Color, Lighting

305890
Spring 2010
4/16/2010
Kyoung Shin Park

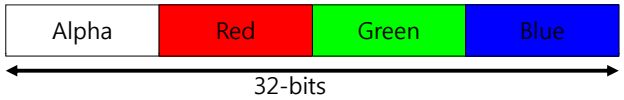
Overview

- Representing Color
- Vertex Colors
- Flat Shading, Gouraud Shading, Phong Shading
- Lighting
  - Light-Material Interaction
  - Diffuse/Ambient/Specular Lighting
  - Light sources: Point Light/Directional Light/Spot Light

Representing Colors

- D3DCOLOR
  - DWORD (32 bits) type
    - 4(Alpha, Red, Green, Blue) 8-bit color
  - D3DCOLOR_ARGB(a, r, g, b) macro
    - alpha, red, green, blue
  - D3DCOLOR_XRGB: red, green, blue (set alpha 0xff)
    - #define D3DCOLOR_XRGB(r, g, b) D3DCOLOR_ARGB(0xff, r, g, b)

```
D3DCOLOR brightRed = D3DCOLOR_ARGB(255, 255, 0, 0)
D3DCOLOR yellowGreen = D3DCOLOR_ARGB(255,128, 255, 0)
```



Representing Colors

- D3DCOLORVALUE
  - 4 float(0 ~ 1) 128-bit color structure

```
typedef struct _D3DCOLORVALUE {
    float r;    // red, 0.0~1.0
    float g;    // green, 0.0~1.0
    float b;    // blue, 0.0~1.0
    float a;    // alpha, 0.0~1.0
} D3DCOLORVALUE;
```

- D3DCOLOR: D3DCOLORVALUE + operators

Representing Colors

- D3DXCOLOR

```
typedef struct D3DXCOLOR {
#ifdef __cplusplus
public:
    D3DXCOLOR() {}
    D3DXCOLOR(DWORD argb);
    D3DXCOLOR(CONST FLOAT *);
    D3DXCOLOR(CONST D3DXFLOAT16*);
    D3DXCOLOR(CONST D3DXCOLORVALUE&);
    D3DXCOLOR(FLOAT r, FLOAT g, FLOAT b, FLOAT a);

    operator DWORD() const;
    operator FLOAT* ();
    operator CONST FLOAT* () const;
    operator D3DCOLORVALUE* ();
    operator CONST D3DCOLORVALUE* () const;
    operator D3DCOLORVALUE& ();
    operator CONST D3DCOLORVALUE& () const;
```

Representing Colors

```
    D3DXCOLOR& operator += (CONST D3DXCOLOR&); //assignment
    D3DXCOLOR& operator -= (CONST D3DXCOLOR&);
    D3DXCOLOR& operator *= (FLOAT);
    D3DXCOLOR& operator /= (FLOAT);
    D3DXCOLOR operator + () const;                    // unary operator
    D3DXCOLOR operator - () const;
    D3DXCOLOR operator + (CONST D3DXCOLOR&) const; // binary operator
    D3DXCOLOR operator – (CONST D3DXCOLOR&) const;
    D3DXCOLOR operator * (FLOAT) const;
    D3DXCOLOR operator / (FLOAT) const;
    friend D3DXCOLOR operator * (FLOAT, CONST D3DXCOLOR&);
    BOOL operator == (CONST D3DXCOLOR&) const;
    BOOL operator != (CONST D3DXCOLOR&) const;
#endif // _cplusplus
    FLOAT r, g, b, a;
} D3DXCOLOR, *LPD3DXCOLOR;
```

