CMSC 425: Lecture 11
Light Modeling for Games
Tuesday, March 12, 2013

Reading: Chapt 10 of Gregory, *Game Engine Architecture*.

Lighting in Games: In order to produce a realistic rendering of 3-dimensional scene, we need to specify how the various elements of the scene are to be illuminated or shaded. Lighting and shading serve a number of roles in a computer game:

Illumination: Accurate illuminate can enhance realism and help encourage a feeling of immersion.

Revelation of form: Good lighting can be used to compensate for the lack of accuracy in a geometric models. (For example, adding a bumpy lighting pattern to a flat surface can suggest a stone road.)

Focus: Highlighting can be used to direct the viewer's attention to a desired region of focus.

Mood: Lighting can be used to set the tone of a scene (e.g., safe, threatening, foreboding, pastoral), which may be useful in establishing the narrative elements of a game. One of the simplest applications of this idea is the use of low-contrast versus high-contrast lighting (see Fig. 1):

High-Key Lighting: Bright, warm, soft, and high-set lights tend to provide a feeling of safety

Low-Key Lighting: Dim, cool, harsh, and low-set lights produce a feeling of danger

Fig. 1: High-key versus low-key lighting.

Local/Global Light Models: Lighting is a very complex phenomenon that involves the interaction of physical elements (photons, reflection and refraction, chromatic dispersion), physiological elements (color perception), and psychological elements (e.g., the feelings engendered by certain colors).

Basic OpenGL supports a very simple lighting and shading model, and hence can achieve only limited realism. This was done primarily because speed is of the essence in interactive graphics. OpenGL assumes a *local illumination model*, which means that the shading of a point depends *only* on its relationship to the light sources, without considering the other objects in the scene.

This is in contrast to a *global illumination model*, in which light reflected or passing through one object might affects the illumination of other objects. Global illumination models deal with many

affects, such as shadows, indirect illumination, color bleeding (colors from one object reflecting and altering the color of a nearby object), caustics (which result when light passes through a lens and is focused on another surface).

For example, OpenGL's lighting model does not model shadows, it does not handle indirect reflection from other objects (where light bounces off of one object and illuminates another), it does not handle objects that reflect or refract light (like metal spheres and glass balls). OpenGL's light and shading model was designed to be very efficient. Although it is not physically realistic, the OpenGL designers provided many ways to "fake" realistic illumination models. Modern GPUs support programmable shaders, which offer even greater realism.

The Phong Lighting Model: A detailed discussion of light and its properties would take us more deeply into physics than we care to go. For our purposes, we can imagine a simple model of light consisting of a large number of photons being emitted continuously from each light source.

Each photon has an associated energy, which (when aggregated over millions of different reflected photons) we perceive as *color*. Although color is complex phenomenon, for our purposes it is sufficient to consider color to be a modeled as a triple of red, green, and blue components. On hitting a surface, one of three things can happen (see Fig. 2):

Reflection: The photon can be reflected or scattered back into the atmosphere. At a microscopic level, surfaces tend to be smooth (as with polished metals) or irregular (as with cloth). The former surfaces reflect light in a *specular* (or shiny) manner and surfaces of the latter type reflect light in a *diffuse* (or uniform) manner. We thus distinguish three different varieties of reflection:

Pure reflection: Perfect mirror-like reflectors

Specular reflection: Imperfect reflectors like brushed metal and shiny plastics

Diffuse reflection: Uniformly scattering, like cloth

Absorption: The photon can be absorbed into the surface (and hence dissipates in the form of heat energy). We do not see this light.

Transmission: The photon can pass through the surface. This happens perfectly with *transparent* objects (like glass and polished gem stones) and with a significant amount of scattering with *translucent* objects (like human skin or a thin piece of tissue paper).

