

Lecture on Light, Material and the Phong Model

Reading: Most of what I know about Light, Material and the Phong Model, I learned from Angel, chapter 6, and the Red Book (chapter 5 in the 3rd edition, chapter 6 in the 1st edition). Some of the figures in this reading are drawn from Angel.

1 Lighting Models

You'll notice that when we color objects directly using RGB, there is no shading or other realistic effects. They're just "cartoon" objects. In fact, since there is no shading, it's impossible to see where two faces meet unless they are different colors.

Lighting models are a replacement for "direct color" (where we directly specify what color something is using RGB). Instead, the actual RGB values are *computed* based on properties of the object, the lights in the scene, and so forth.

There are several kinds of lighting models used in Computer Graphics, and within those kinds, there are many algorithms. Let's first lay out the landscape, and then explore what's available in OpenGL. The two primary categories of lighting are

- Global: take into account properties of the whole scene
- Local: take into account only
 - material
 - surface geometry
 - lights (location, kind and color)

Global lighting models take into account interactions of light with objects in the room. For example:

- light will bounce off one object and onto another, lighting it.
- objects may block light from a source
- shadows may be cast
- reflections may be cast
- diffraction may occur

Global lighting algorithms fall into two basic categories:

- Ray-tracing: conceptually, the algorithm traces a ray from the light source onto an object in the scene, where it bounces onto something else, to something else, . . . , until it finally hits the eye.

Often the ray of light will split, particularly at clear surfaces, such as glass or water, so you have to trace two light rays from then on.

Most rays of light won't intersect the eye. For efficiency, then, algorithms may trace the rays backwards, from the eye into the scene, back towards light sources (either lights or lit objects). Figure 1 illustrates this.

- Radiosity: any surface that is not completely black is treated as a light source, as if it glows. Of course, the color that it emits depends on the color of light that falls on it. The light falling on the surface is determined by direct lighting from the light sources in the scene and also indirect lighting from the other objects in the scene. Thus, every object's color is determined by every other object's color.

You can see the dilemma: how can you determine what an object's color is if it depends on another object whose color is determined by the first object's color? How to escape?

Radiosity algorithms typically work by iterative improvement (successive approximation): first handling direct lighting, then primary effects (other objects' direct lighting color), then secondary effects (other objects' indirect lighting color) and so on, until there is no more change.

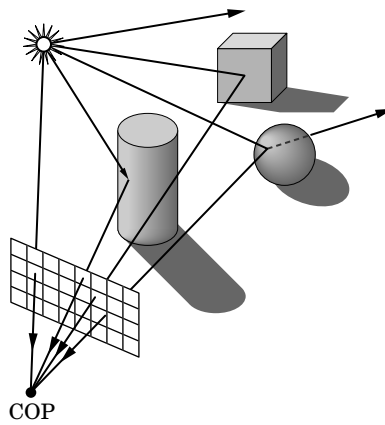


Figure 1: This figure illustrates how ray tracing works, tracing light rays back into the scene.

Global lighting models are very expensive to compute. According to Tony DeRose, rendering a single frame of the Pixar movie *Finding Nemo* takes *four hours*. For *The Incredibles*, the new Pixar movie, rendering each frame takes *ten hours*, which means that the algorithms have gotten more expensive even though the hardware is speeding up.

Local lighting models are perfect for a pipeline architecture like OpenGL's, because very little information is taken into account in choosing the RGB. This enhances speed at the price of quality. To determine the color of a polygon, we need the following information:

- material: what kind of stuff is the object made out of? Blue silk is different from blue jeans. Blue jeans are different from black jeans.
- surface geometry: is the surface curved? how is it oriented? What direction is it facing? How would we even define the direction that a curved surface is facing?
- lights: what lights are in the scene? Are they colored? How bright are they? What directions does the light go?

The rest of this document describes local lighting models, as in OpenGL. The mathematical lighting model that is used by OpenGL, and which we'll be developing, is called the "Phong model." We're going to proceed in a "bottom-up" fashion, first explaining the conceptual building blocks (section 2), before we see how they all fit together (section 5).

2 Local Lighting

To see a demo of what we'll be able to accomplish with material and lighting, run the lit teddy bear demo:

`~cs307/public_html/demos/material-and-lighting/TeddyBearLit`

Once the program is running, right-click to get a menu, and then choose "enable lighting." You can also choose "show lights." By turning the lighting on and off, you can see the effects of light and material versus direct color.

2.1 Material Types

Because local lighting is focussed on speed, a great many simplifications are made. Many of these may seem simplistic or even bizarre.

The first thing is to say that there are only three ways that light can reflect off a surface. Figure 2 illustrates this.

- Diffuse: These are rough surfaces, where an incoming ray of light scatters in all directions. The result is that direction that the material is viewed doesn't matter much in determining its color and intensity. Examples:

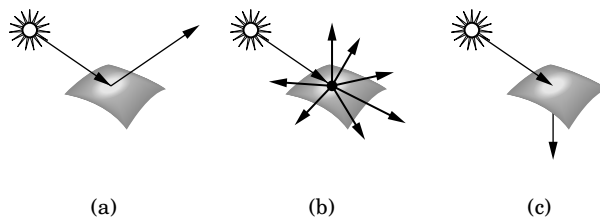


Figure 2: Ways that a light ray can interact with a surface.

