Chapter 9

Lighting

This chapter describes the GL lighting facility and tells you how to create lighted scenes.

- Section 9.1, “Introduction to GL Lighting,” presents some terminology and basic principles of lighting that you need to know to understand GL lighting.
- Section 9.2, “Setting Up GL Lighting,” tells you how to configure lighting parameters.
- Section 9.3, “Binding Lighting Definitions,” tells you how to activate your lighting definitions.
- Section 9.4, “Changing Lighting Settings,” tells you how to change lighting definitions that you have set up.
- Section 9.5, “Default Settings,” describes the default lighting settings and tells you how to create your own default definitions.
- Section 9.6, “Advanced Lighting Features,” tells you how to use special lighting features for more complex lighting effects.
- Section 9.7, “Lighting Performance,” gives you some programming hints to help you get the best performance from lighting.
- Section 9.8, “Color Map Lighting,” tells you how to use lighting in color map mode, primarily for machines with limited RGB capability.

Code fragments are used extensively to encourage a learn-by-example approach. A complete sample program is included at the end of the chapter.

9.1 Introduction to GL Lighting

In the real world, when light falls on an opaque object, some of this light is absorbed by the object and the rest of the light is reflected. Our eyes use this reflected light to interpret the shape, color, and other details about the object. The light that strikes the object from the light source is called *incident* light; the light that bounces off the object's surface is called *reflected* light.

In GL lighting, as in the real world, the characteristics of the light source determine the direction, intensity, and wavelength(color) of the incident light. The characteristics of the object geometry and its surface material determine the direction, intensity, and color of the reflected light. How light is reflected depends on the interaction between the incident light and the surface material.

The interaction between light and matter is far more complicated than can be simulated in real time. Lighting on the IRIS achieves a balance between realistic appearance and real-time drawing speed. The GL achieves this balance by performing lighting calculations only at geometry vertices, rather than calculating lighting for each pixel rendered.

You control GL lighting by creating a lighting definition that allows you to specify and manipulate the characteristics of the incident light, the surface properties of lighted objects, and the effects of the surrounding environment. In a lighting definition, you set up the color, intensity, and position of light sources, the properties of surface materials, and the characteristics of the lighting environment in your scene. Once you have your lighting definition set up, you can turn lights on and off, use different object materials, and use advanced lighting techniques for interesting effects.

9.1.1 Color

A lighting definition determines how the color of incident light is modified when it reflects from surfaces. For example, an object can appear blue because:

- A white light source shines on an object which reflects only the wavelengths that our eyes interpret as blue. Although other wavelengths of light are present, the object absorbs them and reflects only blue.
- The object reflects blue light and possibly other wavelengths, but only blue light is shining on it. In this case, there are no other wavelengths for the object to reflect.

GL lighting allows both of these methods: you can define a blue light and shine it on a white object, or you can define a white light and shine it on a blue object.

9.1.2 Reflectance

The ratio of incident light to reflected light is called *reflectance*. Some of the factors that determine reflectance are inherent in an object's geometry; other properties that affect reflectance are controlled by GL lighting.

There are three reflectance properties that determine how a surface reflects light:

- *diffuse*, which shows up as a matte, or flat, reflection
- *specular*, which shows up as highlights
- *ambient*, which simulates indirect light

Diffuse Reflectance

Diffuse reflectance gives the appearance of a matte, or flat, reflection from a surface. The direction of the light as it falls on the surface determines how bright the diffuse reflection is. Diffuse reflection is brightest where the incident light strikes the perpendicular to the surface.

For example, consider a distant light shining directly on the north pole of a sphere representing the earth. The diffuse reflectance of the sphere causes the sphere to appear brightest at the north pole. The brightness falls off as you look farther down the sides of the sphere. South of the equator there is no diffuse reflectance at all.

Because diffuse light reflects equally in all directions, the viewer's perception of diffuse intensity is not affected by the viewing angle.

Specular Reflectance

Specular reflectance creates highlights and is dependent on the position of the viewpoint. For example, consider the glare in your rear view mirror from the headlights of a car behind you. If you shift your head a few inches to the right or left, you cannot see the glaring headlights in your mirror. The intensity of specular reflection is typically highest along the direct angle of reflection.

Ambient Reflectance

Diffuse and specular reflectance simulate how objects in the scene reflect light that comes directly from a light source. Ambient reflectance simulates light reflected from other objects in the scene, rather than directly from the light source. For example, if you look under your desk (presuming that you have a light on your desk that does not shine directly under it), you can still see things, even though the area under your desk is not directly illuminated. In reality, this ambient light is reflected from other surfaces in the room.

The ambient component is most noticeable in areas of the scene that receive no direct illumination.

Emission

Emission is the quality of giving off light. Adding an emission component is useful for simulating the appearance of lights in a scene. A GL object that emits light does not also automatically act as a light source; that is, it does not add illumination to a scene. For this object to act as a light source in the scene, you must add a light to the lighting model and position this light at the same location as the object that is emitting light.

9.2 Setting Up GL Lighting

This section introduces fundamental GL lighting concepts, and tells you how to create a static lighting environment.

Points, lines, polygons, and character strings can all be lighted. As a general rule, any GL primitive that uses the current color can also be lighted. The lighting calculation is performed at each vertex of a primitive.