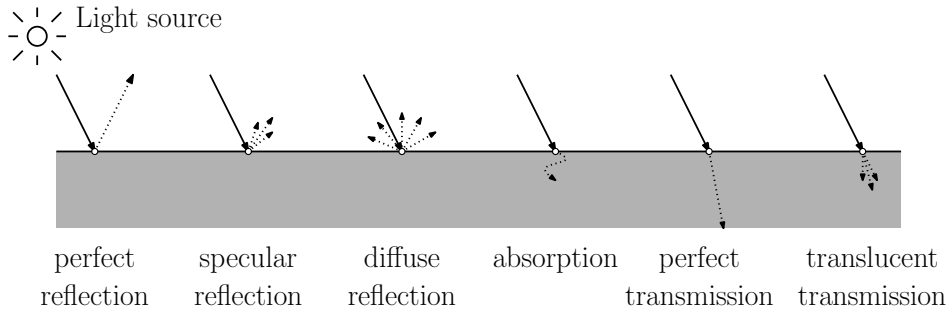

Fig. 2: The ways in which a photon of light can interact with a surface.

All of the above involve how incident light reacts with a surface. Another way that light may result from a surface is through emission, which will be discussed below.

Light Sources in OpenGL: OpenGL assumes that each light source is a *point*, and that the energy emitted can be modeled as an RGB triple, called a *luminance function*. This is described by a vector with three components $L = (L_r, L_g, L_b)$, which indicate the intensities of red, green, and blue light respectively. We will not concern ourselves with the exact units of measurement, since this is very simple model. Note that, although your display device will have an absolute upper limit on how much energy each color component of each pixel can generate (which is typically modeled as an 8-bit value in the range from 0 to 255), in theory there is no upper limit on the intensity of light.

Lighting in real environments usually involves a considerable amount of indirect reflection between objects of the scene. If we were to ignore this effect and simply consider a point to be illuminated only if it can see the light source, then the resulting image in which objects in the shadows are totally black. In indoor scenes we are accustomed to seeing much softer shading, so that even objects that are hidden from the light source are partially illuminated. In OpenGL (and most local illumination models) this scattering of light modeled by breaking the light source's intensity into two components: *ambient emission* and *point emission*.

Ambient emission: Refers to light that does not come from any particular location. Like heat, it is assumed to be scattered uniformly in all locations and directions. A point is illuminated by ambient emission even if it is not visible from the light source.

Point emission: Refers to light that originates from a single point. In theory, point emission only affects points that are directly visible to the light source. That is, a point p is illuminate by light source q if and only if the open line segment $\overline{pq}$ does not intersect any of the objects of the scene.

Unfortunately, determining whether a point is visible to a light source in a complex scene with thousands of objects can be computationally quite expensive. So OpenGL simply tests whether the surface is facing towards the light or away from the light. Surfaces in OpenGL are polygons, but let us consider this in a more general setting. Suppose that have a point p lying on some surface. Let n denote the normal vector at p, directed *outwards* from the object's interior, and let ℓ denote the directional vector from p to the light source ($\ell = q - p$), then p will be illuminated if and only if the angle between these vectors is acute. We can determine this by testing whether their dot produce is positive, that is, $n \cdot \ell > 0$. (The dot product btween two vectors is positive if and only if the angle between them is acute.)

For example, in the Fig. 3, the point p is illuminated. In spite of the obscuring triangle, point p' is also illuminated, because other objects in the scene are ignored by the local illumination model. The point p'' is clearly not illuminated, because its normal is directed away from the light.

Attenuation: The light that is emitted from a point source is subject to *attenuation*, that is, the decrease in strength of illumination as the distance to the source increases. Physics tells us that the intensity of light falls off as the inverse square of the distance. This would imply that the intensity at some (unblocked) point p would be

$$I(p, q) = \frac{1}{\|p - q\|^2} I(q),$$

where $\|p - q\|$ denotes the Euclidean distance from p to q. However, in OpenGL, our various simplifying assumptions (ignoring indirect reflections, for example) will cause point sources to appear unnaturally dim using the exact physical model of attenuation. Consequently, OpenGL uses an attenuation function that has constant, linear, and quadratic components. The user specifies constants a, b