- carpet, cloth
- dirt, rock
- dry grass

Look at the lit bear from different angles and you'll see.

- Specular: These are smooth, shiny surfaces, where an incoming ray of light might bounce, mirror-like, and proceed on. The result is that, if the camera is lined up with the reflected rays, we'll see a bright spot caused by that reflection. This is called a specular highlight. Examples:
 - plastic
 - metal
 - polished leather

Look at the lit bear's eyes, and you'll see a specular highlight.

- Translucent: These are surfaces that transmit as well as reflect light. These can really only be handled properly using ray tracing. Local lighting can do transparency after a fashion. However, we will not be talking more about transparency in this course. Examples:
 - water
 - glass

So, we really only need to understand *specular* and *diffuse* surfaces.

3 Kinds of Light

In talking about kinds of material, we divided them into diffuse and specular (and translucent, but we're not going to talk about that). Of course, most materials have some of each: you get color from the diffuse properties of, say, leather, but a shine of specular highlight at the right angle.

A major part of the Phong light model, then, is light interacting with these two properties of material. The model, therefore, divides *light* into different kinds, so that the *diffuse* light interacts with the *diffuse* material property and the *specular* light interacts with the *specular* material property.

The three kinds of light are:

- ambient
- diffuse
- specular

As we just said, the diffuse and specular light components interact with the corresponding material properties.

What is “ambient” light? As you might guess from the name, it’s the light all around us. In most real-world scenes, there is lots of light around, bouncing off objects and so forth. We can call this “ambient” light: light that comes from no where in particular. Thus, ambient light is *indirect* and *non-directional*. It’s the local-lighting equivalent of “radiosity.”

Even though in local lighting, we don’t trace ambient light rays back to a specific light source, there is still a connection. This is because, in the real world, when you turn on a light in a room, the whole room becomes a bit brighter. Thus, each OpenGL light source can add a bit of ambient light to the scene, brightening every object.

That ambient light interacts with the “ambient” property of a material. Because of the way it’s used, a material’s ambient property is often exactly the same color as the diffuse property, but they need not be.

Thus, each material also has the three properties: ambient, diffuse, and specular. We’ll get into the exact mathematics later, but for now, you can think of these properties as *colors*. For example, the ambient property of brown leather is, well, brown, so that when white ambient light falls on it, the leather looks brown. Similarly, the diffuse property is brown. The specular property of the leather is probably gray (colorless), because when white specular light reflects off shiny leather, the reflected light is colorless, not brown.

4 Light Sources

In OpenGL, we can have global ambient light, plus up to 8 particular light sources, each of which can be one of three different types. These light sources are not accurate models of the real world, but they are reasonable approximations.

4.1 Global Ambient Light

As we said above, ambient light is generalized, non-directional light that illuminates all objects equally, regardless of their physical or geometrical relationship to any light source.

In OpenGL, you can specify a “global ambient” value. The default is 0.3. That’s as if there were a uniform gray light of (0.3,0.3,0.3) falling on every object. The following code reduces that to 0.2:

```
GLfloat global_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient);
```

Note that the function `glLightModelfv` ends in a “v”: this means that the argument is a *vector* (that is, an array). In this case, it is a four-place vector of RGBA values. RGBA? We’ll talk about RGBA below in section 6.

In addition, each of the eight light sources (see the following sections), can have its own ambient value. When the light source is enabled, that ambient light is added to the global ambient light, brightening the scene.

4.2 Point Sources

The most common source of light in OpenGL is a “point source.” You can think of it as a small light bulb, radiating light in all directions around it. This is somewhat unrealistic, but it’s easy to compute with.

Note that `GL_LIGHT0` is one of the eight OpenGL light sources. They are named `GL_LIGHT0` through `GL_LIGHT7`. If we wanted to make `GL_LIGHT0` be a dim red light (suitable for the red light district in Amsterdam?) we would do the following:

```
GLfloat dimRed = { 0.5, 0, 0, 0 };
glLightfv(GL_LIGHT0, GL_AMBIENT, dimRed );
glLightfv(GL_LIGHT0, GL_DIFFUSE, dimRed );
glLightfv(GL_LIGHT0, GL_SPECULAR, dimRed );
```

Our light needs a location, too, which is a point. (That’s the definition of being a point source.) To make `GL_LIGHT0` be at location (10,20,30), we would do the following:

```
GLfloat light0_place = { 10, 20, 30, 1 };
glLightfv(GL_LIGHT0, GL_POSITION, light0_place);
```

(The 1 is the w component of the homogeneous coordinates for the light's position.)

The intensity of light “attenuates” (falls off) with the square of the distance. This is because the area of the surface of a sphere is proportional to the square of the radius, so the photons are spread out over a wider area. For example, if the Earth is twice as far from the sun as Venus, it would get 1/4th as much solar energy as Venus gets. (Actually, the earth is about 1.4 times as far from the sun, so it would get about half the solar energy: $1/(1.4^2) = 0.51$.)

This is just physics. Thus, that equation is incorporated into OpenGL:

$$I(p, l) = \frac{1}{d^2} I(l)$$

The intensity at a point p from the light source l is the intensity of the light source divided by the square of the distance between p and l .

