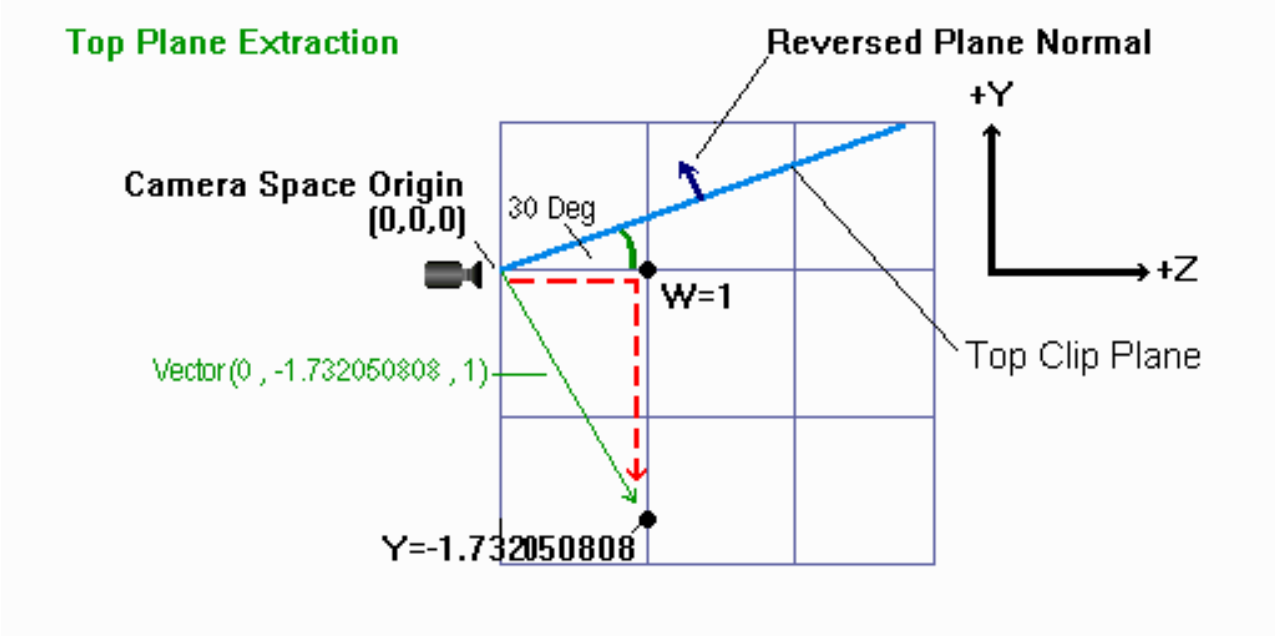


Figure 4.28

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.

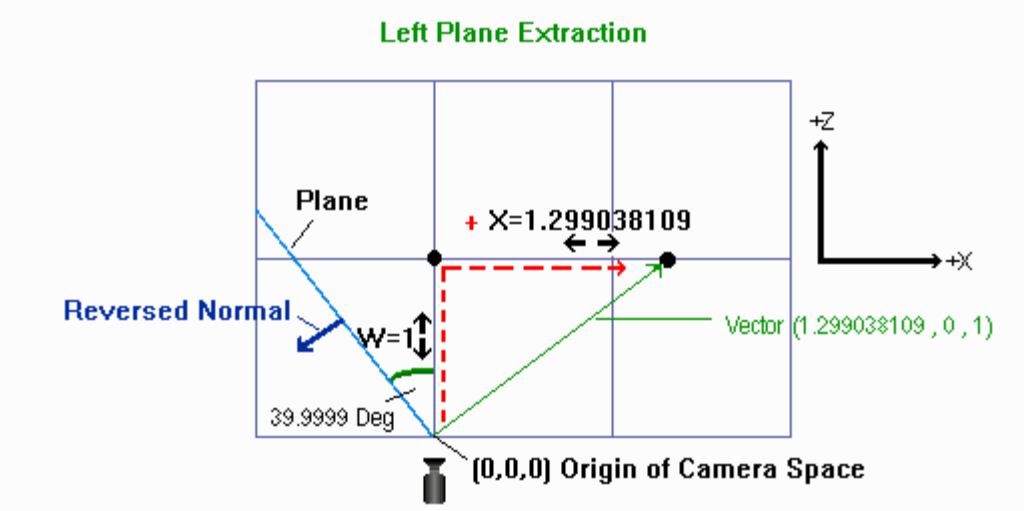


Figure 4.29

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:

$$M = \begin{bmatrix} 1.299038109 & 0 & 0 & 0 \\ 0 & 1.732050808 & 0 & 0 \\ 0 & 0 & 1.11111111 & 1 \\ 0 & 0 & -11.11111111 & 0 \end{bmatrix} \quad \begin{matrix} & & \mathbf{Z} & \mathbf{W} \end{matrix}$$

First let us see what subtracting the first three rows of W and Z produces as a direction vector:

-(Column_4 – Column_3) = -((0,0,1)-(0,0,1.11111111)) = (0 , 0 , 0.11111111)

This result certainly seems correct, for if the vector were normalized it would become:

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):

Distance = -(m44 – m43) = -(0 - -11.11111111)= -11.1111111111

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 One 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:

$$VectorLength = \sqrt{X^2 + Y^2 + Z^2}$$

If for example, we had a vector of (3,0,9), the length of that vector would be calculated as:

$$VectorLength = \sqrt{(3*3) + (0*0) + (9*9)} = 9.4868329$$

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:

$$UnitVector = \left(\frac{3}{9.4868329}, \frac{0}{9.4868329}, \frac{9}{9.4868329} \right) = (0.316227768,0,0.9486833)$$

Just to verify that this worked and that we indeed have a unit normal, let us calculate its magnitude:

$$VectorLength = \sqrt{(0.316227768*0.316227768) + (0*0) + (0.9486833*0.9486833)} = 1.0$$

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:

Far Plane->Normal.x = - (m14 - m13) = - (0 – 0) =0
Far Plane->Normal.y = - (m24 - m23) = - (0 – 0) =0
Far Plane->Normal.z = - (m34 - m33) = - (1 - 1.11111111) =0.11111111
Far Plane->Distance = - (m44 – m43)= - (0 - -11.11111111) = -11.1111111111

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 3rd column of the matrix and we have our near plane.

Our 3rd 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, -1.11111111)**. Remember that the value **1.11111111** was our scaling ratio to map the input z value to a Z-Buffer value. Also recall that the 4th 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).

$$\text{Vector Length}=\sqrt{0.0^2+0.0^2+-1.11111111^2}=1.11111111$$

$$\text{Near Plane->Normal}=\left(\frac{0}{1.11111111},\frac{0}{1.11111111},\frac{-1.11111111}{1.11111111}\right)=\textbf{(0, 0, -1)}$$

$$\text{Near plane->Distance}=\frac{11.11111111}{1.11111111}=\textbf{10.0}$$

4.4.4 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 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.

4.4.5 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

$$\begin{matrix} \textit{RIGHT} & \textit{UP} & \textit{LOOK} & \\ \begin{bmatrix} 0.707107 & 0 & 0.707107 & 0 \\ 0 & 1 & 0 & 0 \\ -0.707107 & 0 & 0.707107 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{matrix}$$

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.

$$\begin{matrix} \textit{RIGHT} & \textit{UP} & \textit{LOOK} & \\ \begin{bmatrix} 0.707107 & 0 & 0.707107 & 0 \\ 0 & 1 & 0 & 0 \\ -0.707107 & 0 & 0.707107 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{matrix} \times \begin{matrix} \textit{X} & \textit{Y} & \textit{Z} & \textit{W} \\ \begin{bmatrix} 1.299038109 & 0 & 0 & 0 \\ 0 & 1.732050808 & 0 & 0 \\ 0 & 0 & 1.11111111 & 1 \\ 0 & 0 & -11.11111111 & 0 \end{bmatrix} \end{matrix}$$

View Matrix **Projection Matrix**

The resulting combined matrix is shown next:

$$\begin{matrix} \textit{X} & \textit{Y} & \textit{Z} & \textit{W} \\ \begin{bmatrix} 0.918559 & 0 & 0.785674 & 0.707107 \\ 0 & 1.732051 & 0 & 0 \\ -0.918559 & 0 & 0.785674 & 0.707107 \\ 0 & 0 & -11.111111 & 0 \end{bmatrix} \end{matrix}$$

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:

Left Plane->Normal.X = - (m14 + m11) = ***1.62566***
Left Plane->Normal.Y = - (m24 + m21) = 0
Left Plane->Normal.Z = - (m34 + m31) = ***-0.211452***
Left Plane->Distance = - (m44 + m41) = 0

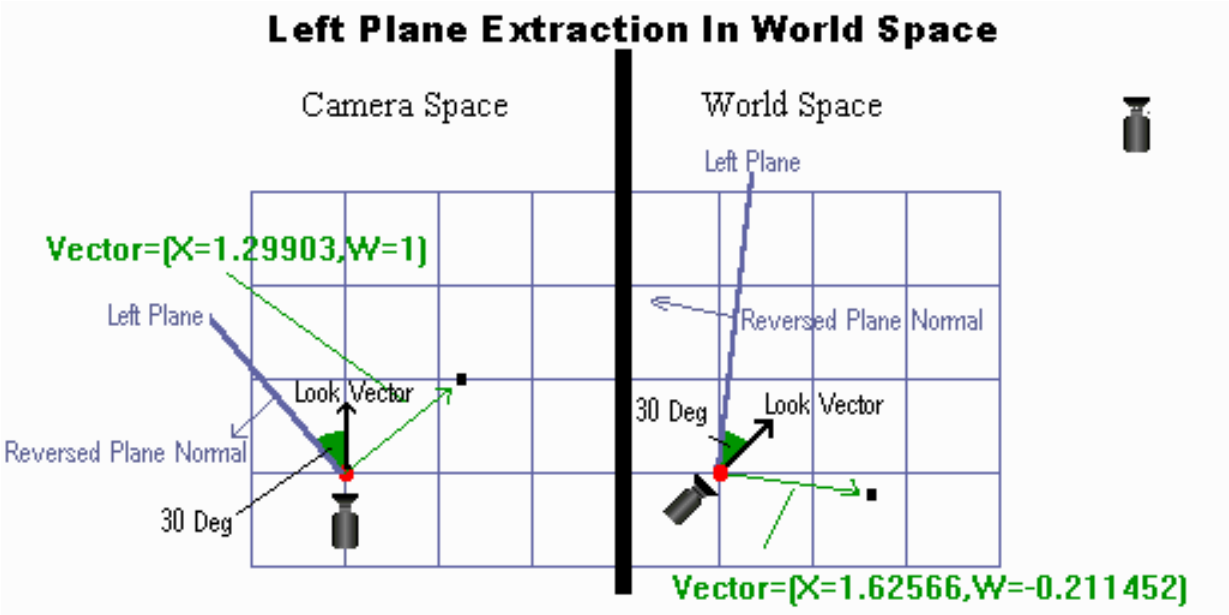


Figure 4.30

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.

4.4.6 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:

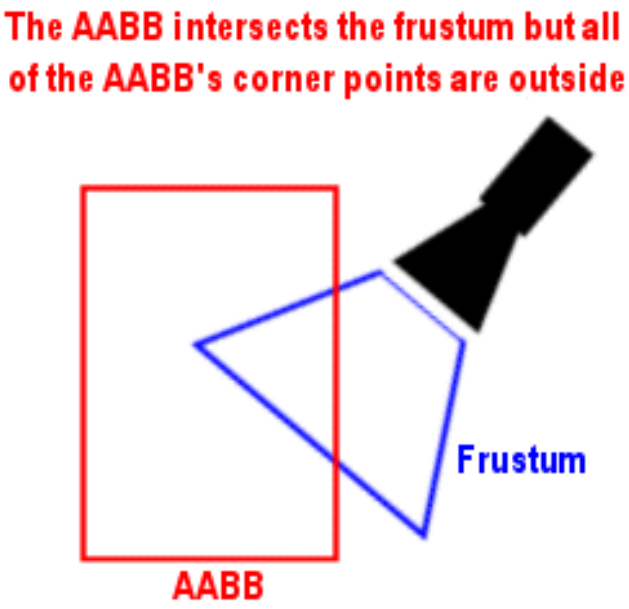


Figure 4.31

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.

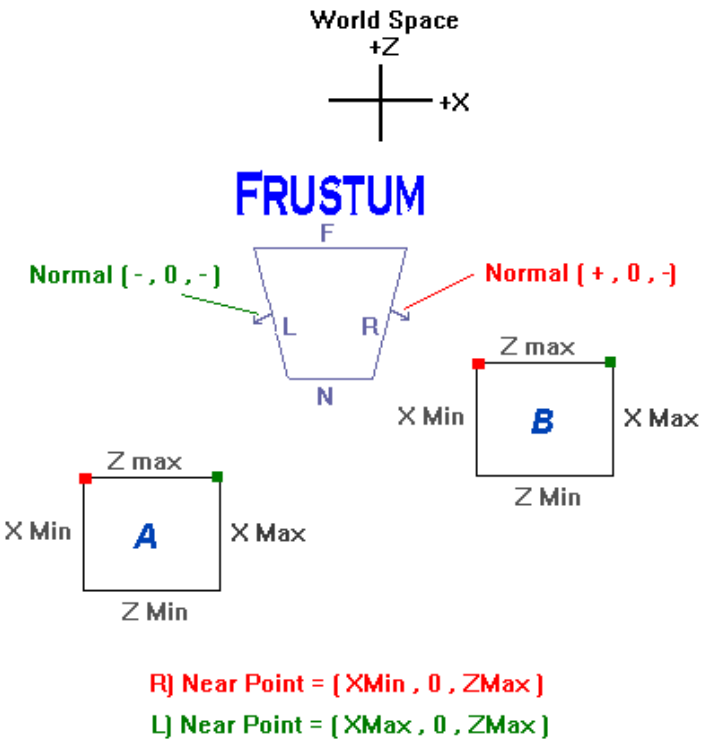
- If the 'y' component of the plane normal is negative, then we use the bounding box's y maximum point as our near point's y component. Otherwise, we use the bounding box's y minimum component.

- 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 AABB is 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.

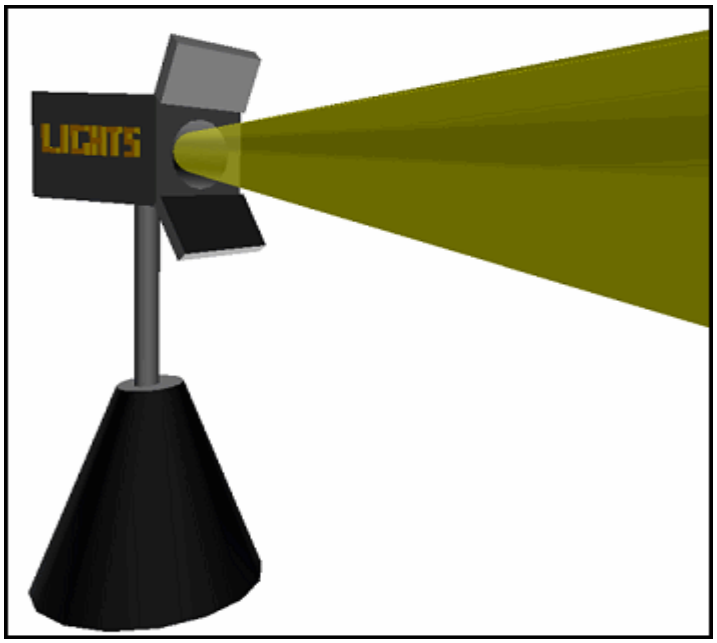
This concludes our coverage of frustum culling for this course. However, in the next course in this series, our frustum culling code will take a significant leap forward. Not only will it allow for more bounding volume types to be tested, but it will also add code that distinguishes between 'fully inside' 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.

Chapter Five:

Lighting



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 light bulb. 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.

5.1 Lighting Models

5.1.1 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.

5.1.2 Ambient 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.

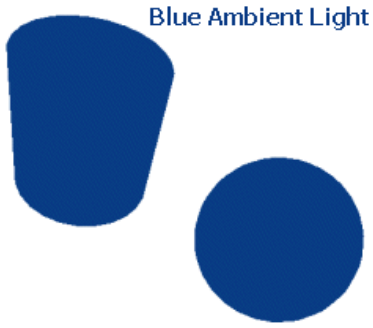


Figure 5.1

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.