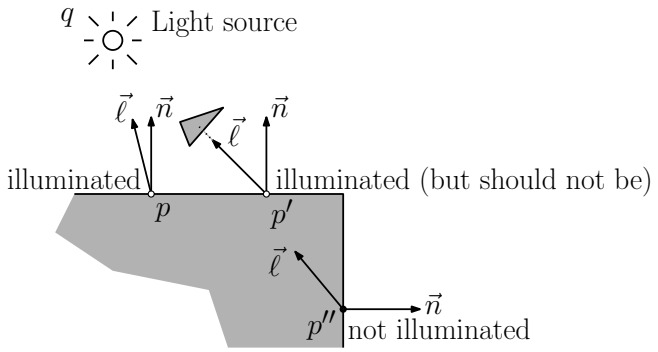

Fig. 3: Point light source visibility using a local illumination model. Note that p' is illuminated in spite of the obscuring triangle.

and c. Let $d = \|p - q\|$ denote the distance to the point source. Then the attenuation function is

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

In OpenGL, the default values are $a = 1$ and $b = c = 0$, so there is no attenuation by default.

Types of light reflection: The next issue needed to determine how objects appear is how this light is *reflected* off of the objects in the scene and reach the viewer. So the discussion shifts from the discussion of light sources to the discussion of object surface properties. We will assume that all objects are opaque. The simple model that we will use for describing the reflectance properties of objects is called the *Phong model*. The model is over 20 years old, and is based on modeling surface reflection as a combination of emission (glowing) and the ambient, diffuse, and specular reflection from the light sources in the scene.

Let $L = (L_r, L_g, L_b)$ denote the illumination intensity of the light source. OpenGL allows us to break this light's emitted intensity into three components: *ambient* L_a, *diffuse* L_d, and *specular* L_s. Each type of light component consists of the three color components, so, for example, $L_d = (L_{dr}, L_{dg}, L_{db})$, denotes the RGB vector (or more generally the RGBA components) of the diffuse component of light. As we have seen, modeling the ambient component separately is merely a convenience for modeling indirect reflection. The diffuse and specular intensities of a light source are usually set equal to each other.

An object's color determines how much of a given intensity is reflected. Let $C = (C_r, C_g, C_b)$ denote the object's color. These are assumed to be normalized to the interval $[0, 1]$. Thus we can think of C_r as the fraction of red light that is reflected from an object. Thus, if $C_r = 0$, then no red light is reflected. When light of intensity L hits an object of color C, the amount of reflected light is given by the *component-wise product* of the L and C vectors. Let us define $L \odot C$ to be this product, that is,

$$L \odot C = (L_r \cdot C_r, L_g \cdot C_g, L_b \cdot C_b).$$

For example, if the light is white $L = (2, 1, 1)$ and the color is red $C = (0.25, 0, 0)$ then the reflection is $L \odot C = (0.5, 0, 0)$ which is a dark shade of red. However if the light is blue $L = (0, 0, 1)$, then $L \odot C = (0, 0, 0)$, and hence the object appears to be black.

In OpenGL, rather than specifying a single color for an object (which indicates how much light is reflected for each component) you instead specify the amount of reflection for each type of illumination: C_a, C_d, and C_s. Each of these is an RGBA vector. This seems to be a rather extreme bit of generality, because, for example, it allows you to specify that an object can reflect only red light ambient light and only blue diffuse light. Although they can each be set separately, OpenGL provides a single command that sets both the ambient and diffuse components of reflection for an object.

What about the specular color? It is interesting to note that specular color is typically the color of the light source, not the color of the object. (Consider the shiny white spot on a black billiard ball.) For this reason, the components of the specular color of an object are typically set to $(1, 1, 1)$, meaning that the light's color is reflected perfectly.

Lighting and Shading in OpenGL: To describe lighting in OpenGL there are three major steps that need to be performed: setting the general parameters of the lighting and shading model, defining the lights (their positions, colors, and properties), and finally defining objects and specifying their material properties.

