# 6 Lighting

The simplest way to perform lighting is by computing it per-vertex, which would place responsibility for most of the work squarely on the shoulders of the vertex shader. If lighting is performed this way, the color is computed based on light and material properties that determine the color of each vertex based on the standard *ambient-diffuse-specular* (*ADS*) lighting model. This per-vertex color can be used for either flat or smooth shading. However, if a more complex shading model is to be used, such as Phong or anisotropic shading, the color computation will probably be deferred until the fragment shader, where per-pixel color can be computed.

In this chapter, we will discuss both per-vertex and per-fragment lighting methods.

## The ADS Lighting Model

This lighting model is the basis for fixed-function OpenGL lighting, and we want to see how to handle this in shaders you write yourself. You were probably introduced to this in your beginning computer graphics course, but let's review it to be sure we're all using the same terminology and notation. The three kinds of light used in this model are

- *Ambient* light, or light that is always present at all points in a scene.
- *Diffuse* light, or light that comes directly from a light source.
- *Specular* light, or light that is reflected in a "shiny" way from a light source by an object.

Each of these kinds of light contributes to the overall lighting at any point in a separate way. The general context for these contributions is shown in Figure 6.1, which illustrates a point on a surface with normalized (unit) vectors from the point to the eye, $\hat{E}$; from the point to a light source, $\hat{L}$; the normal to the surface at the point, $\hat{N}$; and the reflected light direction $\hat{R}$.

Ambient light contributes to the lighting as a product of the ambient light itself $L_A$ and the ambient light color of the material being lighted $M_A$:

$$A = L_A * M_A .$$

Diffuse light contributes to the lighting as a product of the diffuse light itself $L_D$, the diffuse light color of the material being lighted $M_D$, and the cosine of the angle $\Theta$ between the light and the normal, $(\hat{L} \bullet \hat{N})$:

$$D = L_D * M_D * (\hat{L} \bullet \hat{N}).$$

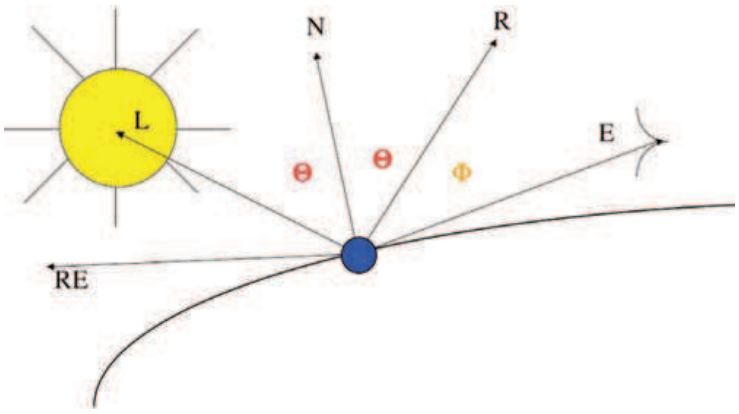


**Figure 6.1.** The setup for ADS lighting.

Specular light contributes to the lighting as a product of the specular light itself $L_S$, the specular light color of the material being lighted $M_S$, and a power (the "shininess" coefficient *SH*) of the cosine of the angle Φ between the eye vector and the light reflection vector, $(\hat{R} \bullet \hat{E})^{SH}$:

$$S = L_S * M_S * (\hat{R} \bullet \hat{E})^{SH}.$$

Then the total lighting at the point is the sum of these:

$$A + D + S = L_A * M_A + L_D * M_D * (\hat{L} \bullet \hat{N}) + L_S * M_S * (\hat{R} \bullet \hat{E})^{SH}.$$

The reflection vector R is calculated by $R = 2(\hat{N} \bullet \hat{L})\hat{N} - \hat{L}$. Details on how these individual formulas are derived may be found in any introductory graphics text, such as [14]. Also, GLSL has a built-in function called `reflect( )`, which will do this for you.

This model can also take into account attenuation, or the reduction in light intensity with distance. OpenGL models this with three factors: a constant attenuation $A_C$, a linear attenuation $A_L$, and a quadratic attenuation $A_Q$. If a point is at a distance *D* from a light, the overall attenuation *A* is calculated as

$$A = \frac{1}{A_C + A_L D + A_Q D^2}.$$

The distance can be calculated from the light and vertex positions in eye space, and this value of *A* then multiplies the diffuse and specular terms above.

