Laboratory 4
Lights

Light

In the real world, light comes at us from all sides and with all colors and, when combined, can create the details and rich scenes of everyday life. OpenGL doesn't attempt to duplicate anything like the real-world lighting models, because those are very complex and time-consuming. But it can approximate it in a way that is certainly good enough for our purposes.

The lighting model used in OpenGL ES permits us to place several lights of varying types around our scene. We can switch them on or off at will, specifying direction, intensity, colors, and so on. But that's not all, because we also need to describe various properties of our model and how it interacts with the incoming light. *Lighting* defines the way light sources interact with objects and the materials those objects are created with. *Shading* specifically determines the coloring of the pixel based on the lighting and material. Notice that a white piece of paper will reflect light completely differently than a mirrored Christmas ornament. Taken together, these properties are bundled up into an object called a *material*. Blending the material's attributes and the light's attributes together generates the final scene.

The colors of OpenGL lights can consist of up to three different components:

- Diffuse
- Ambient
- Specular

Diffuse light can be said to come from one direction such as the sun or a flashlight. It hits an object and then scatters off in all directions, producing a pleasant soft quality. When a diffuse light hits a surface, the reflected amount is largely determined by the angle of incidence. It will be at its brightest when directly facing the light but drops as it tilts further and further away.

Ambient light is that which comes from no particular direction, having been reflected off all the surfaces that make up the environment. Look around the room you are in, and the light that is bouncing off the ceiling, walls, and your furniture all combine to form the ambient light. If you are a photographer, you know how important ambient lighting is to make a scene much more realistic than a single point source can, particularly in portrait photography where you would have a soft "fill light" to offset the brighter main light.

Specular light is that which is reflected off a shiny surface. It comes from a specific direction but bounces off a surface in a much more directed fashion. It makes the hot spot that we would see on a disco ball or a newly cleaned and waxed car. It is at its brightest when the viewpoint is directly in line with the source and falls off quickly as we move around the object.

When it comes to both diffuse and specular lighting, they are typically the same colors. But even though we are limited to the eight light objects, having different colors for each component actually means that a single OpenGL light can act like three different ones at the same time. Out of the three, you might consider having the ambient light be a different color, usually one that is opposing the main coloring so as to make the scene more visually interesting.

Note that you don't have to specify all three types for a given light. Diffuse usually works just fine in simple scenes.

Shading

The diffuse shading model gives a very smooth look to objects. It uses something called the *Lambert lighting model.* Lambert lighting states simply that the more directly aimed a specific face is to the light source, the brighter it will be. The ground beneath your feet is going to be brighter the higher the sun is in the sky. Or in the more precise technical version, the reflective light increases from 0 to 1 as the angle, Θ, between the incident light, I, and the face's normal, N, decrease from 90 to 0 degrees based on $\cos(\Theta)$. See Figure 1. Here's a quick thought experiment: when Θ is 90 degrees, it is coming from the side; $\cos(90°)$ is 0, so the reflected light along N is naturally going to be 0. When it is coming straight down, parallel to N, $\cos(0°)$ will be 1, so the maximum amount will be reflected back. And this can be more formally expressed as follows:

$$I_d = k_d I_i \cos(\Theta),$$

where I_d is the intensity of the diffuse reflection, I_i is the intensity of the incoming ray of light, and k_d represents the *diffuse reflectance* that is loosely coupled to the roughness of the object's material. *Loosely* means that in a lot of real-world materials, the actual surface may be somewhat polished but yet translucent, while the layers immediately underneath perform the scattering. Materials such as this may have both strong diffuse and specular components. Also, each color band may have its own k value in real life, so there would be one for red, green, and blue.