Representing Colors

- Representing Colors

```
const D3DXCOLOR WHITE (D3DXCOLOR_XRGB(255, 255, 255));
const D3DXCOLOR BLACK (D3DXCOLOR_XRGB(0, 0, 0));
const D3DXCOLOR RED (D3DXCOLOR_XRGB(255, 0, 0));
const D3DXCOLOR GREEN (D3DXCOLOR_XRGB(0, 255, 0));
const D3DXCOLOR BLUE (D3DXCOLOR_XRGB(0, 0, 255));
const D3DXCOLOR YELLOW (D3DXCOLOR_XRGB(255, 255, 0));
const D3DXCOLOR CYAN (D3DXCOLOR_XRGB(0, 255, 255));
const D3DXCOLOR MAGENTA (D3DXCOLOR_XRGB(255, 0, 255));
```

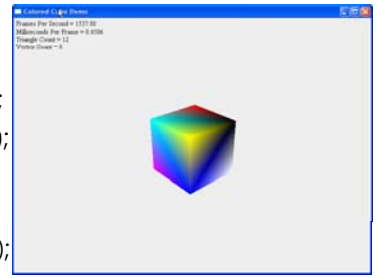
Vertex Color

- Vertex structure include a color to each vertex

```
struct VertexCol {
    float _x, _y, _z;              // XYZ
    D3DCOLOR _color;               // color
    static IDirect3DVertexDeclaration9* Decl;
};

D3DVERTEXELEMENT9 VertexColElements[] = {
    {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_POSITION, 0},
    {0, 12, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_COLOR, 0},
    D3DDECL_END()
};
```

Color Cube Demo



```
// fill the VB with color vertex
VertexCol* v = 0;
HR(m_CubeVB->Lock(0, 0, (void**)&v, 0));
v[0] = VertexCol(-1.0f, -1.0f, -1.0f, WHITE);
v[1] = VertexCol(-1.0f, 1.0f, -1.0f, BLACK);
v[2] = VertexCol( 1.0f, 1.0f, -1.0f, RED);
v[3] = VertexCol( 1.0f, -1.0f, -1.0f, GREEN);
v[4] = VertexCol(-1.0f, -1.0f, 1.0f, BLUE);
v[5] = VertexCol(-1.0f, 1.0f, 1.0f, YELLOW);
v[6] = VertexCol( 1.0f, 1.0f, 1.0f, CYAN);
v[7] = VertexCol( 1.0f, -1.0f, 1.0f, MAGENTA);
HR(m_CubeVB->Unlock());
// disable lighting
_Device->SetRenderState(D3DRS_LIGHTING, false);
```

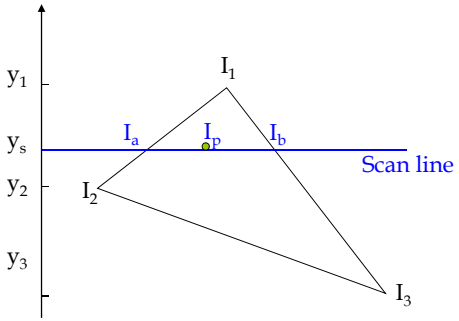
Shading

- Shading is the process of determining the colors of the pixels in a primitive.
- SetRenderState(D3DRS_SHADEMODE, mode) enables the shade mode render state
  - Flat shading: fills the polygon face with the first vertex color
  - Gouraud shading (smooth shading): interpolates the polygon face with the colors at each vertex (Default)

```
VertexCol t[3];
t[0]._color = D3DCOLOR_XRGB(255, 0, 0);
t[1]._color = D3DCOLOR_XRGB(0, 255, 0);
t[2]._color = D3DCOLOR_XRGB(0, 0, 255);
// flat shading
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
// Gouraud shading
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
```

Gouraud Shading

- Gouraud shading
  - The color of a pixel is linearly interpolated using the colors of all vertices in the primitives



