Chapter 5

DirectX

Chas Boyd

Chapter 5 Hardware Shading with Direct3D

This white paper provides a short history and overview of the hardware accelerated shading models and the mechanisms for unifying them that are provided by the DirectX graphics library known as D3DX for use with the Direct3D low-level API.

Goals

Direct3D was developed to meet requirements for game programmers. They have two primary needs:
1) Deliver novel visual experiences based on the increasing features and performance available from graphics accelerator fabrication processes.
2) Support enough total systems to enable a reasonable sales volume.

This last requirement means the API needs to span hardware implementations along two axes. It must not only support multiple manufacturers, but also the different generations of hardware from each manufacturer. Due to the rapid rate of innovation used to deliver novel experiences, there are often multiple generations of a given brand of hardware in the consumer marketplace, and while the latest version of a given accelerator can be very important to a game developer to target, the publishers of each game title would like to take advantage of a larger installed base by including earlier hardware.

While creating an API that can help applications span multiple brands of hardware is difficult, helping them span consecutive generations is actually the tougher problem. Direct3D has typically sorted the features differences into those that can be accommodated by relatively localized code branches (for which capability bits are provided), and those that represent generational changes.

Much of the recent work on DirectX Graphics has been dedicated to helping applications with this latter task. A key result of this work is the D3DX Effect Framework, a mechanism for helping applications manage different rendering techniques enabled by the various generations of hardware.

Effects

With the change in hardware model for multi-texture, it became clear that a successful game would need to provide different code paths for each of the 2 or 3 generations of hardware it targeted. This is because each new generation of hardware has not only more advanced features, but also improved performance. This makes it very difficult to emulate the newer generation's features on older hardware. The performance of emulating a technique is usually worse than the true technique, yet that older hardware already has less performance.

For example, many multi-texture blending operations can be emulated by multi-pass frame-buffer blending operations. However, the older hardware that requires multi-pass

emulation is so much slower that to provide acceptable performance it really should be used for fewer passes, not more.

As a result, separate code paths and in some cases separate art content (texures and models) are often needed for each generation of hardware. This level of impact on application architecture is non-trivial, however, many applications began to be implemented using this type of architecture

Starting in DirectX7, support was provided for effects in the D3DX library of extensions to Direc3D. Effects are a way of architecting rendering code that isolates and manages implementation-specific details from the application, allowing it to more easily span generations of hardware, or other variations in implementations. An effect object is a software abstraction that gathers together a collection of different techniques which could be used to implement a common logical visual effect.

There is very little policy associated with them. The different techniques used to implement an effect can be selected based on any criterion, not just feature set. Criteria such as performance, distance from the camera, user preferences etc. are also commonly used, and sometimes combined together.

Effect objects can be loaded from and persisted in .fx files. This enables them to serve as collections of binding code that map various rendering techniques into applications.

Because of the efficiency of the effect loading process, they can be dynamically loaded while an application is running. This is extremely useful for efficient application development. When an effect is modified while the application is running, it not-only saves the time required to reload the app that would be required using a compile-based process, but more importantly, the time for the app to reload all the models and textures it is using.

Like OpenGL, Direct3D does not have a concept of geometric objects to which shaders are assigned to directly. The API is designed to be a flexible substrate for implementation of higher-level object models such as hierarchical scene graphs, cell-portal graphs, BSPs, etc. The data model is based on vertices, textures, and the state that controls the device, of which shaders are a part.

This flexibility is preserved in the Effect binding mechanism. Effects manage only the state information, so the application is free to render its content using any low-level rendering calls it considers appropriate.

Effects provide a clean way to make an application scale from single-texture to multi-texture to programmable shaders. Or to manage hardware vs software vertex transformation pipelines.

The following listings show the evolution of the hardware and API generations using multi-texture, assembly level shaders, and the C-level language.

DirectX 6 Multi-Texture

The following listing shows the code required to implement a specular per-pixel bump map using the dependent read mechanism introduced in late 1998 in DirectX 6.0, and supported in hardware on the Matrox G400 in 1999, the ATI Radeon in 2000, and the nVidia GeForce 3 in 2001, among others.

```
/*  Render()
Implements the following specular bump mapping rendering using multitexture syntax

0   MODULATE( EarthTexture, Diffuse );   // light the base texture
1   BUMPENVMAP( BumpMap, _ );            // sample bump map (other arg ignored)
2   ADD( EnvMapTexture, Current );       // sample envt map using bumped texcoords
                                         // and add to result
*/
HRESULT CMyD3DApplication::Render()
{
    m_pd3dDevice->Clear( 0L, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                         0x00000000, 1.0f, 0L );

    m_pd3dDevice->BeginScene();

    m_pd3dDevice->SetRenderState( D3DRS_WRAP0, D3DWRAP_U | D3DWRAP_V );

    m_pd3dDevice->SetTexture( 0, m_pEarthTexture );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_MODULATE );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );

    m_pd3dDevice->SetTexture( 1, m_psBumpMap );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,   D3DTOP_BUMPENVMAP );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT00,   F2DW(0.5f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT01,   F2DW(0.0f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT10,   F2DW(0.0f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT11,   F2DW(0.5f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLSCALE,  F2DW(4.0f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLOFFSET, F2DW(0.0f) );

    m_pd3dDevice->SetTexture( 2, m_pEnvMapTexture );
    m_pd3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX, 0 );
    m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLOROP,   D3DTOP_ADD );
    m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG2, D3DTA_CURRENT );


    m_pd3dDevice->SetStreamSource( 0, m_pEarthVB, 0, sizeof(BUMPVERTEX) );
    m_pd3dDevice->SetFVF( BUMPVERTEX::FVF );

    if( FAILED( m_pd3dDevice->ValidateDevice( &dwNumPasses ) ) )
    {
        // The right thing to do when device validation fails is to try
        // a different rendering technique.  This sample just warns the user.
```

```
        m_bDeviceValidationFailed = TRUE;
    }
    else
    {
        m_bDeviceValidationFailed = FALSE;
    }

    // Finally, draw the Earth
    m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, m_dwNumSphereVertices );

    return;
}
```

DirextX 8 Pixel Shaders

The following listing shows an effect file containing a single technique that implements the homomorphic factorization method of McCool, Ang, and Ahmad for rendering BRDFs. It uses DirectX 8 pixel shader version 1.1 and vertex shader version 1.1.

```
//////////////////////////////////////////////////////////////////////////
// BRDF Effect File
// Copyright (c) 2000-2002 Microsoft Corporation. All rights reserved.
//

vector lhtR;            // Light Direction from app
vector MaterialColor;   // Object Diffuse Material Color


matrix mWld;     // World
matrix mTot;     // Total


texture BRDFTexture1;
texture BRDFTexture2;
texture BRDFTexture3;
texture ObjectTexture;

// These strings are for the app loading the fx file

// Technique name for display in viewer window:
string tec0 = "BRDF Shader";

// Background Color
DWORD  BCLR = 0xff0000ff;

// model to load
string XFile = "sphere.x";

// BRDF technique
technique tec0
{
    pass p0
    {
        //load matrices
        VertexShaderConstant[0] = <mWld>;           // World Matrix
        VertexShaderConstant[4] = <mTot>;           // World*View*Proj Matrix

        //Material properties of object
        VertexShaderConstant[9]  = <MaterialColor>;

       // Light Properties.
       // lhtR, the light direction, is input from the shader app
       // for BRDFs, these color constants are built into the texture maps
        VertexShaderConstant[16] = <lhtR>;            // light direction

        Texture[0] = <BRDFTexture1>;
        Texture[1] = <BRDFTexture2>;
        Texture[2] = <BRDFTexture3>;
```

```
Texture[3] = <ObjectTexture>;

// Only one colour being used
ColorOp[1]   = Disable;
AlphaOp[1]   = Disable;

AddressU[0] = clamp;       // set up clamping for cube maps
AddressV[0] = clamp;
AddressW[0] = clamp;

AddressU[1] = clamp;
AddressV[1] = clamp;
AddressW[1] = clamp;

AddressU[2] = clamp;
AddressV[2] = clamp;
AddressW[2] = clamp;

// object's detail texture
AddressU[3] = wrap;
AddressV[3] = wrap;


// Definition of the vertex shader, declarations then assembly.
VertexShader =
decl
{
    stream 0;
    float v0[3];        // Position
    float v3[3];        // Normal
    float v7[3];        // Texture Coord1
    float v8[3];        // Tangent
}
asm
{
    vs.1.1              // version number

    m4x4 oPos, v0, c4   // transform point to projection space
    m4x4 r4,v0,c0       // transform point to world space
    m3x3 r0,v3,c0       // transform Normal to World Space, result in r0
    m3x3 r1,v8,c0       // transform Tangent to World Space

    mul r2,-r1.zxyw,r0.yzxw;      // comptue binorm with cross product
    mad r2,-r1.yzxw,r0.zxyw,-r2;

    // get negative view vector
    add r4.xyz,-r4.xyz,c10
    dp3 r5.x,r4.xyz,r4.xyz
    rsq r5.x,r5.x
    mul r4.xyz,r5.xxx,r4.xyz

    //compute the half angle and normalize
    add r5.xyz,r4.xyz,-c16
    dp3 r6.x,r5.xyz,r5.xyz
    rsq r6.x,r6.x
    mul r5.xyz,r6.xxx,r5.xyz
```

```
            //now inverse transform everything
            //r1 = tangent
            //r2 = binormal
            //r0 = normal

            //t0 = view
            dp3 oT0.x,r4,r1
            dp3 oT0.y,r4,r2
            dp3 oT0.z,r4,r0

            //t1 = -light
            dp3 oT1.x,-c16,r1
            dp3 oT1.y,-c16,r2
            dp3 oT2.z,-c16,r0

            //t2 = half angle
            dp3 oT2.x,r5,r1
            dp3 oT2.y,r5,r2
            dp3 oT2.z,r5,r0

            //object might have its own texture
            mov oT3.xy,v7.xy

            //move diffuse color in
            mov oD0,c9
        };

        pixelshader =
        asm
        {
            ps.1.1
            tex t0
            tex t1
            tex t2
            tex t3

            mul r0,t0,t1     // combine 1st 2 brdf factors
            mul r0,r0,t2     // multiply in 3rd factor
            mul r0,r0,t3     // apply detail texture
            mul r0,r0,v0     // light using diffuse primary color
        };
    }
}
```

DirectX 9 High Level Shading Language

The following is a very simple wood shader from the Renderman Companion. Implemented as a DirectX 9.0 effect file, it demonstrates the integration of the "C"-like high-level shading language into the effect framework. This should run in 1 pass of the pixel shader 2.0 model supported in DirectX 9.0.

```
fx.2.0

//////////////////////////////////////////////////////////////////////////////
// Effect parameters /////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

float ringscale = 10.0f;
float point_scale = 1.0f, turbulence = 1.0f;

vec3 lightwood = vec3(0.3f, 0.12f, 0.03f);
vec3 darkwood  = vec3(0.05f, 0.01f, 0.005f);

float Ka = 0.2;
float Kd = 0.4;
float Ks = 0.6;
float roughness = 0.1;

vec3 ambient_color;
vec3 diffuse_color;
vec3 specular_color;

volumetexture noise;

vec3 L;
vec3 I;


//////////////////////////////////////////////////////////////////////////////
// Renderman-like helper functions ///////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

float smoothstep(float a, float b, float s)
{
    float s2 = s * s;
    float s3 = s * s2;

    float sV1 = 2.0f * s3 - 3.0f * s2 + 1.0f;
    float sT1 = s3 - 2.0f * s2 + s;
    float sV2 = -2.0f * s3 + 3.0f * s2;
    float sT2 = s3 - s2;

    return s < 0.0f ? a : s > 1.0f ? b : sV1 * a + sT1 + sV2 * b + sT2;
}

vec3 diffuse(vec3 N)
{
    return diffuse_color * dot(N, L)
}
```

```
vec3 specular(vec3 N, vec3 eye, float roughness)
{
    vec3 H = (L + eye) / len(L + eye);
    return specular_color * pow(dot(H, N), 1 / roughness);
}



////////////////////////////////////////////////////////////////////////////
// Wood Shader ////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

struct PS_INPUT
{
    vec3 Color    : COLOR0;
    vec3 Position : TEXCOORD0;
    vec3 Normal   : TEXCOORD1;
};

struct PS_OUTPUT
{
    vec4 Color : COLOR0;
};



PS_OUTPUT Wood(const PS_INPUT v)
{
    /* Perturb P to add irregularity */
    vec3 PP = v.Position * point_scale;
    PP += sample3d(noise, PP) * turbulence;

    /* Compute radial distance r from PP to axis of tree */
    float r = sqrt(PP.y * PP.y + PP.z * PP.z);

    /* Map radial distance r into ring position [0,1] */
    r *= ringscale;
    r += abs(sample1d(noise, r) * turbulence);
    r  = frc(r);

    /* Use r to select wood color */
    r = smoothstep(0.0f, 0.8f, r) - smoothstep(0.83f, 1.0f , r);

    /* Shade using r to vary brightness of wood grain */
    PS_OUTPUT o;

    o.Color.rgb = (ambient_color * Ka + diffuse(v.Normal) * Kd) *
        lerp(lightwood, darkwood, r) * v.Color.a +
        specular(v.Normal, -I, roughness) * (Ks * (0.3f * r + 0.7f));

    o.Color.a = v.Color.a;

    return o;
};
```

Additional Resources

Chas. Boyd
chasb@microsoft.com

DirectX email contact for applications to beta program:
directx@microsoft.com

Developer web site with links, whitepapers, and SDK downloads:
http://msdn.microsoft.com/DirectX

Presentations on techniques at previous conferences:
http://www.microsoft.com/mscorp/corpevents/meltdown2001/presentations.asp

http://www.microsoft.com/mscorp/corpevents/gdc2001/developer_day.asp

Per-Pixel Lighting

Philip Taylor

Microsoft Corporation

November 13, 2001

Download the source code for this article.