In the ADS lighting function in the next section, we use the reflected-light formulation because we have access to the reflection for each pixel, using the GLSL function `reflect( )` to compute the reflection vector. However, fixed-function OpenGL uses the half-angle formulation for specular light because it is easier to compute for each vertex.

## The ADS Lighting Model Function

Below is a function that computes the color at a vertex based on the ADS lighting model with standard light and material definitions. It is intended for use with *glman*, so it uses stubs for the values it would get from another source. These stubbed values would come from system uniform variables, as noted in the function's comments.

You can use this function in a vertex shader if you are computing the color at each vertex, as you would if you were planning to interpolate the color across the graphics primitive, as in smooth shading, or you can use it in a frag-

ment shader if you are computing the color at each pixel for Phong shading. These two kinds of shading were discussed earlier in this chapter. Because we have not yet talked about the GLSL programming API, we have stubbed in the light and materials definitions in the function, indicating where they would come from if this were part of a graphics application.

```
//Assumed context:
//uniform variables uLightsource[i] and uFrontMaterial are
//stubbed with constant values below. These would probably be
//passed into the shader function if used in an application.
//
//variables myNormal and myPosition are passed in; in a vertex
//shader these would be computed and used directly, while in a
//fragment shader these would be set by the associated vertex
//shader.
//
//the ADS color is returned from the function

vec3 ADSLightModel( in vec3 myNormal, in vec3 myPosition )
{
  const vec3 myLightPosition     = vec3( 1. , 0.5, 0.  );

  const vec3 myLightAmbient      = vec3( 0.2, 0.2, 0.2 );
  const vec3 myLightDiffuse      = vec3( 1. , 1. , 1 . );
  const vec3 myLightSpecular     = vec3( 1. , 1. , 1.  );

  const vec3 myMaterialAmbient   = vec3( 1. , 0.5, 0.  );
  const vec3 myMaterialDiffuse   = vec3( 1. , 0.5, 0.  );
  const vec3 myMaterialSpecular  = vec3( 0.6, 0.6, 0.6 );

  const float myMaterialShininess = 80.;

//normal, light, view, and light reflection vectors
  vec3 norm   = normalize( myNormal );
  vec3 lightv = normalize( myLightPosition - myPosition);
  vec3 viewv  = normalize( vec3(0.,0.,0.) - myPosition );
  vec3 refl   = reflect( vec3(0.,0.,0.) - lightv, norm );
//ambient light computation
  vec3 ambient = myMaterialAmbient*myLightAmbient;

//diffuse light computation
  vec3 diffuse = max(0.0, dot(lightv, norm)) * myMaterialDiffuse
                 *myLightDiffuse;

//Optionally you can add a diffuse attenuation term at this
//point
```

```
//specular light computation
  vec3 specular = vec3( 0.0, 0.0, 0.0 );
  if( dot(lightv, viewv) > 0.0)
  {
    specular = pow(max(0.0, dot(viewv,refl)),
                   myMaterialShininess)*myMaterialSpecular*
                   myLightSpecular;
  }
  return clamp( ambient + diffuse + specular, 0.0, 1.0);
}
```

This calculation does not take into account lighting attenuation. If you want to include attenuation, you can enhance this computation by computing the distance to the light and getting the light's constant, linear, and quadratic attenuation terms as uniform variables, and then computing

```
1./(constant + linear*distance + quadratic*distance*distance)
```

as a multiplier of the diffuse and specular components, as described above. (Attenuation does not act on the ambient light component.)

These computations use simple vector addition and subtraction, not homogeneous addition and subtraction, because we want to keep this simple. If you want to make them fully general, you would need to replace these with homogeneous vector addition and subtraction, as we discussed in Chapter 1. This would be necessary, for instance, if you have a directional light source (which acts as if it were placed at infinity).