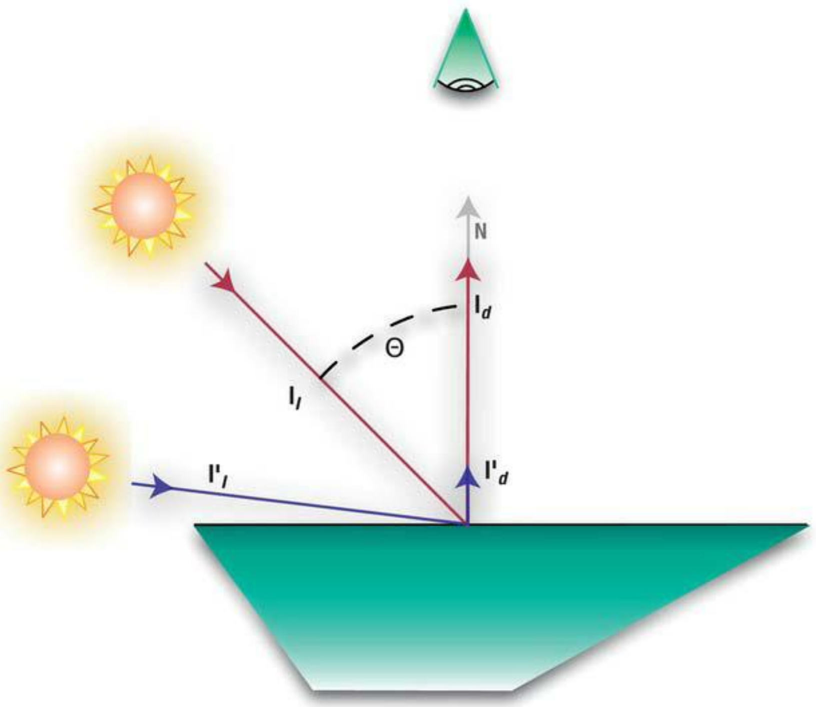

Figure 1: The reflected intensity of an incoming beam for a perfectly diffuse surface

Specular Reflections

Specular reflections serve to give your model a shiny appearance besides the more general diffuse surface. Few things are perfectly flat or perfectly shiny, and most lay somewhere in between. In fact, the earth's oceans are good specular reflectors, and on images of the earth from long distances, the sun's reflection can clearly be seen in the oceans.

Unlike a diffuse reflection, which is equal in all directions, a specular reflection is highly dependent on the viewer's angle. We have been taught that the *angle of incidence=angle of reflectance*. This is true enough for the perfect reflector. But with the exception of mirrors, few things are perfect reflectors. And as such, there will be a slight scattering of the incoming ray; see Figure 2.

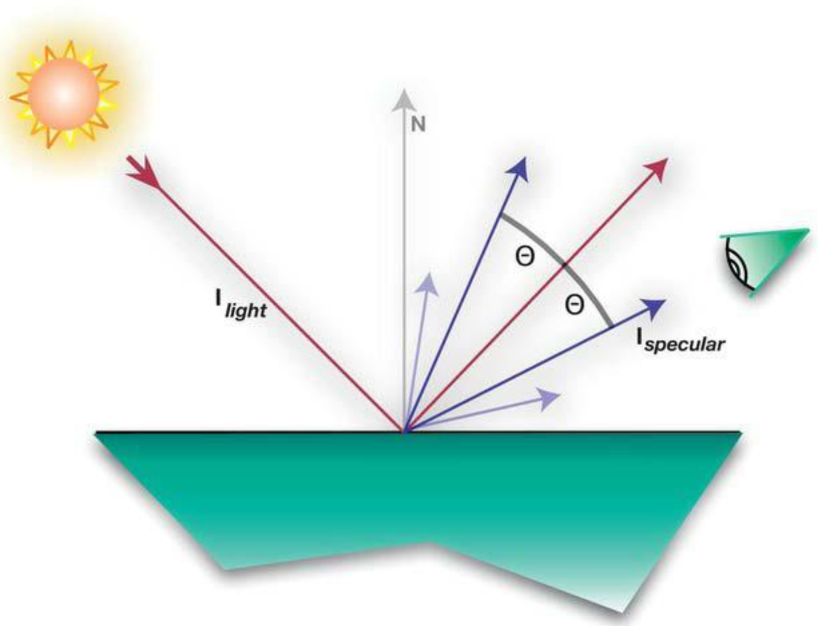

Figure 2: For a specular reflection, the incoming ray is scattered but only around the center of its reflected counterpart

The equation of the specular component can be expressed as follows:

$$I_{specular} = W(q) I_{light} \cos^n(\Theta),$$

where I_{light} is the intensity of the incoming ray, $W(q)$ is how reflective the surfaces is based on the angle of I_{light}, n is the *shininess factor*, Θ is the angle between the reflected ray and the ray hitting the eye.