However, the inverse square law can be harsh, with the light falling off “too fast.” Therefore, it's often softened like:

$$I(p, l) = \frac{1}{a + bd + cd^2} I(l)$$

where a , b and c are parameters that the OpenGL programmer can control:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, b);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, c);
```

If you're physics-minded, you'll object that this isn't realistic, and you'd be right, but it's one of those approximations that are made in local lighting.

If we wanted `GL_LIGHT0` to have no attenuation (which is the default), we would do:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 1);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0);
```

4.3 Spotlights

A spotlight is just a point source limited by a cone of angle θ . (Think of those “Luxo” lights, made famous by Pixar.) Intensity is greatest in the direction that the spotlight is pointing and trailing off towards the sides, but dropping to zero when you reach the cutoff angle θ . One way to implement that is let α be the angle between the spotlight direction and the direction that we're interested in. When $\alpha = 0$, the intensity is at its maximum ($I = I_0$). The cosine function has that property, so we use it as a building block. To give us additional control of the speed at which the intensity drops off (therefore, how concentrated the spotlight is), we allow the user to choose an exponent, e , for the cosine function. The resulting function is:

$$I = \begin{cases} I_0 \cos^e(\alpha) & \text{if } \alpha < \theta \\ 0 & \text{otherwise} \end{cases}$$

To do this in OpenGL, we use the following functions:

```
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, vector);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, angle);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, exp);
```

To have `GL_LIGHT0` pointing downwards and to the left and in a cone of angle 90 degrees (like a Luxo lamp) with an exponent of 2, we would have to do the following *in addition* to all the other information:

```
GLfloat spotDir = { -1, -1, 0, 0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spotDir );
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 2);
```

Remember that some of these parameters need arrays (and therefore use `glLightfv`) and others need scalars (and therefore use `glLightf`).

Figure 3 plots some of these functions for us.

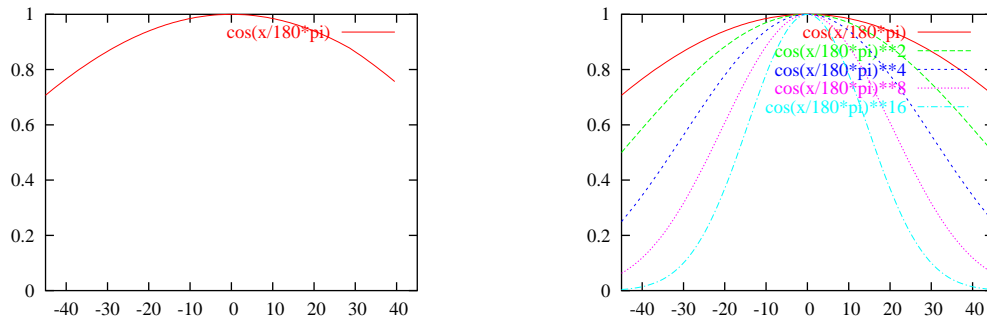
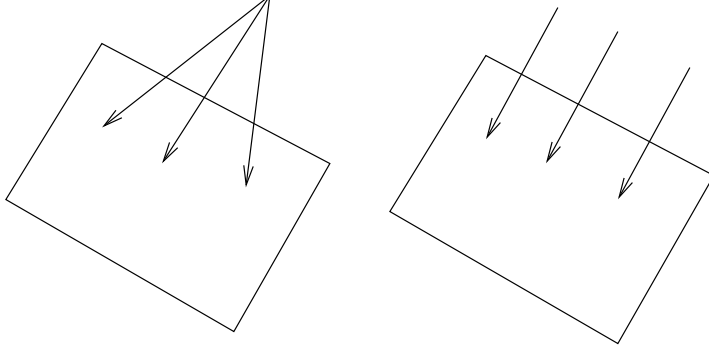


Figure 3: Spotlight intensity curves

4.4 Distant Lights

If a light source is nearby, the vector from the surface point to the light source changes from place to place. Since that vector is important in the lighting computation (the Phong model), having a nearby light means that the calculation must be re-done for each point on the surface. However, if the light is *distant*, the vector doesn't change, and so the computation can be done only once.



Therefore, the main idea is to speed up the computation by avoiding having to recompute the angle with light for each point on the object. Light comes from “infinity” from a particular direction. For example, any outdoor scene with sunlight (or even moonlight) would benefit from this.

In OpenGL, this is done by giving the position of the light in homogeneous coordinates: distant lights are vectors (last component is zero) and near lights are points (last component is unity).

4.5 Color Sources

So far, we've been only talking about the intensity of light as a function of light distance, angle and so forth. Intensity is a scalar (one-dimensional) quantity ranging from zero to one, so at this point we're really talking about black-and-white images.

To handle color, we will treat each of the three primary colors the same way, so we will just have *scalar* equations (for now). The actual color of a light or an object is just the intensity of each of the three primary colors:

$$\mathbf{I} = \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix}$$

5 Phong Model

Phong's model combines all of the above into one big, hairy equation. It's a reasonable approximation of the real physics that can be computed in a reasonable amount of time. In the section, we'll work top-down, attacking it one

piece at a time.