## Types of Lights

Since the fixed-function pipeline does all the color computations at the vertex processing stage, whenever you use shaders to replace fixed-function operations, you must handle lighting yourself. Besides the full ADS lighting model, there are other issues in lighting because OpenGL supports spot lights and directional lights, as well as positional lights. To be able to replace fixed-function lighting computations, you must have ways to handle all the options that you plan to use. If you are using lighting, you are probably using material properties as well.

Overall, the OpenGL API gives you ways to define color, lights, and material properties that are treated globally in the graphics system. So you may define a light position, a color, etc. using the API calls to set their global properties, so that any shader can pick them up. We have often used an alternate approach of

Recall our assumption that in our example shader code, we use general attribute and uniform variables with our first-letter naming convention instead of the built-in OpenGL variable names. These names are close enough to the built-in variable names that you can easily convert them if you are working in compatibility mode.

setting discrete uniform variables in our examples, because we can then put them on sliders so that you can experiment with them. In applications, though, you should probably take the more global OpenGL API approach. This will be described in Chapter 14.

### Positional Lights

The most common kind of lighting in OpenGL scenes is with positional lights. Each light has position, color, and a number of other values.

For positional lights, the primary consideration is the direction from a vertex to the light source, and you can get that by a simple vector subtraction so you can make it an `out` vector in the vertex shader and pass it to the fragment shader. Alternately, you can make the vertex position in eye space an `out` variable so the fragment shader can use the ADS lighting function. Your choice will probably depend on the effect you are trying to achieve. As we will see in examples below, you can get traditional smooth shading by computing the light direction at each vertex and defining the color as an `out` variable in a vertex (or tessellation) shader, while you can get Phong shading by defining the normal as an `out` variable and interpolating either the vertex position or the light direction for each pixel.

| Lighting Method | Vertex Shader Does | Rasterizer Interpolates | Fragment Shader Does |
|---|---|---|---|
| Per-vertex | Lighting model | Color | Applies color |
| Per-fragment | Setup | Normal and EC position | Lighting model |

### Directional Lights

If you use directional lights or spot lights, the necessary data for using these kinds of lights can be found in the components of the built-in uniform `uLightSource[i]` struct. Directional lights, also called *parallel light sources*, are

treated in almost the same way as positional lights, except that the direction to the light is always the same, regardless of the position of a point. This simplifies the light direction in any lighting computation by letting you use the light direction directly, instead of computing the direction between the point and the light position. Conceptually, for a directional light, you simply treat the light as a homogeneous point at infinity.

## Spot Lights

Spot lights include specifications for the direction, cutoff, and attenuation. To use a spot light, you must compute the angle between the light direction and the direction from the light to the vertex. By comparing that to the light's cutoff angle and using the light's attenuation, you can then determine the value of the light at the vertex. This requires the vertex shader to send both the light position and the light direction to the fragment shader, and the fragment shader must calculate the angle between the light direction and the vector from the light to the point in order to see whether to use the light in the color computation.

In the vertex shader example below, you can see the kind of computation that is needed to compute the light intensity for a spot light. The color always includes the ambient light, and it uses diffuse and specular light for the particular light source only if the point is close enough to the light direction. The effect of spot lighting is shown in Figure 6.2, where the light shines on only part of the geometric primitive, but we omit the specular contribution in this case to simplify the computation.

A vertex shader for lighting with a spot light or directional light (or both) requires us to manage that lighting function ourselves. The fixed-function OpenGL spot light on the standard teapot is shown in Figure 6.2 (top), while we can use the capabilities of GLSL and the vertex shader to create the "fuzzy" spot light shown in Figure 6.2 (bottom). The vertex shader for this example has only three things to do:

**Figure 6.2.** The effect of a spot light on a teapot that lies on the edge of the light's illumination area. Traditional OpenGL spot light (top) and a spot light with a fuzzy edge (bottom).

- Copy the color from the attribute variable `aColor` to an `out` variable such as `vColor`.
- Set an `out` variable such as `vLightIntensity` with the light intensity based on diffuse lighting computations at this vertex.
- Set an `out` variable such as `vECposition` with the eye coordinates of the vertex.

