Using Shaders for Lighting

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
Lighting Definitions

\[ \text{Color} = \text{LightColor} \times \text{MaterialColor} \]

- **Ambient** = Light intensity that is “everywhere”
- **Diffuse** = Light intensity proportional to \( \cos(\Theta) \)
- **Specular** = Light intensity proportional to \( \cos^s(\Phi) \)
- **A-D-S** = Lighting model that includes Ambient, Diffuse, and Specular

- **Flat Interpolation** = Use a single polygon normal to compute one A-D-S for the entire polygon
- **Smooth Interpolation** = Use a normal at each vertex to compute one A-D-S for each vertex

- **Per-vertex lighting** = Compute A-D-S using each vertex normal and then interpolate the summed intensity over the entire polygon
- **Per-fragment lighting** = Interpolate the vertex normals across the entire polygon and then compute A-D-S at each fragment

- **CubeMap Reflection** = Using the Eye Reflection Vector (ER) to look-up the reflection of a “wall texture”
A-D-S Lighting

Ambient:  \( K_a \)

Diffuse:  \( K_d \cdot \cos \theta \)

Specular:  \( K_s \cdot \cos^s \varphi \)
Ambient-only  
Diffuse-only  
Specular-only  

ADS – Shininess=50  
ADS – Shininess=1000  
ADS – Shininess=1000 -- Flat
A-D-S Lighting with Flat Interpolation

Each polygon has a single lighting value applied to every pixel within it.

N = Normal
L = Light vector
E = Eye vector
R = Light reflection vector
ER = Eye reflection vector
Color = LightColor * MaterialColor

```
vec3 ambient = Color.rgb;
diffuse  = max( dot(L,N), 0. ) * Color.rgb;
vec3 R = normalize( reflect( -L, N ) );
vec3 spec = LightColor * pow( max( dot( R, E), 0. ), Shininess );
gl_FragColor.rgb = Ka*ambient + Kd*diffuse + Ks*spec;
```
What you see depends on the light color and the material color

ER = LR * MR
EG = LG * MG
EB = LB * MB

This is how you implement subtractive coloring.
A-D-S Lighting with Smooth Interpolation

Note: In per-vertex lighting, the light intensity is computed at each vertex and interpolated throughout the polygon. This creates artifacts such as Mach Banding and the fact that the bright spot is “jagged”.

You can do this in stock OpenGL or in a shader.

\[
\begin{align*}
N &= \text{Normal} \\
L &= \text{Light vector} \\
E &= \text{Eye vector} \\
R &= \text{Light reflection vector} \\
ER &= \text{Eye reflection vector} \\
\text{Color} &= \text{LightColor} \times \text{MaterialColor}
\end{align*}
\]

Vertex Shader

\[
\begin{align*}
\text{vec3 ambient} &= \text{Color.rgb;}
\text{diffuse} &= \max(\ \text{dot}(L,N), 0. ) \times \text{Color.rgb;}
\text{vec3 R} &= \text{normalize( reflect( -L, N ) );}
\text{vec3 spec} &= \text{LightColor} \times \text{pow}( \max( \text{dot}( R, E), 0. ), \text{Shininess } );
\end{align*}
\]

Fragment Shader

\[
\text{gl_FragColor.rgb} = \text{Ka*ambient} + \text{Kd*diffuse} + \text{Ks*spec};
\]

Smooth-rasterize ambient, diffuse, spec
A-D-S Lighting with Normal Interpolation

In per-fragment lighting, the normal is interpolated throughout the polygon. The light intensity is computed at each fragment. This avoids Mach Banding and makes the bright spot smooth.

You can only do this in a shader.