Note This download requires DirectX 8.1.

This column is based on material contained in the "Per-Pixel Lighting" talk, developed and delivered by Dan Baker and Chas Boyd at GDC 2001. In the interests of space, I am not going to cover several advanced topics covered in the slide material (available at http://www.microsoft.com/mscorp/corpevents/meltdown2001/presentations.asp and http://www.microsoft.com/corpevents/gdc2001/developer_day.asp) like anisotropic lighting, and per-pixel environment mapping.

Instead, this column will focus on the fundamentals of pixel lighting, the standard models, the process of defining new models, and provide an example of defining and implementing a new lighting model using pixel shaders. It's in the area of custom or "do-it-yourself" lighting models that pixel shaders really shine—but lets not get ahead of ourselves.

Fundamentals of Per-Pixel Lighting

First, I assume everyone is familiar with basic diffuse and specular lighting. This assumes a physical model, like that shown in Figure 1. Let's examine the standard lighting model, and define the system and the terminology.

Figure 1 below is a diagram showing the standard lighting setup used to describe Direct3D's fixed-function lighting. There is a vertex, defined by the position P, a Light, defined by the L vector, the View position defined by the V vector, and the Normal defined by the N vector. In addition, the diagram shows the "half-vector" H, part of the Blinn simplification to Phong shading. These elements are sufficient to describe both the diffuse and specular reflectance lighting system.

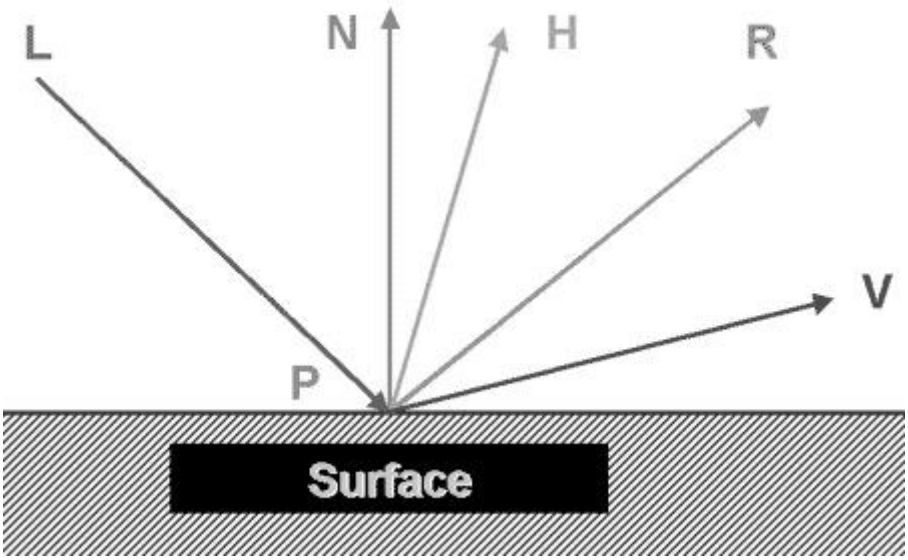

Figure 1. Standard lighting diagram

P = Vertex position
N = unit normal vector of Vertex P
L = unit vector from Light to the Vertex position
V = unit vector from Vertex P to the View position V
R = unit vector representing light reflection R
H = unit vector halfway H, between L and V, used for Blinn simplification

Diffuse lighting uses the relationship N dot L, where dot is the dot product, to determine the diffuse contribution to color. The dot product represents the cosine of the angle between the two vectors, so when:

The angle is acute, or small, the cosine value is large, and so this component contribution to the final color is larger.

The angle is obtuse, or large, the cosine value is small, and so this components contribution to the final color is smaller.

The Phong formulation for specular lighting uses a reflected vector R, representing the direction the light is reflected, with the Light vector L, R dot V, raised to a power n. The power value n allows simulation of a variety of surfaces, so when:

The power value n is large, the resulting highlight is tight and shiny, simulating a glossy surface.

The power value n is small, the resulting highlight is large and dull, simulating a less glossy surface.

The Blinn simplification replaces the R vector with the vector H halfway between V and L, and modifies the power value n to produce a result sufficiently similar to the more complex calculation for good image

quality purposes—but at a significantly lower computational cost, since H is much cheaper to calculate than R.

Direct3D lighting, whether in hardware or software, uses these equations at the vertex level. Unfortunately, vertex lighting can have two undesired properties:

1. Vertex lighting requires extra tessellation to look good; otherwise the coarseness of the underlying geometry will be visible.
2. Vertex lighting causes all applications that use it to have a similar look.

Tessellation becomes critical for vertex lighting to look good, since the triangle rasterizer linearly interpolates the vertices without a deep understanding of local geometry. If the geometry is too coarse, or the geometry contains a lot of variation in a short distance, then the hardware can have a problem producing values that result in good image quality. Increasing tessellation, however, reduces performance. Couple that with the fact that vertex lighting always has a telltale visual signature, and it's rarely compelling. Exceptions are when vertex lighting is used for global ambient, or in addition to per-pixel lighting.

Now, with that understanding of lighting, it is easier to see what all the fuss is about with respect to per-pixel lighting. Of course everyone wants per-pixel lighting, as it really is that much better.

There are two approaches for per-pixel lighting:

1. Pixel lighting in world space.
2. Pixel lighting in tangent space.

Now, in looking at approach two, you may say, "Wait a minute there, Phil, what is this 'tangent space,' and where did that come from?"

Per-pixel lighting uses texture mapping to calculate lighting. That's not new, as light mapping using textures has been utilized in games for years to generate lighting effects. What is new, however, is that in pixel shaders, the values in the texture map can be used to perform a lighting calculation, as opposed to a texture-blend operation.

Now that a lighting calculation is involved, great care must be taken to ensure that the lighting computation is done in the correct 3d basis (also called a coordinate space, or "space" for short). All lighting math must be calculated in the same coordinate space. If a normal is in a different space than the light direction, any math between them is bogus. It would be like multiplying feet by meters; it just doesn't make sense.

With this requirement in mind, any lighting approach needs to manipulate the source values to make sure all components of the calculations are in the same space. In our case here, there are two sets of inputs:

1. Normal and bump maps, stored in texture or "tangent" space.
2. Light directions and environment maps, stored in object or world space.

For normal maps, the texels in a texture map don't represent colors, but vectors. Figure 2 below shows the coordinate space the normals are in. The standard u and v directions along the textures width and height are joined with a "w" direction that is normal to the surface of the texture, to finish the basis (u,v,w). That is, a texel of 1,0,0 actually translates to 1 component of the u vector, 0 of the v vector, and 0 of the w vector. If u, v, and w are perpendicular to each other, this is called an orthonormal basis. This generates a texture space basis, which by convention is called the "tangent space basis."

One thing you may have wondered about u, v, and w, is where in the world are these vectors? At first, it looks like they are part of the texture. In reality, these vectors get pasted onto the real world object that is being textured. At each vertex, u, v, and w might point in an entirely different direction! So, two places on a mesh that have the same piece of texture might have completely different vectors, since u, v, and w are different. Remember, an x, y, z vector in the normal map really means x*u + y*v + z*w. If u, v, w are different, then the vectors are different even if x, y and z are the same. The hard part is finding u, v, and w; after that, everything else is pretty straightforward.

Figure 3 shows the relative coordinate spaces for both objects and texture maps. In either approach to pixel lighting, all source data needs to be moved into one space or the other, either from tangent space into world space, or from world space into tangent space.

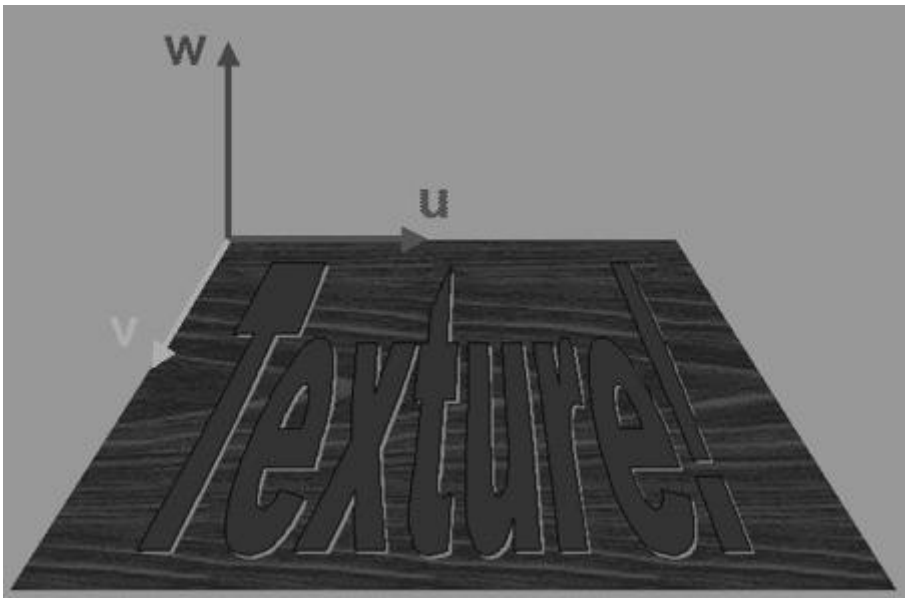

Figure 2. Texture coordinate system

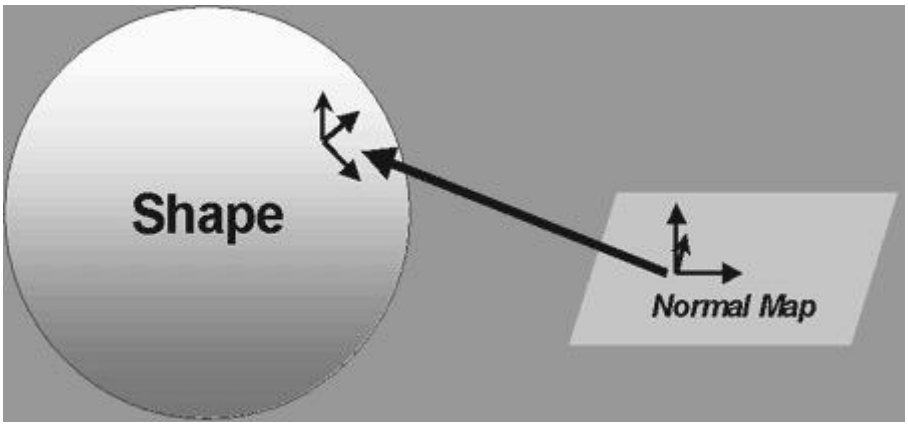

Figure 3. Basis computation diagram

Now that a tangent space (u,v,w) has been defined (more on how to find these vectors later), it's time to look at world-space pixel lighting. In world-space pixel lighting, it's necessary to transform the texture data—for example, the texture map texels into world space. Here the light vectors are not transformed in the vertex shader; they are left in world space, as it's the texture data that needs transformation. Simply pass the tangent basis transform into the pixel shader as three texture coordinates. Note that it's possible to generate the binormal by a cross product to save bandwidth. The iterators will interpolate this matrix to each pixel. Then use the pixel-shader and the tangent space basis to perform per-pixel lighting.

