Chapter 4

Shader Examples

In the previous chapters, I detailed the ins-and-outs of the HLSL shader language. The one thing I had not bothered demonstrating is how to actually write shaders. Although this is not a book on how to write shaders, it was fitting to give a few examples of basic shaders. These shaders will come in handy later as I start using the effect framework and to illustrate core concepts.

Keep in mind that to create complete shaders, you not only need the shader code but a set of render states and variables which need to be setup based on proper semantics. Since we have not made it to the topic of building complete effects yet, I will only concentrate on the vertex and pixel fragment codes at this point. I will expand on these later in this book.

The Bare Minimum Shader

To have something rendering on the screen, a bare minimum task needs to be accomplished. First, you will need to take the incoming vertex positions (which are in world space) and convert them to screen space. This is performed by using the world-view-projection matrix, containing all the necessary transformations to take an object from its local space and bringing it into projected space corresponding to your screen coordinates. For now, we'll assume that this combined transformation matrix is contained in a variable named *view_proj_matrix*.

The first thing we need to do is define the structure used to pass the vertex information from our vertex shader onto our pixel shader. You can define this in a structure called *VS_OUTPUT*, or any name you wish. All we need at this point is the position of the vertex.

```
struct VS_OUTPUT
{
   float4 Pos:     POSITION;
};
```

You may notice the *POSITON* semantics attached to the *Pos* variable; this tells the effect system how this value should be passed to the pixel shader. Semantics will be discussed in a later chapter. The only thing missing at this point is the vertex shader code. This code will take the incoming vertex position and transform it with the *view_proj_matrix*, which can be done by using the

mul built-in function. Putting this code in a function called *vs_main* would yield the following output:

```
VS_OUTPUT vs_main( float4 inPos: POSITION )
{
   VS_OUTPUT Out;

   // Output a transformed and projected vertex position
   Out.Pos = mul(view_proj_matrix, inPos);

   return Out;
}
```

Take note that the input parameter *inPos* is also followed by the semantics declaration *POSITION*. This tells the vertex shader how to map the geometry stream information to this input parameter. This brings us to the second component of a bare minimum shader. Now that you know where the vertices will be on-screen, you need to define what they should look like. The easiest way is to use a single constant color. Within a pixel shader, the return value of the shader is usually defined as a *float4* which will be used as the screen color for this pixel in the form of red, green, blue and alpha. Putting the appropriate code in a function named *ps_main*, yielded the following:

```
float4 ps_main( void ) : COLOR0
{
   //  Output constant color
   float4 color;
   color[0] = color[3] = 1.0; // Red and Alpha on
   color[1] = color[2] = 0.0; // Green and Blue off
   return color;
}
```

The pixel shader code is simple enough, and you can notice the *COLOR0* semantic on the return of the function, telling our compiler that this function returns the color to be applied to this particular pixel. It's nothing too complicated, nor is it intended to be in this chapter. This code, as-is, is not very useful. As I mentioned above, we'll use this code as the basis to develop full-blown shaders and effects later in the book.

Making it Colorful

Now that we have the code for basic rendering, how about adding a little more color by applying a texture to the geometry? For the shader to be able to use a texture, it will need a global variable of type *sampler*. We'll explain later how you can use semantics and the effect framework to set up textures. For now, this is how the texture would be set up:

```
sampler Texture0;
```

Before you can use the texture, you need to know where to sample the textures in the first place. To do this, your pixel will need some texture coordinates. Texture coordinates will come in to your vertex shader (from your geometry data), processed and passed on to the pixel shader. Texture coordinates are passed in parameters, which used the *TEXCOORDx* semantics. This tells the hardware how to exchange the data between the vertex and pixel shaders. Here is the resulting vertex shader code with all our adjustments.

```
struct VS_OUTPUT
{
   float4 Pos:     POSITION;
   float2 Txr1:    TEXCOORD0;
};

VS_OUTPUT vs_main(
       float4 inPos: POSITION,
       float2 Txr1: TEXCOORD0
)
{
   VS_OUTPUT Out;

   // Output our transformed and projected vertex
   // position and texture coordinate
   Out.Pos = mul(view_proj_matrix, inPos);
   Out.Txr1 = Txr1;

   return Out;
}
```

The pixel shader is easy to define. With the sampler variable already created, you will need to sample the texture by using the *tex2D* built-in function. The final pixel shader code with all the changes is shown below.

```
sampler Texture0;
float4 ps_main(
       float4 inDiffuse: COLOR0,
       float2 inTxr1: TEXCOORD0
) : COLOR0
{
   // Output the color taken from our texture
   return tex2D(Texture0,inTxr1);
}
```

Shining Some Light

Although rendering objects with a basic texture might be enough, it definitely fails to be realistic. In the quest to create realistic scenes, one of the steps usually taken is to apply lighting to objects.