The fragment shader carries out all the interesting computations that simulate spot lighting for *glman* use. The positions of the light, the eye, and a focal point of the light are set in eye space to define two vectors that meet at the focal point, and uniform slider variables are used to set the angle of the light and the horizontal location (the variable `LeftRight`) of the light focal point. The cosine of the angle set by the vectors is compared with the cosine of the cutoff angle in a `smoothstep( )` function to determine the amount of diffuse light to include for each pixel. The simulation uses a number of parameters that would normally be taken from the uniform lighting variables provided by the system. See the GLSL API for more details.

```
uniform float uAngle;
uniform float uLeftRight;
uniform float uWidth;

in vec4  vColor;
in float vLightIntensity;
in vec3  vECposition;

out vec4 fFragColor;

const vec4  LIGHTPOS  = vec4(0.,0.,40.,1.);
const float AMBCOEFF  = 0.5;
        // simulate ambient reflection coefficient
const float DIFFCOEFF = 0.6;
        // simulate diffuse reflection coefficient

void main( )
{
  // stubs for data in system attribute variables
  // simulate MC light position

  vec3  ECLightTarget  = vec3( uModelViewMatrix *
                    vec4( uLeftRight, 0., 1.5, 1. ) );
  vec3  LightDirection = normalize( ECLightTarget - LIGHTPOS );
  vec3  EyeDirection   = normalize( vECposition   - LIGHTPOS );

  // Ambient only
```

```
    fFragColor = vLightIntensity*AMBCOEFF*vColor;

    // Add diffuse light based on spotlight
    float myAngleCosine  = dot( LightDirection, EyeDirection );
    float CutoffCosine   = cos( radians(uAngle) );
    float BlendFactor    = smoothstep( CutoffCosine - uWidth,
                        CutoffCosine + uWidth, myAngleCosine);

    fFragColor += DIFFCOEFF*BlendFactor*vColor*vLightIntensity;
}
```

Of course, in an application, `uAngle` and `uWidth` would be passed to the shader as uniform variables from the application, and it would be better to compute the value of `CutoffCosine` there, instead of for each pixel. We do it as above in order to take advantage of *glman*.

## Setting Up Lighting for Shading

Shading is the process of determining the color of each pixel in each primitive in your scene. This is actually carried out in the fragment processing part of the graphics processor that we described in Figure 1.5, but the vertex processor must set up the right environment for the kind of shading that you will implement. In this section, we will discuss some kinds of shading and how they are set up. In our discussion, we will draw on several shader concepts from Chapter 2.

The standard shading models available in fixed-function OpenGL are limited. They are *flat shading*, where a polygon is given a single color, and *smooth shading*, where the colors at the vertices of the polygon are interpolated to fill its interior. These are far from the only kinds of shading that have been used in the graphics field, but they are enough for many kinds of graphics work. More sophisticated shading is discussed later in this chapter and in Chapter 8.

Recall from the discussions in Chapter 1 that the fixed-function vertex processor must set a color for each vertex, and that the fragment processor can only interpolate vertex colors. This gives us our first two kinds of shading: flat shading and smooth shading. However, if we have vertex and fragment shaders, we can set up `out` variables in the vertex shader so that the fragment shader can interpolate other information and compute each pixel's color directly. This gives us two other kinds of shading: Phong shading and anisotropic shading.

## Flat Shading

Flat shading is a type of per-vertex color computation. In order to use flat shading for a graphics primitive, the vertex shader will determine a color for a particular vertex (called the *provoking vertex*) and pass it forward to the fragment processor. The color will not be interpolated across the fragments. The color can come from an `aColor` attribute variable, or it could come from a lighting calculation, as described below.

In early versions of GLSL, it was not possible to specify flat shading, and flat shading was seen as an operation that would be done by fixed-function processing outside the GLSL shaders. However, GLSL has added a keyword `flat` to the GLSL language, defining a variable type called *flat out variables*. These variables may be passed to a fragment shader and call for the variable's value not to be interpolated across a graphics primitive during fragment processing. Our familiar teapot is shown in Figure 6.3 with flat shading, a look that may be familiar from your own beginning graphics work.