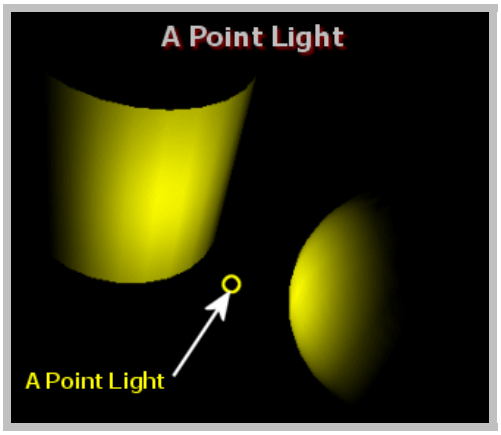


Figure 5.2

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:

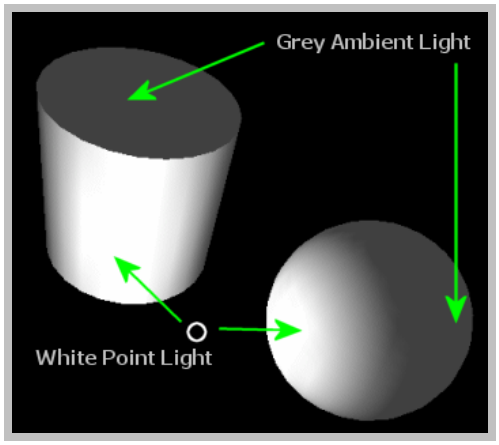


Figure 5.3

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.

5.1.3 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.

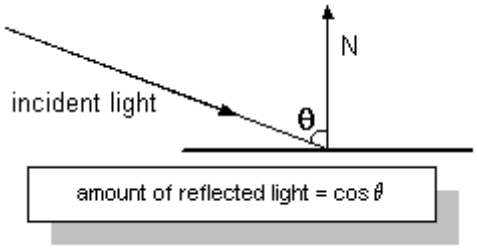


Figure 5.4

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.

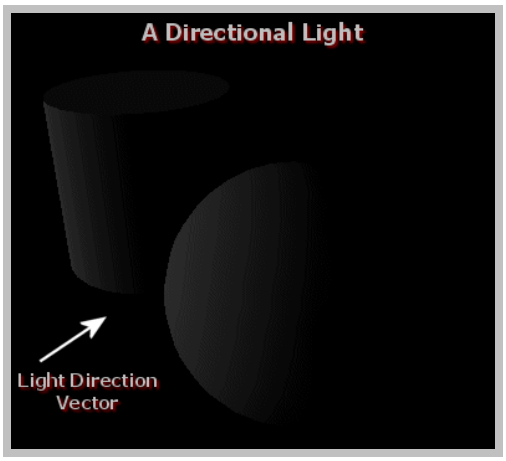


Figure 5.5

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.

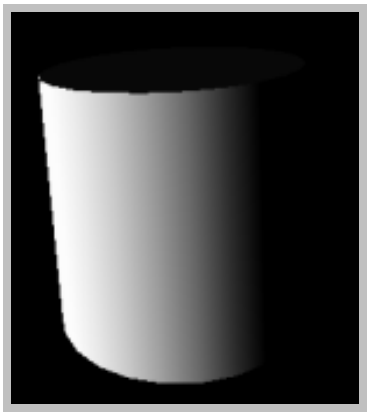


Figure 5.6

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.

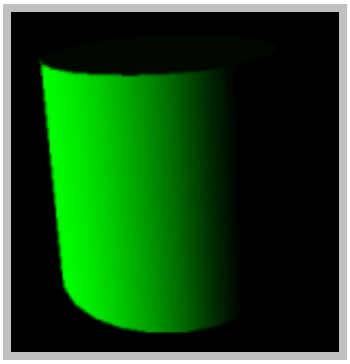


Figure 5.7

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.

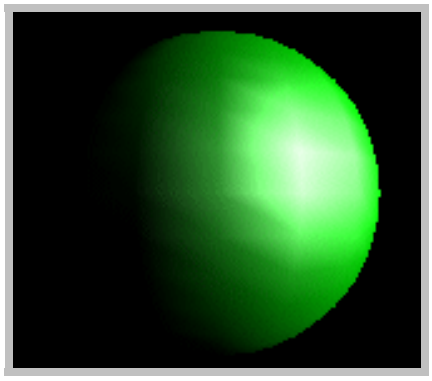


Figure 5.8

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.

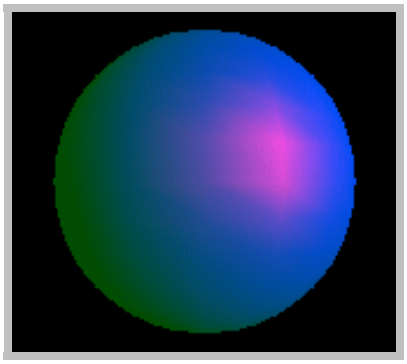


Figure 5.9

5.1.4 The Basic Lighting Equation

The total illumination (**I**) level of a vertex can now be described by:

$$\mathbf{I = A + D + S + E}$$

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.

5.2 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.

5.2.1 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.

5.2.2 Lighting Vertex Formats

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 GILES™ 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 GILES™ 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.

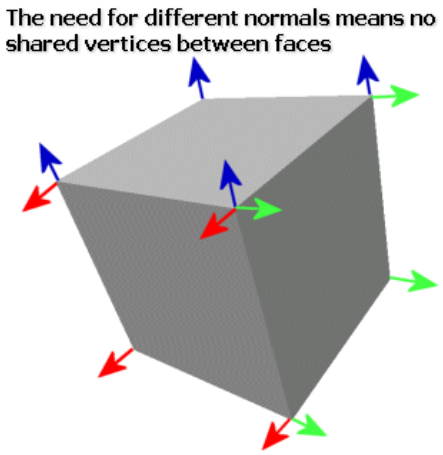


Figure 5.10

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 is 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.

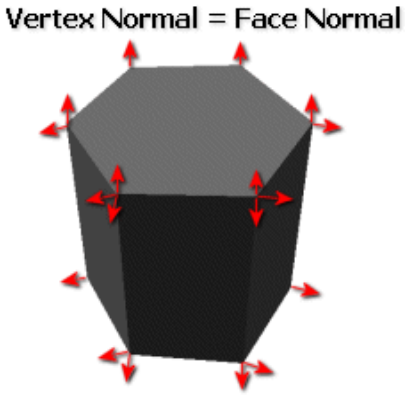


Figure 5.11

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.

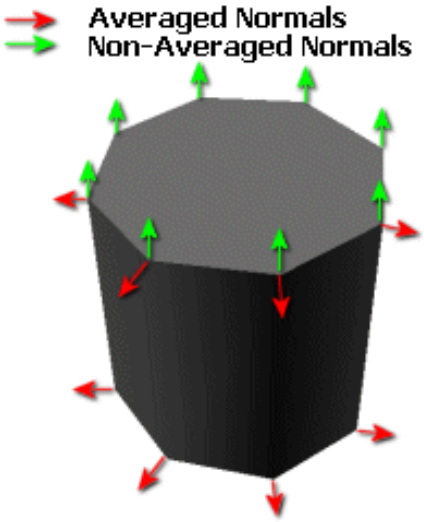


Figure 5.12

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 code snippet above 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 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 storing its own normal, the pipeline line has what it needs to accurately the 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.

5.2.3 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 off-putting 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 GILES™ 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.

5.3 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;
```

5.3.1 Point Lights

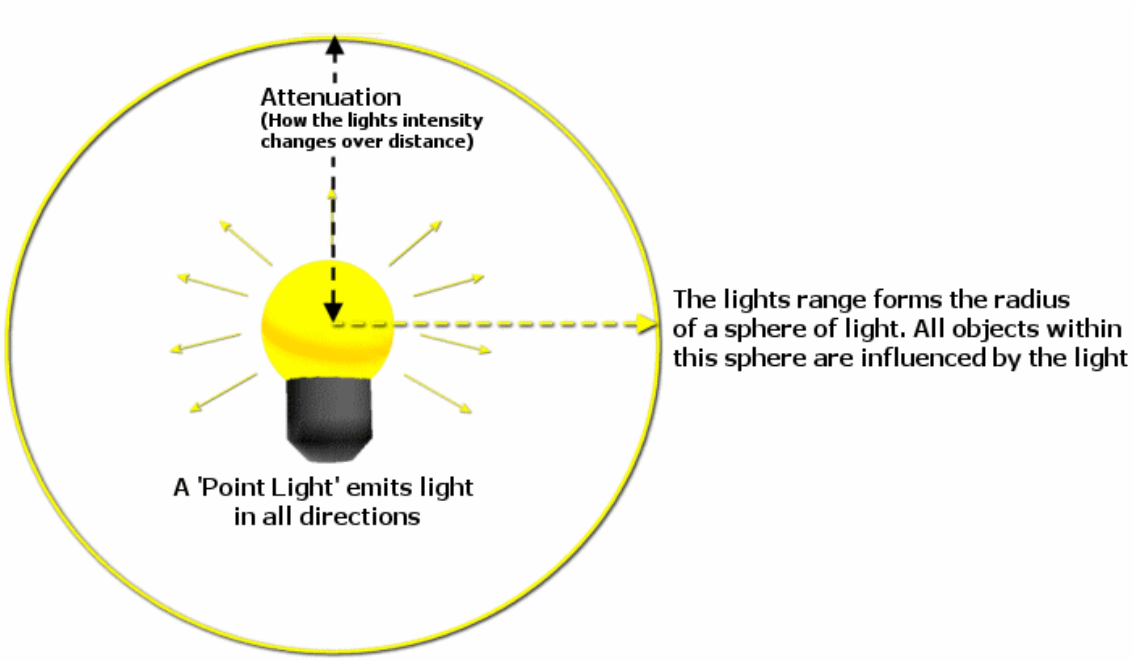


Figure 5.13

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.

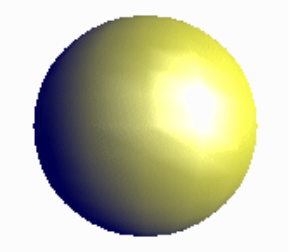


Figure 5.14

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:

$$\text{Color}_{(\text{current})} = \frac{1}{Attenuation1 + (Attenuation2 \times D) + (Attenuation3 \times D^2)} * \text{Color}_{(\text{orig})}$$

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:

Attenuation1 = 1.0 Attenuation2 = 0.0 Attenuation3 = 0.0

$$\text{Attenuation Value} = \frac{1}{1 + (0 \times D) + (0 \times D^2)} = 1.0$$

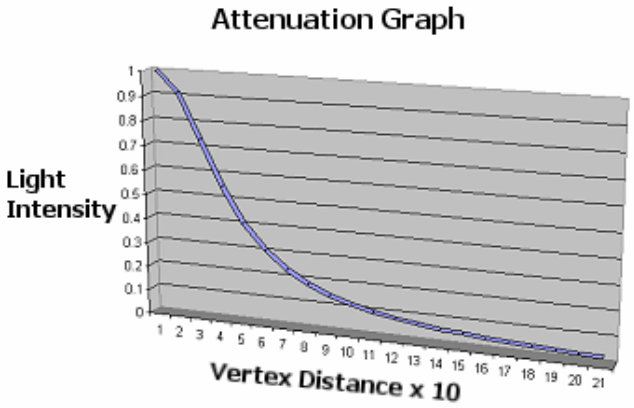
$$\text{Color}_{(\text{current})} = \text{Attenuation Value} * \text{Color}_{(\text{orig})}$$

$$\text{Color}_{(\text{current})} = \text{Color}_{(\text{orig})}$$

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: Attenuation1 = 1.0 Attenuation2 = 0.0002 Attenuation3 = 0.0009

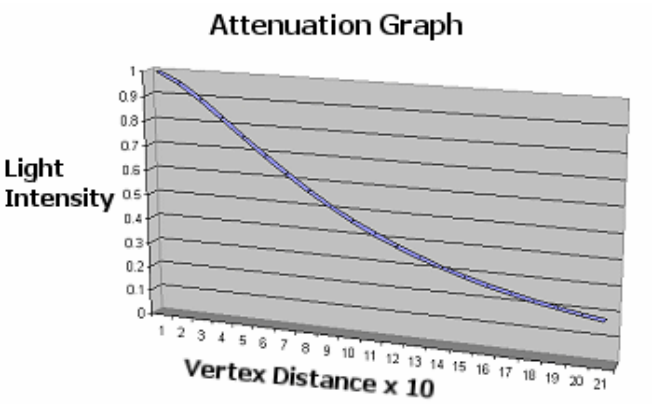


The above graph shows how the color emitted from the light would be scaled by distance. At a distance of 200 units it is close enough to zero not to produce a noticeable illumination discontinuity at the edge of the light's range. Changing the attenuation values allows you to modify the shape of the 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.

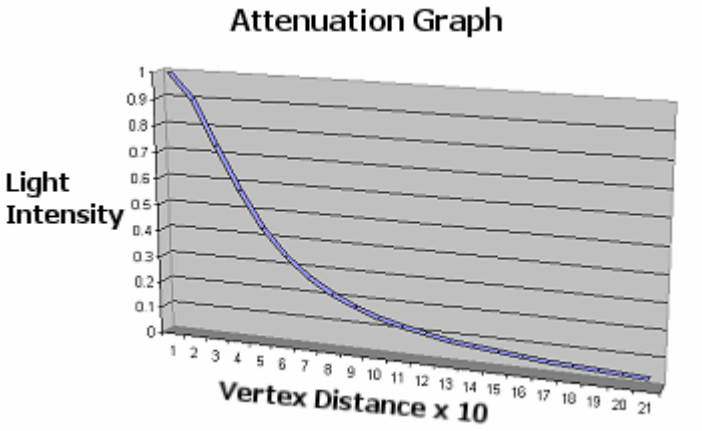
Example 2: **Attenuation1 = 1.0 Attenuation2 = 0.004 Attenuation3 = 0.0001**



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
```

5.3.2 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.

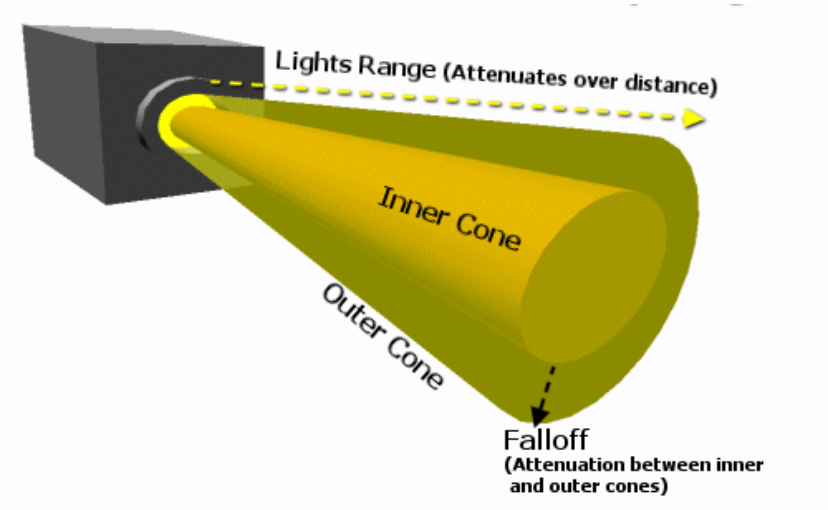


Figure 5.15

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*.

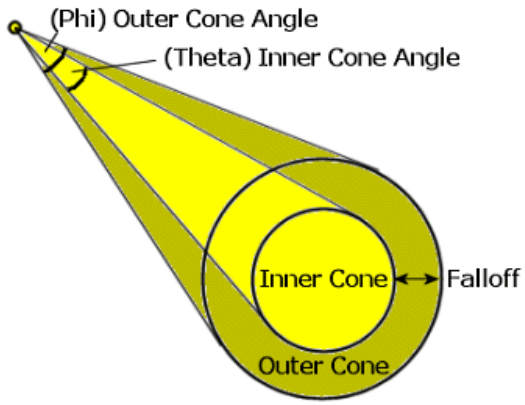


Figure 5.16

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 not any 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.

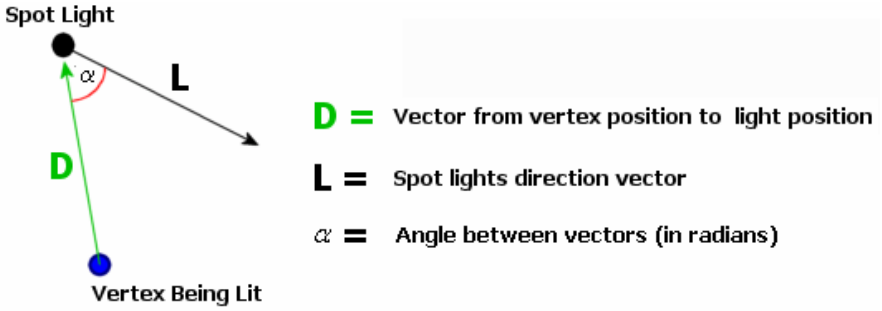


Figure 5.17

The vector **L** is negated and the dot product performed to return the cosine of the angle between the two vectors.

$$\cos\alpha = -L \bullet D$$

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 α because they describe total cone angles when we actually want the angle relative to the direction vector **L** at the center if the inner cone. The cones have an angle that sweeps out α /2 on each side of vector **L**.