The real world is filled with lights, from the sun to light bulbs. Without light, there wouldn't be much to see in the first place!

Although lighting can be a complex topic of its own, in the realm of 3D graphics, lights are generally simplified to a few basic types.

- **Ambient lighting**: a gross approximation of the total level of lighting in a scene coming from a multitude of discreet lights. This serves as a good guess significantly reduce the number of lights to consider within a particular scene. Ambient lighting generally manifests itself as a constant colored light, affecting the object uniformly.
- **Diffuse lighting**: Materials presenting a microscopic rough surface has the effect of reflecting incoming light uniformly in all directions. This has the result of causing the perceived lighting to be the same from any viewing angle.
- **Specular lighting**: When the surface of a material is smooth and presents little roughness, light reflects off the surface in a non-uniform way. This means that for specular lighting, the light intensity will not only depend on the light to surface angle but also on the viewing angle.

In addition to the influence that light can have on an object surface, you also need to consider how lights can be classified themselves. Although light generally emits from a surface, such as a light bulb or the sun, we can usually consider light as either being directional or coming from a point source.

Directional lights are the simplest form of lighting available. They do not have a position and simply assume that all light rays are parallel to one another and heading in the same direction. What kind of light behaves like this? Well in reality, none. A directional light only has the purpose of estimating the case where a uniform source of light is sufficiently far away and that you can estimate it by assuming that all rays of light are parallel to one another.

The best example of this would be sunlight. Considering the sun can be treated as a point light and that it is located million of miles away from earth, rays that hit the surface of the earth are almost parallel and considered directional light. This concept is shown in Figure 4.1.

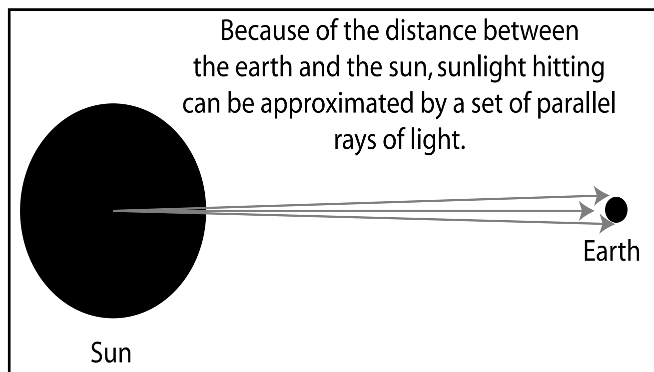

Figure 4.1: Light rays from the sun can be considered parallel because of the sun to earth distance.

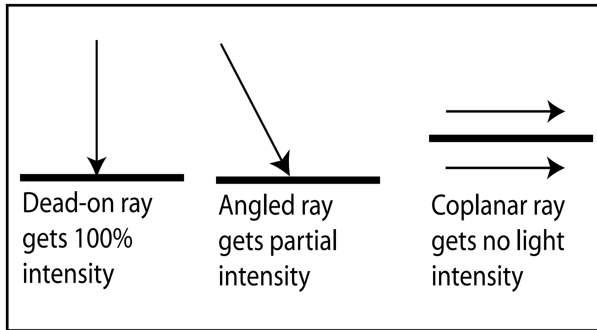

Figure 4.2: The intensity of light on a surface relates to the angle between the light direction and the surface normal.

Another consequence of the fact that a directional light has no position means that it does not attenuate over distance. The only factors to consider in the end are the direction and color of the light rays. One question that may come to mind is how this light influences the surface. As Figure 4.2 shows, you only need to consider the angle inbetween the light rays and the surface normal.

Knowing this, you can simply find out the intensity of the light on a particular point of the surface by taking the dot product inbetween the light direction and surface normal and factoring in the light color. This would yield the following shader code:

```
Color = Light_Color * saturate(dot( Light_Direction, inNormal ));
```

Note that in the above code that we use the *saturate* function. This function allows us to ensure a positive result as a face away from the light would get a negative lighting value. You could have also used the `clamp` function, but to clamp between zero and one, the `saturate` function will yield more efficient code.

Most of the light in a scene comes in the form of a light bulb, torch or similar lighting element. When you think about it, most of those sources are small, finite and located at a discreet position within your scene. For simplification, you may consider all of them as being a source of light contained as a single discreet point in your scene, therefore a point light.