**Figure 6.3.** The familiar teapot with flat shading.

Vertex shaders that use `flat out` varying variables differ little from those you are already familiar with. An example vertex shader is shown below, which computes light intensity from the standard diffuse technique and passes this intensity to a fragment shader through the `flat out` variable `vLightIntensity`. Compare this with the vertex shader you saw early in the book to create Figure 2.2.

```
uniform vec3 uLightPos;

flat out float vLightIntensity;

void main( )
{
  vec3 transNorm  = normalize( uNormalMatrix * aNormal );
  vec3 ECposition = ( uModelViewMatrix * aVertex ).xyz;
  vLightIntensity = dot(normalize(uLightPos-
                                  ECposition),transNorm);
  vLightIntensity = abs(vLightIntensity);
  gl_Position     = uModelViewProjectionMatrix * aVertex;
}
```

## Smooth (Gouraud) Shading

Smooth shading is another kind of per-vertex color computation. In order to use smooth shading (also known as *Gouraud shading*) for a graphics primitive, the vertex shader must determine a color for each vertex as above and pass that color as an `out` variable to the fragment processor. The color can be determined from the ADS lighting model by using the function we gave earlier in this chapter, or it can simply be defined in an application through a color attribute variable. Because the color is passed to the fragment shader as an `in` varying variable, it is interpolated across the fragments that make up the primitive, thus giving the needed smooth shading. Below, we see a very simple vertex shader that computes the `out` variable `vColor` using the `ADSLightModel` function and makes it available to a fragment shader. Figure 6.4 shows the familiar teapot with Gouraud shading; it is clear that this is the smooth shading we are used to seeing in fixed-function shading.

**Figure 6.4.** The familiar teapot with smooth (Gouraud) shading.

```
out vec3 vColor;

// use vec3 ADSLightModel here

void main( )
{
  vec3 transNorm = normalize( uNormalMatrix * aNormal );
  vec3 ECpos     = ( uModelViewMatrix * aVertex ).xyz;

  vColor         = ADSLightModel( transNorm, ECpos );

  gl_Position    = uModelViewProjectionMatrix * aVertex;
}
```

The specular highlight in Gouraud-shaded figures are often not smooth, but show the typical smooth-shading effect of differing interpolations across neighboring primitives that leads to Mach banding on polygon edges. We will see much better results in the next section when we develop Phong shading.

## Phong Shading

*Phong shading* is a per-fragment color computation, and is a capability missing from the fixed-function OpenGL system. In true Phong shading, the vertex normals are interpolated across a graphics primitive, and the ADS lighting model is applied separately at each individual pixel. In order to do that, the lighting model's key variables must be evaluated and set up as `out` variables during vertex processing. The vertex shader code below sets up the normal and position data for the ADS lighting model function in `out` variables, so that a fragment shader can interpolate these variables and use them in the `ADSLightModel( )` function to compute the color. The actual fragment shader that implements this lighting is shown in Chapter 8. In Figure 6.5, you can see the smooth specular highlight that you expect from Phong shading.

**Figure 6.5.** The familiar teapot with Phong shading.

```
out vec3 vNormal;
out vec3 vECpos;

void main( )
{
  vNormal     = normalize( uNormalMatrix * aNormal );
  vECpos      = ( uModelViewMatrix * aVertex ).xyz;
  gl_Position = uModelViewProjectionMatrix * aVertex;
}
```

This specular computation uses the unit reflection vector, $\hat{R}$, which changes with each pixel. An alternative approach computes the "half angle"—the vector $\hat{H}$ halfway between the light $\hat{L}$ and the eye $\hat{E}$ vectors—and uses the cosine of the angle $\Phi$ between $\hat{H}$ and the normal $\hat{N}$. If the angle $\Phi$ is zero, the cosine is 1 and the light is reflected directly to the eye. As the angle increases, the cosine decreases. Again, a power of that cosine is used to control the size of the specular highlight. So we could replace the specular term in the model by the expression

$$S = L_S * M_S * (\hat{N} \bullet \hat{H})^{SH}.$$

The half angle vector $\hat{H}$ is computed as the average of the unitized `L` and `E` vectors, which in GLSL is expressed as `normalize(L + E)`, and the term $(\hat{N} \bullet \hat{H})^{SH}$ that provides the shiny appearance of specular light is slightly differ-

**Figure 6.6.** Specular lighting with the half-angle formulation (left) and full-angle formulation (right).

ent from the similar term in the reflection vector formulation. In general, the half-angle formulation for specularity gives a slightly less-focused specular highlight than the reflected-light version. Since the shininess coefficient *SH* is simply an approximation that is adjusted for visual effect anyway, the difference is only qualitative. You can see this qualitative difference in Figure 6.6, which shows the half-angle formulation on the left, and the full-angle formulation on the right.