Lighting/Shading model: There are a number of global lighting parameters and options that can be set through the command glLightModel*(). It has two forms, one for scalar-valued parameters and one for vector-valued parameters.

```
glLightModelf(GLenum pname, GLfloat param);
glLightModelfv(GLenum pname, const GLfloat* params);
```

Perhaps the most important parameter is the global intensity of ambient light (independent of any light sources). Its pname is GL_LIGHT_MODEL_AMBIENT and params is a pointer to an RGBA vector.

One important issue is whether polygons are to be drawn using *flat shading*, in which every point in the polygon has the same shading, or *smooth shading*, in which shading varies across the surface by interpolating the vertex shading. This is set by the following command, whose argument is either GL_SMOOTH (the default) or GL_FLAT.

```
glShadeModel(GL_SMOOTH);    --OR--    glShadeModel(GL_FLAT);
```

In theory, shading interplation can be handled in one of two ways. In the classical *Gouraud interpolation* the illumination is computed exactly at the vertices (using the above formula) and the values are interpolated across the polygon. In *Phong interpolation*, the normal vectors are given at each vertex, and the system interpolates these vectors in the interior of the polygon. Then this interpolated normal vector is used in the above lighting equation. This produces more realistic images, but takes considerably more time. OpenGL uses Gouraud shading. Just before a vertex is given (with glVertex*()), you should specify its normal vertex (with glNormal*()).

The commands glLightModel and glShadeModel are usually invoked in your initializations.

Create/Enable lights: To use lighting in OpenGL, first you must enable lighting, through a call to glEnable(GL_LIGHTING). OpenGL allows the user to create up to 8 light sources, named GL_LIGHT0 through GL_LIGHT7. Each light source may either be enabled (turned on) or disabled (turned off). By default they are all disabled. Again, this is done using glEnable() (and glDisable()). The properties of

each light source is set by the command glLight*(). This command takes three arguments, the name of the light, the property of the light to set, and the value of this property.

Let us consider a light source 0, whose position is $(2, 4, 5)$. This would be presented to OpenGL in homogeneous coordinates. Recall that this means that we set the last component to 1. Thus, it would be $(2, 4, 5, 1)$ in homogeneous coordinates. Suppose we want this to generate a bright red ambient intensity, given as the RGB triple $(0.9, 0, 0)$, and white diffuse and specular intensities, given as the RGB triple $(1.2, 1.2, 1.2)$. (Normally all the intensities will be of the same color, albeit of different strengths. We have made them different just to emphasize that it is possible.) There are no real units of measurement involved here. Usually the values are adjusted manually by a designer until the image "looks good."

Light intensities are actually expressed in OpenGL as RGBA, rather than just RGB triples. The 'A' component can be used for various special effects, but for now, let us just assume the default situation by setting 'A' to 1. Here is an example of how to set up such a light in OpenGL. The procedure glLight*() can also be used for setting other light properties, such as attenuation.

Setting up a simple lighting situation

```
glClearColor(0.0, 1.0, 0.0, 1.0);           // intentionally background
glEnable(GL_NORMALIZE);                     // normalize normal vectors
glShadeModel(GL_SMOOTH);                    // do smooth shading
glEnable(GL_LIGHTING);                      // enable lighting
                                            // ambient light (red)
GLfloat ambientIntensity[4] = {0.9, 0.0, 0.0, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientIntensity);

                                            // set up light 0 properties
GLfloat lt0Intensity[4] = {1.5, 1.5, 1.5, 1.0};  // white
glLightfv(GL_LIGHT0, GL_DIFFUSE, lt0Intensity);
glLightfv(GL_LIGHT0, GL_SPECULAR, lt0Intensity);

GLfloat lt0Position[4] = {2.0, 4.0, 5.0, 1.0};   // location
glLightfv(GL_LIGHT0, GL_POSITION, lt0Position);
                                           // attenuation params (a,b,c)
glLightf (GL_LIGHT0, GL_CONSTANT_ATTENUATION,  0.0);
glLightf (GL_LIGHT0, GL_LINEAR_ATTENUATION,    0.0);
glLightf (GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.1);
glEnable(GL_LIGHT0);
```

Defining Surface Materials (Colors): When lighting is in effect, rather than specifying colors using glColor() you do so by setting the material properties of the objects to be rendered. OpenGL computes the color based on the lights and these properties. Surface properties are assigned to vertices (and not assigned to faces as you might think). In smooth shading, this vertex information (for colors and normals) are interpolated across the entire face. In flat shading the information for the first vertex determines the color of the entire face.