With such lights, rays will emerge in a radial fashion as illustrated in Figure 4.3. This means that any object in any position relative to the light will get affected in the same way. Since the intensity of light on a surface is determined by the relationship between the light rays and the surface normal, the only extra step to take is to discover the direction of the rays of light in the first place. Determining

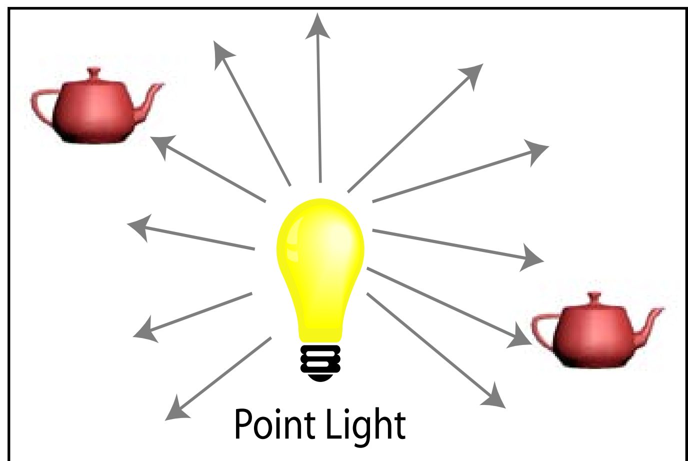

Figure 4.3: How rays of light emerge from a Point Light.

the vector from the source of light to the 3D position of the surface point you are considering can easily do this. Finding out the angle based attenuation for a point light yields the following code:

```
// Compute the normalized light direction vector and use it to determine
// the angular light attenuation.
float3 Light_Direction = normalize( inPos - Light_Position );
float AngleAttn = saturate(dot( inNormal, Light_Direction ));
```

Another consideration with point lights is that the light attenuates based on distance. The first step is to determine the distance from the surface to the light. This can be easily done with the following shader code:

```
float Distance = length( inPos - Light_Position );
```

Generally, point light will attenuate proportionally to the square of the distance from the light. To allow flexibility within your scene, however, you would adjust the light based on the inverse of the distance, distance and distance squared. Thiscode illustrates how it can be done:

```
// Compute distance based attenuation. This is defined as:
// Attenuation = 1 / ( LA.x + LA.y*Dist + LA.z*Dist*Dist )
float DistAttn = saturate(1 / ( LightAttenuation.x +
                               LightAttenuation.y * Dist +
                               LightAttenuation.z * Dist * Dist ));
```

Integrating the pieces of code developed earlier for a point light with the vertex shader gives you the following vertex shader code:

```
struct VS_OUTPUT
{
   float4 Pos:       POSITION;
   float2 TexCoord:  TEXCOORD0;
   float2 Color:     COLOR0;
};

float4 Light_PointDiffuse(float3 VertPos, float3 VertNorm, float3 LightPos,
                         float4 LightColor, float4 LightAttenuation)
{
   // Determine the distance from the light to the vertex and the direction
   float3 LightDir = LightPos - VertPos;
   float  Dist = length(LightDir);
   LightDir = LightDir / Dist;

   // Compute distance based attenuation. This is defined as:
   // Attenuation = 1 / ( LA.x + LA.y*Dist + LA.z*Dist*Dist )
   float DistAttn = saturate(1 / ( LightAttenuation.x +
                                  LightAttenuation.y * Dist +
                                  LightAttenuation.z * Dist * Dist ));
```

```
    // Compute surface/light angle based attenuation defined as dot(N,L)
    // Note: This must be clamped as it may become negative.
    float AngleAttn = saturate(dot(VertNorm, LightDir) );

    // Compute final lighting
    return LightColor * DistAttn * AngleAttn;
}

VS_OUTPUT vs_main(float4 inPos: POSITION, float3 inNormal: NORMAL,
                  float2 inTxr: TEXCOORD0)
{
    VS_OUTPUT Out;

    // Compute the projected position and send out the texture coordinates
    Out.Pos = mul(view_proj_matrix, inPos);
    Out.TexCoord = inTxr;

    float4 Color;

    // Compute light contribution
    Color = Light_PointDiffuse(inPos, inNormal, Light1_Position,
                               Light1_Color, Light1_Attenuation);

    // Output Final Color
    Out.Color = Color;

    return Out;
}
```

The function *Light_PointDiffuse* is used to factor out the lighting code so you may write shaders using multiple lights. We'll do this in later chapters.

Now that we have a diffuse point light shader, we need to consider the specular portion to the lighting equation. The major difference with specular lighting is that the lighting intensity is not only influenced by the surface to light angle but also by the light to viewer angle.

A common way in which this is implemented is with what is called the half vector. This vector is essentially the halfway vector in-between the view vector and the incident light vector.