Fragment Shader

```
vec3 ambient = Color.rgb;
diffuse = max( dot(L,N), 0. ) * Color.rgb;
vec3 R = normalize( reflect( -L, N ) );
vec3 spec = LightColor * pow( max( dot( R, E ), 0. ), Shininess );
gl_FragColor.rgb = Ka*ambient + Kd*diffuse + Ks*spec;
```
The Difference Between Per-Vertex Lighting and Per-Fragment Lighting

Per-vertex

Per-fragment
The Difference Between Per-Vertex Lighting and Per-Fragment Lighting

Per-vertex

Per-fragment
Flat shading

Normal interpolation
A-D-S Lighting with Normal Interpolation and a CubeMap Reflection

vec3 ambient = Color.rgb;
diffuse  = max( dot(L,N), 0. ) * Color.rgb;
vec3 R = normalize( reflect( -L, N ) );
vec3 spec = LightColor * pow( max( dot( R, E ), 0. ), Shininess );
vec3 reflcolor = textureCube( ReflectUnit, R ).rgb;
gl_FragColor.rgb = Ka*ambient + Kd*diffuse + Ks*spec + Kr*reflcolor.rgb;

Note: A cube map reflection is blended in, given a stronger impression that the surface is shiny.

N = Normal
L = Light vector
E = Eye vector
R = Light reflection vector
ER = Eye reflection vector
Color = LightColor * MaterialColor

Smooth-rasterize N, L, E

Fragment Shader

vec3 ambient = Color.rgb;
diffuse  = max( dot(L,N), 0. ) * Color.rgb;
vec3 R = normalize( reflect( -L, N ) );
vec3 spec = LightColor * pow( max( dot( R, E ), 0. ), Shininess );
vec3 reflcolor = textureCube( ReflectUnit, R ).rgb;
gl_FragColor.rgb = Ka*ambient + Kd*diffuse + Ks*spec + Kr*reflcolor.rgb;
A-D-S Anisotropic Lighting with Normal Interpolation


Note: The bright spot is not circular because the material has different properties in different directions. Materials such as fur, hair, and brushed metal behave this way.

\[
\text{vec3 ambient = Color.rgb;}
\text{float dl = dot( T, L );}
\text{vec3 diffuse = sqrt( 1. - dl*dl ) * Color.rgb;}
\text{float de = dot( T, E );}
\text{vec3 spec = LightColor * pow( dl * de + sqrt( 1. - dl*dl ) * sqrt( 1. - de*de ), Shininess );}
\text{gl_FragColor.rgb = Ka*ambient + Kd*diffuse + Ks*spec;}
\]
Summary

Flat

Normal

Anisotropic

Smooth

Reflection
#version 330 compatibility

uniform float uLightX, uLightY, uLightZ;

flat out vec3 vNf;
    out vec3 vNs;
flat out vec3 vLf;
    out vec3 vLs;
flat out vec3 vEf;
    out vec3 vEs;

vec3 eyeLightPosition = vec3( uLightX, uLightY, uLightZ );

void main( )
{
    vec4 ECposition = gl_ModelViewMatrix * gl_Vertex;

    Nf = normalize( gl_NormalMatrix * gl_Normal );     // surface normal vector
    Ns = Nf;

    Lf = eyeLightPosition - ECposition.xyz;           // vector from the point
    Ls = Lf;                                         // to the light position

    Ef = vec3( 0., 0., 0. ) - ECposition.xyz;         // vector from the point
    Es = Ef;                                         // to the eye position

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
#version 330 compatibility

uniform float uKa, uKd, uKs;
uniform vec4 uColor;
uniform vec4 uSpecularColor;
uniform float uShininess;
uniform bool uFlat;

flat in vec3 vNf;
in vec3v Ns;
flat in vec3 vLf;
in vec3v Ls;
flat in vec3 vEf;
in vec3 vEs;

void main()
{
  vec3 Normal;
  vec3 Light;
  vec3 Eye;

  if( uFlat )
  {
    Normal = normalize(vNf);
    Light  = normalize(vLf);
    Eye    = normalize(vEf);
  }
  else
  {
    Normal = normalize(vNs);
    Light  = normalize(vLs);
    Eye    = normalize(vEs);
  }

  Per-fragment lighting: the fragment shader, I
vec4 ambient = uKa * uColor;

float d = max( dot(Normal,Light), 0. );
vec4 diffuse = uKd * d * uColor;

float s = 0.;
if( dot(Normal,Light) > 0. ) // only do specular if the light can see the point
{
    vec3 ref = normalize( 2. * Normal * dot(Normal,Light) - Light );
    s = pow( max( dot(Eye,ref),0. ), uShininess );
}

vec4 specular = uKs * s * uSpecularColor;

gl_FragColor = vec4( ambient.rgb + diffuse.rgb + specular.rgb, 1. );