Tangent space pixel lighting is the reverse. Here it's necessary to transform the light and environment map data into tangent space. The light vectors are transformed in the vertex shader, and then passed down to the pixel shader. The environment map data is transformed in the pixel shader, and then the pixel shader performs per-pixel lighting.

Let me discuss one other terminology convention. The normal, the w, and the z axis vectors are all defined to be the same vector; the tangent is the u-axis vector, and the binormal is the v-axis vector. It's possible, and necessary, to pre-compute the tangent vector u as the vector that points along the u-axis of the texture at every point.

Now it's time to examine the tangent space transformation process.

First, recognize this is an inverse transformation. From linear algebra, if one has an orthonormal basis, the inverse matrix is the transpose. So, given (u,v,w) are the basis vectors of the surface, and L is the light, the transform is:

```
   [U.x U.y U.z] * [-L.x]
```

```
[V.x V.y V.z] * [-L.y]

[w.x W.y W.z] * [-L.z]
```

Expanding out, this becomes

```
L.x' = DOT3(U,-L)

L.y' = DOT3(V,-L)

L.z' = DOT3(W,-L)
```

where

U = the tangent along the x-axis of the texture;
V = the normal;
W = the binormal U x V (cross product).

Note that one performs the dot product with the negative of the Light direction.

Computing a tangent space vector is just like computing a normal. Remember that w is defined as the normal, so we should already have that piece of information. Now, we need to generate u and v. To generate the tangent space basis vectors (u and v), use the following equation:

```
Vec1 = Vertex3 – Vertex2

Vec2 = Vertex1 – Vertex2

DeltaU1 = Vertex3.u - Vertex2.u

DeltaU2 = Vertex1.u - Vertex2.u

DirectionV = |DeltaU2*Vec1-DeltaU1*Vec2|

DirectionU = |DirectionV x Vertex.N|

DirectionW = |DirectionU x DirectionV|
```

Where

X indicates taking a cross product;
|| indicates taking a unit vector;
Vertex1-3 are the vertices of the current triangle.

Usually, tangents and normals are calculated during the authoring process, and the binormal is computed in the shader (as a cross product). So, the only field we are adding to our vertex format is an extra u vector. Additionally, if we assume the basis is orthonormal, we don't need to store v either, since it is just u cross w.

A couple of points to keep in mind: Tangents need to be averaged—and be careful about texture wrapping (since it modifies u and v values). Look for D3DX in DirectX 8.1 to include new methods to help with tangent space operations; check the documentation.

For character animation, as long as you skin the normals and the tangents, this technique works fine. Again, generate the binormal in the shader after skinning, so you can skin two vectors instead of three. This will work just fine with indexed palette skinning and 2/4-matrix skinning, as well as with vertex animation (morphing). In terms of performance, tangent space lighting is good, since the transform can be done at a per-vertex level. It's less clocks than vertex lighting, and the per-pixel dp3 dot product is as fast as any pixel operation, so there is no loss of performance there either. To perform diffuse and specular in the same pass, compute the light vector and the half-angle vector, and transform both into tangent space. The perturbation and dot product are then done in the pixel pipeline, either in the pixel shader or by using multi-texture.

Below is a section of a vertex shader that shows how to generate a tangent space light vector:

```
// v3 is normal vector
// v8 is tangent vector
// c0-c3 is World Transform
// c12 is light dir

//tangent space basis generation
m3x3 r3,v8,c0      // transform tan to world space
m3x3 r5,v3,c0      // transform norm to world space

mul r0,r3.zxyw,r5.yzxw   // cross prod to generate binormal
mad r4,r3.yzxw,r5.zxyw,-r0

dp3 r6.x,r3,-c12      // transform the light vector,
dp3 r6.y,r4,-c12      // by resulting matrix
dp3 r6.z,r5,-c12      // r6 is light dir in tan space
```

This can simply be repeated for any vector that needs to be transformed to tangent space.

One other thing to be aware of is the range of values required by dot product lighting. Dot product operations contain data represented in the range [-1,1] to perform lighting operations, or signed data. Standard texture formats contain data represented in the range [0,1] to map color values, and thus contain unsigned data. Both pixel shaders and the fixed-function pipeline define modifiers that remap texture data to bridge this gap so texture data can be used effectively in dot product operations. Pixel shaders define the **_bx2** argument modifier, that remaps input data from unsigned to signed. So the input arguments to the dot product operation usually have this modifier applied to them. It's also useful to clamp the results of the dot product to black using the **_sat** instruction modifier. Here is a typical dot product pixel shader instruction:

```
dp3_sat r0, t0_bx2, v0_bx2    // t0 is normal map, v0 is light dir
```

For the fixed-function pipeline, the similar process is done with the texture argument modifier flag **D3DTA_COMPLEMENT** for texture inputs, and the texture operators **D3DTOP_ADDSIGNED** for results in the range [-0.5,0.5], and **D3DTOP_ADDSIGNED2X** for results in the range [-1.0,1.0] range.

With this understanding of the basics of per-pixel lighting, it's time to examine the standard lighting models, and how diffuse and specular lighting work in pixel shaders.

Standard Lighting Models

The standard lighting models include diffuse and specular lighting. Each lighting model can be done with both pixel shaders and fixed-function multi-texture fallback techniques. Understanding these techniques and the fallbacks allows development of a shader strategy that can cope with the differing generations of graphics cards. DirectX 6.0 generation cards are multi-texture capable—almost all can do subtractive blending, and some can do dot product blending. Examples include TNT2, Rage128, Voodoo 5, and G400. DirectX 7.0 generation cards are both hardware transform and multi-texture capable, and almost all can do both subtractive and dot product blending. Examples include geForce2 and Radeon. All DirectX 8.0 cards can do vertex and pixel shaders in hardware. Examples include geForce3 and Radeon8500.

Per-pixel diffuse is consistent with standard lighting models with no specular. It's nice, because there is no need to modulate against another lighting term; each pixel is correctly lit after the per-pixel diffuse calculation. **Note that filtering can be a major problem, since normals cannot be filtered for a variety of reasons.** Below is a vertex shader fragment to perform setup for per-pixel diffuse lighting, including calculating the light direction in tangent space, biasing it for the per-pixel dot product, and setting up the texture coordinates for the pixel shader.

```
//v0 = position

//v3 = normal (also the w vector)
```

```
//v7 = texture coordinate
//v8 = tangent (u vector)

vs.1.1
//tranform position
m4x4 oPos,v0,c4

//tsb generation
m3x3 r3,v8,c0          //gen normal
m3x3 r5,v3,c0          //gen tangent

//gen binormal via Cross product
mul r0,-r3.zxyw,r5.yzxw;
mad r4,-r3.yzxw,r5.zxyw,-r0;

//diffuse, transform the light vector
dp3 r6.x,r3,-c16
dp3 r6.y,r4,-c16
dp3 r6.z,r5,-c16

//light in oD0
mad oD0.xyz,r6.xyz,c20,c20   //multiply by a half then add half

//tex coords
mov oT0.xy, v7.xy
mov oT1.xy, v7.xy
```

Next, a typical diffuse pixel shader is shown below:

```
ps.1.1
```

```
tex t0                  //sample texture

tex t1                  //sample normal


//diffuse

dp3_sat r1,t1_bx2,v0_bx2          //dot(normal,light)


//assemble final color

mul r0,t0,    r1             //modulate against base color
```

This is prototypical usage of dp3 to calculate N dot L. Conceptually, this is a good way to lay out the pixel shader. Figure 4 contains a screenshot of this shader in action. Notice the separation of the calculation and final color assembly. The following renderstates are used, shown in effects file format syntax:

```
VertexShaderConstant[0]  = World Matrix (transpose)

VertexShaderConstant[8]  = Total Matrix (transpose)

VertexShaderConstant[12] = Light Direction;

VertexShaderConstant[20] = (.5f,.5f,.5f,.5f)


Texture[0]          = normal map;

Texture[1]          = color map;
```

Note how simple the pixel shader is. Figure 4 shows an example diffuse per-pixel lighting image. The take-away from this is that with per-pixel diffuse lighting, it is easy to get good-looking results. All pixel shader cards support the **dp3** operator, so this technique is good to go on pixel shader hardware.

For previous generation cards, two primary fallbacks exist. The first fallback is to use the **D3DTOP_DOTPRODUCT3** fixed-function operator, which some of the better previous generation cards support, since this operator was first enabled in DirectX 6.0. Be sure to check the **D3DTEXOPCAPS_DOTPRODUCT3** capability bits for support of this multi-texture capability. Using **ValidateDevice()** is also a good idea. Below is the multi-texture setup (using the effects framework syntax) for a **D3DTOP_DOTPRODUCT3** fixed-function operation

```
ColorOp[0]   = DotProduct3;

ColorArg1[0] = Texture;
```

```
ColorArg2[0] = Diffuse;


ColorOp[1]   = Modulate;
ColorArg1[1] = Texture;
ColorArg2[1] = Current;


VertexShaderConstant[0]  = World Matrix (transpose)
VertexShaderConstant[8]  = Total Matrix (transpose)
VertexShaderConstant[12] = Light Direction;


Texture[0]   = normal map
Texture[1]   = color map;
```

Where colorop indicates the texture stage operation, and colorarg[n] indicates the texture stage arguments. MIP mapping and filtering need to be set as well, but I ignore these settings due to space considerations. Remember, the **D3DTOP_DOTPRODUCT3** operator in the fixed-function pipeline automatically applies the **_sat** and **_bx2** operations automatically, which means:

You must use biased art for the normal maps for **_bx2** to generate correct results.

The automatic use of **_sat** (clamping) means no signed result can be generated.