After Phi and Theta have been divided by 2, the pipeline compares them with α to see if the vertex is within the cones. If α 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 α is compared against Theta/2. If α 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:

$$\mathbf{S} = \left(\frac{\cos(\alpha) - \cos(\phi)}{\cos(\theta) - \cos(\phi)} \right)^{Falloff}$$

α = Angle between vectors L and D
ϕ = Phi / 2 (Half the outer cone angle)
θ = 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.

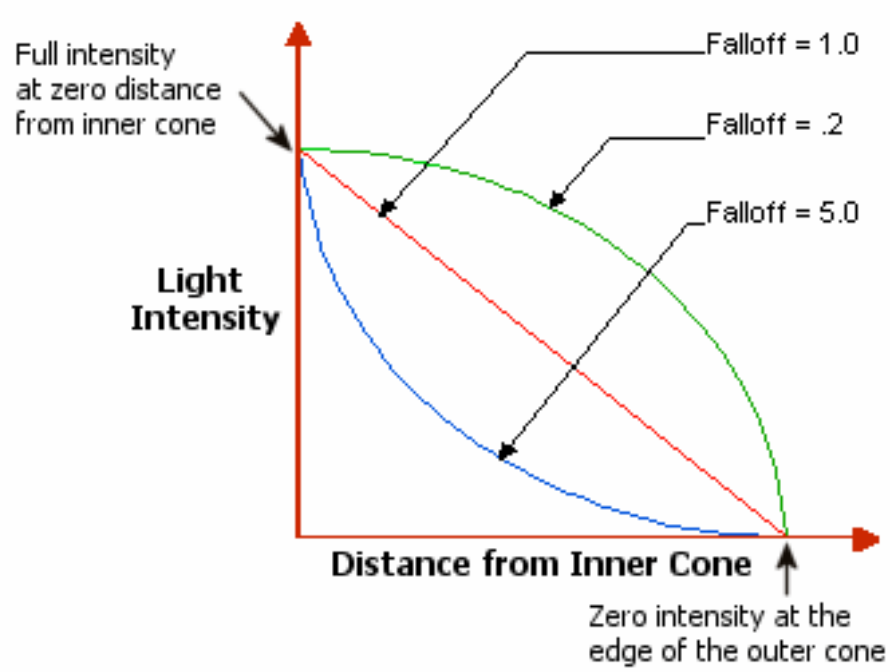


Figure 5.18

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.

5.3.3 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.

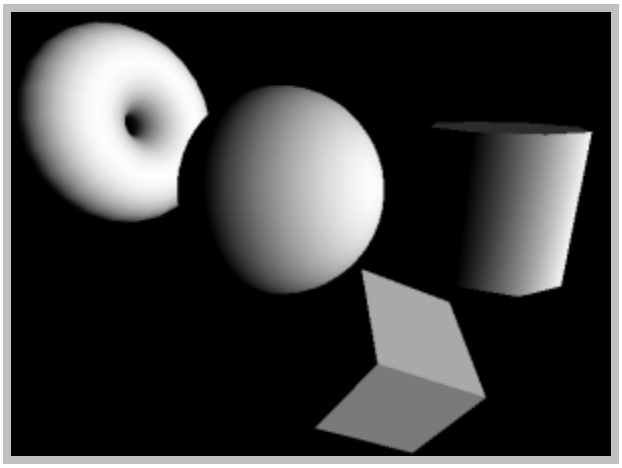


Figure 5.19

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.

5.4 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.

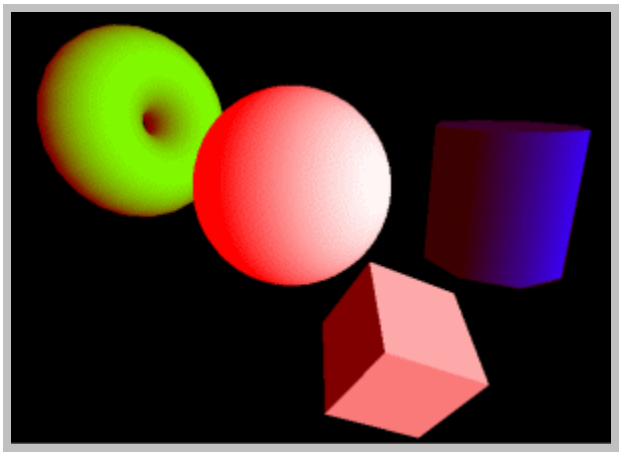


Figure 5.20

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:

Vertex Color = (AmbientLight * A) + (DiffuseLight * D) + (SpecularLight * S) + 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:

```
VertexColor =  (AmbientLight(a,r,g,b) * A(a,r,g,b))  +
               (DiffuseLight  (a,r,g,b) * D(a,r,g,b)) +
               (SpecularLight(a,r,g,b) * S(a,r,g,b)) + E(a,r,g,b)
```

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:

```
Vertex Color = (AmbientLight * A) ….
Vertex Color = (1.0, 1.0, 1.0, 1.0) * (1.0, 0.0, 0.5, 0.0)
Vertex Color = (1.0, 0.0, 0.5, 0.0)
```

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.

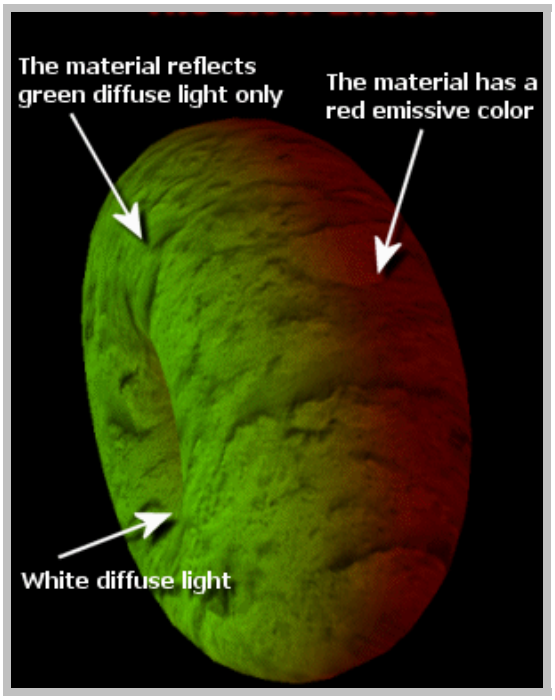


Figure 5.21

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.

5.4.1 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 render state:

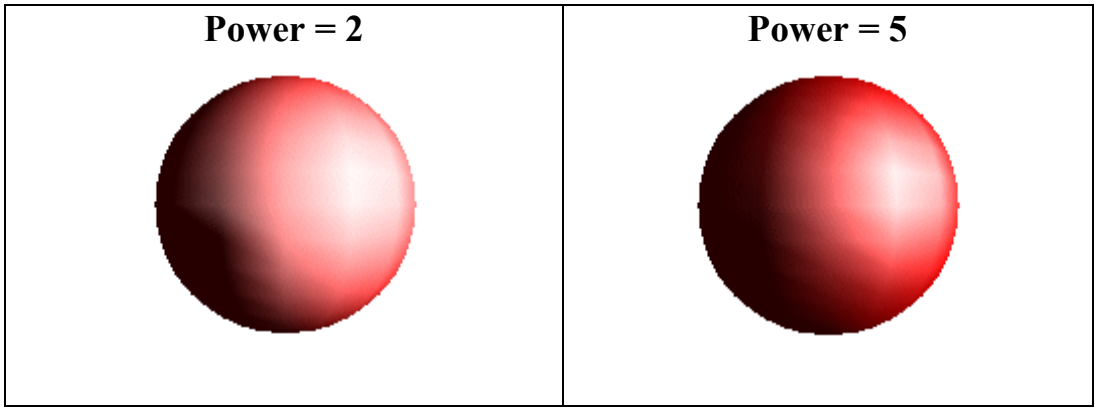
```
m_pDevice->SetRenderState (D3DRS_SPECULARENABLE , TRUE);
```

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** render state 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:



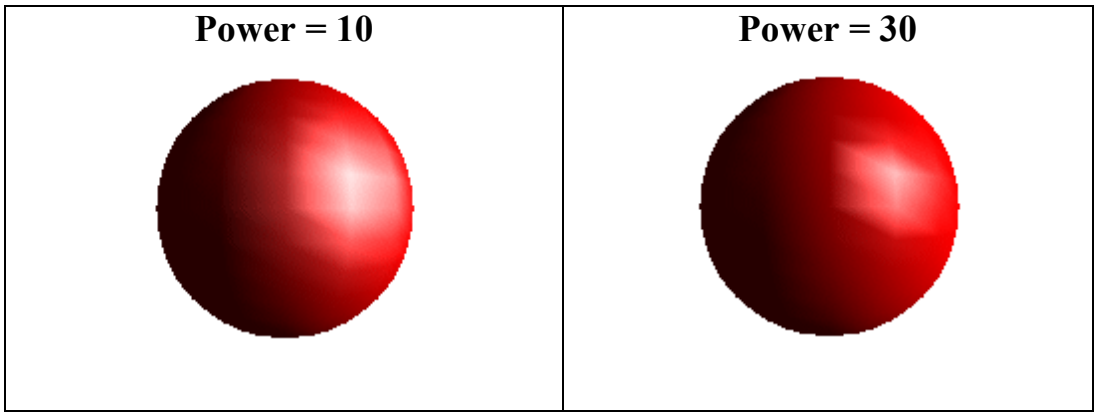


Figure 5.22

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.

5.4.2 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:

Vertex Color = (AmbientLight * A) + (DiffuseLight * D) + (SpecularLight * S) + 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 (A, D, S, and 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 (A) and diffuse (D) reflection, but use the two colors stored in each vertex as the specular (S) and emissive (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 A, D, S and 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_AMBIENTMATERUALSOURCE , D3DMATERIALCOLORSOURCE)
SetRenderState ( D3DRS_SPECULARMATERIALSOURCE , D3DMATERIALCOLORSOURCE)
SetRenderState ( D3DRS_EMISSIVEMATERUAL 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** render state 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** render state 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 render states. 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 Color Source Render State	**Default Value**	**Description**
D3DRS_DIFFUSEMATERIALSOURCE	D3DMCS_COLOR1	If the vertex has a diffuse component (**D3DFVF_DIFFUSE**), this color is used as the diffuse reflectance property (**D** 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 (**D3DFVF_SPECULAR**), this color is used as the specular reflectance property (**S** 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 **A** in the lighting equation.
D3DRS_EMISSIVEMATERIALSOURCE	D3DMCS_MATERIAL	The Emissive member of the currently set material is used as **E** 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_EMISSIVEMATERIALSOURCE , 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_EMISSIVEMATERIALSOURCE , 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.

5.5 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 GILES™ level editor certainly allows you to do all these things quickly and easily. Because GILES™ 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 GILES™ is almost identical to setting them in DirectX. So you can more easily understand all of our discussions simply by firing up GILES™ and experimenting. Using GILES™ to create you 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 GILES™ 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.

5.6 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 mesh has a spotlight shining on it.

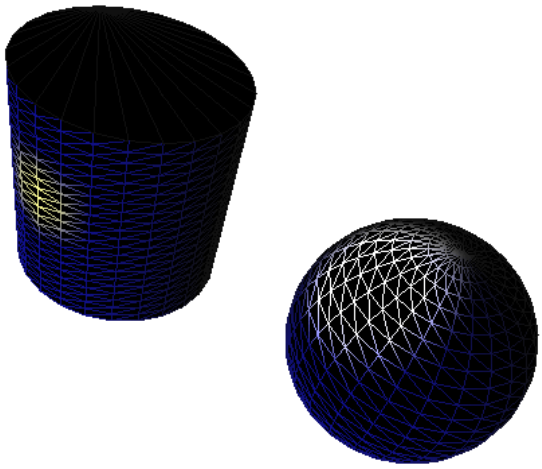


Figure 5.23

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:

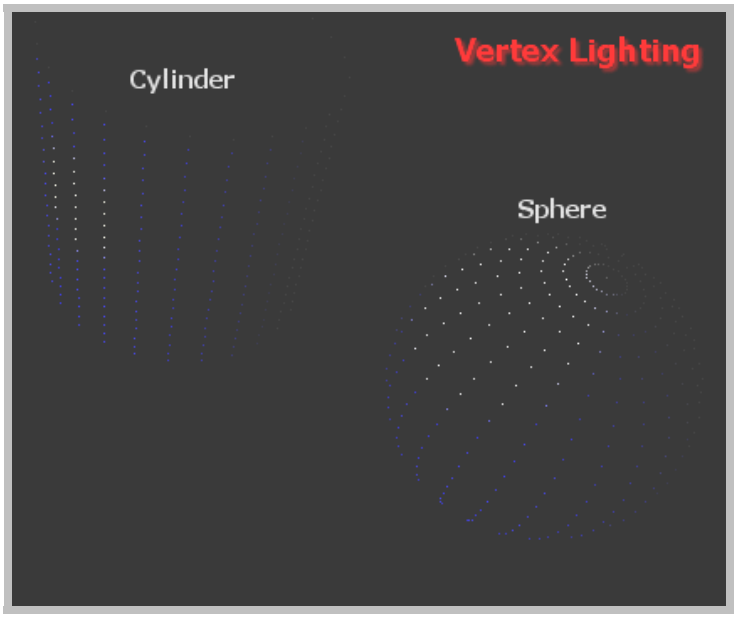


Figure 5.24

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.

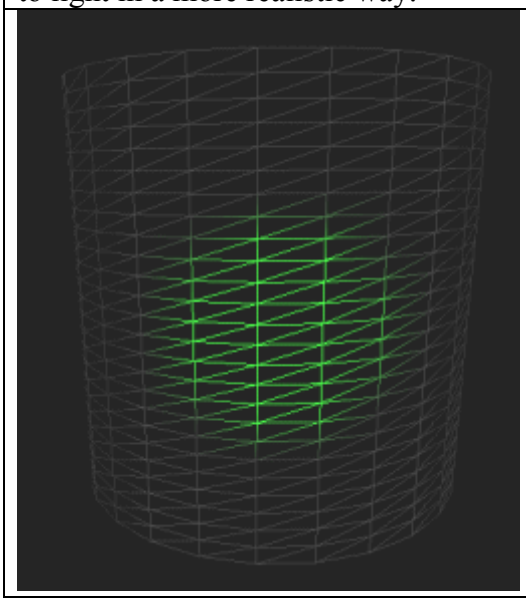
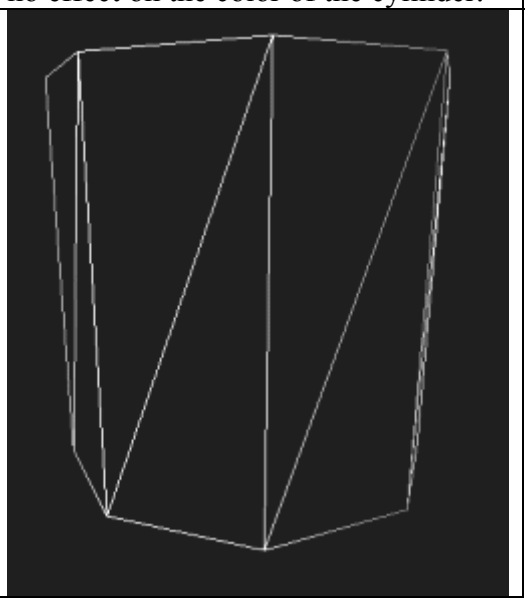
High Vertex Count Cylinder	**Low Vertex Count Cylinder**
Vertices inside the cone receive the green light. Because the surface of the cylinder is made up of many vertices in close proximity, the surface reacts to light in a more realistic way.	Vertices are placed only at the top and bottom of the cylinder and as such, none of the vertices fall within the cone of light. Therefore, the light has no effect on the color of the cylinder.
	

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.

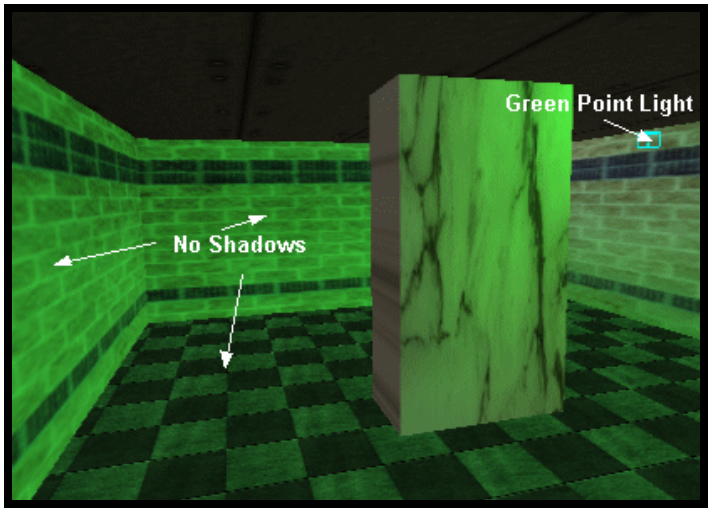


Figure 5.26

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 GILES™

Chapter Six:

Texture Mapping

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 it 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.