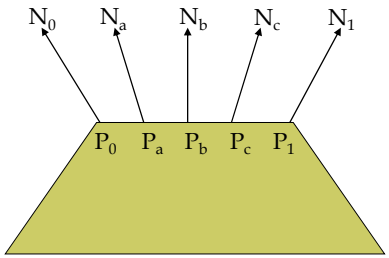
$$I_a = I_1 - (I_1 - I_2)\frac{y_1 - y_s}{y_1 - y_2}$$

$$I_b = I_1 - (I_1 - I_3)\frac{y_1 - y_s}{y_1 - y_3}$$

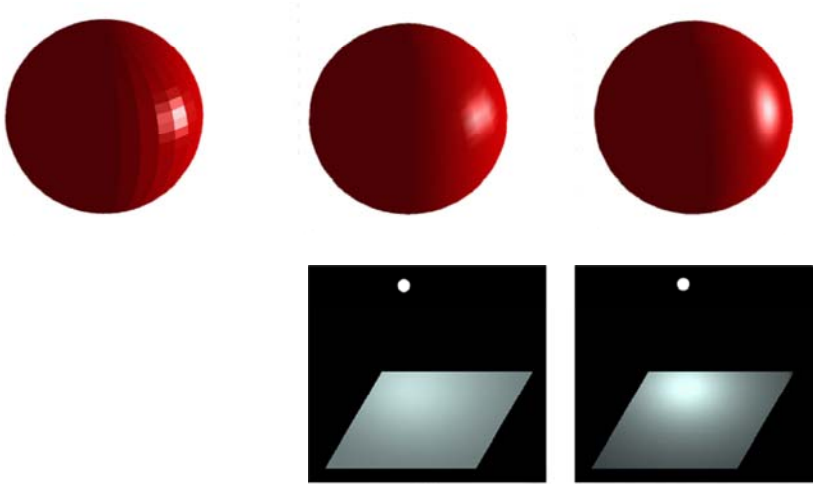
$$I_p = I_b - (I_b - I_a)\frac{x_b - x_p}{x_b - x_a}$$

Phong Shading

- Normal-vector interpolated shading
- Phong shading computes the normal vector for each pixel on the polygon



Flat, Gouraud, and Phong Shading

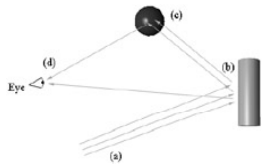


Lighting

- Light Components
- Materials
- Vertex Normals
- Light Sources

Light and Material Interaction

- When using lighting, we no longer specify vertex colors ourselves; rather, we specify materials and lights, and then, apply a lighting equation, which computes the vertex colors for us based on light/material interaction.
- Materials can be thought of as the properties that determine how light interacts with an object.
- We model lights by an additive mixture of red, green, and blue light (RGB); we can simulate many light colors.



Lighting Component

- Light
  - Ambient light – Ambient light comes from no particular direction. It reflects equally in all directions.
  - Diffuse light – The basic lighting effects is diffuse lighting. The intensity of diffuse lighting depends on the orientation of the surface relative to the light source. Diffuse light is reflected equally in all directions.
  - Specular light – Specular light is the light that is directly reflected off the surface to the camera. Its intensity depends on the orientation of the surface relative to camera, as well as to the light source.
    - Device->SetRenderState(D3DRS_SPECULARENABLE, true);

Lighting Component

- Light source color representation using D3DXCOLOR or D3DCOLORVALUE structure

```
D3DXCOLOR redAmbient(1.0f, 0.0f, 0.0f, 1.0f);      // alpha is not used in light
D3DXCOLOR blueDiffuse(0.0f, 0.0f, 1.0f, 1.0f);
D3DXCOLOR whiteSpecular(1.0f, 1.0f, 1.0f, 1.0f);
```

Materials

- Material properties define how a surface reflects light. Basically, they represent the surface color.
- *When lighting is active, the material is used instead of color.*
- D3DMATERIAL9 structure

```
typedef struct _D3DMATERIAL9 {
    D3DCOLORVALUE Diffuse, Ambient, Specular, Emissive;
    float Power;
} D3DMATERIAL9;
```