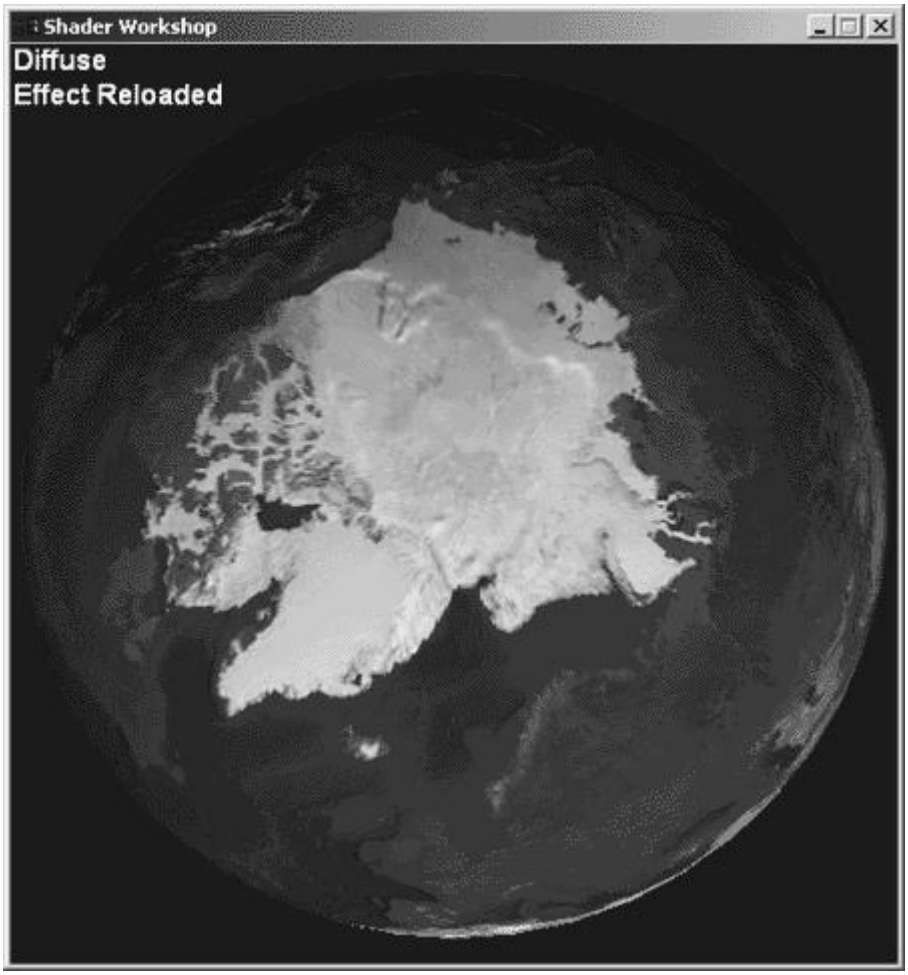

Figure 4. Diffuse per-pixel lighting

The second fallback is to use emboss bump mapping. The only hardware requirement is for a dual texture unit with a subtract operation, as shown by the presence of the **D3DTEXOPCAPS_SUBTRACT** capability bit. Again, whenever using the fixed-function multi-texture pipeline, it's a good idea to use **ValidateDevice()**. Emboss bump-mapping works by shifting a height map in the direction of the light vector, and subtracting this from the base map. The results can be very convincing, but can take quite an effort to fine tune. A vertex shader fragment for a typical emboss operation is shown below:

```
//v0 = position

//v3 = normal (also the w vector)

//v7 = texture coordinate

//v8 = tangent (u vector)
```

```
vs.1.1          // for emboss
m4x4 oPos, v0,c08      // generate output position

//diffuse
m3x3 r3, v8, c0       // transform tan to world space
m3x3 r5, v3, c0      // transform norm to world space

mul r0,r3.zxyw,r5.yzxw  // cross prod to generate binormal
mad r4,r3.yzxw,r5.zxyw,-r0

dp3 r6.x,r3,c12      // tangent space light in r6
dp3 r6.y,r4,c12
//   dp3 r6.z,r5,c12  don't need this
//                    -only x and y shifts matter

// set up the texture, based on light direction:
mul r1.xy,  r6.xy, -c24.xy
mov oT0.xy, v7.xy          // copy the base height map
add oT1.xy, v7.xy,  r1.xy    // offset the normal height map

// simple dot product to get global darkening effects:
dp3 oD0.xyz,v3.xyz,c12.xyz
```

Next is the multi-texture state setup (again using the effects framework syntax) for a **D3DTOP_ADDSIGNED** fixed-function operation, using the complement input argument modifier flag:

```
ColorOp[0]   = SelectArg1;
ColorArg1[0] = Texture;

ColorOp[1]   = AddSigned;
```

```
ColorArg1[1] = Texture | Complement;

ColorArg2[1] = Current;


VertexShaderConstant[0]  = World Matrix (transpose)

VertexShaderConstant[8]  = Total Matrix (transpose)

VertexShaderConstant[12] = Light Direction;

VertexShaderConstant[24] = Offset Constant


Texture[0]   = base height map;

Texture[1]   = normal height map;
```

Again, MIP mapping and filtering need to be set as well, but I ignore these settings due to space considerations. In conclusion, emboss-style bump mapping can be used for a diffuse fallback technique for hardware that does not support the dot product multi-texture operator. This includes most DirectX 6.x generation cards—a huge percentage of the installed base. For ideal results, this technique requires modified artwork, and textures should be brightened on load. An alternative is to use a **D3DTOP_MODULATE2X** operator to scale the result up, which has the visual effect of brightening. Also note that filtering can be applied to this technique more easily than to normal maps, so this technique may result in a better appearance than the dot product multi-texture technique, even on hardware that supports dot product operations.

Per-pixel specular is similar to diffuse, but requires a pixel shader. Instead of the light direction, an interpolated half-angle vector H is used; which is computed in the vertex shader. In the pixel shader, the H is dotted with the pixel normal, and then raised to a pre-determined power. The specular result is added to the other passes. Also, remember that there is only one output value of a pixel shader, in **r0**, so make sure to add the specular result into **r0**.

One question you may be asking at this point: How is exponentiation performed? Two techniques are used, multiply-based and table-based. One is simpler and is acceptable for small exponents, and one is more work, but looks nicer for higher exponents. Both techniques I cover here use the following renderstates (again using effects framework syntax):

```
VertexShaderConstant[0] = World Matrix

VertexShaderConstant[8] = Total Matrix
```

```
VertexShaderConstant[12] = Light Direction

VertexShaderConstant[14] = Camera Position (World)

VertexShaderConstant[33] = (.5f,.5f,.5f,.5f)


Texture[0]   = normal map

Texture[1]   = color map
```

Now, it's time to examine the two pixel shader specular techniques, starting with the multiply-based exponentiation technique. Below is a vertex shader fragment that shows (in addition to the diffuse actions of calculating the light direction in tangent space, biasing it for the per-pixel dot product, and setting up the texture coordinates for the pixel shader) the actions of computing the half vector, using the view direction and the light direction, and scaling/biasing it for the dot product calculations used by multiply-based exponentiation:

```
vs.1.1

//tranform position

m4x4 oPos,v0,c4


//tsb generation

m3x3 r3,v8,c0            //gen normal

m3x3 r5,v3,c0            //gen tangent


//gen binormal via Cross product

mul r0,-r3.zxyw,r5.yzxw;

mad r4,-r3.yzxw,r5.zxyw,-r0;


//specular

m4x4 r2,v0,c0            //transform position

//get a vector toward the camera

add r2,-r2,c24
```

```
dp3 r11.x,  r2.xyz,r2.xyz   //load the square into r11
rsq r11.xyz,r11.x         //get the inverse of the square
mul r2.xyz, r2.xyz,r11.xyz   //multiply, r0 = -(camera vector)
add r2.xyz,r2.xyz,-c16      //get half angle

//normalize
dp3 r11.x,r2.xyz,r2.xyz      //load the square into r1
rsq r11.xyz,r11.x         //get the inverse of the square
mul r2.xyz,r2.xyz,r11.xyz   //multiply, r2 = HalfAngle

//transform the half angle vector
dp3 r8.x,r3,r2
dp3 r8.y,r4,r2
dp3 r8.z,r5,r2

//half-angle in oD1
mad oD1.xyz,   r8.xyz,c20,c20   //mutiply by a half, add half

//tex coords
mov oT0.xy,   v7.xy
mov oT1.xy,   v7.xy
```

Below is a pixel shader fragment that shows per-pixel specular, using the multiply-based exponentiation technique:

```
ps.1.1            // pow2 by multiplies
tex t0            // color map
tex t1            // normal map

//  specular lighting dotproduct
```

```
dp3_sat r0,t1_bx2,v1_bx2   // bias t0 and v1 (light color)


mul r1,r0,r0            //  2nd power
mul r0,r1,r1            //  4th power
mul r1,r0,r0            //  8th power
mul r0,r1,r1            // 16th power!


//assemble final color
mul r0,r0,t0         // modulate by color map
```

Note the use of the **_bx2** modifier. Again, this is to enable the input data to be processed as a signed quantity, while reserving dynamic range (used by the specular calculation) before overflow clamping that can occur on implementations limited to the range [-1, 1]. Figure 5 shows an image generated using multiply-based specular exponentiation. Notice the banding in the highlight. This is due to loss of precision in the calculations, since each result channel is only 8-bits. Here is where higher precision texture formats and higher precision for internal calculations will increase image quality. In conclusion, multiply-based per-pixel specular is easy to implement, but can involve precision problems, so don't try to use the technique for powers greater than 16. On graphics chips with higher precision, this may not be an issue.

The next technique is table-lookup based specular exponentiation. The example used here performs this operation with a 3x2 table. The texture is used as a table of exponents, storing the function y = pow (x).

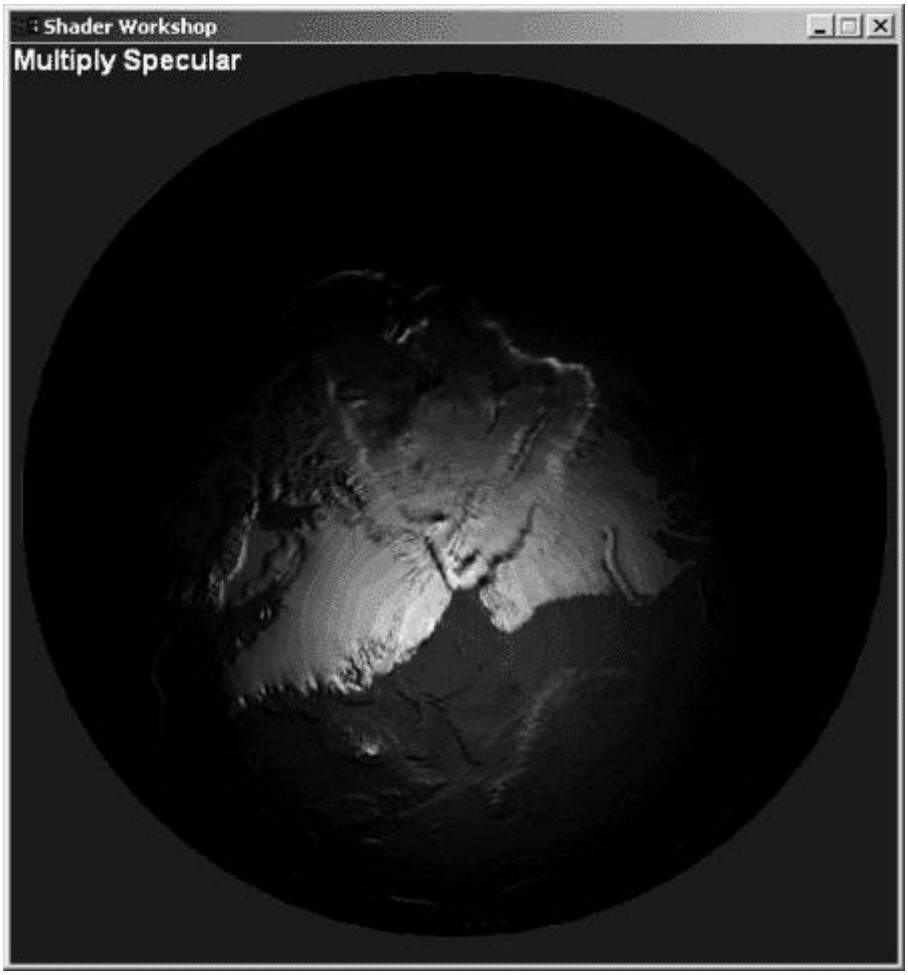

Figure 5. Multiply-based specular exponentiation

This technique also uses the dependent texture read capability of the `texm3x2tex` instruction. Note the 3x2 multiply is also 2 dot products, so this technique can do specular and diffuse, or two light sources, simultaneously. Below is the vertex shader for this technique:

```
vs.1.1
//tranform position
m4x4 oPos,v0,c4

//tsb generation
m3x3 r3,v8,c0             //transform normal
m3x3 r5,v3,c0             //and tangent
```

```
//Cross product
mul r0,-r3.zxyw,r5.yzxw;
mad r4,-r3.yzxw,r5.zxyw,-r0;

//specular
m4x4 r2,v0,c0            //transform position

//get a vector toward the camera
add r2,-r2,c24

dp3 r11.x,r2.xyz,r2.xyz   //load the square into r11
rsq r11.xyz,r11.x         //get the inverse of the square
mul r2.xyz,r2.xyz,r11.xyz //multiply, r0 = -(camera vector)

add r2.xyz,r2.xyz,-c16    //get half angle

//normalize
dp3 r11.x,r2.xyz,r2.xyz   //load the square into r1
rsq r11.xyz,r11.x         //get the inverse of the square
mul r2.xyz,r2.xyz,r11.xyz //multiply, r2 = HalfAngle

//transform the half angle vector
dp3 r8.x,r3,r2
dp3 r8.y,r4,r2
dp3 r8.z,r5,r2

//tex coords
mov oT0.xy, v7.xy         //coord to samp normal from
```

```
mov oT1.xyz,r8            //Not a tex coord, but half

mov oT2.xyz,r8            //angle

mov oT3.xy, v7.xy
```

The table-lookup vertex shader is identical to the multiply-based vertex shader through the half-angle normalization calculation. From there, this technique uses texture coordinates to pass down vectors, as well as true texture coordinates used to index the color and normal maps. The half-angle is passed down as texture coordinates for stage 2, then texture coordinates for stage 1 are used to pass down the light direction, and texture coordinates for stage 0 and stage 3 are used for the normal and color maps respectively.