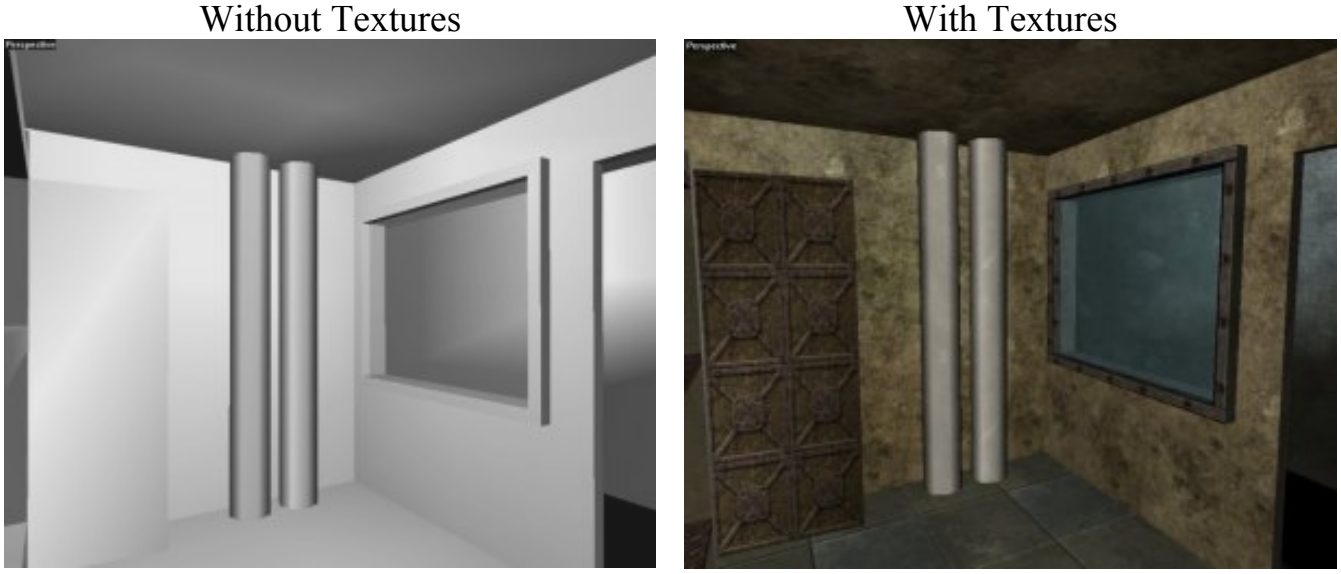


Figure 6.2

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.

6.1 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.

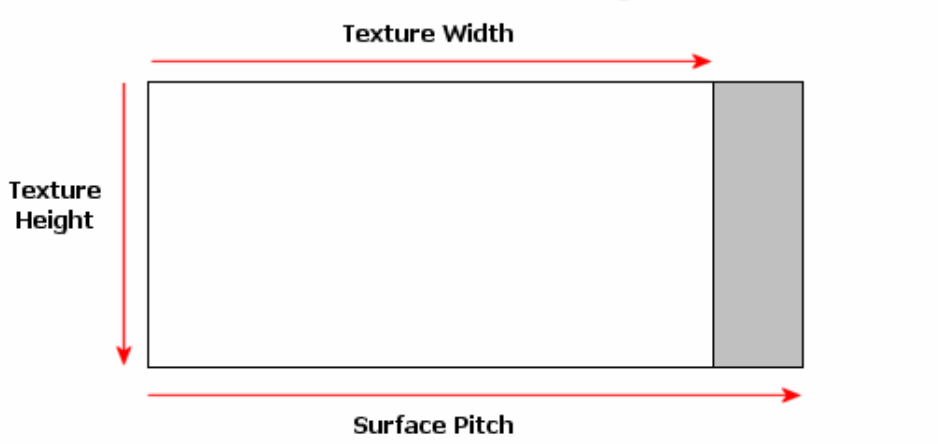


Figure 6.4

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.

6.1.1 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

D3DFORMAT Member	Description
32-bit Surface Formats	
D3DFMT_A8R8G8B8	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.
D3DFMT_X8R8G8B8	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 to not support 24 bit textures so this allows us to use a 32-bit surfaces to store 24 bit color.
D3DFMT_A2B10G10R10	2 bits for alpha, 10 bits for each red, green and blue.
D3DFMT_A8B8G8R8	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.
D3DFMT_X8B8G8R8	8 bits for red, green and blue stored in XBGR format. There are 8 bits of each pixel not used (X8)
D3DFMT_G16R16	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.
D3DFMT_A2R10G10B10	2 bits for alpha and 10 bits for each red, green and blue component. Rarely used.
16-bit Surface Formats	
D3DFMT_R5G6B5	This 16-bit RGB format has 5 bits reserved for red, 6 bits for green and 5 bits for blue.
D3DFMT_X1R5G5B5	5 bits for each red, green and blue component with one bit unused.
D3DFMT_A1R5G5B5	This ARGB surface stores 1 bit for alpha and 5 bits for each red, green and blue component.
D3DFMT_A4R4G4B4	4 bits for each alpha, 4 bits for red, 4 bits for green and 4 bits for blue.
D3DFMT_A8R3G3B2	Rarely used ARGB format with 8 bits for alpha, 3 bits each for both red and green color components and 2 bits for blue.
D3DFMT_X4R4G4B4	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 covert 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_SYSLOCK
- 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)
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 = 00000000111111110000000000000000 // 0x00FF0000

                                  &

SrcColor = 11111111001100100110110010010100 // 0xFF326C94

                                  =

TempColor = 00000000001100100000000000000000 // 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 = 00000000001100100000000000000000 // 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 color 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 = 0000111100000000 // 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;
```

6.1.2 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.

6.2 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:

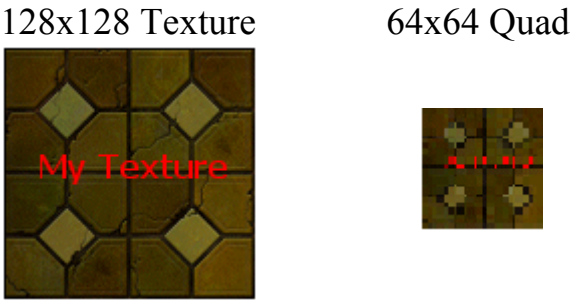


Figure 6.5

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 problem 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.
	

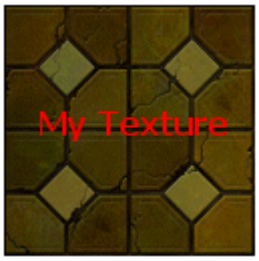
Figure 6.6

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.

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.

	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.
	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.

6.3 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.

6.3.1 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
);
```

LPDIRECT3DEVICE9 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:

128 x 128	**Level 0**
64 x 64	**Level 1**
32 x 32	**Level 2**
16 x 16	**Level 3**
8 x 8	**Level 4**
4 x 4	**Level 5**
2 x 2	**Level 6**
1 x 1	**Level 7**

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 use 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.

D3DX_FILTER_NONE	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.
D3DX_FILTER_POINT	Each texel gets its color from the nearest equivalent pixel in the file image. This can cause resizing artifacts.
D3DX_FILTER_LINEAR	Each pixel in the texture surface is computed by averaging the four nearest pixels in the source image.
D3DX_FILTER_TRIANGLE	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 **D3DX_DEFAULT** for either the **Filter** or **MipFilter** parameters.
D3DX_FILTER_BOX	Each pixel is computed by averaging a 2×2(×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).
D3DX_FILTER_DITHER	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,        // Our Device
                            "Brickwall.bmp",  // File name of texture
                            256,  256,        // We want a 256x256 texture
                            D3DX_DEFAULT,     // Create MIP chain to 1x1
                            0,                // No special usage flags
                            D3DFMT_X8R8G8B8,  // Desired texture format
                            D3DPOOL_MANAGED,  // 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.

6.3.2 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
);
```

6.3.3 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).

6.3.4 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
);
```

6.3.5 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
);
```

6.3.6 IDirect3DDevice9::CreateTexture

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.

6.4 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::SetMateral 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.

6.5 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.

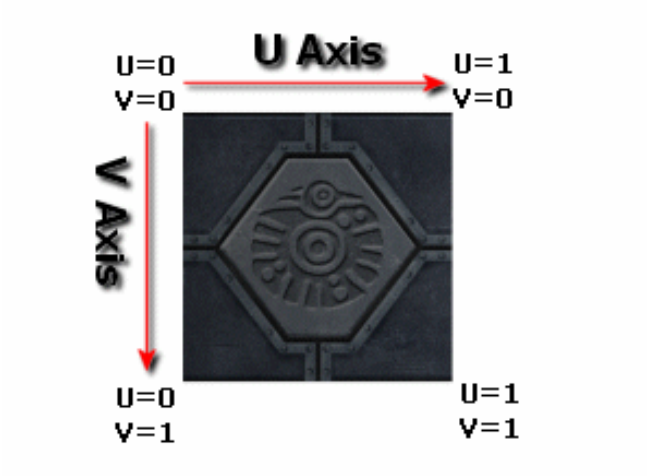


Figure 6.8

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.

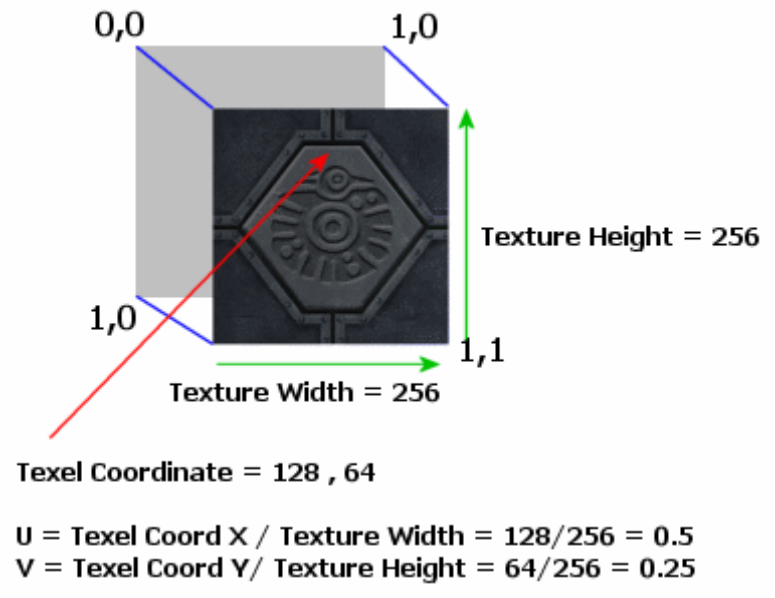


Figure 6.10

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:

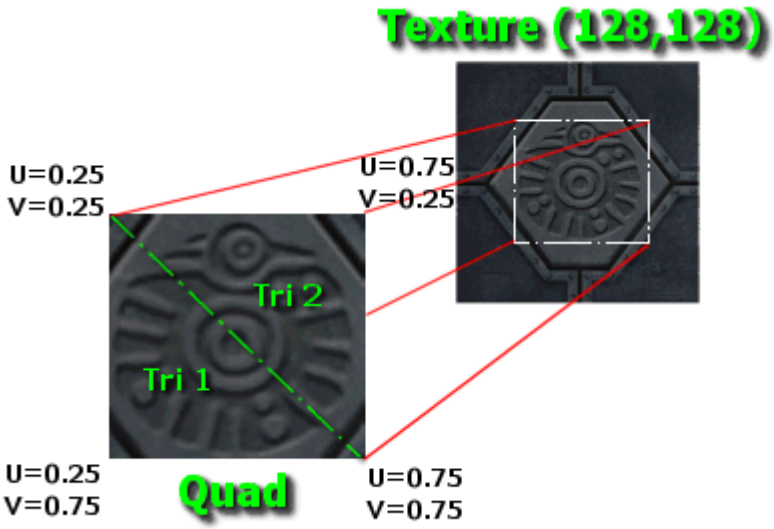


Figure 6.11

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.

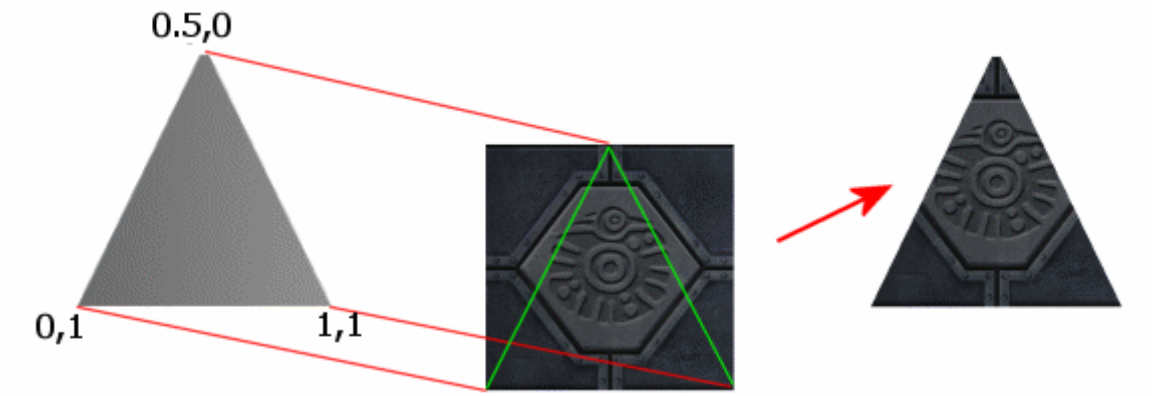


Figure 6.12

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
D3DFVF_XYZ	Informs the device that the vertices are untransformed and will need to be sent through the transformation pipeline.
D3DFVF_XYZRHW	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.
D3DFVF_NORMAL	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.
D3DFVF_DIFFUSE	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.
D3DFVF_SPECULAR	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.
D3DFVF_TEX0 through D3DFVF_TEX8	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; floaty; floatz;     // 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.

6.6 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
);
```

6.6.1 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 **D3DTEXTUREADDRESS** enumerated type:

```
typedef enum _D3DTEXTUREADDRESS
{
    D3DTADDRESS_WRAP = 1,
    D3DTADDRESS_MIRROR = 2,
    D3DTADDRESS_CLAMP = 3,
    D3DTADDRESS_BORDER = 4,
    D3DTADDRESS_MIRRORONCE = 5,
    D3DTADDRESS_FORCE_DWORD = 0x7fffffff
} D3DTEXTUREADDRESS;
```

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.

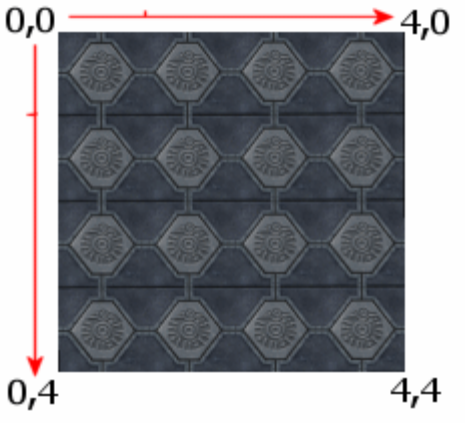


Figure 6.13

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 GILES™ 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.

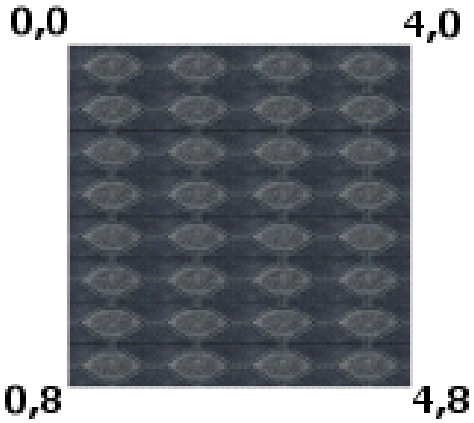


Figure 6.14

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.