First, we need some notational building blocks:

- **n**: the normal vector for the surface. In this context, “normal” means *perpendicular* (you’ll also sometimes hear the term *orthogonal*), so the normal vector is a vector that is perpendicular to the surface. The normal vector is how we define the “orientation” of a surface — the direction it’s “facing.” The normal vector is the same over a whole plane, but may change over each point on a curved surface.
- **l**: the vector towards the light source; that is, a vector from the point on the surface to the light source. (Not used for ambient light.) Doesn’t change from point to point for distant lights.
- **v**: the vector towards the viewer; that is, the Center of Projection (COP). If **v** says that the surface faces away from the view, the surface is invisible and OpenGL can skip the calculation.
- **r**: the reflection direction of the light. If the surface at that point were a shiny plane, like a mirror, **r** is the direction that **l** would bounce to.

Figure 4 illustrates these vectors.

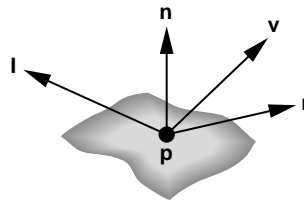


Figure 4: The vectors of the Phong model

5.1 Cosine

In the handout on geometry, we learned how simple it can be to find the cosine of the angle between two vectors, especially if they are normalized.

Assume these are normalized, so that we can compute cosines by a simple dot product: three multiplies and two adds.

5.2 Kinds of Light

Now we do something weird: we break light into

- three colors of light: (red, green and blue) (RGB)
- three *kinds* of light: (ambient, diffuse, specular) (ADS)

Thus, we have $3 \times 3 = 9$ different light intensities to worry about. We can throw them into a matrix:

$$\begin{bmatrix} L_{ra} & L_{ga} & L_{ba} \\ L_{rd} & L_{gd} & L_{bd} \\ L_{rs} & L_{gs} & L_{bs} \end{bmatrix}$$

But we’re not actually going to do any matrix equations, so that’s not the issue. We can treat each color of light the same way, so let’s just drop that subscript.

So, the L values here are the intensity of the light. Turn ’em up and the light gets brighter.

We also have to worry about how much of the incoming light gets reflected. Let this be a number called R . This number is a fraction, so if $R = 0.8$, that means that 80 percent of the incoming light is reflected. (We actually have 9 such numbers, such as the reflection fraction for specular red, ambient green, and so on for all 9 combinations.)

As we discussed earlier, in general, R can depend on:

- material properties: cotton is different from leather
- orientation of the surface
- direction of the light source
- distance of the light source

Of course, R is really a *function* of those four parameters.

The light that gets reflected is the product of the incoming light intensity, L , and the fraction R :

$$I = LR$$

That is, the intensity of light that is reflected (and ends up on the image plane and the framebuffer) is the incoming light intensity multiplied by the reflection number.

Note that the previous equation is just shorthand for:

$$I = L_{ra}R_{ra} + L_{ga}R_{ga} + L_{ba}R_{ba} + L_{rd}R_{rd} + L_{gd}R_{gd} + L_{bd}R_{bd} + L_{rs}R_{rs} + L_{gs}R_{gs} + L_{bs}R_{bs}$$

And that's just for one light! If we have multiple lights, we have (returning to our shorthand):

$$I = \sum_i L_i R_i$$

This leads to the problem of *overdriving* the lighting, where every material turns white because there's so much light falling on it. This happens sometimes in practice: you have a decently lit scene, and you add another light, and then you have to turn down your original lights (and your ambient) to get the balance right.

Why does R depend on i ? That is, why does the reflection fraction depend on which light we're talking about? Because the direction and distance change.

But since all the light sources work the same way, we're not going to worry about i and we'll just have

$$I = L_a R_a + L_d R_d + L_s R_s \quad (1)$$

That is, the intensity of the light coming from an object is

- the ambient light falling on it, multiplied by the reflection amount for ambient light, plus
- the diffuse light falling on it, multiplied by the reflection amount for diffuse light, plus
- the specular light falling on it, multiplied by the reflection amount for specular light

Equation 1 is our abstract Phong model. Now, let's see how to compute the three R values.

5.3 Ambient

Reflection of ambient light obviously doesn't depend on direction or distance or orientation, so it's solely based on the material property: is the material dark or light? Note that it can be dark for blue and light for red and green. If white light falls on such a material, what does it look like? So, R_a is a simple constant, which we will call k_a , just to remind ourselves that it's a constant:

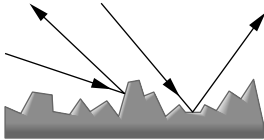
$$R_a = k_a \quad (2)$$

Note that $0 \leq k_a \leq 1$. Why?

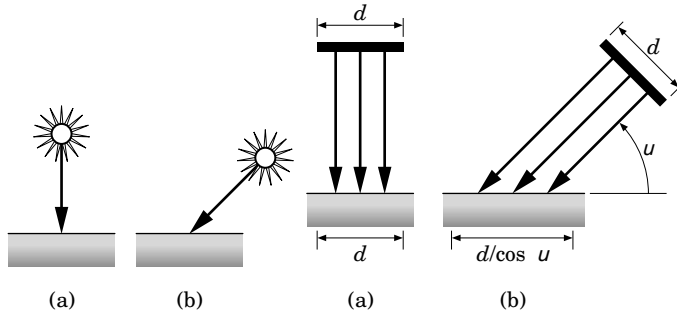
This k_a constant is chosen by the OpenGL programmer as part of the material properties for an object, in the same way that you choose color. There are actually three such values, one each for red, green, and blue.