Next is shown the 3x2 table-lookup specular lighting pixel shader:

```
ps.1.1          // exponentiation by table lookup


//  texcoord t1      // the diffuse light direction

//  texcoord t2      // half-angle vector

//  texture at stage t2 is a table lookup function


tex t0          // sample the normal map

texm3x2pad t1, t0_bx2     // 1st row of mult, 1st dotproduct=u

texm3x2tex t2, t0_bx2     // 2nd row of mult, 2nd dotproduct=v


//asemble final color

mov r0,t2         // use (u,v) above to get intensity

mul r0,r0,t3            //blend terms
```

The key detail of this shader is the use of the **texm3x2tex** instruction with the **texm3x2pad** instruction to perform a dependent read. The **texm3x2pad** instruction is used in conjunction with other texture address operators to perform 3x2 matrix multiplies. It is used to represent the stage where only the texture coordinate is used, so there is no texture bound at this stage (in this shader, that is, **t1**). The input argument, **t0**, should still be specified.

The **texm3x2pad** instruction takes the specified input color (**t0** here) and multiplies that by the subsequent stages' (**t1** here) texture coordinates (u, v, and w) to calculate the 1st row of the multiply (a dot product) to generate a u coordinate. Then the **texm3x2tex** instruction takes the specified input color (**t0** again) and the texture coordinates of the stage specified (**t2** here) to calculate the second row of the multiply (again a dot product) to generate the v coordinate. Lastly, this stage's texture (**t2** here) can be used to sample the texture by a dependent read at (u, v) to produce the final color.

That leaves the question of how to generate the lookup-table texture. Using D3DX, one could use the following code fragment to generate the table-lookup texture:

```
void LightEval(D3DXVECTOR4 *col, D3DXVECTOR2 *input
                          D3DXVECTOR2 *sampSize, void *pfPower)
{
    float fPower = (float) pow(input->y,*((float*)pfPower));
    col->x = fPower;
    col->y = fPower;
    col->z = fPower;
    col->w = input->x;
}

D3DXCreateTexture(m_pd3dDevice, 256,256, 0,0,
   D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, &pLightMap100);

float fPower = 100;
D3DXFillTexture(m_pLightMap100,LightEval,&fPower);
```

Figure 6 below shows the results of table-lookup specular exponentiation. A careful examination does indeed show less banding effects and a better-looking image. This technique will support exponents greater than 100, which is important for some visual effects. There is higher precision in the texture read unit, and the table-lookup texture ends up nicely filtered, so that the banding is reduced to a tolerable level. Note that other functions can be used besides exponents.

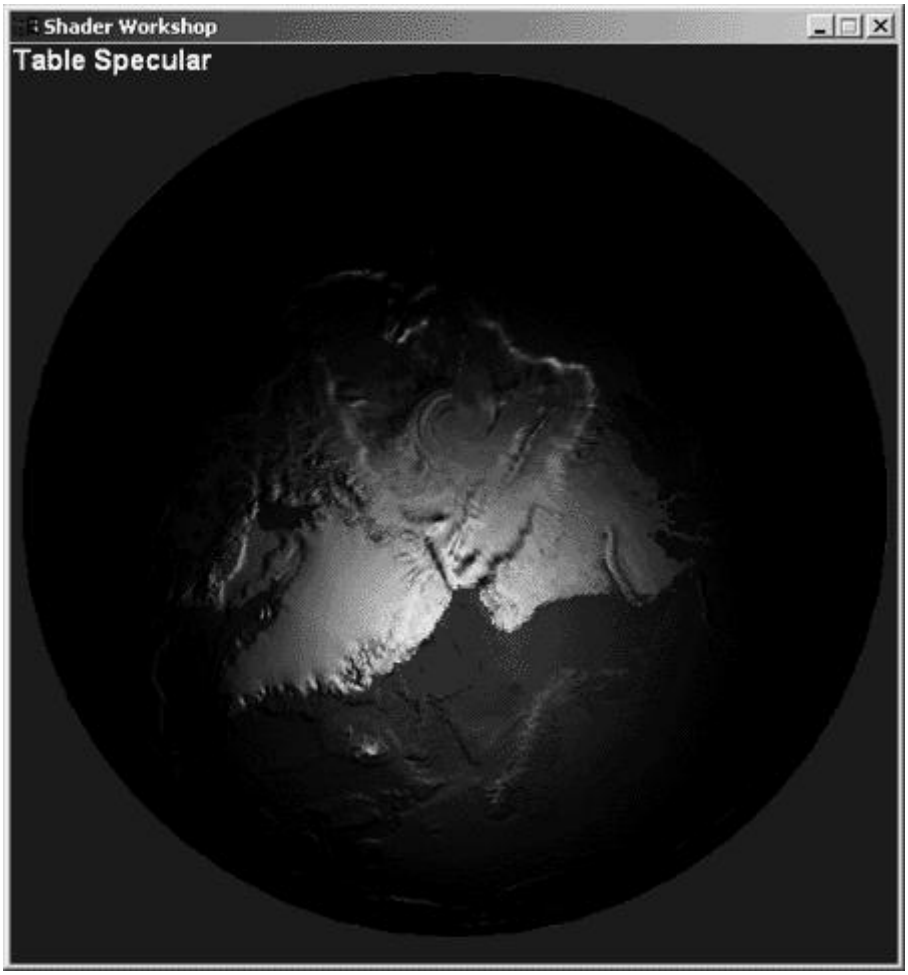

Figure 6. Table lookup specular exponentiation

The fallback using the fixed-function multi-texture pipeline is an analog of emboss for the specular term, where the light map highlight is subtracted from the height-field normal map. Specular highlights do need to be manually positioned, but that's not hard to do in the vertex shader. Then composite the values in the pixel shader using subtract, and add the result as the per-pixel specular term. Figure 7 shows a diagram of the two textures and how the subtract result gives the desired effect.

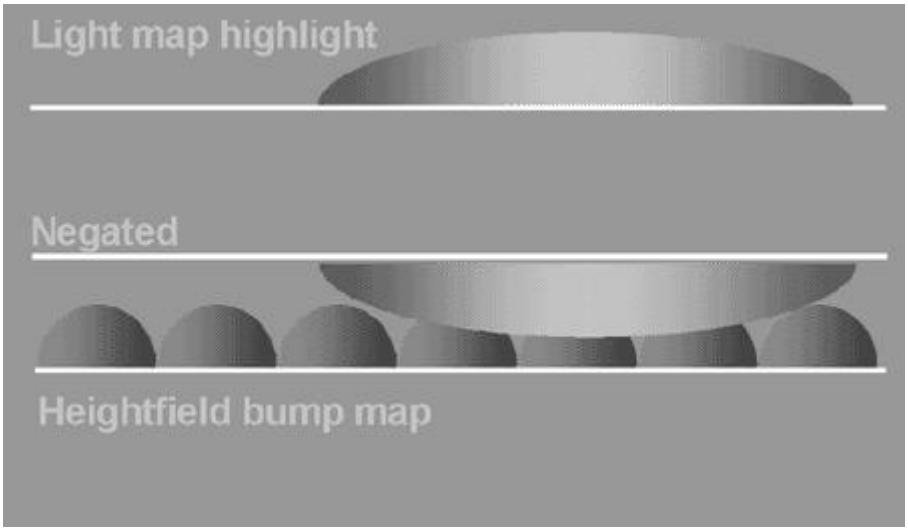

Figure 7. Specular emboss fallback diagram

In the interests of space, I am skipping the implementation of the specular fallback; you should be getting the idea. That's the end of the coverage of the standard diffuse and specular per-pixel lighting models, and how to realize them in pixel shaders. Now it's on to "do-it-yourself" lighting models, where I show how to develop your own, custom lighting models and implement them.

Custom Per-Pixel Lighting

With this summary of techniques for the legacy—that is, with standard lighting models behind us—it's now time to consider custom, "do-it-yourself" lighting models, or DIY lighting. The beauty of pixel shader hardware is that it frees the developer from being stuck with the legacy lighting model, and opens up a brave new world of custom lighting. Remember, the legacy lighting models are just a set of equations someone came up with at some point in time that did a reasonable job of modeling the physics of lighting. There is no magic there. So do not be restricted by the basic lighting model. While diffuse and specular are easy to calculate, and generally produce acceptable results, they do have the drawback of producing a result that looks overly familiar. When the goal of advanced lighting is to stand out, looking the same is a major drawback.

So what does DIY lighting mean? It means not being bound by the basic lighting model, and instead using creativity and an understanding of the principles of the basic lighting tasks. Using the SIGGRAPH literature, hardware vendor Web sites, and information about game engine rendering approaches, there are huge amounts of material available as a guide to interesting alternative approaches to the basic diffuse and specular lighting model.

The key is to understand the process of the basic lighting model, and to use a process in developing a lighting model. This allows a consistent basis upon which to evaluate lighting ideas. This discussion focuses on local lighting, since there isn't space to cover attenuation effects, atmospheric effects, or anything else.

The process of the basic lighting model is based on the physics of lighting. Figure 8 shows the physical model the calculations are intended to reproduce. There is a light source generating light, a surface geometry upon which the incident light energy is received, and upon which reflected light departs (called the incident and reflectant geometry), and a camera to record the lighting.

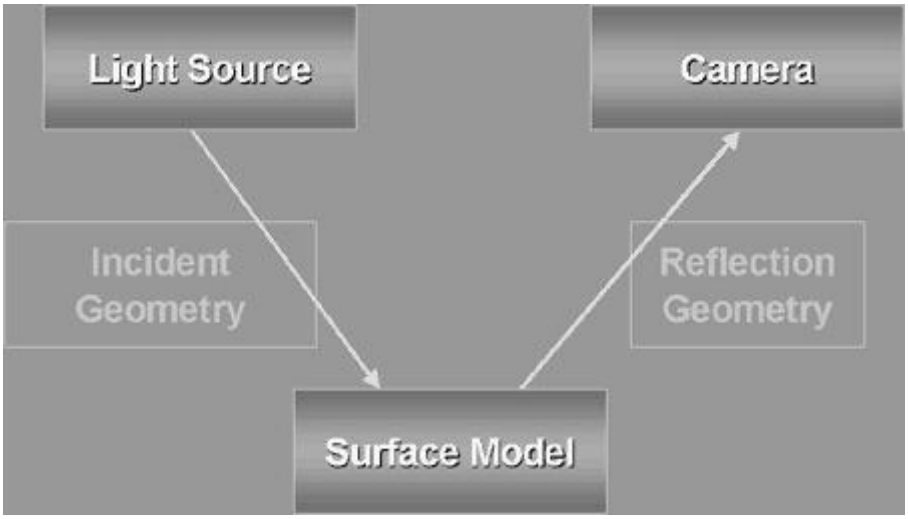

Figure 8. Lighting Process

Incident angle and reflection angle play a key part in the lighting calculation. The geometries' interaction with light is also controlled by a surface model that defines the physical properties of the surface that lighting is incident on and reflected from. There are many different elements that can be used in surface models. The most common model is the Lambert model for the diffuse light term. The Lambert model defines microfacets across the surface where intensity depends on input energy and the area of the microfacets perpendicular to the light direction. Lambert diffuse is easily calculated using the dot product. Figure 9 illustrates the Lambert model. The key point is that there is a physical basis for this model, from which the calculations can be identified.