The view vector is determined by taking a view position and transforming it into object space. In this case we'll assume a vector at (0,0,10). This object-space view vector can then be combined to the vertex position. The following code illustrates how this is done:

```
EyeVector = -normalize(mul(inv_view_matrix,float4(0,0,10,1))+inPos);
```

With this *EyeVector*, you can determine the half vector by combining it with the incident light vector and then normalizing the result. Since both vectors are normalized to start with, the result is equivalent to averaging them with *(A+B)/2*. The following code shows how you can do this:

```
HalfVect = normalize(LightDir-EyeVector);
```

Once you have the half vector, the angle-based light intensity can simply be determined by computing the dot product between the half vector and the surface normal and raising the result to the power of *m*. Raising the result to a certain power controls the specular exponent of the light. The higher the value, the smaller and sharper the lighting highlight will be. For this example shader, we'll assume a specular power of 32, but this can be adjusted as needed.

Combining all these elements into a function called *LightPoint_Specular* gives the following vertex shader code:

```
struct VS_OUTPUT
{
   float4 Pos:       POSITION;
   float2 TexCoord:  TEXCOORD0;
   float2 Color:     COLOR0;
};

float4 Light_PointSpecular(float3 VertPos, float3 VertNorm, float3 LightPos,
                           float4 LightColor, float4 LightAttenuation,
                           float3 EyeDir)
{
   // Determine the distance from the light to the vertex and the direction
   float3 LightDir = LightPos - VertPos;
   float  Dist = length(LightDir);
   LightDir = LightDir / Dist;

   // Compute half vector
   float3 HalfVect = normalize(LightDir-EyeDir);

   // Compute distance based attenuation. This is defined as:
   // Attenuation = 1 / ( LA.x + LA.y*Dist + LA.z*Dist*Dist )
   float DistAttn = saturate(1 / ( LightAttenuation.x +
                                   LightAttenuation.y * Dist +
                                   LightAttenuation.z * Dist * Dist ));

   float SpecularAttn =  pow( saturate(dot(VertNorm, HalfVect)),32);

   // Compute final lighting
   return LightColor * DistAttn * SpecularAttn;
}

VS_OUTPUT vs_main(float4 inPos: POSITION, float3 inNormal: NORMAL,
                  float2 inTxr: TEXCOORD0)
{

   VS_OUTPUT Out;
```

```
    // Compute the projected position and send out the texture coordinates
    Out.Pos = mul(view_proj_matrix, inPos);
    Out.TexCoord = inTxr;

    // Output the ambient color
    float4 Color = Light_Ambient;

    // Determine the eye vector
    float3 EyeVector = -normalize(mul(inv_view_matrix,float4(0,0,10,1))+inPos);

    // Compute light contribution
    Color = Light_PointSpecular(inPos, inNormal, Light1_Position,
                                Light1_Color, Light1_Attenuation,
                                EyeVector);

    // Output Final Color
    Out.Color = Color;

    return Out;
}
```

When considering lighting, most people think that lighting an object per-vertex is enough. This is generally true for highly tessellated objects but fails on complex shapes and low detail ones. The reason behind this is the way interpolation of colors between vertices happens.

When you light your object per-vertex, the lighting color is calculated once for each vertex and then linearly interpolated across the polygon. The reality, however, is that the lighting values are dependent on the incident light angle, the surface normal and the viewer position (for specular lighting). The interpolation of color resulting from the per-vertex lighting calculations does not give the same results as interpolating each component individually, especially on large polygons.

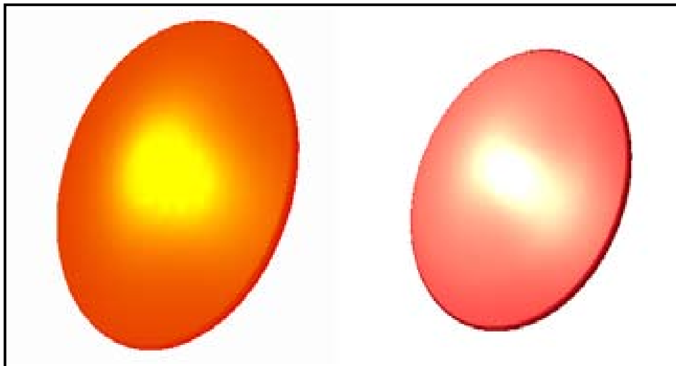

Figure 4.4: Vertex and pixel-based lighting for a simple flat surface lit by a single point light.

Imagine the following case where you have one flat surface being lit by a single point light. With per-vertex lighting, the color will be determined for each vertex then interpolated. With per-pixel lighting, you will interpolate each component needed for lighting then calculate on a per-pixel basis. Figure 4.4 shows you an example of the difference between vertex and pixel based lighting for a simple flat surface.