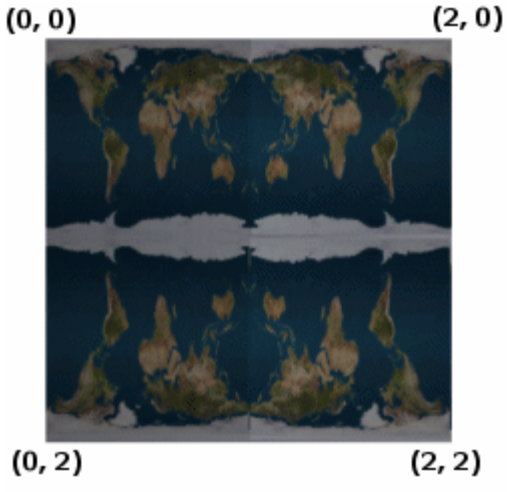


Figure 6.15

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 2nd and 4th tiles mirrored and the 1st and 3rd 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);
```

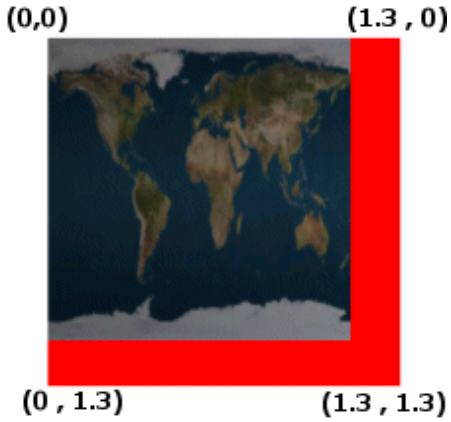


Figure 6.16

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).

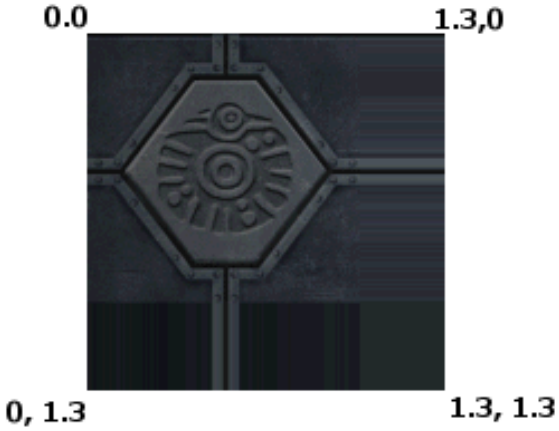


Figure 6.17

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 with the D3D Device

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: UV Wrapping Disabled

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: U Wrapping Enabled

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).

Figure 6.20: V Wrapping Enabled

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:

Figure 6.21: UV Wrapping Enabled

Wrapping can be very useful when generating texture coordinates to wrap a texture around a sphere or cylinder (Fig 6.22).

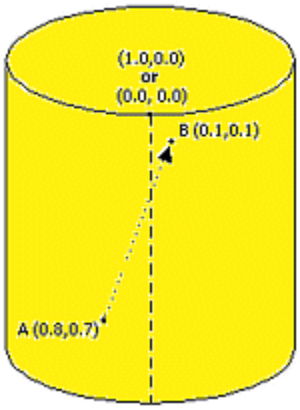


Figure 6.22

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 3rd 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 1st texture coordinate set stored at a vertex.

```
d3dDevice->SetRenderState(D3DRS_WRAP0, D3DWRAPCOORD_1);
```

The next example shows how to enable texture wrapping for the 3rd set of texture coordinates (for vertices that have them) on the U axis.

```
d3dDevice->SetRenderState(D3DRS_WRAP2, D3DWRAPCOORD_0);
```

6.6.2 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.

In this image the camera is moved so close to the quad that a small area in the center of the texture is mapped to the entire frame buffer.

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/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 x ImageWidth – 0.5
>
> V = v x 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).

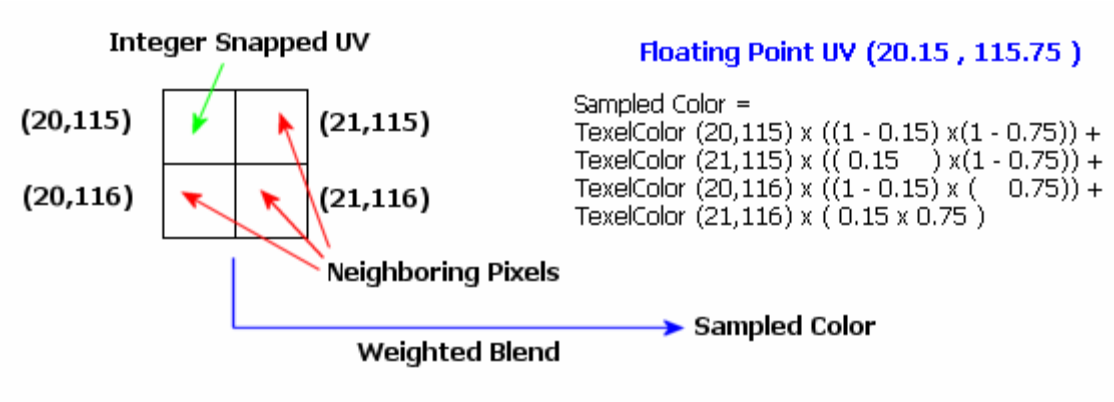


Figure 6.24

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).

Figure 6.25

When used for minification, this helps alleviate the problems caused by skipping pixels in the source image. Pixels that would otherwise we 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 anistropic 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.

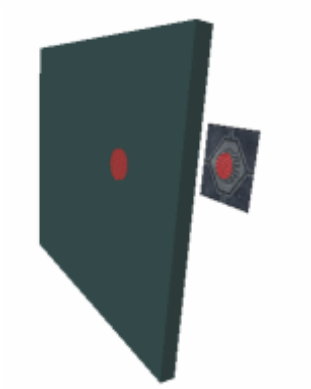


Figure 6.26

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).

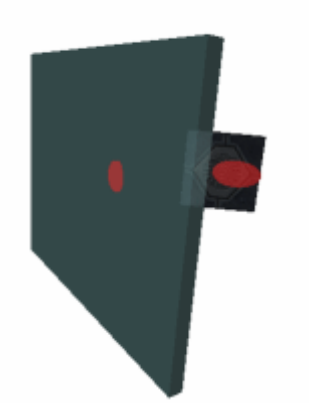


Figure 6.27

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 `D3DTEXTUREFILTERTYPE` 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.

6.6.3 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.

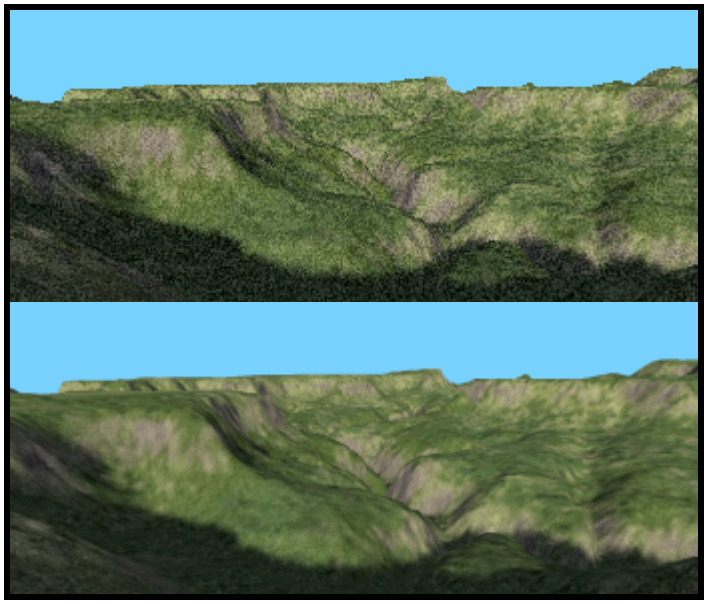


Figure 6.28

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**.

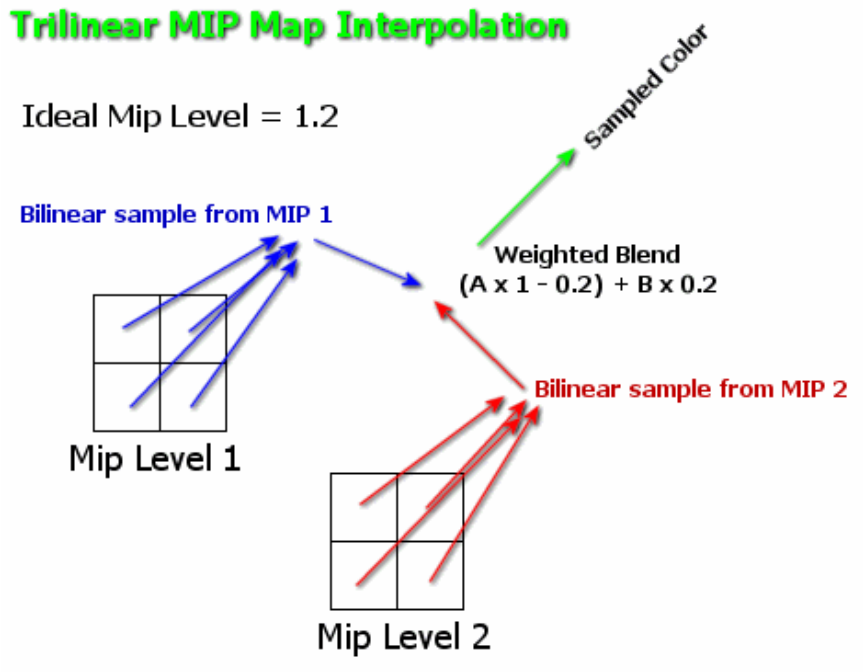


Figure 6.29

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.

6.7 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.

6.7.1 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.

6.7.2 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.

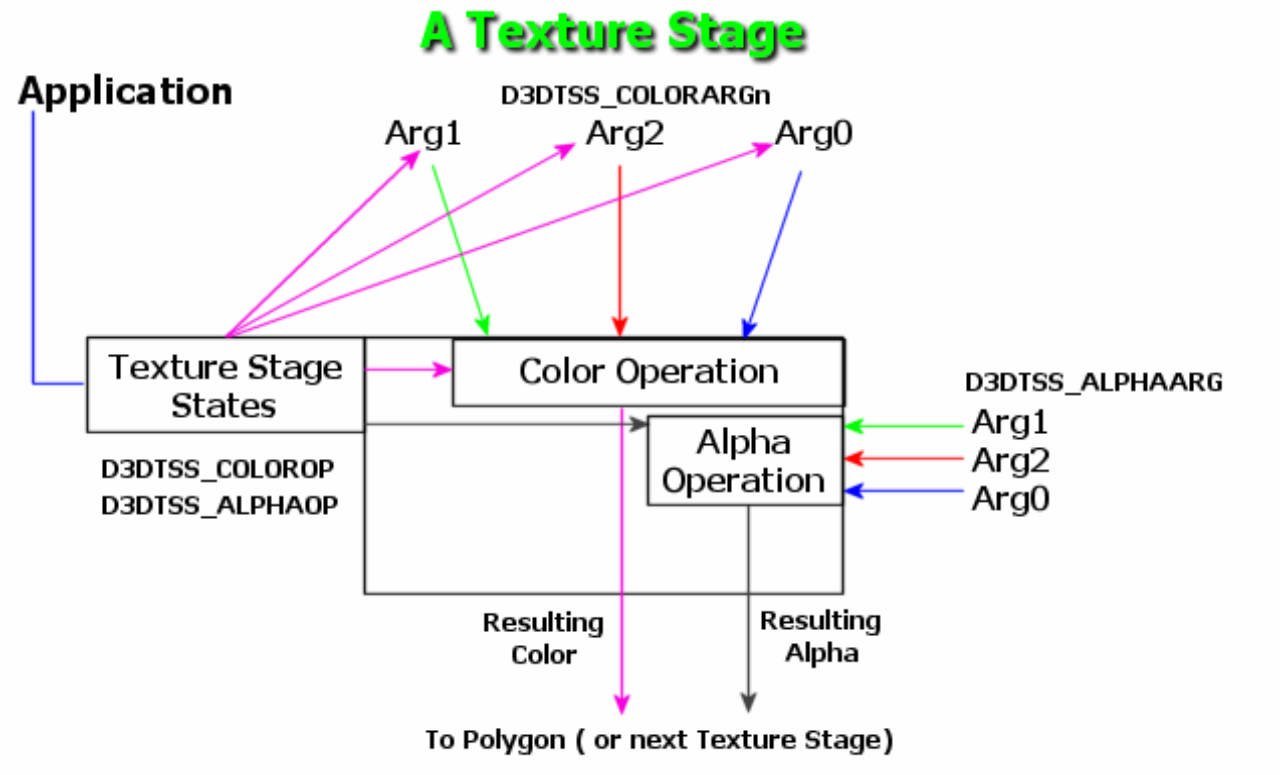


Figure 6.30

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).

6.7.3 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 to 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->SetTextureStateState(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->SetTextureStateState(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 behavior 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->SetTextureStateState(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 **D3DPMISCCAPS_PERSTAGECONSTANT** bit will be set in the PrimitiveMiscCaps member of the D3DCAPS9 structure:

```
D3DCAPS9 caps;
pDevice->GetDeviceCaps(&caps);
if(Caps.PrimitiveMiscCaps & D3DPMISCCAPS_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_BUMPENVMAP = 22,
    D3DTOP_BUMPENVMAPLUMINANCE = 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) x (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.

$$R = (ARG1 \times ARG2) << 1$$

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.

$$R = (ARG1 \times ARG2) << 2$$

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.

$$R = (ARG1 + ARG2 - 0.5) << 1$$

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.

6.7.4 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);
```

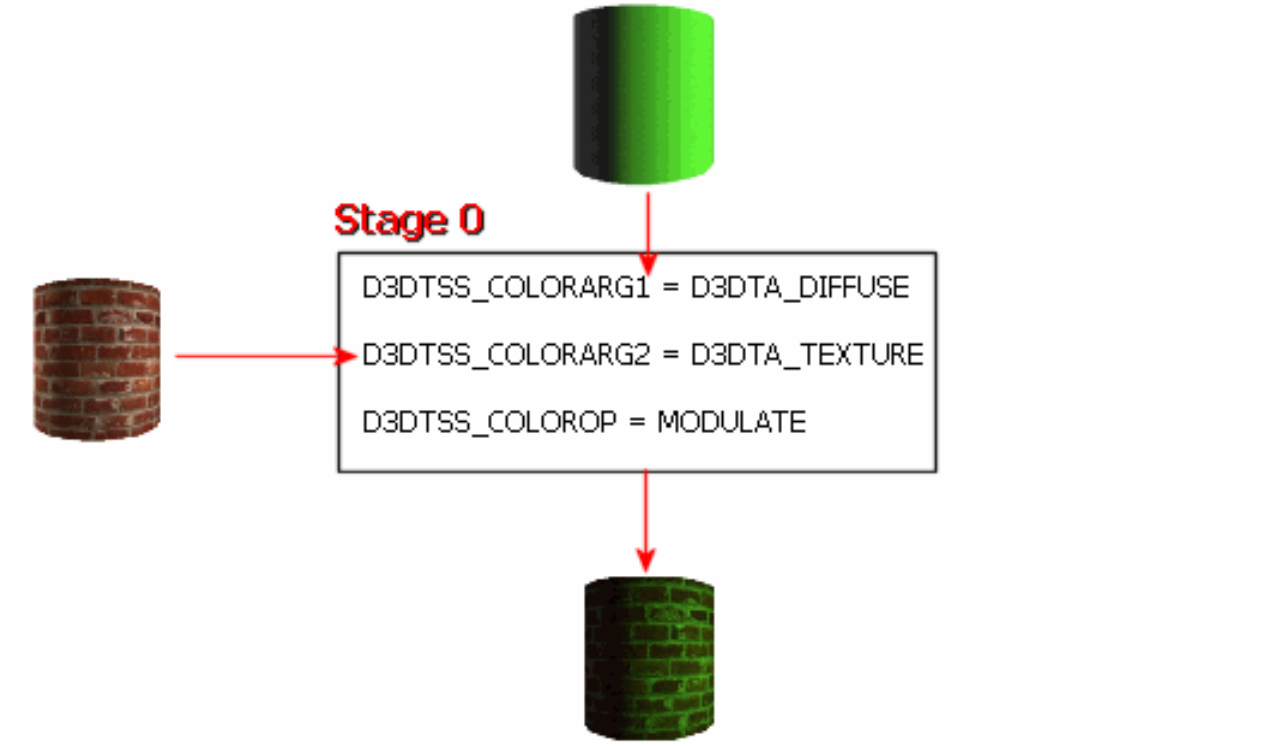


Figure 6.31

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);
```

6.8 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.