This is actually based on what's called the *Fresnel Law of Reflection*, which is where the $W(q)$ value comes from. Although $W(q)$ is not directly used OpenGL ES 1.1 because it varies with the angle of incidence and is therefore a little more complicated than the specular lighting model, it could be used in a shader for version OpenGL ES 2.0. In that case, it would be particularly useful in doing reflections off the water, for example. In its place is a constant that is based on the specular values from the material setting.

The shininess factor n, also called the *specular exponent*, in real life can go far higher than the maximum of 128 in OpenGL ES 1.1.

Attenuation

In the real world, light decreases the farther an object is from the light source. OpenGL ES can model this factor using three kinds of attenuation: constant, linear, and quadratic. Linear attenuation can be used to model attenuation caused by factors such as fog. The quadratic attenuation models the natural falloff of light as the distance increases, which changes exponentially. As the distance of the light doubles, the illumination is cut to one quarter of the previous amount.

The total attenuation k_t is calculated as follows

$$k_t = \frac{1}{k_c + k_l d + k_q d^2},$$

where k_c is the constant, k_l is the linear value, k_q is the quadratic component, and d stands for the distance from the light and an arbitrary vertex.

Summing It All Up

So, now you can see that there are many factors in play to merely generate the color and intensity of that color for any of the vertices of any models in our scene. These include the following:

- Attenuation because of distance
- Diffuse lights and materials
- Specular lights and materials
- Spotlight parameters
- Ambient lights and materials
- Shininess
- Emissivity of the material

You can think of all of these as acting on the entire color vector or on each of the individual R, G, and B components of the colors. So, the final vertex color will be as follows:

$$color = ambient_{world\,model} ambient_{material} + emissive_{material} + intensity_{light},$$

where

$$intensity_{light} = \sum_{i=1}^{n} (attenuation\ factor)_i (spotlight\ factor)_i [ambient_{light} ambient_{material}$$

$$+ \cos(\Theta)^{shininess} specular_{light} specular_{material}]$$

In other words, the color is equal to the some of the things not controlled by the lights added to the intensity of all the lights once we take into account the attenuation, diffuse, specular, and spotlight elements.

When calculated, these values act individually on each of the R, G, and B components of the colors in question.

Normals

Once the lights are turned on, the predefined vertex colors are ignored. With that in mind, our model needs an extra layer of data to tell the system just how to light its facets, and that is done through an array of normals for each vertex.

What is a vertex normal? *Face* normals are normalized vectors that are *orthogonal* (perpendicular) to a plane or face. But in OpenGL, vertex normals are used instead because they provide for better shading down the road. It sounds odd that a vertex can have a "normal" of its own. After all, what is the "direction" of a vertex? It is actually quite simple conceptually, because vertex normals are merely the normalized sum of the normals of the faces adjacent to the vertex. See Figure 3.

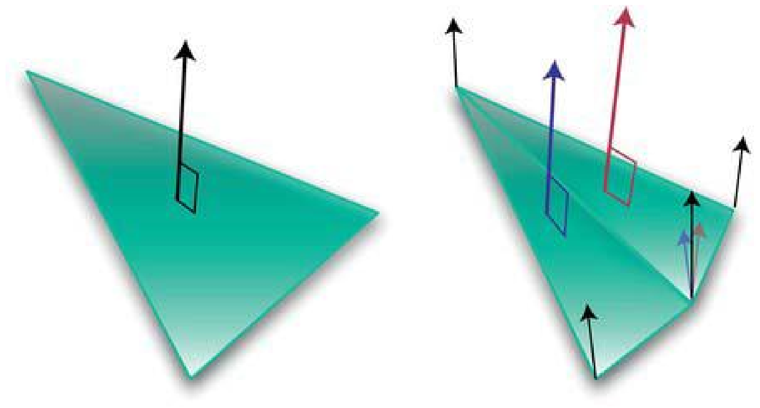

Figure 3: A face normal is illustrated on the right, while vertex normals for a triangle fan are on the left

OpenGL needs all of this information to tell what "direction" the vertex is aiming at so it can calculate just how much illumination is falling on it, if at all. It will be its brightest when aiming directly at the light source and dim as it starts tilting away. This means we need to modify our `Cube` class to create a normal array along with the vertex and color arrays.