When dealing with high polygon counts, per-vertex lighting does well because each individual polygon covers little of

the screen, and each pixel suffers from little interpolation error. When dealing with lower detail geometry, however, pixel- based lighting is the way to go for correct lighting!

Another added advantage of per-pixel lighting is the ability to add detail, which doesn't exist in the original mesh. By using bump maps and normal maps, you can add bumpiness to your geometry on a per-pixel level, which doesn't exist on the geometry but which creates the illusion of such detail by using appropriate lighting.

Starting with diffuse lighting, you will need to decide what lighting components can be interpolated to be sent to the pixel shader and which ones need to be processed per-pixel. From the end to the beginning, what defines diffuse lighting is the dot product in-between the surface normal and the light vector.

This dot product defines the intensity of the lighting, but the results of such an operation cannot be interpolated properly. Because of this, the dot product will need to be processed for each pixel and moved to the pixel shader.

The two components of this dot product are the surface normal and light vector. Vectors will interpolate correctly when put in on a per-vertex basis. This means you can calculate them ahead of time for each vertex and pass them on to the pixel shader, which will take care of computing the final dot product.

Note

Although normals can be interpolated linearly, such interpolation can yield vectors that are not normalized. To correct this, it is a good idea to renormalize the vector by using the built-in HLSL `normalize` function for normals, which are interpolated in the transition from a vertex to a pixel shader.

For this to happen, you will need to add those two components to the vertex shader output structure. They can be passed to the pixel shader using the *TEXCOORD1* and *TEXCOORD2* semantics. Applying the modifications will yield the following output structure:

```
struct VS_OUTPUT
{
   float4 Pos:       POSITION;
   float2 TexCoord:  TEXCOORD0;
   float3 Normal:    TEXCOORD1;
   float3 LightDir:  TEXCOORD2;
};
```

You will also need to change the vertex shader to pass those values to the output structure. Since you will be dealing with a single light, it will be easier at this point to remove the lighting function and

copy the relevant code in the *vs_main* function. Keep in mind that we will be dealing with multiple lights later on in this book.

One thing you may have noticed that was not mentioned is the distance based attenuation. Although the interpolation of this component is not perfect, lights have a sufficient range of action, and small differences in distance will not to make a significant difference in the result, and hence does not need to be computed per-pixel.

Since this attenuation factor is a single scalar value, the easiest way to pass it to the pixel shader without wasting another register is to simply make the *LightDir* vector a *float4* and put this result in the *W* component of this vector. Applying all those changes should give you the following vertex shader:

```
struct VS_OUTPUT
{
   float4 Pos:       POSITION;
   float2 TexCoord:  TEXCOORD0;
   float3 Normal:    TEXCOORD1;
   float4 LightDir:  TEXCOORD2;
};

VS_OUTPUT vs_main(float4 inPos: POSITION, float3 inNormal: NORMAL,
                  float2 inTxr: TEXCOORD0)
{
   VS_OUTPUT Out;

   // Compute the projected position and send out the texture coordinates
   Out.Pos = mul(view_proj_matrix, inPos);
   Out.TexCoord = inTxr;

   // Move the normal to the pixel shader
   Out.Normal = inNormal;

   // Compute and move the light direction to the pixel shader
   float4 LightDir;
   LightDir.xyz = Light1_Position - inPos;
   float Dist = length(LightDir.xyz);
   LightDir.xyz = LightDir.xyz / Dist;

   // Compute the distance based attenuation. Distance can be interpolated
   // fairly well so we will precompute it on a per-vertex basis.
   LightDir.w = saturate(1 / ( Light1_Attenuation.x +
                               Light1_Attenuation.y * Dist +
                               Light1_Attenuation.z * Dist * Dist ));

   // Output the light direction
   Out.LightDir = LightDir;
   return Out;
}
```

On the pixel shader end, the task is simply a matter of taking the incoming vectors and computing the dot product and the final lighting color. For the convenience, the lighting code can be put in a function called *Light_PointDiffuse* as was done earlier in the per-vertex lighting shader. In essence, the lighting computation code is the same as the vertex shader version.