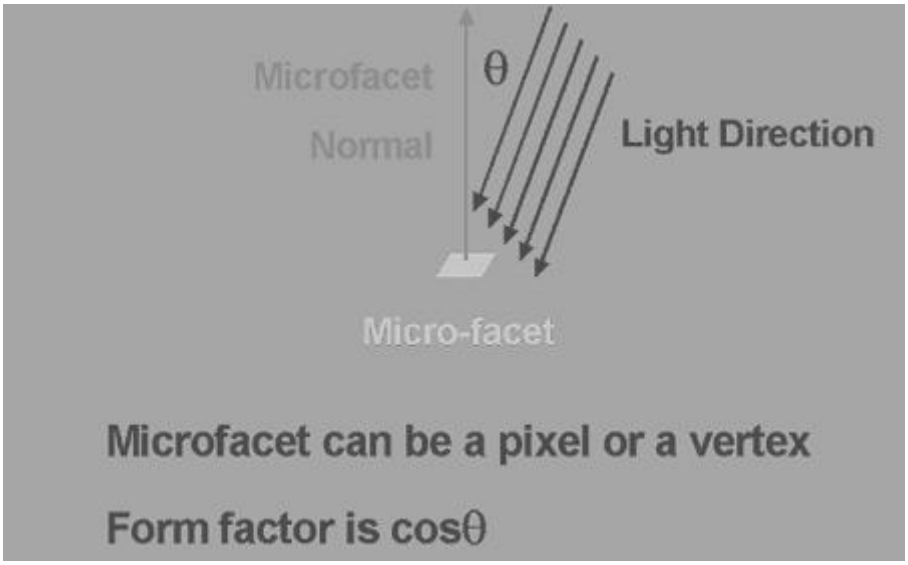

Figure 9. Lambert Model

With an understanding of how this works in the basic lighting model, lets begin an exploration of 'DIY" Lighting. In a simple "DIY" evaluation process, first, there is an identification stage. Here you analyze what the key effects you wish to generate in each scene are. Then these are prioritized. Once this has been accomplished, that data can be used to characterize the lighting environment for each scene and each effect.

How many light sources are needed for these effects? What shape are they? What shadows, and what reflections result? Now with the key lighting effects identified and characterized, algorithms to implement the effect need to be generated. In those algorithms, terms that make important contributions to the final result are kept, and terms that don't contribute significantly to image quality are dropped. Experimentation is usually need to determine this, but in some cases, running limit calculations can lead to an expectation that a terms contribution is meaningful or meaningless. These pieces can often be defined and tested independently, but at some point the individual parts will need to be blended together for a "final" evaluation of the model. Another part of the process consists of determining whether a piece of the calculation is object-based (author time calculation), vertex-based (vertex shader time calculation), or pixel-based (pixel shader time calculation).

Finally, understanding the range of values that serve as input to a model, and understanding the expected range of output values, can lead to finding equivalencies substitutions in making calculations, where the substitute calculation is simpler conceptually, simpler in cost, or simply good enough. This is an important point, to not be hide-bound by convention, and instead keep an open mathematical mind to take advantage of whatever one can.

So, with that in mind, it's time to walk through the process for a simple yet effective 'DIY" lighting model. Here, local lighting is the focus, attenuation effects, atmospheric effects, and others are not considered. The model shown is a variation of an area or distributed lighting effect. It's primarily a diffuse effect, but with a more interesting ambient term. It provides "shadow detail," and shows that an object is grounded in a real-world scene by providing for an influence factor from the world itself. This is in essence an image-based lighting technique.

Figure 10 is a Lightwave image showing a target scene for this model, where how close to the target the model comes provides an evaluation of the model. Many ray-trace tools support a similar model, where the renderer integrates at each point, casts rays from the microfacet pixel to all points of the hemisphere, and accumulates color from ray intersections from other parts of the same object and the environment map image. This can take hours. The goal here is to get a significant percent of the same quality in real-time.

Figure 10. Target image for evaluating DIY model

The distributed lighting model used here, shown in Figure 11, works on large area light sources; there is no single direction vector. Energy is based on a number of directions, the fraction of possible directions that can illuminate the microfacet. When using this model for outdoor scenes, there are two light sources, the sun and the sky. The sun is a directional light source, throwing sharp shadows. The sky is an omni-directional light source, throwing soft shadows. It's useful to consider the area light source as an enclosing

hemisphere, and then the lighting of objects inside the scene reduces to considering what possible percent of the hemisphere can shine on the object.

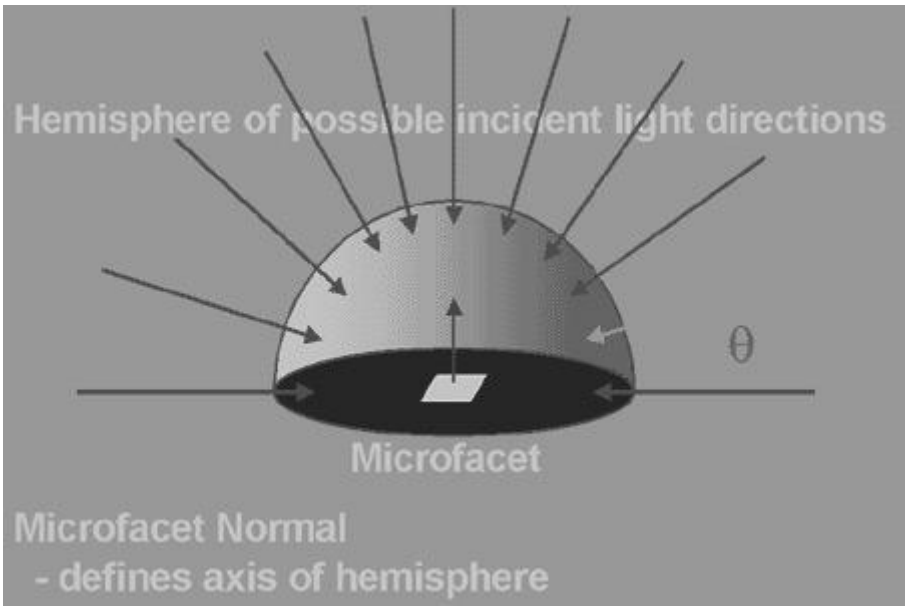

Figure 11. Distributed lighting model

Figures 12, 13, and 14 illustrate this for a cube, a hemisphere, and a sphere. It's pretty easy to see how objects in scenes that use this model are lit. Figure 12 contains a cube, and the lighting intensity is highest on the top face, and gradually decreases down the side faces. Figure 13 contains a hemisphere, and the lighting intensity is greatest at the top polar region, decreasing down to the equator. Figure 14 contains a sphere, and lighting intensity is again greatest at the top polar region, and decreases towards the bottom polar region.

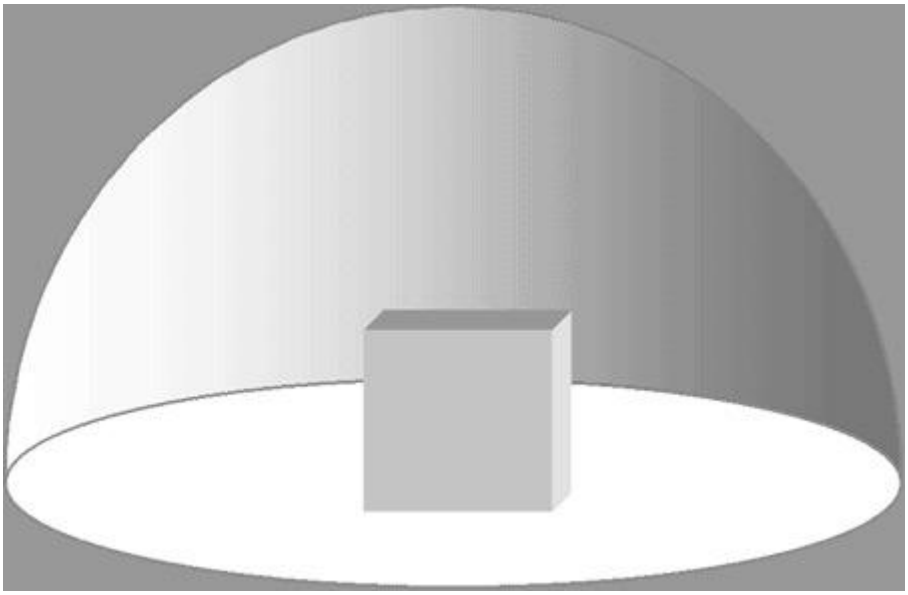

Figure 12. "Hemisphere" lighting for a cube

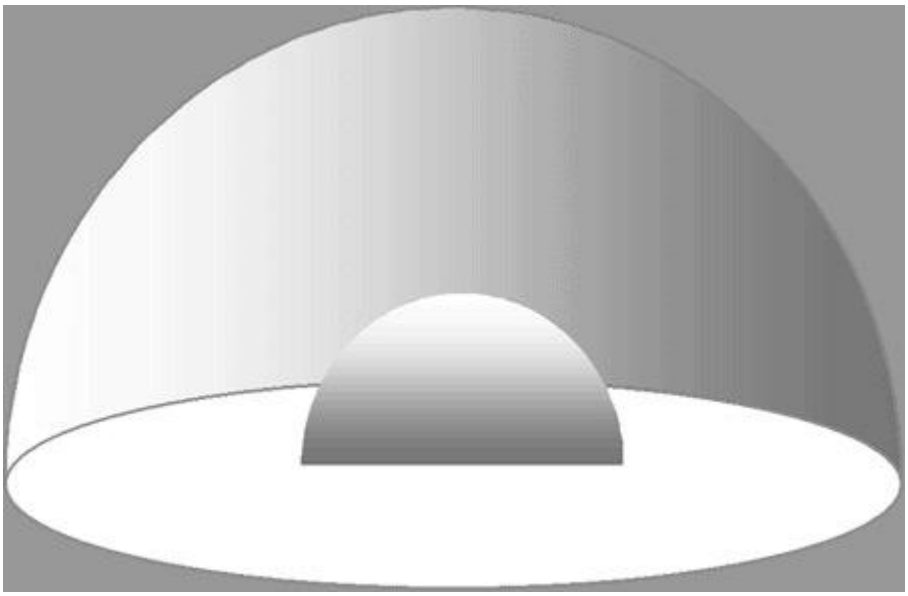

Figure 13. "Hemisphere" lighting for a hemisphere

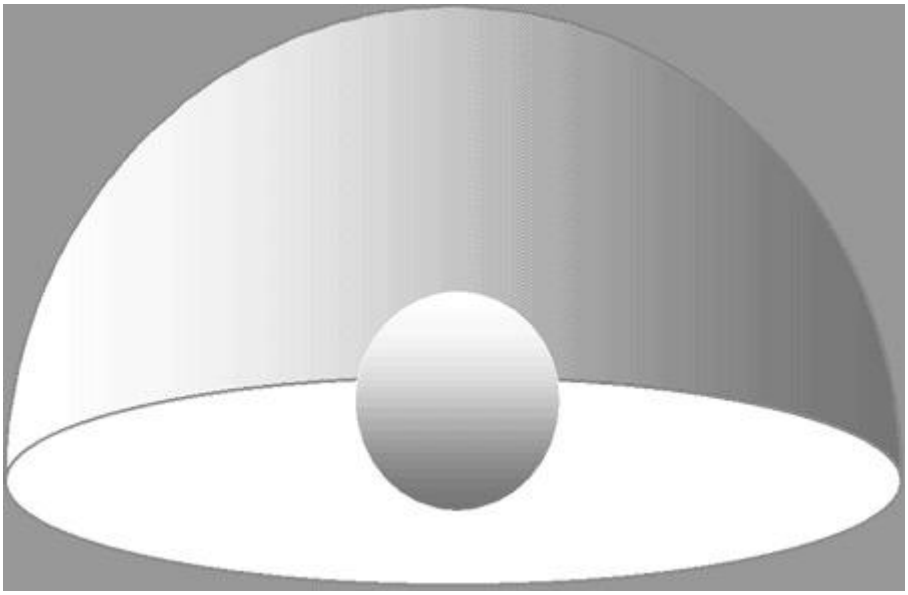

Figure 14. "Hemisphere" lighting for a sphere

Effectively, the light near the plane of the microfacet contributes less energy, so we can use the form factor cos(q) term to scale energy down. Integrating L = 1/p S cos(q) dd for this models' irradiance term, the light source is the far field. Integrating the environment map to get that term is the usual technique. This will work even on DirectX 7-class hardware.

A cube map and its corresponding integral are shown in Figure 15. Notice that the environment map integral is mostly two colors, sky color and ground color. The exact integral calculation is two slow for interactive applications, indicating an authoring time process would be required, and that the integral could not change during the game. That's less than ideal, but can anything be done about that?