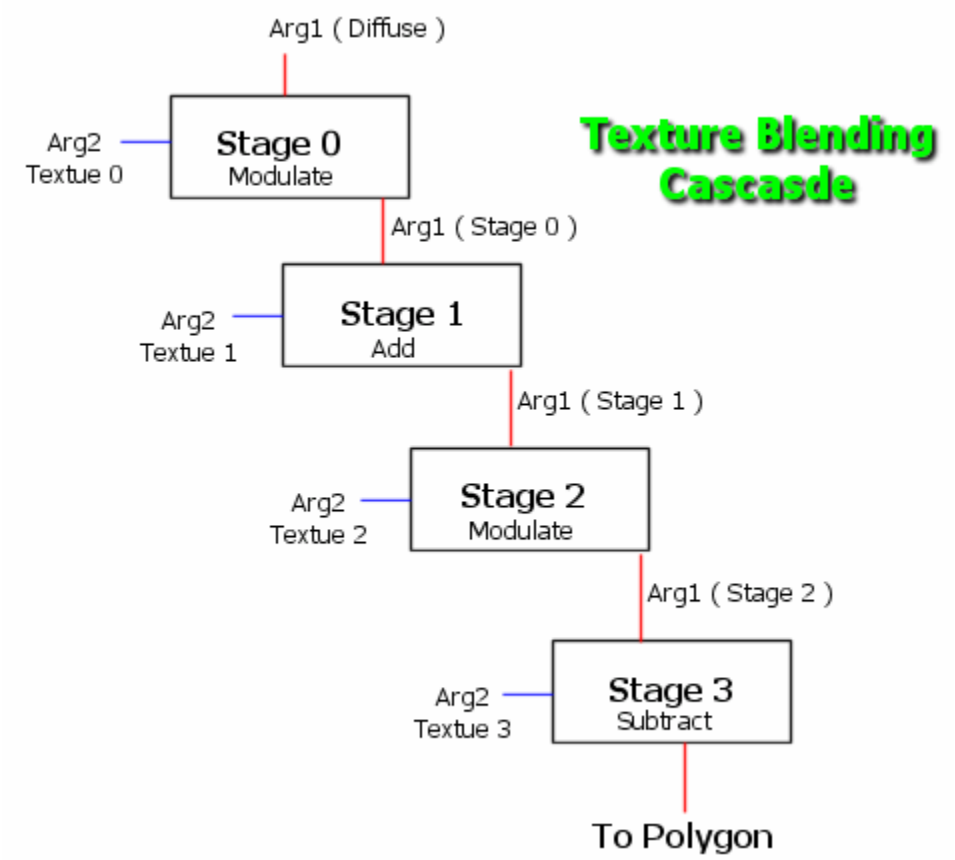


Figure 6.32

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.

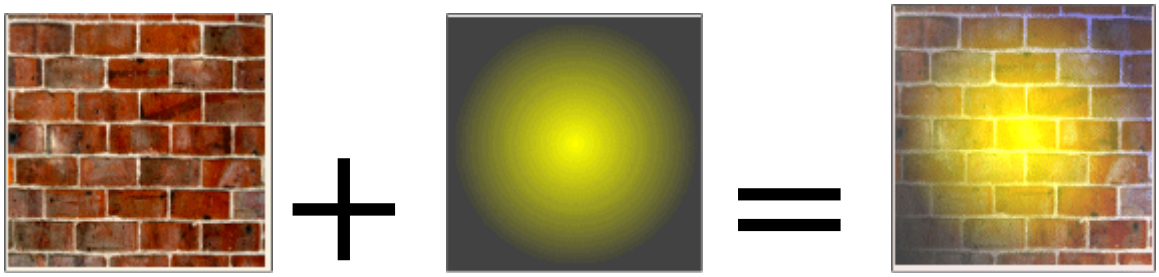


Figure 6.33

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.

6.8.1 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.

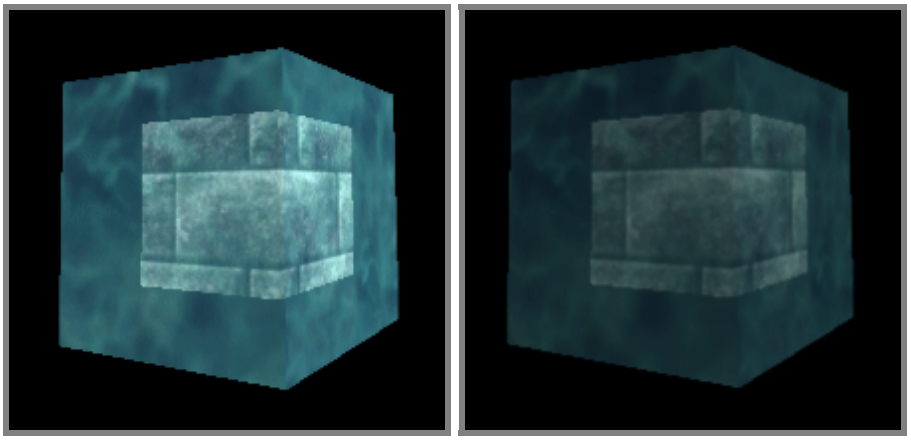


Figure 6.34

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.

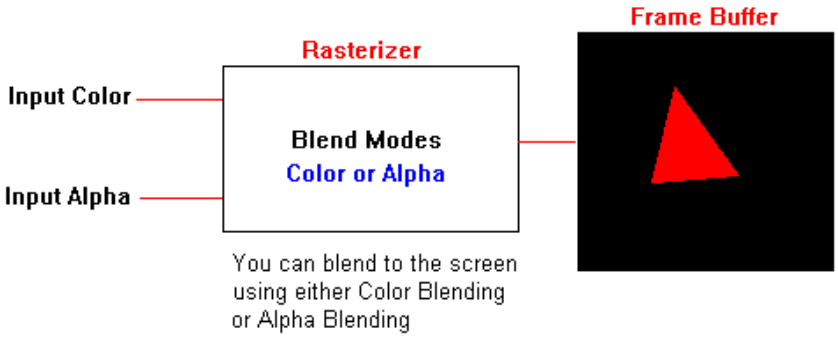


Figure 6.35

When blending is enabled, the following formula calculates the final color written to the frame buffer:

FinalColor = (SrcColor * SrcBlend) + (DestColor * 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(D3DTS_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_INVSRCCOLOR = 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_INVSRCCOLOR = 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.

FrameBufferColor = Source(ARGB) * 0 + Dest(ARGB) * 1
FrameBufferColor = Source(ARGB) * (0,0,0,0) + Dest(ARGB) * (1,1,1,1)
FrameBufferColor = (0,0,0,0) + Dest(ARGB)
= Dest(ARGB)

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:

FrameBufferColor = Source(ARGB) * 0 + Dest(ARGB) * 1
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)
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)

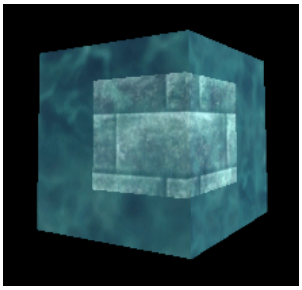


Figure 6.36

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_INVSRCCOLOR);
```

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.

FrameBufferColor = Src(ARGB) * Src(ARGB) + Dest(ARGB) * (1 – Src(ARGB))
FrameBufferColor = (0.25, 0.25, 0.25, 0.25) * (0.25, 0.25, 0.25, 0.25)
+ (0.5, 0.5, 0.5, 0.5) * (1–0.25, 1-0.25, 1-0.25, 1-0.25)
FrameBufferColor = (0.062, 0.062, 0.062, 0.062) + (0.75, 0.75, 0.75, 0.75)
FrameBufferColor = (0.812, 0.812, 0.812, 0.812)

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.

FrameBufferColor = (SourceColor*DestColor) + (DestColor*SourceColor)
FrameBufferColor = (SourceColor * DestColor) * 2

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, 0.75).

FrameBufferColor = (0.5, 0.5, 0.5, 0.5) * (0.75, 0.75, 0.75, 0.75)
+ (0.75, 0.75, 0.75, 0.75) * (0.5, 0.5, 0.5, 0.5)
FrameBufferColor = (0.375, 0.375, 0.375, 0.375) + (0.375, 0.375, 0.375, 0.375)
FrameBufferColor = (0.75, 0.75, 0.75, 0.75)

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.

6.9 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.

6.9.1 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.

6.9.2 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.

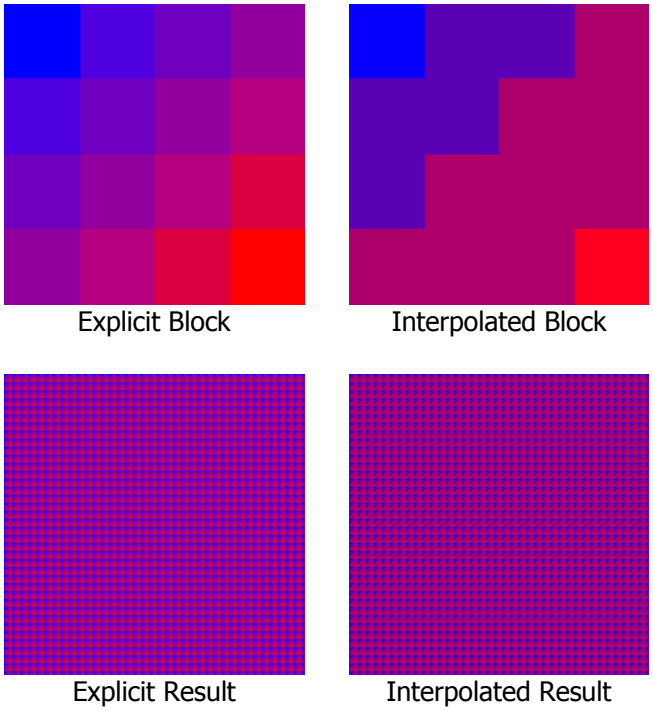
6.9.3 Compressed Data Blocks - Color Data Layout

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**.

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 2/3rd blue and 1/3rd red, and the fourth color is 1/3rd blue and 2/3rd 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 in 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.



Explicit Block

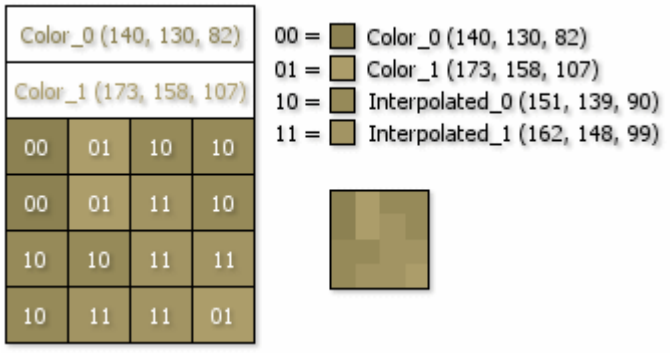
Interpolated Block

Explicit Result

Interpolated Result

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 $2/3^{rd}$ of Color_0 and $1/3^{rd}$ of Color_1. The second interpolated color uses $1/3^{rd}$ of Color_0 and $2/3^{rd}$ of Color_1.

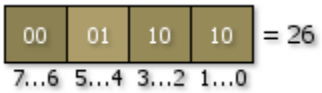
```
Interpolated_0(red) = (2 * (Color_0(red) / 3)) + (1 * (Color_1(red) / 3))
Interpolated_0(red) = (2 * Color_0(red) + Color_1(red)) / 3

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.

6.9.4 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 ($11_{(bin)}$ or $3_{(dec)}$). 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 ½ Color_1 and ½ 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*

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*

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 6/7th Alpha_0 and 1/7th Alpha_1,
Alpha_3 uses 5/7th Alpha_0 and 2/7th Alpha_1,
Alpha_4 uses 4/7th Alpha_0 and 3/7th 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.

6.10 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

$$\begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix}$$

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 * m11 + v * m21 + 1 * m31 + 0 * m41
V = u * m12 + v * m22 + 1 * m32 + 0 * m41
U = u * m11 + v * m21 + 1 * m31
V = u * m12 + v * m22 + 1 * m32

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

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 2 & -5 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

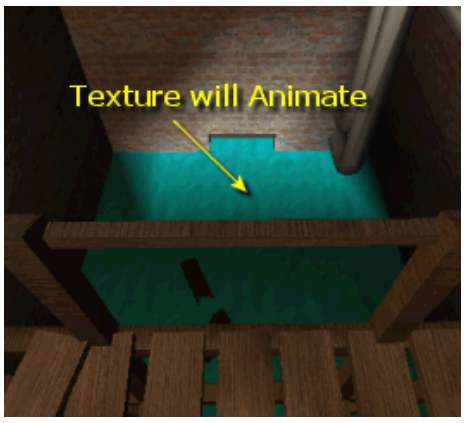
U = 0.5*1 + 0.9*0 + 1*2
U = **2.5**

V = 0.5*0 + 0.9*1 + 1*-5
V = **-4.1**

Next we see how to setup a matrix to translate a 1D texture (u, 1, 0, 0) by **du** and **dv**:

1D Texture Translation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ du & dv & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

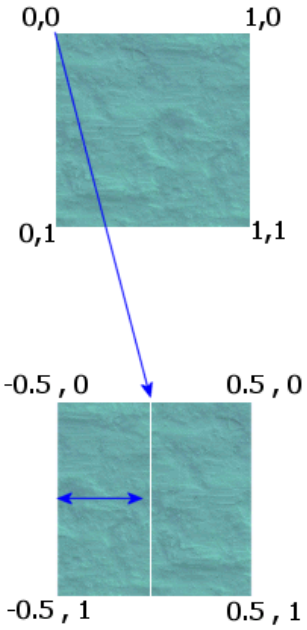


In Lab Project 6.3 we will use texture coordinate animation to make the water underneath the wooden bridge look like it is flowing. We do this by setting the texture stage matrix to a translation matrix that continuously increments the translation amount of the texture coordinates each time the face 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.

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.

D3DTTFF_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.

D3DTTFF_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.

D3DTTFF_COUNT4
The 4D vector output from the matrix multiplication is passed straight to the rasterizer.

D3DTTFF_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 **D3DTTFF_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);
```

6.11 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.

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.

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.

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.

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.

6.12 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, 0xFFFFFF );
::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.

6.12.1 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.

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);
```

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.

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 IDirect3DDevice9::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 (4th 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.

6.12.2 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.

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:

D3DPTFILTERCAPS_MINFPOINT	Device supports point-sample filtering for minifying rectangles. This filter type is requested by calling **IDirect3DDevice9::StretchRect** using *D3DTEXF_POINT*.
D3DPTFILTERCAPS_MAGFPOINT	Device supports point-sample filtering for magnifying rectangles. This filter type is requested by calling **IDirect3DDevice9::StretchRect** using *D3DTEXF_POINT*.
D3DPTFILTERCAPS_MINFLINEAR	Device supports bilinear interpolation filtering for minifying rectangles. This filter type is requested by calling **IDirect3DDevice9::StretchRect** using *D3DTEXF_LINEAR*.
D3DPTFILTERCAPS_MAGFLINEAR	Device supports bilinear interpolation filtering for magnifying rectangles. This filter type is requested by calling **IDirect3DDevice9::StretchRect** using *D3DTEXF_LINEAR*.

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 Render Target	Render Target Surface	Off-screen Plain
Texture Surface	No	No	No	No
Texture Render Target	No	Yes	Yes	No
Render Target Surface	No	Yes	Yes	No
Off-screen Plain	Yes	Yes	Yes	Yes
DX8 Driver (Stretching)	Texture Surface	Texture Render Target	Render Target Surface	Off-screen Plain
Texture Surface	No	No	No	No
Texture Render Target	No	No	No	No
Render Target Surface	No	Yes	Yes	No
Off-screen Plain	No	Yes	Yes	No
DX9 Driver (No Stretching)	Texture Surface	Texture Render Target	Render Target Surface	Off-screen Plain
Texture Surface	No	No	No	No
Texture Render Target	No	Yes	Yes	No
Render Target Surface	No	Yes	Yes	No
Off-screen Plain	Yes	Yes	Yes	Yes
DX9 Driver (Stretching)	Texture Surface	Texture Render Target	Render Target Surface	Off-screen Plain
Texture Surface	No	Yes	Yes	No
Texture Render Target	No	Yes	Yes	No
Render Target Surface	No	Yes	Yes	No
Off-screen Plain	No	Yes	Yes	No

The IDirect3DDevice9::StretchRect function can fail for a number for 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.

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.

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.

6.13 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 D3DXLoadSurfaceFormFile 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.

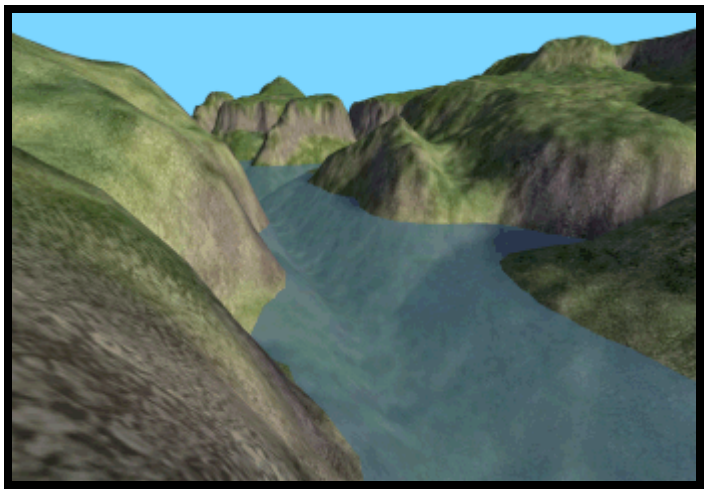
D3DXIMAGE_INFO* pSrcInfo
Pointer to a **`D3DXIMAGE_INFO`** structure to be filled with a description of the data in the source file. We can pass NULL if we are not interested in this information.