You will also need to add the input parameters to the pixel shader main function, *ps_main*, so the interpolated normal and light direction can be read. With all those adjustments, you should end up with the following pixel shader:

```
float4 Light-PointDiffuse(float4 LightDir,
                          float3 Normal,
                          float4 LightColor)
{
   // Compute suface/light angle based attenuation defined as dot(N,L)
   float AngleAttn = saturate(dot(Normal, LightDir.xyz) );

   // Compute final lighting (Color * Distance Attenuation *
   // Angle Attenuation)
   return LightColor * LightDir.w * AngleAttn;
}

float4 ps_main(float3 inNormal:TEXCOORD1,
               float4 inLightDir:TEXCOORD2) : COLOR
{
   // Compute the lighting contribution for this single light
   return Light_PointDiffuse(inLightDir,inNormal,Light1_Color);
}
```

Not too hard yet, huh? Now it is time to tackle the specular per-pixel lighting shader… The basic process is the same as what was done with the diffuse lighting shader. The core of this shader is the dot product between the light vector and the half vector. As with the diffuse lighting shader, this computation will need to be done on the pixel shader.

This means that both the light vector and the half vector will need to be calculated in the vertex shader and passed to the pixel shader. In the same way you have done with the diffuse lighting shader, you will need to add the proper variables to the output structure. With the changes, the output structure should look as follows:

```
struct VS_OUTPUT
{
   float4 Pos:       POSITION;
   float2 TexCoord:  TEXCOORD0;
   float3 Normal:    TEXCOORD1;
   float4 LightDir:  TEXCOORD2;
   float3 HalfVect:  TEXCOORD3;
};
```

Within the vertex shader code, you will need to calculate the normal, light vector, half vector and distance-based attenuation. This is simply a matter taking the specular lighting shader code developed in the previous chapter and applying a few changes. Once you have done this, you should have the following code:

```
struct VS_OUTPUT
{
   float4 Pos:       POSITION;
   float2 TexCoord:  TEXCOORD0;
   float3 Normal:    TEXCOORD1;
   float4 LightDir:  TEXCOORD2;
   float3 HalfVect:  TEXCOORD3;
};

VS_OUTPUT vs_main(float4 inPos: POSITION, float3 inNormal: NORMAL,
                  float2 inTxr: TEXCOORD0)
{
   VS_OUTPUT Out;

   // Compute the projected position and send out the texture coordinates
   Out.Pos = mul(view_proj_matrix, inPos);
   Out.TexCoord = inTxr;

   // Determine the distance from the light to the vertex and the direction
   float4 LightDir;
   LightDir.xyz = Light1_Position - inPos;
   float  Dist = length(LightDir.xyz);
   LightDir.xyz = LightDir.xyz / Dist;

   // Compute the per-vertex distance based attenuation
   LightDir.w = saturate(1 / ( Light1_Attenuation.x +
                               Light1_Attenuation.y * Dist +
                               Light1_Attenuation.z * Dist * Dist ));

   // Determine the eye vector and the half vector
   float3 EyeVector = -normalize(mul(inv_view_matrix,float4(0,0,10,1))+inPos);
   Out.HalfVect = normalize(LightDir-EyeVector);

   // Output normal and light direction
   Out.Normal = inNormal;
   Out.LightDir = LightDir;

   return Out;
}
```

On the pixel shader front, it is simply a matter of creating a function called `Light_PointSpecular`, which will take the input vectors and compute the dot product. With this, you can then consider the lighting color and the distance attenuation. The following pixel shader code implements all these changes:

```
float4 Light_PointSpecular(float3 Normal, float3 HalfVect, float4 LightDir,
                           float4 LightColor)
{
   // Compute suface/light angle based attenuation defined as dot(N,L)
   // Note : This must be clamped as it may become negative.
   float SpecularAttn =  pow( saturate(dot(Normal, HalfVect)),32);

   // Compute final lighting
   return LightColor * LightDir.w * SpecularAttn;
}

float4 ps_main(float3 inNormal:TEXCOORD1, float4 LightDir:TEXCOORD2,
              float3 HalfVect:TEXCOORD3) : COLOR
{
   // Simply route the vertex color to the output
   return Light_PointSpecular(inNormal,HalfVect,LightDir,Light1_Color);
}
```

Bumpy Surfaces

Now that you can light each pixel of your object individually, we'llexplore the topic of normal mapping. Since you are rendering each pixel individually, what prevents you from adding extra information to each pixel, which can add some extra details to the geometry?

The reality is that real objects are not just bound by polygons and that details go much deeper. Imagine a brick wall… Even though you can represent this wall with a simple planar polygon and textures, bricks have much more 3D detail not represented by such geometry. Figure 4.5 shows this in more detail…

All this extra detail could be represented by geometry but would create tremendous amounts of polygons thatwould be almost impossible to manage (at least with today's hardware). Another solution to the problem is to consider the impact of the added detail visually. Since the extra geometry details are small, they do not affect the shape of the object itself but mostly will have an impact on how an object gets lit.