In fact, it is sometimes possible to get even better shading than Phong shading. For some kinds of applications, it is possible to compute exact normals at each pixel instead of simply interpolating vertex normals. We call this *exact shading*, and we discuss it further in Chapter 8.

## Anisotropic Shading

*Anisotropic shading* is another per-pixel color computation that is not available in fixed-function OpenGL. Anisotropic shading is shading in which specular light is not reflected equally in all directions from the surface. An example of this is shown in Figure 6.7, which simulates a sphere for which light is reflected more strongly in a direction perpendicular to the arc from the poles through the point. Note that the bright spot in the figure is not circular because the material has different properties in different directions. Materials such as fur, hair, and brushed metal behave this way [22].

If you are writing shaders to implement anisotropic shading, the vertex shader must send the usual information, such as the normal, the eye position, and the light position, into the fragment shader, in the same way as would be

done for Phong shading. In addition, the fragment shader must get whatever extra information is needed to describe the directional reflection; in this case, that is the tangent vector to the sphere normal to the polar arc through the point. The fragment shader then carries out the ambient and diffuse light computations for regular ADS lighting and computes the specular part of the light based on the new light direction.

The particular kind of anisotropic shading shown in Figure 6.7 is a computer graphics "classic," going back to the late 1980s. The specular reflection is not given by the usual term

$$S = L_S * M_S * (\hat{R} \bullet \hat{E})^{SH},$$

but by the term

$$dl = \hat{T} \bullet \hat{L},$$
$$de = \hat{T} \bullet \hat{E},$$
$$S = L_S * M_S * (dl * de + (\sqrt{1 - dl * dl}) * (\sqrt{1 - de * de}))^{SH},$$

where $\hat{T}$ is the tangent vector (the direction of the brushing or hair), $\hat{L}$ is the light vector, $\hat{E}$ is the eye vector, and $SH$ is the shininess. In the code snippet below, taken from the fragment shader, the values of the tangent, light, and eye vectors, and the value of `vColor`, are assumed to have been computed separately in the associated vertex shader. The anisotropic shading parameters `uKa`, `uKd`, and `uKs` are assumed to be passed into the shader, and the color `vColor` is used for all three components of the ADS lighting model.

**Figure 6.7.** Anisotropic lighting in human hair (left); a sphere with anisotropic shading (right).

```
vec3 ambient = vColor.rgb;

float dl = dot( That, Lhat );
vec3 diffuse = sqrt( 1. - dl*dl ) * vColor.rgb;

float de = dot( That, Ehat );
vec3 spec = uLightColor * pow(dl * de +
     sqrt(1. - dl*dl) * sqrt(1. - de*de), uShininess);

fFragColor = vec4( uKa*ambient + uKd*diffuse + uKs*spe, 1. );
```

## Exercises

1. Compare the tradeoffs between granularity and shading quality, specifically between smooth and Phong shading. Create a model with a granularity you can adjust, and see if you can identify the granularity of smooth shading that is indistinguishable from Phong shading.
2. In the text, we say that the specular light computation using the reflection vector gives you a smaller specular highlight than the computation using the half-angle vector when the same specularity exponent is used. Modify the ADS lighting function in the text to use the half-angle formulation, and verify this statement. Add a slider for the shininess exponent to the GLIB file for the Phong shader, and see if you can quantify the relation between the exponents for the two formulations that give the same look.
3. Modify the ADS light function to use homogeneous vector computations throughout. Is this enough to make it work with directional as well as positional lights? If not, modify it further to support directional lights.
4. In the spotlight example in the text, we simply used ambient and diffuse light. Modify this shader to use the ADS light function and compute specular light as well.
5. Suppose that you had a material that reflected light from a sphere differently from the anisotropic example above: the light is reflected in a direction tangent to the sphere toward the poles. Write a shader to implement this kind of lighting.