  - Diffuse/Ambient/Specular: surface diffuse/ambient/specular reflections
  - Emissive: emissive material (to make object appeared to be self-luminous)
  - Power: sharpness of specular reflection (higher value, smaller highlights)

Materials

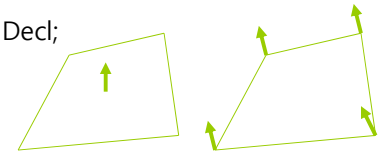
- Red material

```
D3DMATERIAL9 red;
::ZeroMemory(&red, sizeof(red));
red.Diffuse = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f);
red.Ambient = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f);
red.Specular = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f);
red.Emissive = D3DXCOLOR(0.0f, 0.0f, 0.0f, 1.0f); // no emission
red.Power = 5.0f;
```

Vertex Normal

- Normal
  - Lighting computation uses vertex normals
- Vertex structure
  - Use normals instead of colors

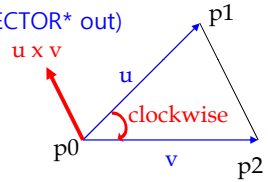
```
struct VertexPN {
    float _x, _y, _z;          // XYZ
    float _nx, _ny, _nz;       // normal
    static IDirect3DVertexDeclaration9* Decl;
};
```

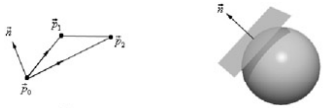


Vertex Normal

- Face Normal
  - Compute the normal vector of a triangle consisting of vertex p0, p1, p2

```
void ComputeNormal(D3DXVECTOR3* p0,
D3DXVECTOR3* p1, D3DXVECTOR3* p2, D3DXVECTOR* out)
{
    D3DXVECTOR3 u = *p1 - *p0;
    D3DXVECTOR3 v = *p2 - *p0;
    D3DXVec3Cross(out, &u, &v);
    D3DXVec3Normalize(out, out);
}
```



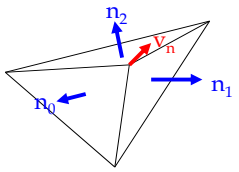


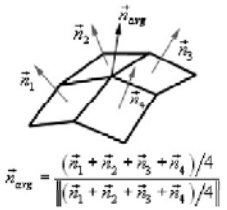
Vertex Normal

- Vertex Normal
  - Vertex normal calculation using adjacent faces

$$v_n = \frac{1}{3}(n_0 + n_1 + n_2)$$

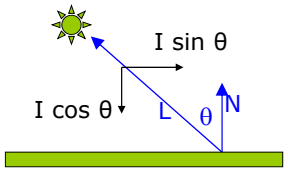
```
Device->SetRenderState(D3DRS_NORMALIZENORMALS, true);
```





Diffuse Lighting

- Lambert's Cosine Law
  - θ between the normal and light vector
  - Maximum intensity when the normal and light vector are perfectly aligned (θ = 0)
  - f(θ) = max(cosθ, 0) = max(L • N, 0)

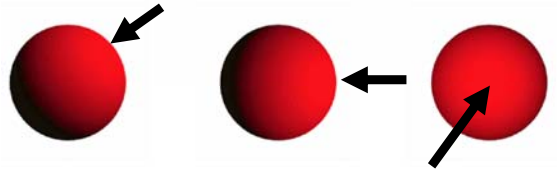


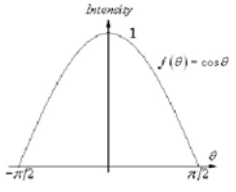
$$Diffuse\ Reflection \propto \cos\theta$$

$$Diffuse\ Reflection = K_d I_d \cos\theta = K_d I_d (N \bullet L)$$

Light vector
Surface normal vector

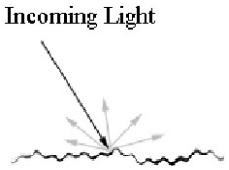
I_d: diffuse light intensity
K_d: diffuse light coefficient