First of all, we need to add the array of normals to the `Cube` constructor:

```
float[] normals =
{
    -1.0f/(float)Math.sqrt(3), 1.0f/(float)Math.sqrt(3), 1.0f/(float)Math.sqrt(3),
    1.0f/(float)Math.sqrt(3), 1.0f/(float)Math.sqrt(3), 1.0f/(float)Math.sqrt(3),
    1.0f/(float)Math.sqrt(3), -1.0f/(float)Math.sqrt(3), 1.0f/(float)Math.sqrt(3),
    -1.0f/(float)Math.sqrt(3), -1.0f/(float)Math.sqrt(3), 1.0f/(float)Math.sqrt(3),
    -1.0f/(float)Math.sqrt(3), 1.0f/(float)Math.sqrt(3), -1.0f/(float)Math.sqrt(3),
    1.0f/(float)Math.sqrt(3), 1.0f/(float)Math.sqrt(3), -1.0f/(float)Math.sqrt(3),
    1.0f/(float)Math.sqrt(3), -1.0f/(float)Math.sqrt(3), -1.0f/(float)Math.sqrt(3),
    -1.0f/(float)Math.sqrt(3), -1.0f/(float)Math.sqrt(3), -1.0f/(float)Math.sqrt(3)
};
```

and then set up the

```
private FloatBuffer mNormalBuffer;
```

class field in the usual way:

```
ByteBuffer nbb = ByteBuffer.allocateDirect(normals.length * 4);
nbb.order(ByteOrder.nativeOrder());
mNormalBuffer = nbb.asFloatBuffer();
mNormalBuffer.put(normals);
mNormalBuffer.position(0);
```

It doesn't look like the normal averaging scheme covered earlier, because, since we are dealing with a very simple symmetrical form of a cube, the normals are identical to the normalized values of the vertices (to ensure they are unit vectors — that is, of length 1.0). You will rarely need to actually generate your own normals. If you do any real work in OpenGL ES, you will likely be importing models from third-party applications, such as 3D-Studio or Strata. They will generate the normal arrays along with the others for you.

Now, in the `draw` method of the `Cube` class, add the following:

```
gl.glNormalPointer(GL10.GL_FLOAT, 0, mNormalBuffer);
gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);
```

which sends the normal data to the OpenGL pipeline alongside the color and vertex information.

Diffuse Lighting

Go back to the `onSurfaceCreated` method in the `BouncyCubeRenderer` and add the following:

```
gl.glDepthMask(false);
initLighting(gl);
```

Then, add the class field

```
public final static int SS_SUNLIGHT = GL10.GL_LIGHT0;
```

Now, create the `initLighting(GL10 gl)` method in the same `BouncyCubeRenderer` class.

The lighting components are in the standard RGBA normalized form. We will add a green diffuse light. The final value of alpha should be kept at 1.0 for now, because it will be covered in more detail later:

```
float[] green = {0.0f, 1.0f, 0.0f, 1.0f};
```

Next, we specify the position of the light:

```
float[] position = {0.0f, 5.0f, 0.0f, 1.0f};
```

We can now set the light's position and the diffuse component to the diffuse color. `glLightfv` is a new call and is used to set various light-related parameters. You can retrieve any of this data at a later time using `glGetLightfv`, which retrieves any of the parameters from a specific light.

```
gl.glLightfv(SS_SUNLIGHT, GL10.GL_POSITION, makeFloatBuffer(position));
gl.glLightfv(SS_SUNLIGHT, GL10.GL_DIFFUSE, makeFloatBuffer(green));
```

where we used the helper method defined below:

```
protected static FloatBuffer makeFloatBuffer(float[] array)
{
    ByteBuffer bb = ByteBuffer.allocateDirect(array.length*4);
    bb.order(ByteOrder.nativeOrder());
    FloatBuffer fb = bb.asFloatBuffer();
    fb.put(array);
    fb.position(0);
    return fb;
}
```

We next specify a shading model. `GL_FLAT` means that a face is a single solid color, while setting it to `GL_SMOOTH` will cause the colors to blend smoothly across the face and from face to face.

```
gl.glShadeModel(GL10.GL_SMOOTH);
```

And finally, we must tell the system we want to use lighting, and then enable the one light we have created.

```
gl.glEnable(GL10.GL_LIGHTING);
gl.glEnable(SS_SUNLIGHT);
```

Material

To add a diffuse red material to the `BouncyCube`, use the following line:

```
gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE, makeFloatBuffer(red));
```

where

```
float[] red = {1.0f, 0.0f, 0.0f, 1.0f};
```

If you run the project, you will see that, in this case, the cube is black. But why is it black? A red object looks red only when the lighting hitting it has a red component, precisely the way our green light doesn't. If you had a red balloon in a dark room and illuminated it with green light on it, it would look black, because no green would come back to you.