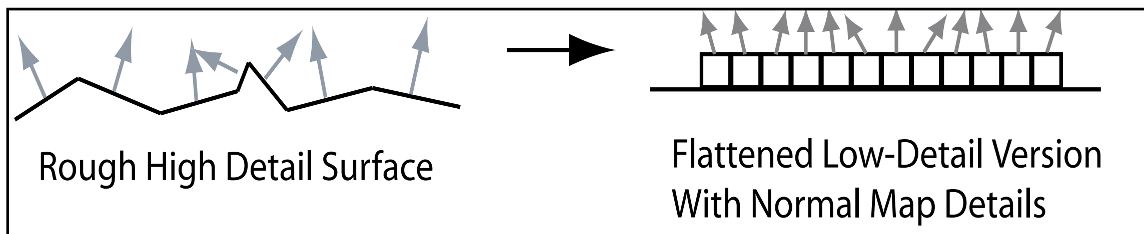

Figure 4.5: Small extra detail compared to the flat polygon representation.

Now that you are rendering each pixel individually nothing prevents you from modifying the normal for each pixel so that it takes account of the added details for this portion of the object. This is where normal mapping comes in to play.

When you are dealing with normal maps, each pixel refers to the current surface normal which is usually either in object space or world space. However, since you are texturing your object with a texture or bump map, which isn't necessarily built for this object, there is no way for the bump map or normal map to specify a new normal without any knowledge of the surface of the object. One solution is to ensure that your texture has a relationship with your object and that the normal map refers to the normals in object-space. This is impractical, however, as it prevents texture reuse and requires special texture authoring for each bumped object in your scene. The fact that the bump texture has knowledge of the surface of the objects will also prohibit you from animating or deforming the surface, restricting your use of normal maps even further.

A more practical approach is to create a new uniform coordinate system, which would be the same for every pixel on your object and from your normal texture can be built. To accomplish this, you need to consider that if your object was not bumped, the normal for each point on the surface would be perpendicular to the surface itself.

When applying a normal map to a surface, you are in essence modifying the normal along the two vectors which go along the `U` and `V` texture coordinates. Putting all this together yields a three vector coordinate system which can be calculated on a per-pixel basis. Figure 4.6 illustrates this coordinate system.

As you can see from Figure 4.6, the vectors along the `U` and `V` texture coordinates are respectively called tangent and binormal. Those vectors are defined as the 3D spatial direction in which the `U` and `V` texture coordinates are headed for a particular point on the surface. How to generate those vectors is outside the scope of this book, but be assured the *D3DXMESH* interface within DirectX gives you functionality to create tangent space vectors.

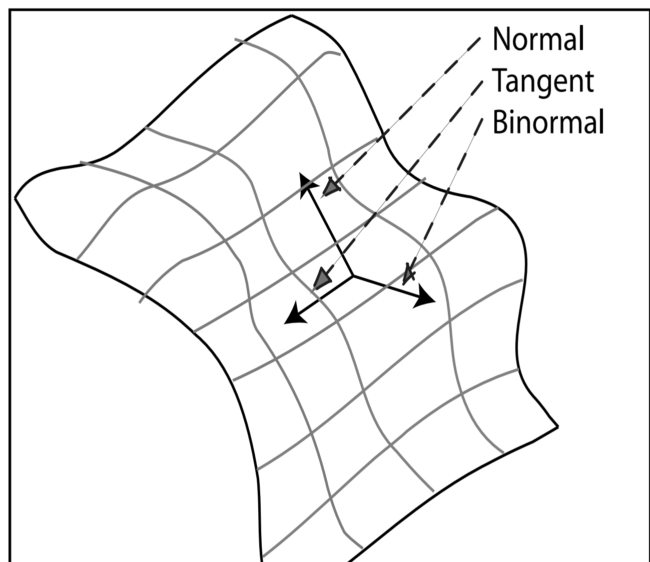

Figure 4.6: How the tangent space coordinate system is generated.

When doing lighting with tangent space, all you need to do is build a tangent space matrix from the surface normal, tangent

and binormal vector. This matrix can then be used to transform any of the lighting components into this coordinate system by multiplying it by the tangent space matrix.

Once this operation has been completed, all your lighting components will be interpolated relative to the local tangent space for each pixel. The following piece of code shows how you can make a tangent space matrix and use it to convert a vector from object space to tangent space:

```
// Build the tangent space matrix
float3x3 TangentSpace;
TangentSpace[0] = inTangent;
TangentSpace[1] = inBinormal;
TangentSpace[2] = inNormal;

// Transform a vector from object space to tangent space
LightDir = mul(TangentSpace,LightDir);
```

When working in tangent space, all lighting vectors become relative to the interpolated surface normal for this pixel. This means that when doing operations such as bump mapping, you can now store the values a simple relative values since you are working with a coordinate system which represents the same thing no matter what point on the surface you are dealing with. In the case of a normal map, the texture uses a three-color component, which represents the actual normal of the surface in tangent space.