Diffuse Lighting

- Diffuse lighting calculation (viewpoint independent)



$$\max(L \bullet N, 0) \bullet (L_a \otimes M_a)$$

L_d: diffuse light color
M_d: diffuse material color
L: light vector
N: vertex normal vector

Ambient Lighting

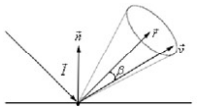
- Ambient lighting calculation

$$(L_a \otimes M_a)$$

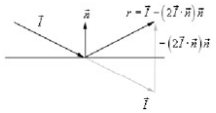
L_a: ambient light color
M_a: ambient material color

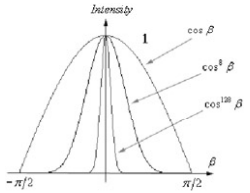
Specular Lighting

- Specular lighting calculation (viewpoint dependent)



$$\cdot \vec{E} \quad (\max(V \bullet R, 0))^p \bullet (L_s \otimes M_s)$$





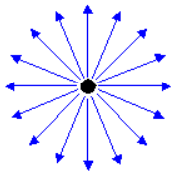
L_s: specular light color
M_s: specular material color
V: view vector
R: light reflection vector, R = L – (2L • N)N

Point/Directional/Spot Light

- D3D light sources
  - Point light
  - Directional light
  - Spot light

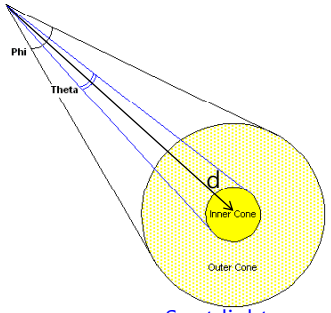
$$L = \frac{(S-P)}{\|S-P\|}$$

L: light vector
S: point/directional light position
P: vertex position



Point light

Directional light



Spot light

Light Attenuation

- Light attenuation
  - Light intensity weakness as a function of distance based on the inverse square law.

$$I(d) = \frac{I_0}{d^2}$$

$$I(d) = \frac{I_0}{a_0 + a_1 d + a_2 d^2}$$

$$d = \|S-P\| (i.e., the\ distance\ between\ P\ and\ S)$$

Light Sources

- D3DLIGHT9 structure

```
typedef struct _D3DLIGHT9 {
    D3DLIGHTTYPE Type;   // light source type (point, spot, directional)
    D3DLIGHTVALUE Diffuse;          // diffuse, specular, ambient
    D3DLIGHTVALUE Specular;
    D3DLIGHTVALUE Ambient;
    D3DVECTOR Position;  // light position. directional light인 경우 무시
    D3DVECTOR Direction; // light direction. spot light인 경우 무시
    float Range;   // 빛이 완전히 소멸할 때까지 진행할 수 있는 거리.
    float Falloff;   // spot light에서 안쪽 원뿔과 바깥쪽 원뿔 간의 빛의 세기 차이.
    float Attenuation0;// 빛의 세기가 거리에 따라 약해지는 정도.
    float Attenuation1;
    float Attenuation2;
    float Theta;    // spot light에서 안쪽 원뿔의 각도를 radian값으로 지정
    float Phi;      // spot light에서 바깥쪽 원뿔의 각도를 radian값으로 지정
} D3DLIGHT9;
```