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.

Chapter Seven:

Alpha Blending and Fog



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

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	23 16	15 8	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 render state:
>
> ```
> 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.

7.1 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.

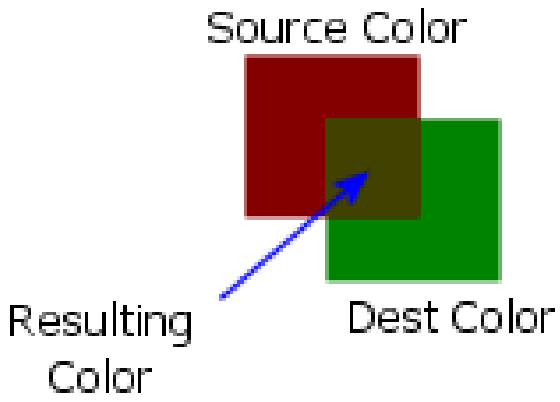


Figure 7.2

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)

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 = (**0.75**, 0.5, 0, 0) * (**0.75**, **0.75**, **0.75**, **0.75**) +
(1.0, 0, 0.5, 0) * (1-**0.75**, 1-**0.75**, 1-**0.75**, 1-**0.75**)
Final Color = (0.56, 0.375, 0, 0) + (0.25, 0, 0.125, 0)
Final Color = (0.81, 0.375, 0.125, 0);

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.

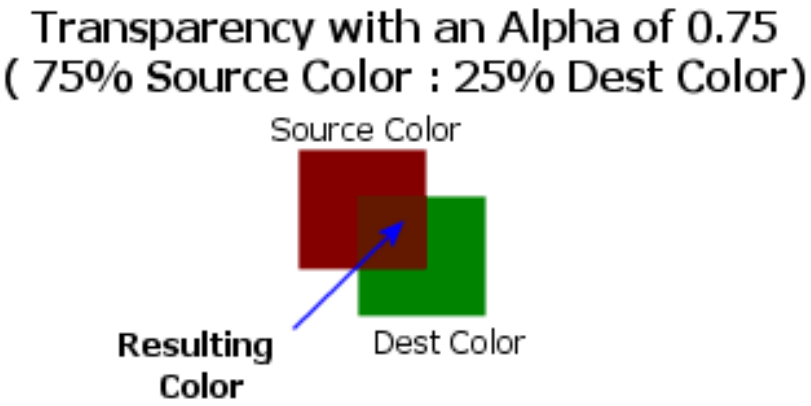


Figure 7.3

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.

7.2 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 though the lab projects accompanying this lesson.

7.2.1 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 ¾ 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 ¾ 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	23 16	15 8	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.

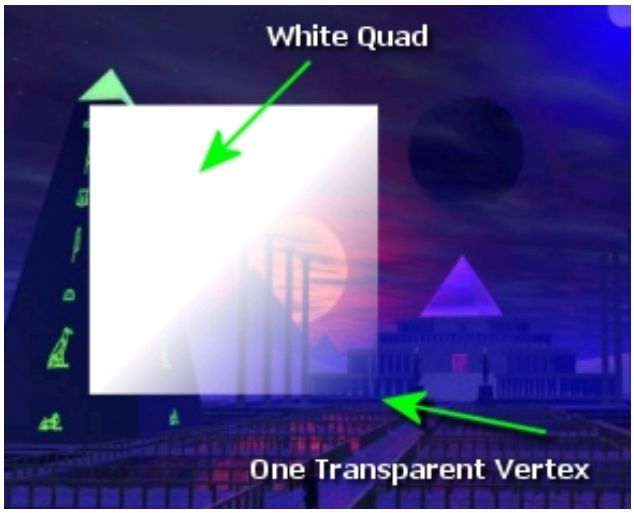


Figure 7.4

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.

7.2.2 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.

7.2.3 Vertex Alpha – Unlit Vertices

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 ¾ transparency, the second vertex has ½ transparency and the third has ¼ 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 ½ transparent green diffuse color and a ¾ 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 ½ transparent and green in color. When no light is shining on the vertices, the polygon will be emissive blue and will be ¾ 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:

Vertex Color = (AmbientLight * A) + (DiffuseLight * D) + (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:

Vertex Color.a = (AmbientLight * A.a) + (DiffuseLight * D.a) + (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.

7.2.4 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 ½ 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.

7.2.5 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.

7.2.6 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 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.

7.3 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.

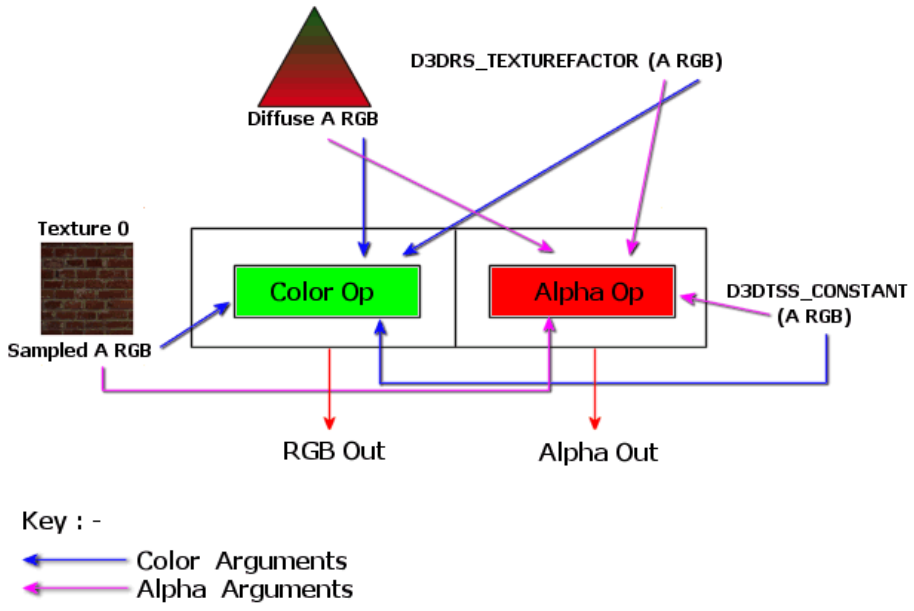


Figure 7.5

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 ¼ 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 ¼ 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 );
```

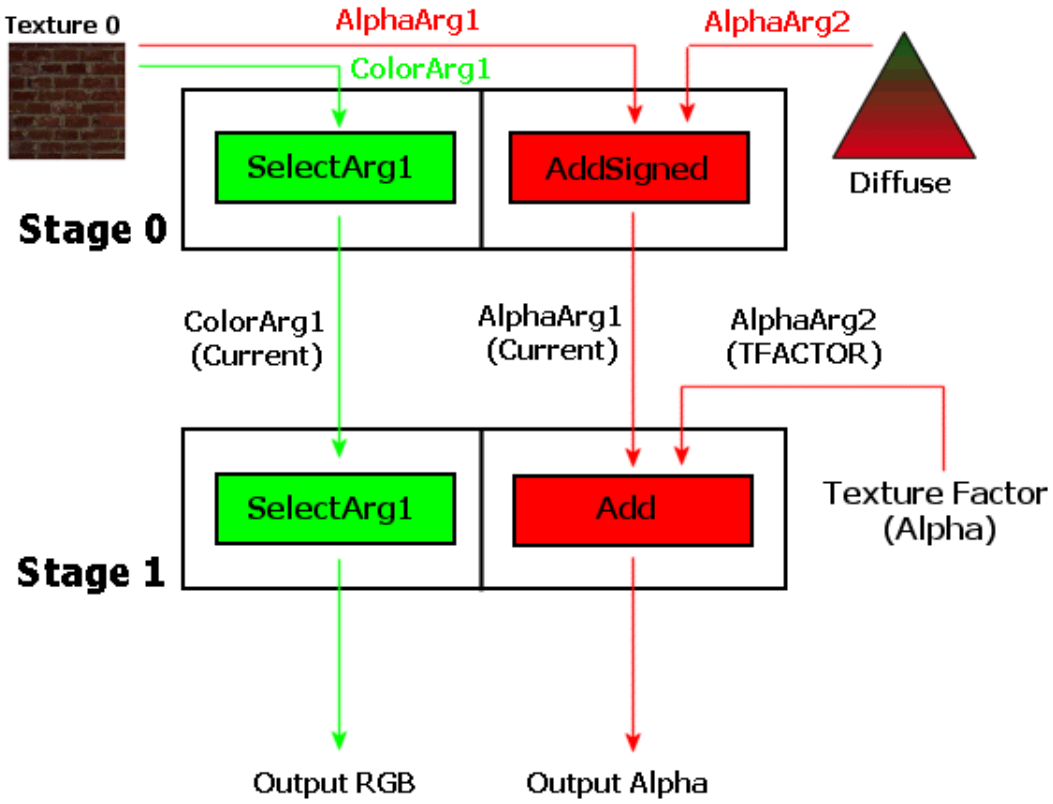


Figure 7.6

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.

7.4 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:

Pixel Color = SourceColor * SrcBlendMode + DestColor * 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:

PixelColor = SourceColor * SourceAlpha + DestColor * (1 – 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_INVSRCCOLOR);
```