Every object in OpenGL is a polygon, and in general every face can be colored in two different ways. In most graphic scenes, polygons are used to bound the faces of solid polyhedra objects and hence are only to be seen from one side, called the *front face*. This is the side from which the vertices are

given in counterclockwise order. By default OpenGL, only applies lighting equations to the front side of each polygon and the back side is drawn in exactly the same way. If in your application you want to be able to view polygons from both sides, it is possible to change this default (using glLightModel() so that each side of each face is colored and shaded independently of the other. We will assume the default situation.

Surface material properties are specified by glMaterialf() and glMaterialfv().

```
glMaterialf(GLenum face, GLenum pname, GLfloat param);
glMaterialfv(GLenum face, GLenum pname, const GLfloat *params);
```

It is possible to color the front and back faces separately. The first argument indicates which face we are coloring (GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK). The second argument indicates the parameter name (GL_EMISSION, GL_AMBIENT, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE, GL_SPECULAR, GL_SHININESS). The last parameter is the value (either scalar or vector). See the OpenGL documentation for more information.

Typical drawing with lighting

```
GLfloat color[]  = {0.0, 0.0, 1.0, 1.0};    // blue
GLfloat white[]  = {1.0, 1.0, 1.0, 1.0};    // white
                                            // set object colors
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, color);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 100);

glPushMatrix();
    glTranslatef(...);                      // your transformations
    glRotatef(...);
    glBegin(GL_POLYGON);                    // draw your shape
        glNormal3f(...); glVertex(...);     // remember to add normals
        glNormal3f(...); glVertex(...);
        glNormal3f(...); glVertex(...);
    glEnd();
glPopMatrix();
```

Recall from the Phong model that each surface is associated with a single color and various coefficients are provided to determine the strength of each type of reflection: emission, ambient, diffuse, and specular. In OpenGL, these two elements are combined into a single vector given as an RGB or RGBA value. For example, in the traditional Phong model, a red object might have a RGB color of $(1, 0, 0)$ and a diffuse coefficient of 0.5. In OpenGL, you would just set the diffuse material to $(0.5, 0, 0)$. This allows objects to reflect different colors of ambient and diffuse light (although I know of no physical situation in which this arises).

Other options: You may want to enable a number of GL options using glEnable(). This procedure takes a single argument, which is the name of the option. To turn each option off, you can use glDisable(). These optional include:

GL_CULL_FACE: Recall that each polygon has two sides, and typically you know that for your scene, it is impossible that a polygon can only be seen from its back side. For example, if you draw

a cube with six square faces, and you know that the viewer is outside the cube, then the viewer will never see the back sides of the walls of the cube. There is no need for OpenGL to attempt to draw them. This can often save a factor of 2 in rendering time, since (on average) one expects about half as many polygons to face towards the viewer as to face away.

Backface culling is the process by which faces which face away from the viewer (the dot product of the normal and view vector is negative) are not drawn.

By the way, OpenGL actually allows you to specify which face (back or front) that you would like to have culled. This is done with glCullFace() where the argument is either GL_FRONT or GL_BACK (the latter being the default).

GL_NORMALIZE: Recall that normal vectors are used in shading computations. You supply these normal to OpenGL. These are assumed to be normalized to unit length in the Phong model. Enabling this option causes all normal vectors to be normalized to unit length automatically. If you know that your normal vectors are of unit length, then you will not need this. It is provided as a convenience, to save you from having to do this extra work.

Tips: Until you have finished debugging your program, I would suggest *disabling* backface culling (in case you accidentally generate the elements of your mesh in an improper CW/CCW orientation). I would also suggest *enabling* normalization of normal vectors.