Example: Point Light Demo

```
const D3DXCOLOR WHITE(1.0f, 1.0f, 1.0f, 1.0f);
const D3DXCOLOR BLACK(0.0f, 0.0f, 0.0f, 1.0f);
const D3DXCOLOR RED(1.0f, 0.0f, 0.0f, 1.0f);
const D3DXCOLOR GREEN(0.0f, 1.0f, 0.0f, 1.0f);
const D3DXCOLOR BLUE(0.0f, 0.0f, 1.0f, 1.0f);

struct Mtrl {
    Mtrl() :ambient(WHITE), diffuse(WHITE), spec(WHITE), specPower(8.0f){}
    Mtrl(const D3DXCOLOR& a, const D3DXCOLOR& d, const D3DXCOLOR& s,
        float power) :ambient(a), diffuse(d), spec(s),specPower(power){}
    D3DXCOLOR ambient;
    D3DXCOLOR diffuse;
    D3DXCOLOR spec;
    float specPower;
};
Mtrl mSridMtrl = Mtrl(BLUE, BLUE, WHITE, 16.0f);
Mtrl mCylinderMtrl = Mtrl(RED, RED, WHITE, 8.0f);
Mtrl mSphereMtrl = Mtrl(GREEN, GREEN, WHITE, 8.0f);
```

Example: Point Light Demo

```
void PointLightDemo::drawCylinders()
{
    D3DXMATRIX T, R, W, WIT;
    D3DXMatrixRotationX(&R, D3DX_PI*0.5f);
    HR(mFX->SetValue(mhAmbientMtrl, & mCylinderMtrl.ambient, sizeof(D3DXCOLOR)));
    HR(mFX->SetValue(mhDiffuseMtrl, &mCylinderMtrl.diffuse, sizeof(D3DXCOLOR)));
    HR(mFX->SetValue(mhSpecMtrl, &mCylinderMtrl.spec, sizeof(D3DXCOLOR)));
    HR(mFX->SetFloat(mhSpecPower, mCylinderMtrl.specPower));

    // ... 중간생략
}
```

Example: Point Light Demo

```
struct OutputVS {
    float4 posH : POSITION0;
    float4 color : COLOR0;
};
OutputVS PointLightVS(float3 post : POSITION0, float3 normalL : NORMAL0)
{
    // Zero out our output.
    OutputVS outVS = (OutputVS)0;
    // Transform normal to world space.
    float3 normalW = mul(float4(normalL, 0.0f), gWorldInvTrans).xyz;
    normalW = normalize(normalW);
    // Transform vertex position to world space.
    float3 posW = mul(float4(posL, 1.0f), gWorld).xyz;
    // Unit vector from vertex to light source.
    float3 lightVecW = normalize(gLightPosW - posW);
    // Ambient Light Computation.
    float3 ambient = (gAmbientMtrl*gAmbientLight).rgb;
    // Diffuse Light Computation.
    float s = max(dot(normalW, lightVecW), 0.0f);
    float3 diffuse = s*(gDiffuseMtrl*gDiffuseLight).rgb;
```

$$(\vec{c}_a \otimes \vec{m}_a) + \frac{\max(\vec{L}\cdot\vec{n},0)\cdot(\vec{c}_d \otimes \vec{m}_d) + \max(\vec{v}\cdot\vec{r},0)^p \cdot (\vec{c}_s \otimes \vec{m}_s)}{a_0 + a_1 d + a_2 d^2}$$

Example: Point Light Demo

```
        // Specular Light Computation.
        float3 toEyeW = normalize(gEyePosW - posW);
        float3 reflectW = reflect(-lightVecW, normalW);
        float t = pow(max(dot(reflectW, toEyeW), 0.0f), gSpecPower);
        float3 spec = t*(gSpecMtrl*gSpecLight).rgb;
        // Attentuation.
        float d = distance(gLightPosW, posW);
        float A = gAttenuation012.x + gAttenuation012.y*d + gAttenuation012.z*d*d;
        // Everything together (Light Equation)
        float3 color = ambient + ((diffuse + spec) / A);
        // Pass on color and diffuse material alpha.
        outVS.color = float4(color, gDiffuseMtrl.a);
        // Transform to homogeneous clip space.
        outVS.posH = mul(float4(posL, 1.0f), gWVP);
        // Done--return the output.
        return outVS;
    }
```

$$spot\left[(\vec{c}_a \otimes \vec{m}_a) + \frac{\max(\vec{L}\cdot\vec{n},0)\cdot(\vec{c}_d \otimes \vec{m}_d) + \max(\vec{v}\cdot\vec{r},0)^p \cdot (\vec{c}_s \otimes \vec{m}_s)}{a_0 + a_1 d + a_2 d^2}\right]$$