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_INVSRCCOLOR = 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;
```

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.

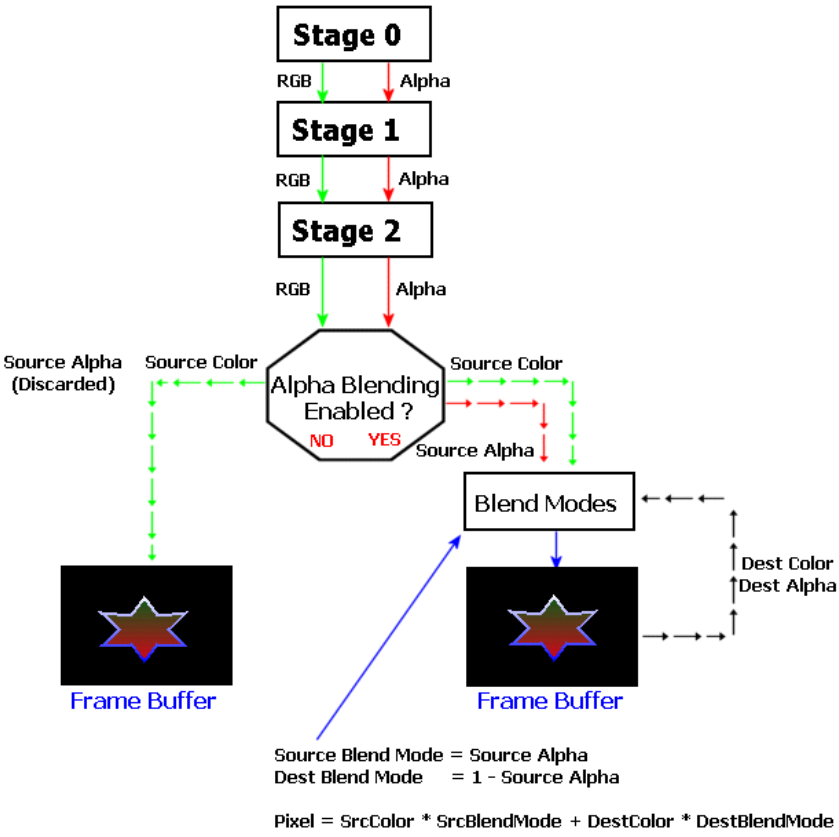


Figure 7.7

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.

7.5 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.

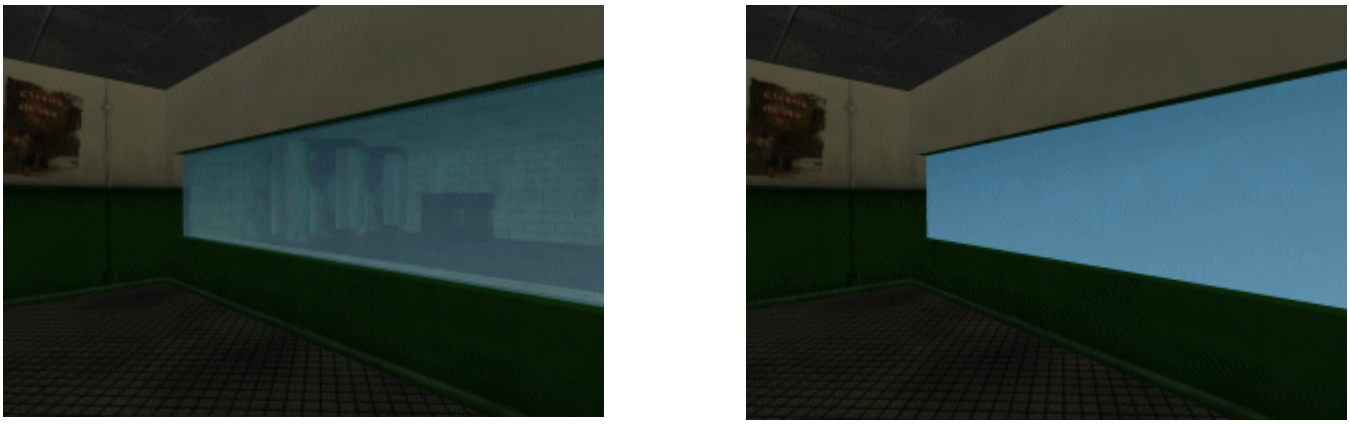


Figure 7.8

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 behavior 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).

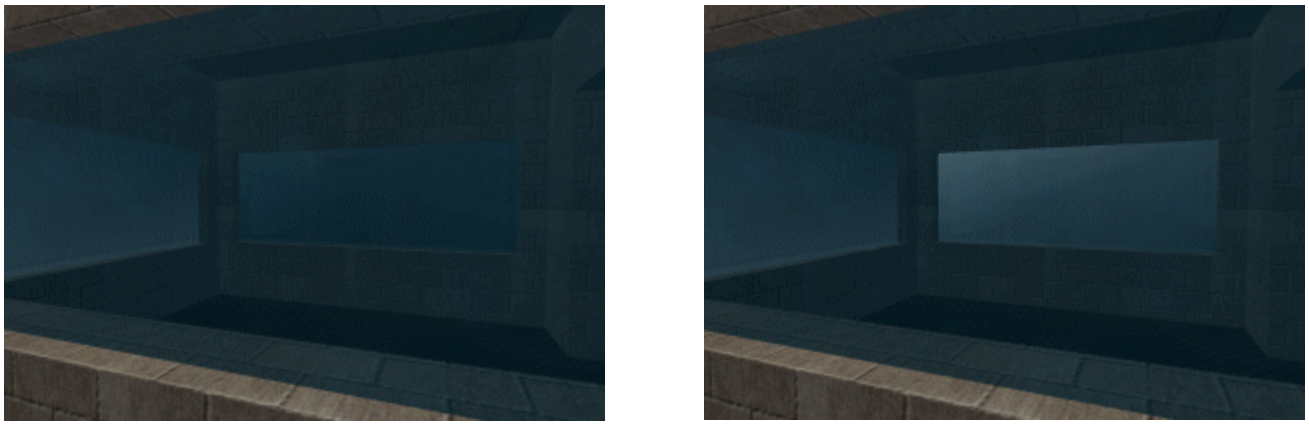


Figure 7.9

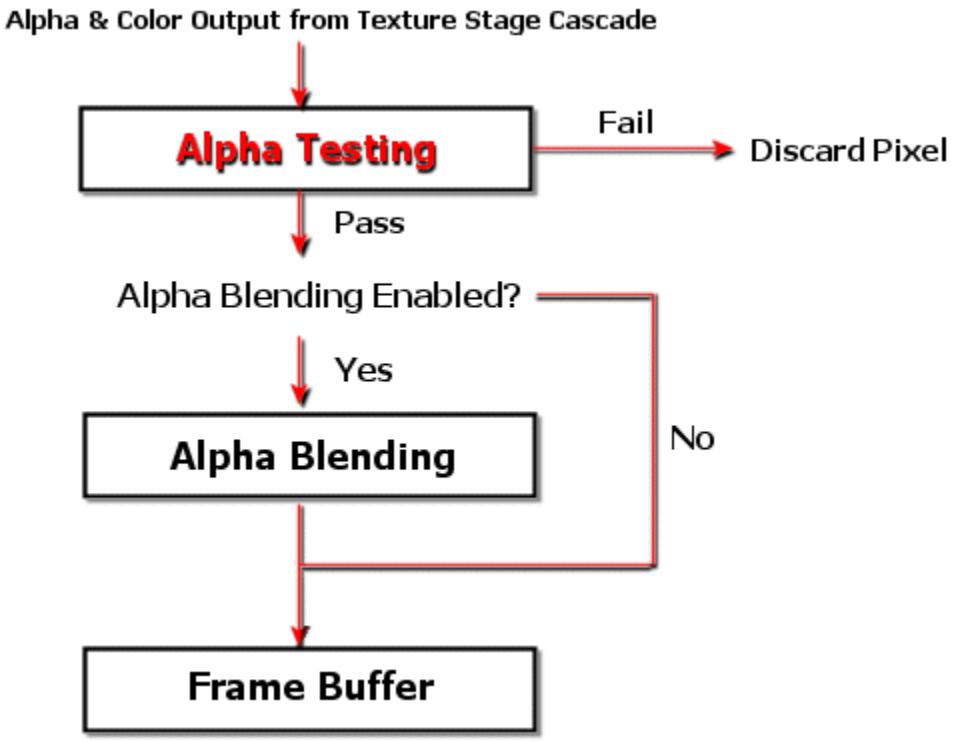
The image on the left has been rendered correctly. The furthest window was rendered first, which tinted the frame buffer pixels appropriately. The closest window was rendered last. Its pixels were blended with the pixels in the frame buffer -- which are themselves the result of a blend between the first 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.

7.6 Alpha Testing

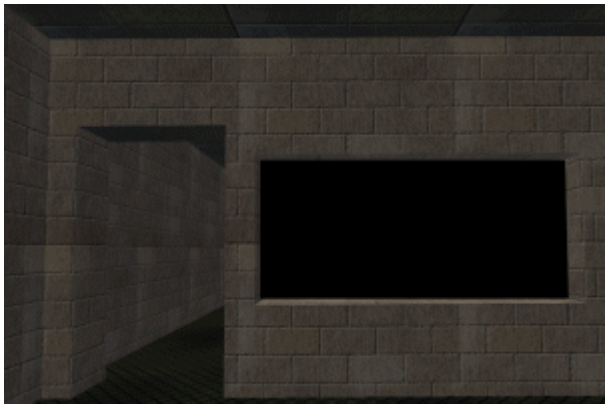
Very often we will be rendering polygons that have a level of transparency such that the pixels are either fully opaque or fully transparent. Think about a texture of a tree mapped to a quad for example. We would want the pixels in between the leaves and branches to be totally transparent so that when 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.

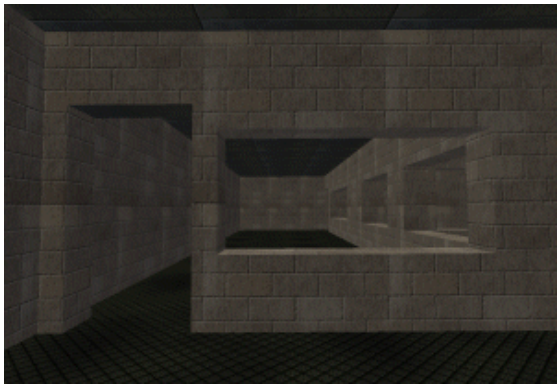
Let us see what happens when we enable alpha testing so that any pixels with an alpha value of anything less than 255 (totally opaque) get rejected by the pipeline – they are never added to the frame buffer or the depth buffer.

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 = 0x7fffffff
} 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.

7.7 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.

Pixel Color = SourceColor * SrcAlpha + DestColor * (1-SrcAlpha)

Example 1: Blue * Green * Red
Blend Green Quad First
Pixel Color = GreenQuad * 0.5 + BlueFrameBuffer * (1 - 0.5)
Pixel Color = (0, 1.0 , 0) * 0.5 + (0 , 0 , 1.0) * 0.5
Pixel Color = (0, 0.5, 0) + (0 , 0 , 0.5)
Pixel Color = (0, 0.5 , 0.5)
Pixel Color = Murky Blue Color

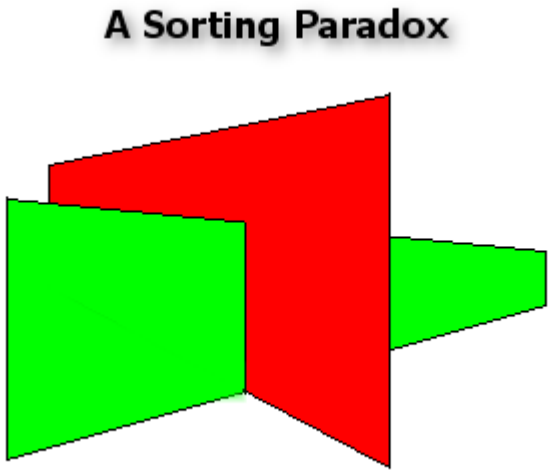
Now Blend Red Quad
PixelColor = RedQuad * 0.5 + MurkyBlue * (1 – 0.5)
PixelColor = (1.0 , 0 , 0) * 0.5 + (0.0 , 0.5 , 0.5) * 0.5
PixelColor = (0.5 , 0 , 0) + (0.0 , 0.25 , 0.25)
Pixel Color = (0.5, 0.25 , 0.25)
Pixel Color = Brownish Color

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.

7.7.1 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.

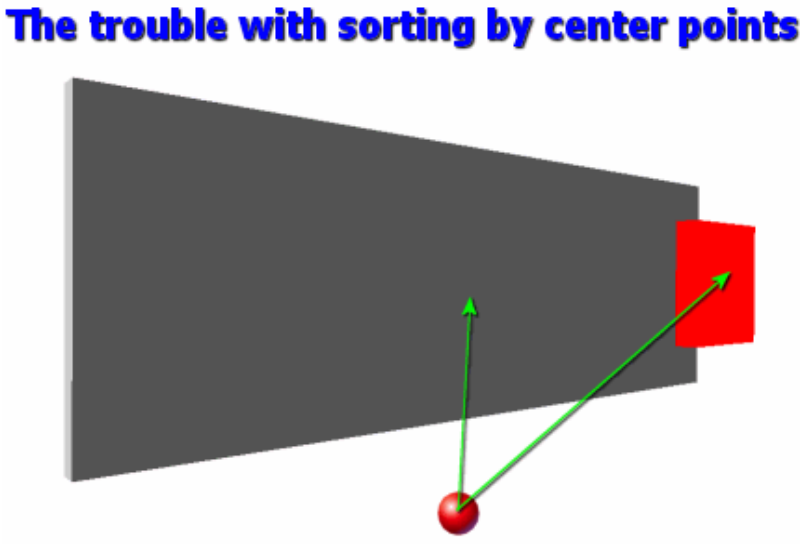


Figure 7.10

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.

7.7.2 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 $O(\text{n})$. 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	*elem1* less than *elem2*
0	*elem1* equal to *elem2*
> 0	*elem1* greater than *elem2*

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.

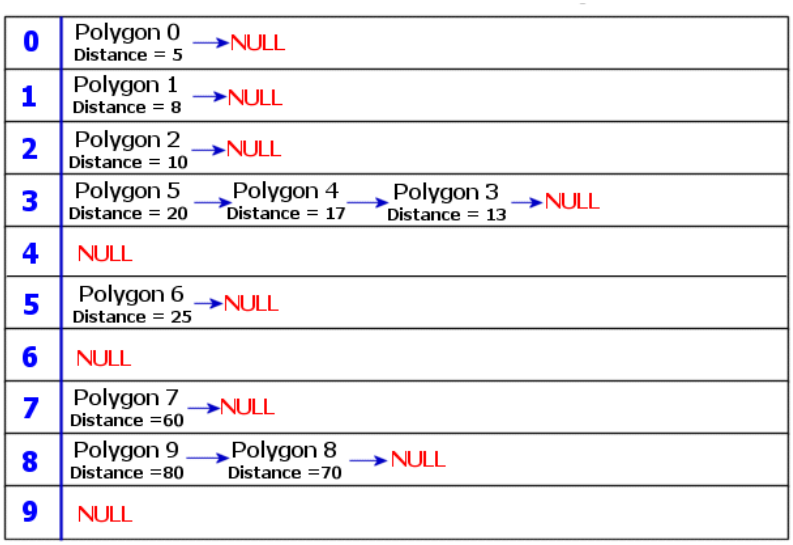


Figure 7.11

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.

7.8 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_ALPHAOP, 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_ALPHAOP, 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 be 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 2nd 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.

7.9 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.

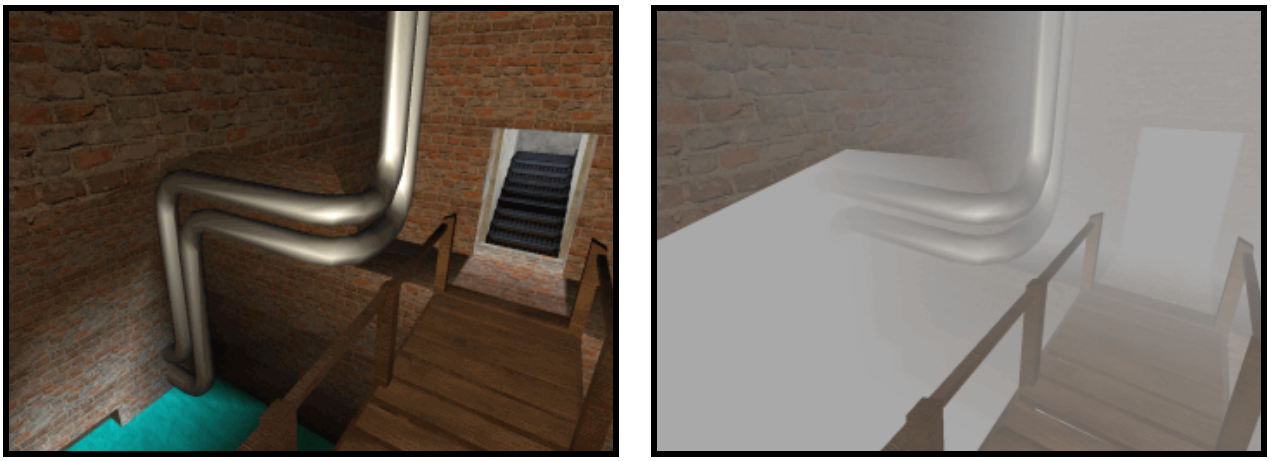


Figure 7.11

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).

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:

$C_{final} = F_{factor} \, x \, C_{original} + (1 - F_{factor}) \, x \, C_{fog}$

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.

7.9.1 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);
```

7.9.2 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.

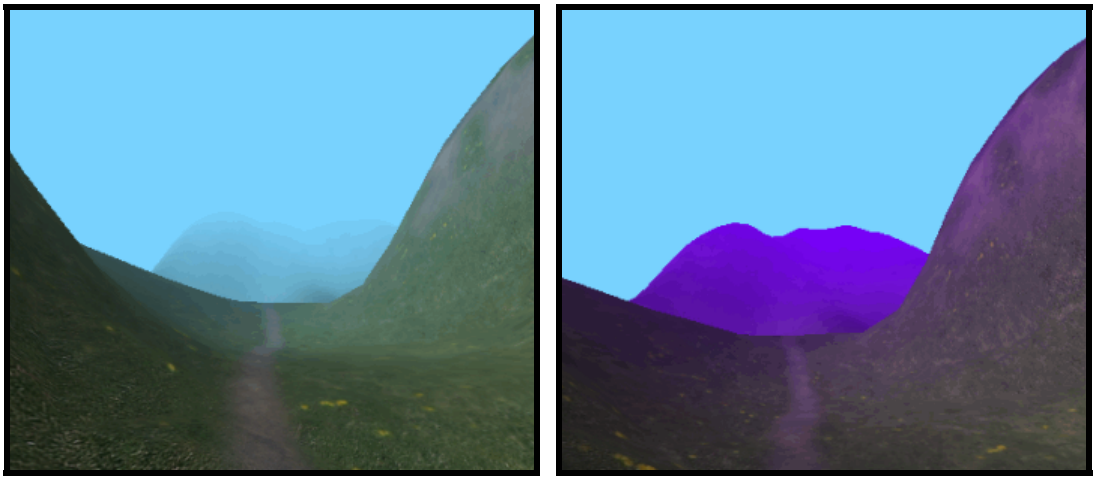


Figure 7.13

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 );
```

7.10 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).

7.10.1 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.

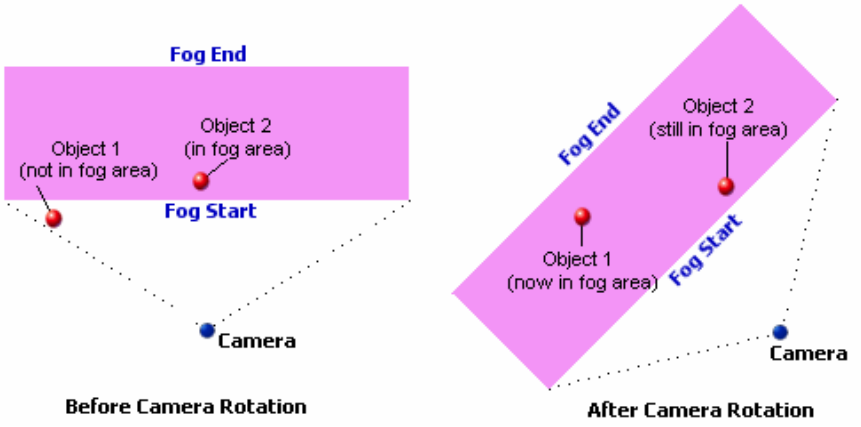


Figure 7.14

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.

7.10.2 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.

7.11 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:

7.11.1 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.

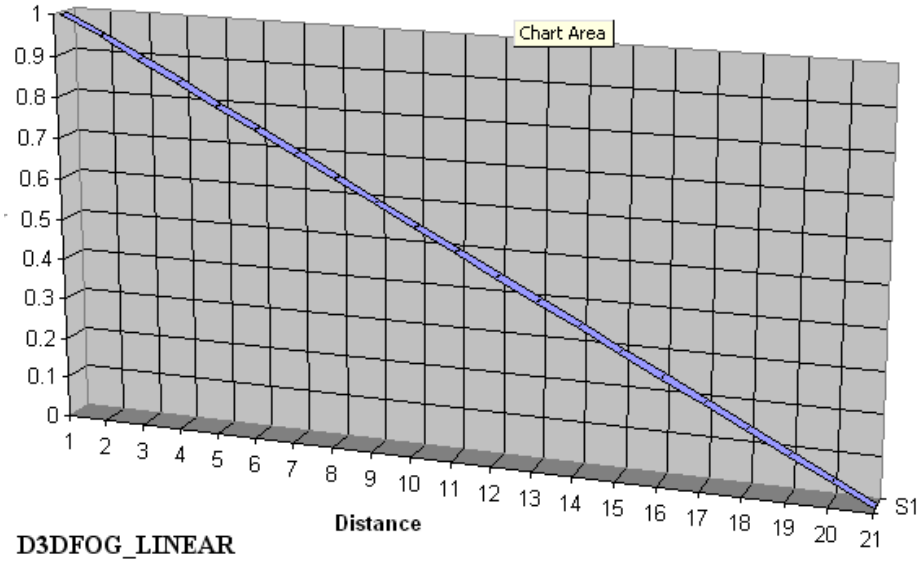


Figure 7.15

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.

7.11.2 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.

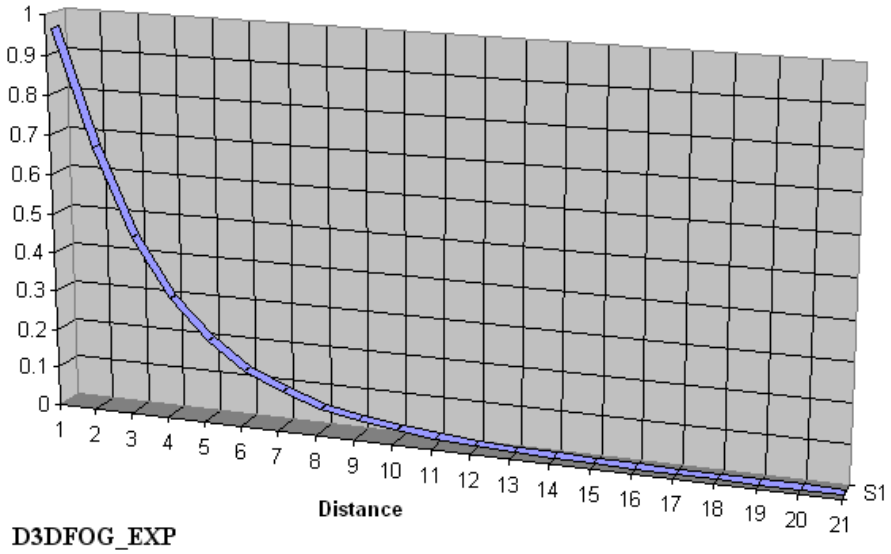


Figure 7.16

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]

7.11.3 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 course in the series.