So, with this understanding, replace the red diffuse material with green. What should you get? Right, the green cube is illuminated again. But you may notice something really interesting. The green now looks a little bit brighter than before adding the material.

Let's do one more experiment. Let's make the diffuse light be a more traditional white. What should now happen with the green? Red? How about blue? Since the white light has all those components, the colored materials should all show up equally well. But if you see the black cube again, you changed the material's color, not the lighting.

Specular Lighting

Add the following line to the lights section:

```
gl.glLightfv(SS_SUNLIGHT,GL10.GL_SPECULAR, makeFloatBuffer(red));
```

To the material section, add this:

```
gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SPECULAR, makeFloatBuffer(red));
```

And change the light's position to the following:

```
float[] position = {10.0f,0.0f,3.0f,1.0f};
```

The first value to `glMaterialfv` must always be `GL_FRONT_AND_BACK`. In normal OpenGL, you are permitted to have different materials on both sides of a face, but not so in OpenGL ES. However, you still must use the front and back values in OpenGL ES, or materials will not work properly.

With the diffuse material green, you should see something that looks like a big mess of something yellowish-reddish. Shorthand for what's happening is that there's yet another value we can use to play with the lighting. Called *shininess*, it specifies just how shiny the object's surface is and ranges from 0 to 128. The higher the value, the more focused the reflection will be, and hence the shinier it appears. But since it defaults to 0, it spreads the specular light across the entire cube. It overpowers the green so much that when mixed with the red, it shows up as yellow. So, in order to get control of this, add this line:

```
gl.glMaterialf(GL10.GL_FRONT_AND_BACK,GL10.GL_SHININESS, 5);
```

We used the value of 5. Next try 25. Shininess values from 5 to 10 correspond roughly to plastics; values greater than that correspond to metals.

Ambient Lighting

Add the following line to `initLighting()`; then compile and run:

```
gl.glLightfv(SS_SUNLIGHT, GL10.GL_AMBIENT, makeFloatBuffer(blue));
```

Now, add the following line:

```
gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT, makeFloatBuffer(blue));
```

Besides the ambient attribute for each light, you can also set a world ambient value. The light-based values are variables, as are all of light parameters, so they vary as a function of the distance, attenuation, and so on. The world value is a constant across your entire OpenGL ES universe and can be set by adding the following to your `initLighting()` routine:

```
float[] colorVector={r, g, b, a};
gl.glLightModelfv(GL10.GL_LIGHT_MODEL_AMBIENT, makeFloatBuffer(colorVector));
```

The default value is a dim gray formed by a color with red=0.2f, green=0.2f, and blue=0.2f. This helps ensure that your objects are always visible no matter what. And while we are at it, there is one other value for `glLightModelfv`, and that is defined by the parameter of `GL_LIGHT_MODEL_TWO_SIDE`. The parameter is actually a boolean float. If it is 0.0, only one side will be illuminated; otherwise, both will. The default is 0.0. And if for any reason you wanted to change which faces were front ones, you may use `glFrontFace` and specify the triangles ordered clockwise or counterclockwise represent the front face. CCW is the default.

Emissive Materials

Another significant parameter we need to cover here that factors into the final color is `GL_EMISSION`. Unlike the diffuse, ambient, and specular ones, `GL_EMISSION` is for materials only and specifies that a material is emissive in quality. An emissive object has its own internal light source such as the sun, which will come in handy in the solar-system model. To see this in action, add the following line to the other material code in `initLighting()` and remove the ambient material:

```
gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_EMISSION, makeFloatBuffer(yellow));
```

Superficially, emissive materials may look just like the results of using ambient lighting. But unlike ambient lights, only a single object in your scene will be affected. And as a side benefit, they don't use up additional light objects. However, if your emissive objects do represent real lights of any sort such as the sun, putting a light object inside definitely adds another layer of authenticity to the scene.