5.4 Diffuse

Diffuse reflection is also called Lambertian, for the developer. For diffuse (matte) surfaces, we assume that light scatters in all directions. If we looked at it up close, we might see:



However, the angle of the light does matter, because the energy (photons) are spread over a larger area:



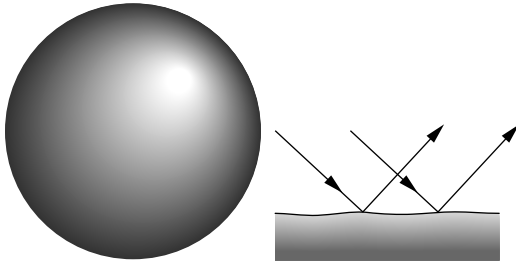
Consequently, we have

$$R_d = k_d(\mathbf{l} \cdot \mathbf{n}) \quad (3)$$

That is, the amount of reflection from a diffuse surface is the product of a constant, chosen by the OpenGL programmer, that is multiplied by the cosine of the angle between \mathbf{l} and \mathbf{n} . (As before, there are actually 3 such constants, one each for red, green and blue.)

5.5 Specular

Specular surfaces are somewhat mirror-like. Imagine the ball on the left in the following picture is a ping-pong ball or a billiard ball: shiny and smooth. The light rays bounce off, and some bounce right to our eyes.



Doing specularly “right” is hard. Again, Phong’s model is a compromise. We assume that the material is “smooth” in the vicinity of the point and that a bunch of light is bouncing in direction \mathbf{r} . If the direction of our view, \mathbf{v} , is near \mathbf{r} , we should get a bunch of that reflected light. Hence:

$$R_s = k_s(\mathbf{r} \cdot \mathbf{v})^e \quad (4)$$

As usual, $0 \leq k_s \leq 1$. The dot product is large when the two vectors are “lined up.” The e exponent is a number that gives the shininess. The higher the shininess, the smaller the spotlight. OpenGL allows $0 \leq e \leq 128$. In addition to e , the OpenGL programmer gets to choose k_s for each of red, green and blue.

5.6 The Phong Model

All together now, add up equations (2), (3), and (4) to get:

$$I = k_a L_a + k_d L_d \mathbf{n} \cdot \mathbf{l} + k_s L_s (\mathbf{r} \cdot \mathbf{v})^e \quad (5)$$

Compare that to equation (1).

To account for distance, which applies to diffuse, specular and ambient (for any particular light source, but not the global ambient), we invent three more constants and use

$$I = \frac{1}{a + bd + cd^2} (k_d L_d \mathbf{n} \cdot \mathbf{l} + k_s L_s (\mathbf{r} \cdot \mathbf{v})^e + k_a L_a) \quad (6)$$

Note that the d in the subscripts of the numerator is for *diffuse*, while the d in the denominator is for *distance*.

6 Transparency

In addition to RGB (red, green, and blue), OpenGL allows a computation on a fourth component, called *alpha* (α), but usually notated A. Thus, a light or a color is an RGBA value. An alpha value of 1 is opaque. An alpha value of 0 is completely transparent, and so the color doesn't matter. Intermediate values mix some color in, but allow others to show through. We'll talk more about this in a later class, devoted to transparency. For now, we'll keep the alpha value to 1. (You see this, for example, in the code for setting the global ambient in section 4.1.)

7 Light and Material in OpenGL

Have you been keeping count of how many parameters there are to control? Let's summarize:

- For the scene as a whole:
 - 3 global ambient primaries
 - 3 attenuation constants
- for each material
 - 3 ambient constants
 - 3 diffuse constants
 - 3 specular constants
 - 1 shininess constant
- for each surface
 - position in 3D
 - orientation (surface normal) in 3D
- for each light
 - 4 position coordinates (homogeneous coordinates)
 - 3 ambient constants
 - 3 diffuse constants
 - 3 specular constants
 - optional direction vector, cutoff angle and concentration exponent

That's a huge number of parameters to play with! It's actually even worse, because OpenGL allows you to play with "transparency" of a color, so color is actually specified in a four-dimensional space: RGBA. The A dimension (sometimes called alpha or α) is the *opacity* of the color, where 1 means completely opaque and 0 means perfectly transparent. We will talk about transparency in a few weeks. For now, I suggest you make your life a little easier and ignore that. However, you will see it in one of the tutors, so I wanted to warn you.

Now let's look at how this actually works in OpenGL. First, please look at two "tutors" for light and material.

- `~cs307/pub/Tutors/lightmaterial`

- `~cs307/pub/demos/materialTutor`

The first tutor uses plain OpenGL functions. I suggest you “cd” to the directory, and run it there, because there are datafiles that it wants to access. This tutor lets you select a variety of figures (right-click in the upper left sub-window), but they are all made of just one material. For that material, you can control 17 parameters.

- 4 ambient constants. These are the constants multiplied by the RGBA intensity of incoming light. They are also multiplied by the global ambient.
- 4 diffuse constants. These are analogous to the ambient, except there’s no “global diffuse”
- 4 specular constants. Similar to ambient.
- 4 emission constants. I haven’t talked about these. You can think of these as constants that make the object glow. The default emission is zero, so you can safely ignore these, which I recommend.
- 1 shininess parameter, which is the exponent in the Phong equation.