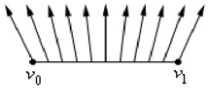
$$spot = \left(\max(-\vec{L}\cdot\vec{d},0)\right)^s$$

Example: Spot Light Demo

```
void SpotLightDemo::buildViewMtx()
{
    float x = mCameraRadius * cosf(mCameraRotationY);
    float z = mCameraRadius * sinf(mCameraRotationY);
    D3DXVECTOR3 pos(x, mCameraHeight, z);
    D3DXVECTOR3 target(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 up(0.0f, 1.0f, 0.0f);
    D3DXMatrixLookAtLH(&mView, &pos, &target, &up);

    HR(mFX->SetValue(mhEyePos, &pos, sizeof(D3DXVECTOR3)));
    // Spotlight position is the same as camera's position.
    HR(mFX->SetValue(mhLightPosW, &pos, sizeof(D3DXVECTOR3)));
    // Spotlight direction is the same as camera's forward direction.
    D3DXVECTOR3 lightDir = target - pos;
    D3DXVec3Normalize(&lightDir, &lightDir);
    HR(mFX->SetValue(mhLightDirW, &lightDir, sizeof(D3DXVECTOR3)));
}
```

Example: Phong Shading Demo




```
struct OutputVS {
    float4 posH : POSITION0;
    float3 normalW : TEXCOORD0;
    float3 posW : TEXCOORD1;
};
// Phong shading directional light
OutputVS PhongVS(float3 posL : POSITION0, float3 normalL : NORMAL0)
{
    // Zero out our output.
    OutputVS outVS = (OutputVS)0;
    // Transform normal to world space.
    outVS.normalW = mul(float4(normalL, 0.0f), gWorldInverseTranspose).xyz;
    outVS.normalW = normalize(outVS.normalW);
    // Transform vertex position to world space.
    outVS.posW = mul(float4(posL, 1.0f), gWorld).xyz;
    // Transform to homogeneous clip space.
    outVS.posH = mul(float4(posL, 1.0f), gWVP);
    // Done--return the output.
    return outVS;
}
```

$$(\vec{c}_a \otimes \vec{m}_a) + \max(\vec{L}\cdot\vec{n},0)\cdot(\vec{c}_d \otimes \vec{m}_d) + (\max(\vec{v}\cdot\vec{r},0))^p \cdot (\vec{c}_s \otimes \vec{m}_s)$$

Example: Phong Shading Demo

```
float4 PhongPS(float3 normalW : TEXCOORD0, float3 posW : TEXCOORD1) : COLOR
{
    // Interpolated normals can become unnormal--so normalize.
    normalW = normalize(normalW);
    // Compute the vector from the vertex to the eye position.
    float3 toEye = normalize(gEyePosW - posW);
    // Compute the reflection vector.
    float3 r = reflect(-gLightVecW, normalW);
    // Determine how much (if any) specular light makes it into the eye.
    float t = pow(max(dot(r, toEye), 0.0f), gSpecularPower);
    // Determine the diffuse light intensity that strikes the vertex.
    float s = max(dot(gLightVecW, normalW), 0.0f);
    // Compute the ambient, diffuse, and specular terms separately.
    float3 spec = t*(gSpecularMtrl*gSpecularLight).rgb;
    float3 diffuse = s*(gDiffuseMtrl*gDiffuseLight).rgb;
    float3 ambient = gAmbientMtrl*gAmbientLight;
    // Sum all the terms together and copy over the diffuse alpha.
    return float4(ambient + diffuse + spec, gDiffuseMtrl.a);
}
```