One further note regarding materials: if your object has had the color vertices specified, like our cube has, those values can be used instead of setting materials. You must use `gl.glEnable(GL10.GL_COLOR_MATERIAL);`. This will apply the vertex colors to the shading system, instead of those specified by the `glMaterialfv` calls.

Attenuation

OpenGL ES uses one or more of the following three attenuation factors:

- `GL_CONSTANT_ATTENUATION`
- `GL_LINEAR_ATTENUATION`
- `GL_QUADRATIC_ATTENUATION`

All three are combined to form one value that then figures into the total illumination of each vertex of your model. They are set using `gLightf`, where the first parameter is your light ID such as `GL_LIGHT0`, the second is one of the three attenuation parameters listed earlier, and the actual value is passed as the third parameter.

Let's just look at one, `GL_LINEAR_ATTENUATION`, for the time being. Add the following line to `initLighting()`:

```
gl.glLightf(SS_SUNLIGHT, GL10.GL_LINEAR_ATTENUATION, 0.025f);
```

And just to make things a little clearer visually, ensure that the emissive material is turned off. What do you see? Now increase the distance down the x-axis from 10 to 50 in the `position` vector.

Spotlights

The standard lights default to an isotropic model; that is, they are like a desk light without a shade, shining equally (and blindingly) in all directions. OpenGL provides three additional lighting parameters that can turn the light into a directional light:

- `GL_SPOT_DIRECTION`

- `GL_SPOT_EXPONENT`
- `GL_SPOT_CUTOFF`

Since it is a directional light, it is up to you to aim it using the `GL_SPOT_DIRECTION` vector. It defaults to $0, 0, -1$, pointing down the –z-axis, as shown in Figure . Otherwise, if you want to change it, you would use a call similar to the following that aims it down the +x-axis:

```
float direction[]={1.0,0.0,0.0};
gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_SPOT_DIRECTION, makeFloatBuffer(direction));
```

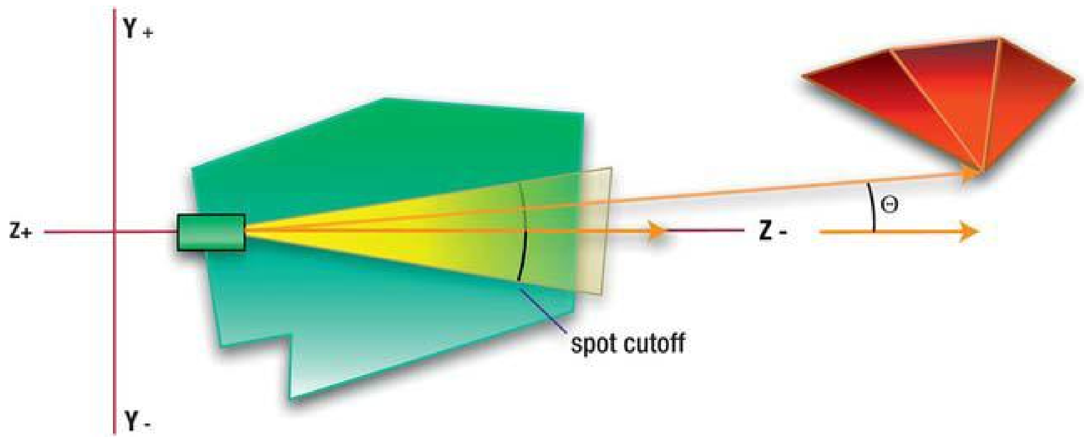

Figure 4: A spotlight aimed at the default direction

`GL_SPOT_CUTOFF` specifies the angle at which the spotlight's beam fades to 0 intensity from the center of the spotlight's cone and is naturally half the angular diameter of the full beam. The default value is 45 degrees, for a beam width of 90 degrees. And the lower the value, the narrower the beam.

The third and final spotlight parameter, `GL_SPOT_EXPONENT`, establishes the rate of drop-off of the beam's intensity, which is still another form of attenuation. OpenGL ES will take the cosine of the angle formed by the beam's center axis and that of an arbitrary vertex, Θ, and raise it to the power of `GL_SPOT_EXPONENT`. Because its default is 0, the light's intensity will be the same across all parts of the illuminated region until the cutoff value is reached, and then it drops to zero.