You can also control 16 parameters for the one light source:

- 4 for position. Recall that we use homogeneous coordinates, so you can make a distant light by setting the W coordinate to zero.
- 4 for the ambient light. Each light produces ambient light (its contribution to the total ambient light?). Strange, but true. Even weirder, the light color is specified as RGBA, so the light can be “transparent,” whatever the heck that means.
- 4 for diffuse light. These get multiplied by the diffuse reflection constants.
- 4 for specular light. Same comments as for diffuse.

Wow. That’s a lot. Try some of the following:

- try moving the light around. Try putting it in the center of the dolphins. Can you see the effects, particularly on the specular highlight?
- try making a distant light and flipping it up and down.
- try playing with the different properties (ambient, diffuse and specular) and the colors.
- try different shininesses

The TW tutor is a bit different, hopefully a bit simpler. Instead of 17 parameters for a single material, TW’s API allows only 5, for the following reasons:

- the ambient triple and diffuse triple are often the same, and based on the intrinsic “color” of the material. In fact, there’s an OpenGL call that allows you to specify them both at once, using a single quadruple of values.
- the specular term is usually gray (all three RGB components have the same value), since the material is acting as a mirror at that point.

This reduces the 17 parameters to 5. TW allows you to specify just those five, without having to build as many matrices and send them to OpenGL:

```
twColor(Triple ambient_and_diffuse, specular, shininess);
```

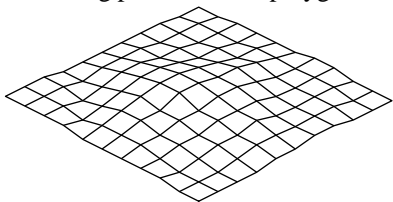
Furthermore, the global ambient is often gray light, so TW allows you to set that using a single function call:

```
twAmbient(value);
```

This API is what the `materialTutor` lets you experiment with. It doesn’t offer a choice of objects, but it does let you “save” a particular set of parameters, so you can do a side-by-side comparison of a particular change.

8 Polygonal Shading

Polygons are easy to shade (compared to curved surfaces) because they're flat over a region (hopefully several pixels). We'll look at three ways to shade a polygon; OpenGL allows two of them. For curved surfaces, we think in terms of representing them as a mesh of polygons, thereby reducing the problem of shading them into that of shading polygons. The following picture is of a polygon mesh.



For additional demos, you can look at the following:

`~cs307/pub/demos/08-TeddyBearLit`

`~cs307/pub/demos/09-JewelAndBall`

The first is one we've looked at before, but as we know, the spheres of the teddy bear are implemented with polyhedra approximation, depending on slices and stacks. You can try modifying those to see the effects on shading.

The second demo contrasts a faceted jewel with a smooth ball, but the remarkable thing is that the objects are identical except for the *shade model*. OpenGL allows two kinds of shading:

- Flat shading, exemplified by the jewel
- Smooth shading, exemplified by the ball

8.1 Flat Shading

To use flat shading in OpenGL, you use the following call. This sets a policy that is applied to all subsequent polygons.

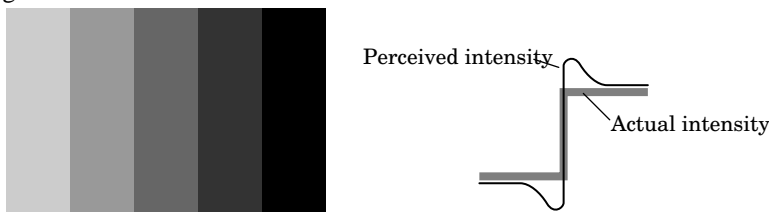
```
glShadeModel(GL_FLAT);
```

The shading assumptions with flat shading are the following:

- \mathbf{n} is constant: that is, the polygonal patch is flat, so the surface normal is the same over the entire patch.
- \mathbf{v} is constant if viewer is far. (We can set the near viewer flag to false, see below.)
- \mathbf{l} is constant if light is far (compared to the size of the polygon), or the light is directional
- \mathbf{r} is constant if \mathbf{l} is.

Thus, all pixels are the same shade, since all the relevant vectors are constant over the polygon. This is *flat* or *constant* shading. Adjoining polygons will have different shades if they have different values for any of the important vectors.

Fortunately or unfortunately, the human eye is remarkably sensitive to differences from one patch to the next: *lateral inhibition* causes *Mach bands*. Lateral inhibition has to do with how the neurons in the eye work, and the effect is that at boundaries where shade changes, the difference or contrast is *enhanced*. Lateral inhibition produces a kind of image enhancement!



The Mach Band effect is why it's so obvious when we use flat shading with the teddy bear. Thus, flat shading is appropriate when the actual object is faceted like a jewel, but if the polyhedron is just an approximation to a real object that is "smooth" like a ball. In cases like that, it's better to use smooth shading.

8.2 Gouraud Shading

Smooth shading is sometimes called “Gouraud” shading, in honor of one of the pioneers. To get smooth shading in OpenGL, use the following call:

```
glShadeModel(GL_SMOOTH);
```

Smooth shading assumes that the surface normal at each vertex is different, and so the shade of each vertex is computed (the \mathbf{l} and \mathbf{v} vectors might also be different). Then, the shade of all the interior pixels is computed by interpolation.