With this knowledge, it is now time to take our per-pixel diffuse/specular shader and use a normal map to enhance the lighting on the object. The changes required to the vertex shader are simple. The first step is to introduce the tangent space matrix code developed in the previous section. Then, you will need to transform both the half-vector and light vector into tangent space before passing them to the pixel shader.

Keep in mind that because you are working in tangent space, the orientation of the surface normal is implicit, and its deviations will be read from the normal map. Because of this, it does not need to be passed to the pixel shader, and you can remove this from the output structure.

With this set of adjustments, you should obtain the following vertex shader code:

```
struct VS_OUTPUT
{
   float4 Pos:       POSITION;
   float2 TexCoord:  TEXCOORD0;
   float4 LightDir:  TEXCOORD1;
   float3 HalfVect:  TEXCOORD2;
};

VS_OUTPUT vs_main(float4 inPos: POSITION, float3 inNormal: NORMAL,
                  float3 inTangent:TANGENT, float3 inBinormal:BINORMAL,
                  float2 inTxr: TEXCOORD0)
```

```
{
   VS_OUTPUT Out;

   // Compute the projected position and send out the texture coordinates
   Out.Pos = mul(view_proj_matrix, inPos);
   Out.TexCoord = inTxr;

   // Determine the distance from the light to the vertex and the direction
   float4 LightDir;
   LightDir.xyz = mul(inv_view_matrix,float3(80,00,-80)) - inPos;
   float  Dist = length(LightDir.xyz);
   LightDir.xyz = LightDir.xyz / Dist;

   // Compute the per-vertex distance based attenuation
   LightDir.w = saturate(1 / ( Light1_Attenuation.x +
                               Light1_Attenuation.y * Dist +
                               Light1_Attenuation.z * Dist * Dist ));

   // Determine the eye vector
   float3 EyeVector = -normalize(mul(inv_view_matrix,float4(0,0,10,1))+inPos);

   // Transform to tangent space and output
   // half vector and light direction
   float3x3 TangentSpace;
   TangentSpace[0] = inTangent;
   TangentSpace[1] = inBinormal;
   TangentSpace[2] = inNormal;
   Out.HalfVect = mul(TangentSpace,normalize(LightDir.xyz+EyeVector));
   Out.LightDir = float4(mul(TangentSpace,LightDir.xyz),LightDir.w);

   return Out;
}
```

On the pixel shader's end, the only change needed is in relation to the surface normal. Since all the lighting components are passed to the pixel shader in tangent space, this means that they are already relative to the interpolated vertex surface normal, which is implicit through the definition of tangent space. The actual normal on any point of the surface will then come from the normal map.

To do this, you will need to sample the normal map texture and convert it from an unsigned value to a signed one. This value will then be the pixel specific normal and can be substituted to the vertex interpolated normal from the previous shader implementations.

The final pixel shader code for this bumped lighting shader is as follows:

```
float4 Light_Point(float3 Normal, float3 HalfVect, float4 LightDir,
                   float4 LightColor)
{
   // Compute both specular and diffuse factors
   float SpecularAttn =  pow( clamp(0, 1,dot(Normal, HalfVect)),16);
```

```
   float DiffuseAttn =  saturate(dot(Normal, LightDir));

   // Compute final lighting
   return LightColor * LightDir.w * (SpecularAttn+DiffuseAttn);
}

float4 ps_main(float2 inTxr:TEXCOORD0,float4 LightDir:TEXCOORD1,
               float3 HalfVect:TEXCOORD2) : COLOR
{
   // Read bump and influence the normal
   float3 normal = tex2D(Bump,inTxr) * 2 - 1;

   // Simply route the vertex color to the output
   return tex2D(Texture0,inTxr)*
          (0.15+Light_Point(normal,HalfVect,LightDir,Light1_Color));
}
```

Summary and what's next?

In this chapter I have showed you a few basic shaders, which will be used throughout the book to help with the construction of shaders. Remember, this book isn't intended to teach you how to build shaders, and I assumed you had some understanding of shaders and lighting throughout this chapter. The important piece of information for you to understand is how you can use the effect system to manage your shaders.

This is exactly what I will start discussing over the next chapters. Now that I've gone over the HLSL shader language, it is time to discuss the effect system and how you can use it to you advantage. So, get ready! It is now time for the main course.