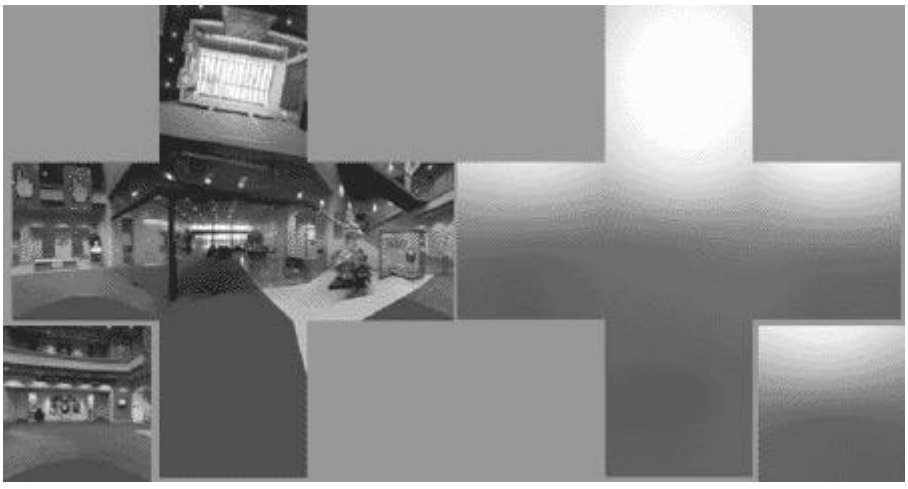

Figure 15. Cubemap and its integral

Let's consider this 2-hemisphere model, where our current understanding of the model is as a calculation with only two important terms—a sky term and a ground term. Figure 16 shows this 2-hemisphere model, and Figure 17 contains a process block diagram.

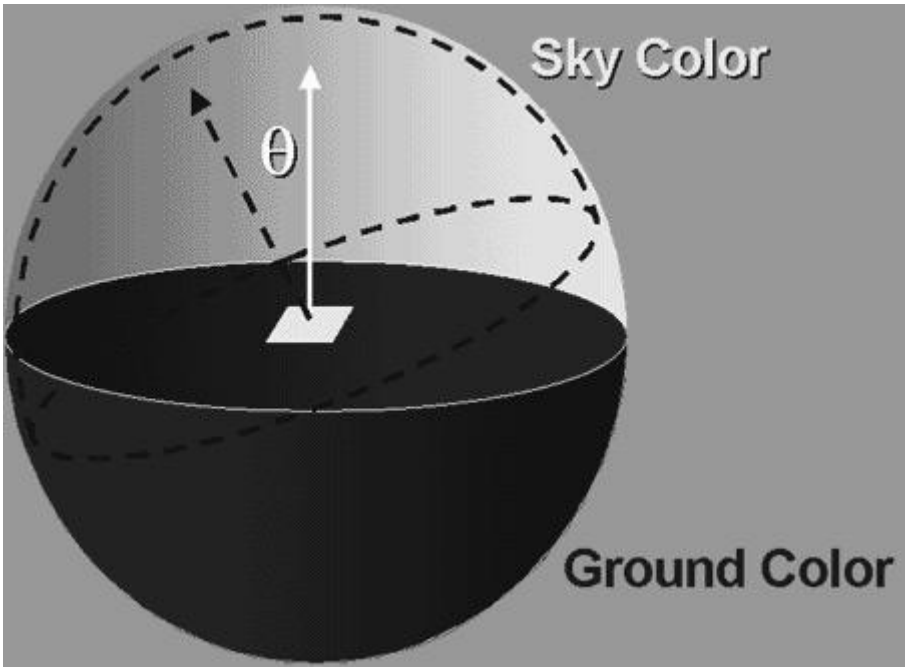

Figure 16. 2-Hemisphere model

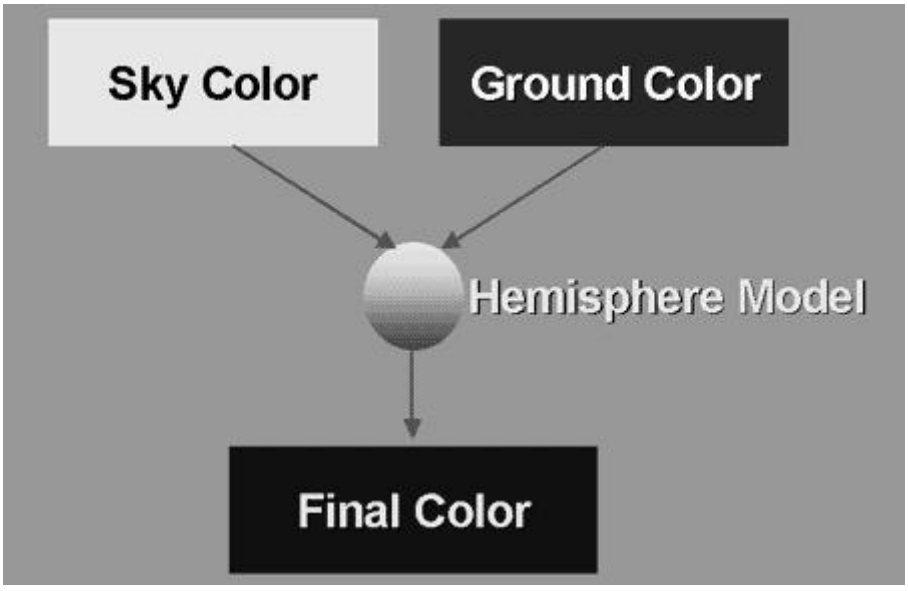

Figure 17. 2-Hemisphere model elements

The actual integral is:

color = a*Skycolor+(1-a)*GroundColor

Where

a = 1-0.5*sin(q) for q<90
a = 0.5*sin(q) for q > 90

Or, if instead of that, the simpler form

a = 0.5+0.5*cos(q)

is used. The resulting simplified integral versus the actual integral is shown in Figure 18. Notice the shape of the curve is the same, but mirrored, and similar amounts of light and dark regions appear below both curves. There is a general equivalency, even if it's not exact.

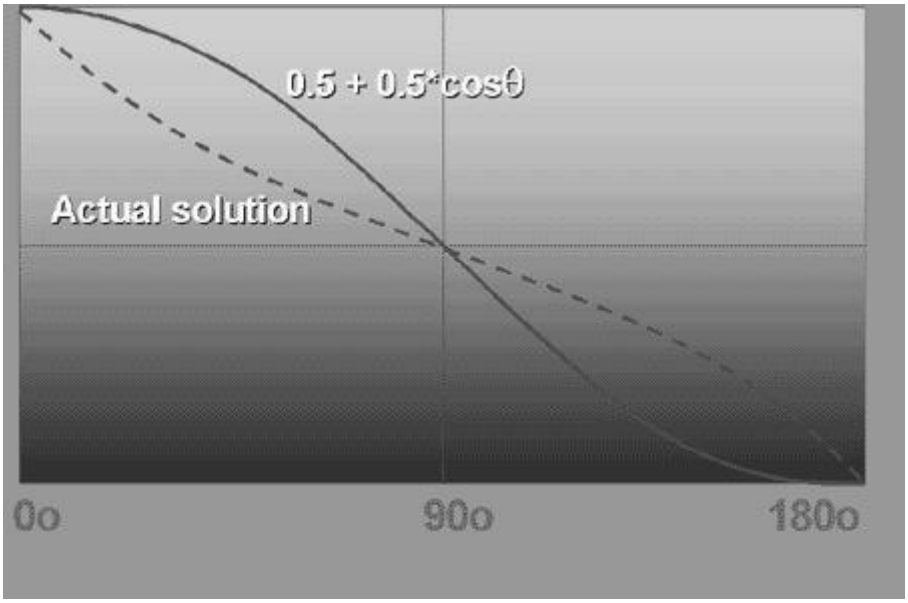

Figure 18. Integral comparison

Herein lies one of the secrets of the shader game: It's okay to substitute similar calculations if they are generally equivalent, because at real-time frame rates the small delta in image quality is usually unnoticeable. If this simplification provides enough of a performance gain, it can be preferred. In this case, the simplification takes 2 clocks, since it uses a **`dp3`** and **`mad`**. While it's not visually identical (this solution provides more bump detail along the equator and less bump detail facing the light) it's good enough to produce the desired effect at a significant gain in performance because, in this case, the actual calculation was too slow to do in real-time versus this simplification with its 2-clock cost. That's a huge win both in terms of clocks and in terms of grasping the essence and beauty of DIY lighting and the shader calculation gestalt.

The 2-term calculation boils down to what percent of incident energy has what color. The far field hemisphere is composed of 2 colors, sky, and ground, in a simple proportion. Even with the environment simplified to two colors like this, the model still allows for dynamic updates, like when a car enters a canyon or a tunnel, and then leaves it; so the pavement or ground color would change, or the sky or roof color would change. The hemisphere implementation can also be done either per-vertex or per-pixel.

The per-vertex implementation can pass the hemisphere axis in as a light direction and use the standard tangent space basis vertex shader that transforms the hemi axis into tangent space. A vertex shader implementation would look something like:

```
vs.1.1          // vertex hemisphere shader


m4x4 oPos,v0,c8   // transform position
m3x3 r0, v3,c0   // transform normal


dp3 r0,r0,c40   // c40 is sky vector
mov r1,c33      // c33 is .5f in all channels
mad r0,r0,c33,r1   // bias operation


mov r1,c42      // c42 is ground color
sub r1,c41,r1      // c41 is sky color
mad r0,r1,r0,c42   // lerp operation


//c44 = (1,1,1,1)
sub r1,c44,v7.zzz   // v7.zzz = occlusion term
mul r0,r0,r1
mul oD0,r0,c43
```

A per-pixel fragment implementation would look like:

```
//   v0.rgb is hemi axis in tangent space
// v0.a is occlusion ratio from vshader
```

```
tex t0          // normal map
tex t1          // base texture


dp3_d2_sat r0,v0_bx2,t0_bx2   // dot normal with hemi axis
add r0,r0,c5      // map into range, not _sat
lrp r0,r0,c1,c2
mul r0,r0,t1      // modulate base texture
```

With that in mind, how does this look? Figure 19 shows the 2-term approach. While this is interesting, there are issues here. The combination of two colors is getting there, but there is obviously too much light in certain areas, like the eye sockets, the nostrils, and behind the teeth.

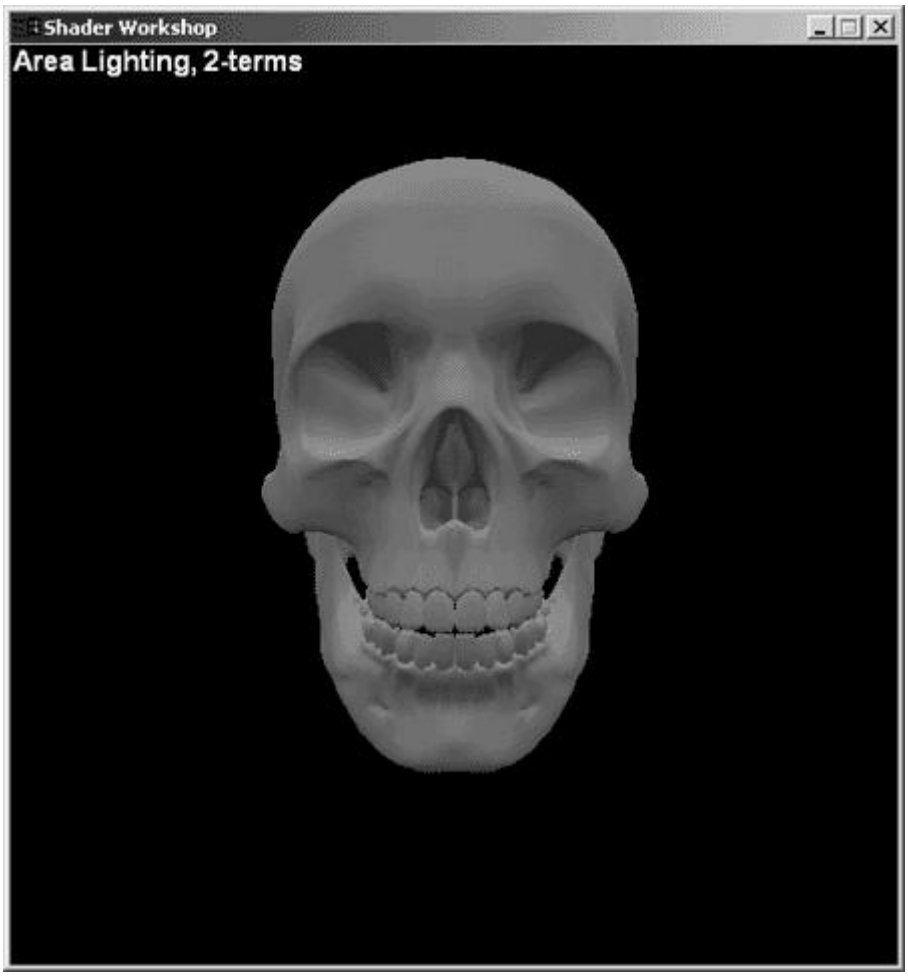

Figure 19. 2-term DIY image

Time to refine the DIY model. How is the model refined? With the addition of another term, of course. What term would that be? Well, the first attempt did not take object self-shadowing into account, and that is the basis of the resulting image being brighter in areas where it shouldn't be. So adding an occlusion term is necessary.