8.3 Phong Shading

A third form of shading, which isn’t available in OpenGL, is Phong Shading. It’s like smooth shading, but instead of interpolating the shade, the vectors are interpolated (particularly the surface normal). Then, the shade of each pixel is computed.

The result is smoother and nicer than Gouraud, but is more computation-intensive, so it’s done off-line.

8.4 Normals

Since all of these computations depend on normals, how do we determine the normal vector at a vertex in OpenGL?

The programmer defines the normal at a vertex using one of the following two calls:

```
glNormal3f(x, y, z);  
glNormal3fv(float*);
```

As with colors, you define the normal *before* you send the vertex down the pipeline. That normal applies to all subsequent vertices. For example, here’s a triangle in the (x, y) plane:

```
glBegin(GL_TRIANGLES);  
glNormal3f(0, 0, 1);  
glVertex3f(0, 0, 0);  
glVertex3f(7, 0, 0);  
glVertex3f(0, 5, 0);  
glEnd();
```

Note that the current transformation matrix (CTM) is applied to normals as well as vertices as they go down the pipeline, so you can transform the triangle above to anywhere in your scene, at any angle, and the normal goes along for the ride. This is really useful.

If you have a more difficult situation, you can always do the following:

- Choose three points on your surface (three points define a plane)
- Compute two vectors by subtracting those points (hint, use `twVector`).
- Compute the cross product of those two vectors (hint, use `twCrossProduct`).
- Normalize the cross product (hint, use `twVectorNormalize`)
- That normalized cross product is the surface normal of the plane, so you can give it to OpenGL using `glNormal3fv`.

The default normal is the vector $(0,0,1)$, which might be completely wrong. If so, your diffuse and specular terms will be nearly zero and the usual result is that the surface is *black*.

You can define a different normal each time you send a vertex down the pipeline. Specifically, if you send vertex V down the pipeline twice, it can have a different normal each time. Why might you do that? Figure 5 shows some of the vertices in the jewel and ball and their normal.

The glut objects, such as `glutSolidSphere`, define the normals for you; yet another reason they’re nice.

Note that OpenGL will usually assume that your normals are normalized. If they’re not, they’ll usually be longer than unit length, resulting in light values that are too high, and your surfaces become white. There are two solutions:

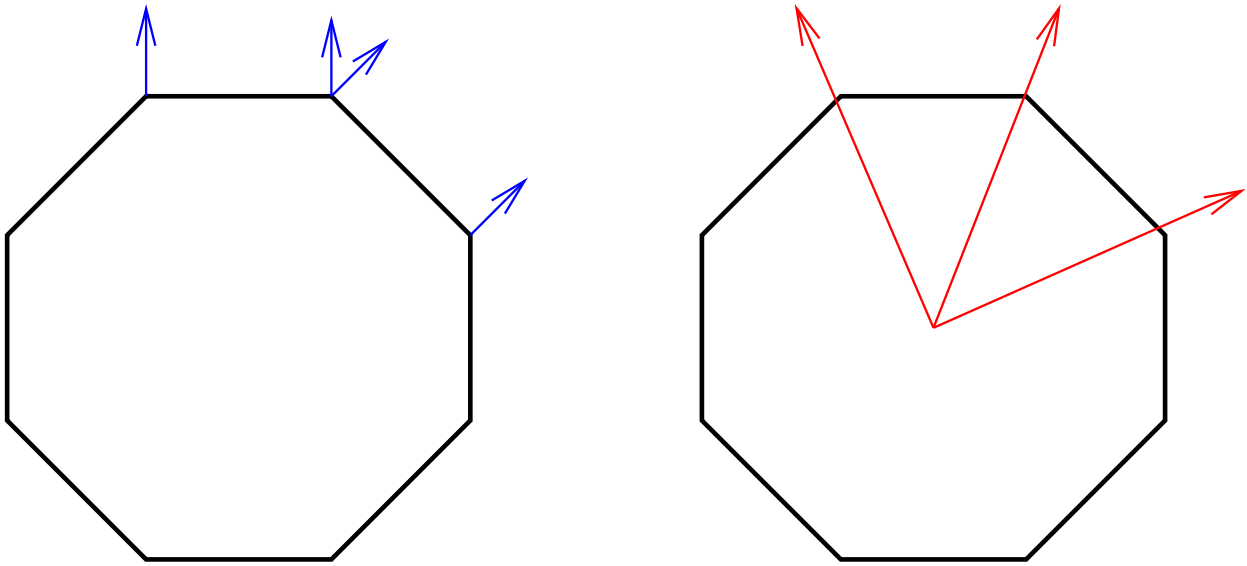


Figure 5: The normals on the jewel change depending on which facet is being send down the pipeline, but the normals for the ball don't.

- Normalize your vectors yourself, as needed
- Tell OpenGL to normalize all vectors automatically, using

```
glEnable(GL_NORMALIZE);
```

The former is more efficient, but less convenient. TW enables `GL_NORMALIZE`, so you don't have to worry about normalizing vectors when using TW.

9 Lighting and Light Sources

First, you must remember to enable lighting. There's so much else to do, it's easy to forget:

```
glEnable(GL_LIGHTING);
```

You can turn lighting off and on, as desired, but typically we set it up once and leave it on.

OpenGL allows all four types of light (ambient, point, spot and directional) and at least eight sources. We have to specify a ton of information, but it's organized by the Phong model. Notice that you have to enable the light as well as specifying a bunch of information about it.

9.1 Global Ambient

```
GLfloat global_ambient[] = {0.2, 0.2, 0.2, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient);
```

9.2 Point Lights

All the following are necessary for each point light. You can have up to at least eight lights: `GL_LIGHT0` through `GL_LIGHT7`. Some OpenGL implementations allow more. (Consult `GL_MAX_LIGHTS`; our implementation allows 8. You can see this with the 00-Limits demo.)

```

glEnable(GL_LIGHT0);
GLfloat light0_place[] = {1,2,3,1}; // x,y,z,w values
GLfloat light0_ambient[] = {1,0,0,1}; // rgba values
GLfloat light0_diffuse[] = {1,0,0,1}; // rgba values
GLfloat light0_specular[] = {1,1,1,1}; // rgba values
glLightfv(GL_LIGHT0, GL_POSITION, light0_place);
glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light0_specular);

```

9.3 Attenuation

We can add distance effects, specifying the *a*, *b* and *c* with one call each:

```

glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, b);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, c);

```

9.4 Spotlights

In addition to all the information for the point lights, you can do the following to set up a spotlight.

```

glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, vector);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, angle);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, exp);

```

OpenGL limits the spotlight exponent to the range [0,128] and the angle to between 0 and 90, with a special angle of 180. The cutoff is the half-angle angle at the top of the light. That is, it's a limit on the angle between the vector to the vertex and the spot direction.

9.5 TW Lights

OpenGL allows you to control all 9 parameters of a light, but in practice most lights are gray, so

```
twLight1(light_id, position, value);
```

sets the position and value (all 9 parameters are set to “value”) for a given light. This may be too restrictive; I’m open to suggestions.

9.6 Distant Viewer

Distant is the OpenGL default, since it’s faster. If we want a near viewer:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

10 Materials

Specifying material of a vertex is a bit like specifying a light, at least the way the function calls work. As with `glLightf()`, there are basically two functions:

```

glMaterialfv(face, parameter, values_vector);
glMaterialf(face, parameter, value);

```

The “face” is one of the following OpenGL constants:

- `GL_FRONT`

- GL_BACK
- GL_FRONT_AND_BACK

If one side of your polygon is a different color or material than the other, you can specify which side you are giving the color of.

How does OpenGL define which side of a polygon is “the front”? When viewed from the front, the vertices of a polygon are *counterclockwise*. You can control this with `glFrontFace`; see the man page of that function for more information.

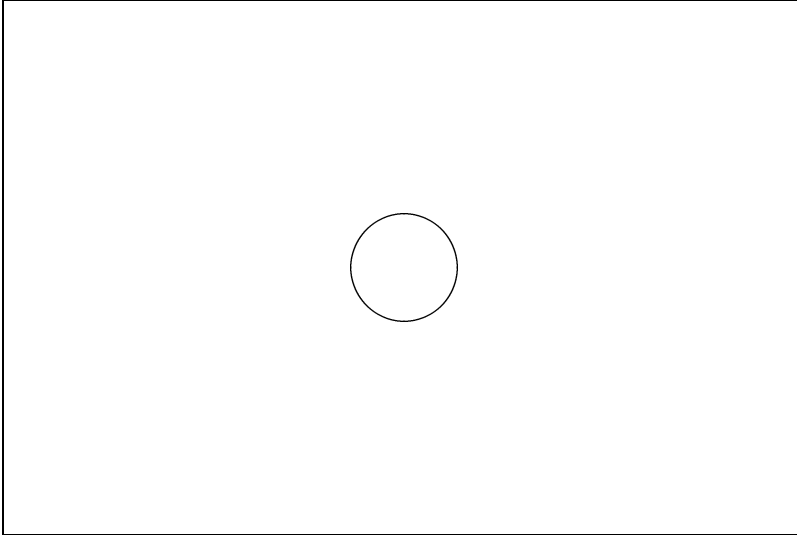
The second argument, called “parameter” is one of the following OpenGL constants.

- GL_AMBIENT
- GL_DIFFUSE
- GL_SPECULAR
- GL_AMBIENT_AND_DIFFUSE
- GL_SHININESS
- GL_EMISSION

You choose one of those based on the property you’d like to set. The value you set the property to the last argument. Some values need to be vectors; in fact, all of them except for shininess are RGBA arrays.

11 Shading Large Areas

We’ll play with the 09-Spotlight demo. This illustrates the important point that OpenGL only knows about *vertices*, so the play of light over a big polygon will be *disappointing*. Suppose you have a spotlight falling like:



All four vertices will be unlit, so any kind of interpolation will result in a rectangle that is completely black, despite the light falling on it. To solve this problem, we break the polygon up into lots of little polygons, in a mesh, and handle it as we have curved surfaces (the slices and stacks idea).

A convenience for this is `twDrawUnitSquare()`, which draws a unit square broken up into small rectangles; you choose how many in each direction.

11.1 Two Sided Planes

By default, the color of the back side of polygons is not computed (so they'll be black). If you want that calculation to be made, use the following OpenGL call:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```