Figure 20 shows a block diagram of this updated DIY lighting model. This calculation can be done vertex-to-vertex by firing a hemisphere of rays from each normal, storing the result as vertex colors; or pixel-to-pixel by firing rays in a height field and storing the result in the alpha channel of a normal map; or both by firing rays from vertices and pixels and storing the result in a texture map.

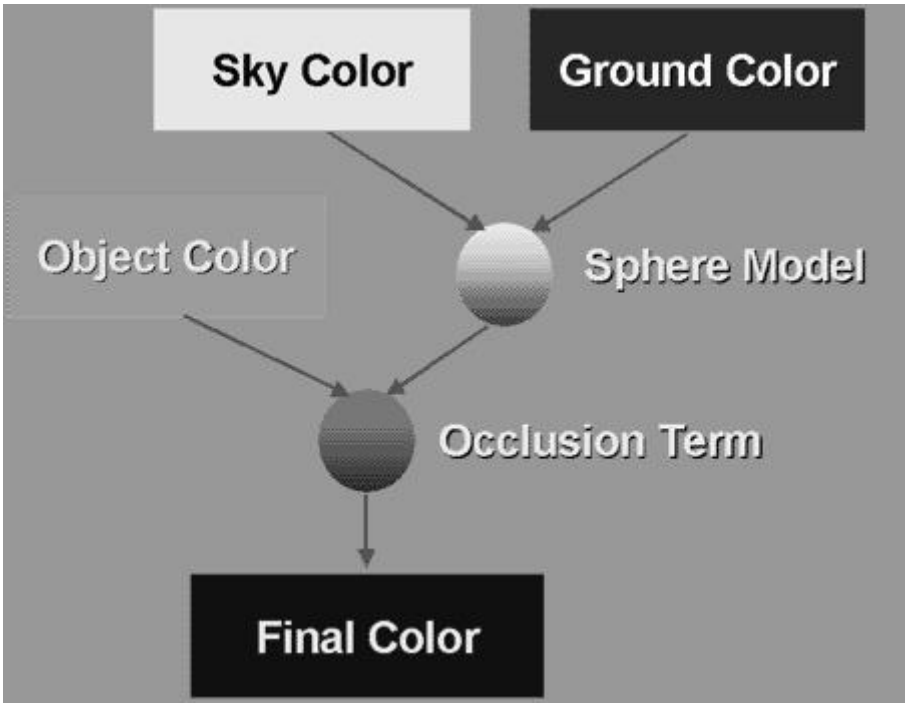

Figure 20. Updated DIY model elements

The sample shown here used an offline rendering process to calculate this occlusion data. Considering the vertex-to-vertex case, the calculation answers the question, "How much do adjacent polygons shadow each other?" The result can be stored in a vertex attribute, and should handle object level effects. Note that looking only at neighbor vertices might be okay.

Considering the pixel-to-pixel case, the calculation similarly answers the question, "How much do adjacent pixels shadow each other?" An example is a bump-mapped earth, where the geometry provides no self-occlusion, since a sphere is everywhere convex. This means all occlusion can be done in a bump map. Figure 21 shows the resulting image with a 3-term DIY lighting model. This is a big improvement, with the problem areas of the eye sockets, nostrils, and interior of the mouth looking much better.

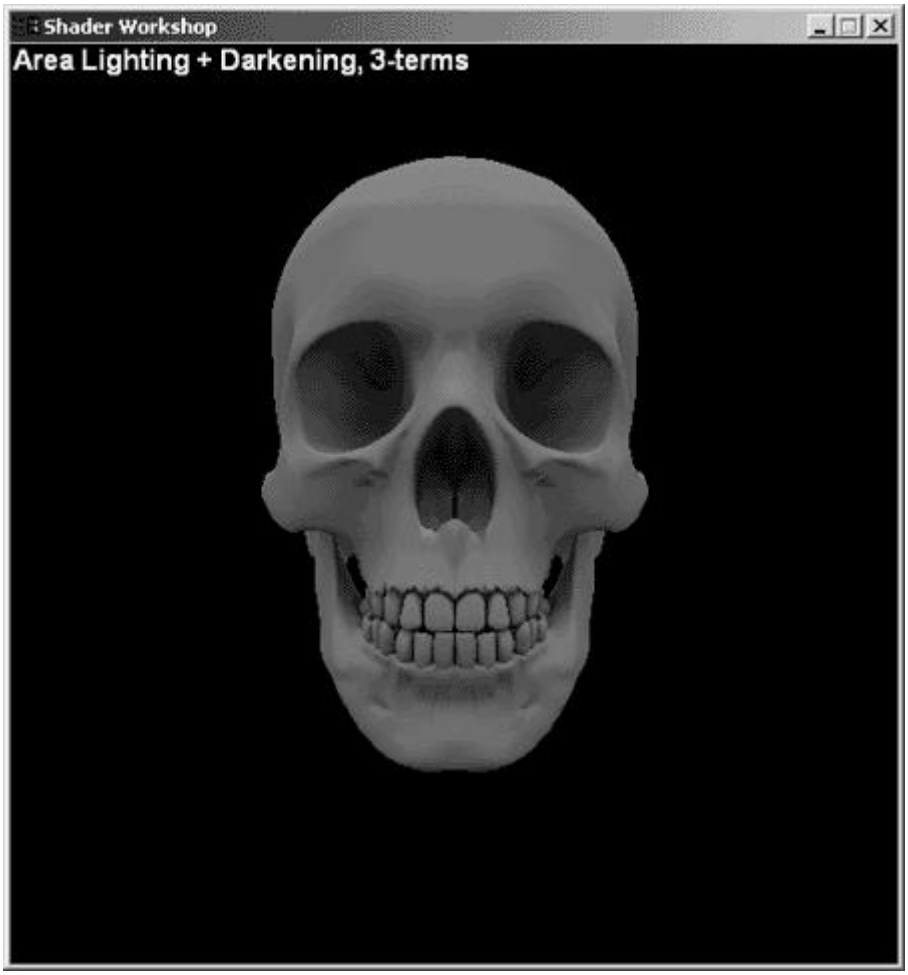

Figure 21. 3-term DIY lighting model image

Finally, combine this with a directional light, as shown in Figure 22, and an amazingly realistic image results for such a relatively simple lighting model.

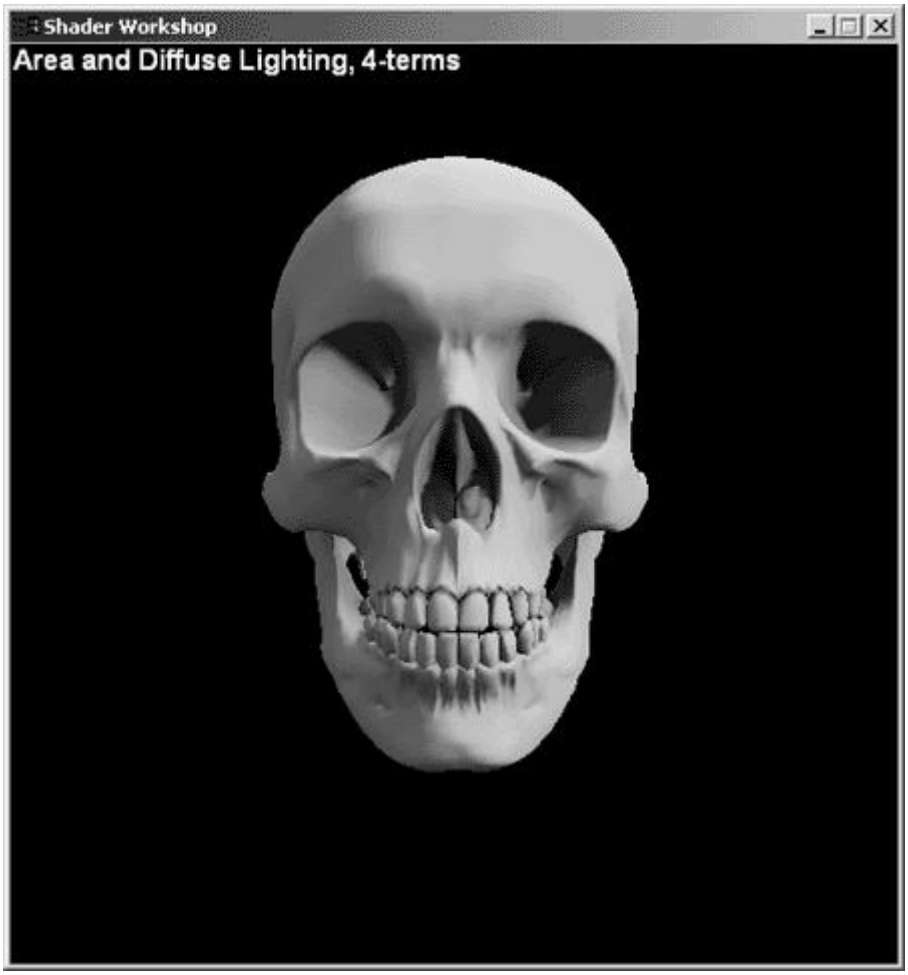

Figure 22. 3-term DIY lighting model image

Now clearly, the process of defining, evaluating, and refining a DIY lighting model is iterative, but it isn't all that difficult, as this article has shown. It's just a matter of clearly thinking through the process of what sort of illumination is necessary to get the desired effect, and working through the iterations until it looks good enough.

There are two lessons here: First, understanding the tangent space basis, so you can correctly perform DOT3 based diffuse and specular lighting, and secondly, understanding how to perform "do-it-yourself" lighting. Each of these lessons is important, and DIY lighting builds on the knowledge of and correct usage of the tangent space basis, but the real kicker is understanding the "lighting black box" and functional equivalencies, so that you feel comfortable using your own approaches to lighting when you want something that appears just a little different.

Last Word

I'd like to acknowledge the help of **Chas Boyd, Dan Baker, Tony Cox,** and **Mike Burrows** (Microsoft) in producing this column. Thanks to Lightwave for the Lightwave images, and Viewpoint Datalabs for the models.

Your feedback is welcome. Feel free to drop me a line at the address below with your comments, questions, topic ideas, or links to your own variations on topics the column covers. Please, though, don't expect an individual reply or send me support questions.

Remember, Microsoft maintains active mailing lists as forums for like-minded developers to share information:

DirectXAV for audio and video issues at http://DISCUSS.MICROSOFT.COM/archives/DIRECTXAV.html.

DirectXDev for graphics, networking, and input at http://DISCUSS.MICROSOFT.COM/archives/DIRECTXDEV.html.

Philip Taylor is the PM for the DirectX SDK, Managed DirectX, the Windows® XP 3D screensavers, and a few more bits and bobs. Previously at Microsoft he was senior engineer in the DirectX evangelism group for DirectX 3.0 to DirectX 8.0, and helped many game ISVs with DirectX. He has worked with DirectX since the first public beta of the GameSDK (DirectX 1.0), and, once upon a time, actually shipped DirectX 2.0 games. In his spare time, he can be found lurking on many 3-D graphics programming mailing lists and Usenet newsgroups. You can reach him at msdn@microsoft.com.