To set up GL lighting:

1. Define the properties for each material, light and lighting model that you want in your scene.
2. Activate (*bind*) your definitions.
3. Draw the scene. Provide surface normals, as discussed in Section 9.2.1, “Defining Surface Normals,” for all primitives to be lighted.

9.2.1 Defining Surface Normals

Surface normals are unit-length vectors that are perpendicular to a given surface at a particular vertex. They serve as input to the lighting formula, and are required for GL lighting.

When using GL NURBS surfaces (see Chapter 14, “Curves and Surfaces”), normals can be generated automatically from the surface descriptions.

Specify a surface normal for each vertex using the `n3f()` subroutine. The GL maintains a *current normal*, which remains the same until you change it. The current normal is analogous to the current color.

This example specifies a point with a normal:

```
static float np[3] = {0, .7071, .7071};
static float vp[3] = {0, 0, -1};

bgnpoint();
   n3f(np);
   v3f(vp);
endpoint();
```

This example specifies a triangle with normals:

```
static float np[3][3] =
   {{-.08716, 0, .9962}, {.08716, 0, .9962}, {0, 0, 1}};
static float vp[3][3] =
   {{-.08716, 0, .9962}, {.08716, 0, .9962}, {0, .1, 1}};

bgnpolygon();
   n3f(np[0]);
   v3f(vp[0]);
   n3f(np[1]);
   v3f(vp[1]);
   n3f(np[2]);
   v3f(vp[2]);
endpolygon();
```

The relationship between the order of the vertices and the direction of the normals is significant. When the order of the vertices of the projected triangle is counterclockwise as seen from the viewer’s perspective, the front face of the triangle is the side toward the viewer. In this case, the normals of the triangle also point toward the viewer. This convention obeys the right-hand rule. The example triangle given above displays this behavior. The right-hand rule

makes it possible to distinguish between the front and the back faces of the triangle. Although it is not necessary for you to follow the right-hand-rule when using lighting, doing so allows you to use backface elimination (see Chapter 8). Two-sided lighting, described in Section 9.6.3, requires you to follow the right-hand-rule.

Non-Unit-Length Normals

The previous section stated that normals must be unit-length, which is the default. The GL can handle non-unit-length normals, but there may be a performance penalty associated with their use. See Section 9.7, "Lighting Performance," for information about performance considerations with non-unit-length normals.

Use the `nmode()` subroutine to handle non-unit-length normals. The default mode is `NAUTO`, which automatically corrects for situations where normals you would expect to be unit-length become non-unit-length. When you know your normals are unit-length, use this mode:

```
nmode(NAUTO);
```

To configure the GL to automatically normalize (correct to unit length) *all* normals (when you are intentionally using non-unit length normals), use:

```
nmode(NNORMALIZE);
```

Note: Lighting does not have to be on to set `nmode()`.The `nmode()` command is not available on all systems, see the `nmode()` man page for details.

9.2.2 Defining Lighting Components

You configure three lighting components to define GL lighting for your scene:

- material - determines how a surface responds to illumination
- light source - determines the characteristics of the incident light
- lighting model - determines the behavior of the lighting environment

The combination of these three components determines the appearance, or more specifically, the color at each lighted vertex. The GL calculates the color at each lighted vertex by summing the total ambient light (scaled by the

material ambient reflectance), the material emitted light, and the contributions of each light source.

The total ambient light is the sum of the ambient light associated with each light source and the ambient light associated with the scene, as given by the lighting model.

The contribution of each light source is the sum of:

1. Light source ambient color, scaled by material ambient reflectance.
2. Light source color, scaled by material diffuse reflection and the *dot product* (see Foley and Van Dam, *Computer Graphics Principles and Practice*) of the vertex normal and the vertex-to-light source vector.
3. Light source color, scaled by material specular reflectance and a function of the angle between the vertex to viewpoint vector and the vertex-to-light source vector.

You configure each of the three components using a two-step process. The first step is *definition*, using `lmdef()`. The second step is *activation*, using `lmbind()`.

Use the `lmdef()` subroutine to establish your material, light source and lighting model definitions.

The ANSI C specification of `lmdef()` is:

```
void lmdef(short deftype, short index, short np, float props[])
```

where:

deftype	indicates whether the properties in the array apply to a material, a light source, or a lighting model.
index	defines an index to be associated with the definition.
np	provides a count of symbols and floating point values in the props array. You can usually set np to 0, in which case it is ignored; however, operation over network connections is more efficient when np is correctly specified.
props	array of floating point symbols that define properties.

You specify properties as an array of floats. The array elements are terminated by the special constant `LMNULL`, which must always be the last float in the array. `lmdef()` associates the properties in your property array with an integer index and copies these properties at the time of the call.

Index 0 is reserved as the null definition for material, light source and lighting model definitions. You use index 0 to turn definitions off, as described in Section 9.4, "Changing Lighting Settings." You cannot assign your own definition to index 0.

Defining a Material

To define a material, you specify its properties in an array of floats.

For example, a greenish plastic-like material is defined like this:

```
static float mat[] = {
    AMBIENT, .1, .1, .1,
    DIFFUSE, 0, .369, .165,
    SPECULAR, .5, .5, .5,
    SHININESS, 30,
    LMNULL
};
lmdef(DEFMATERIAL, 39, 5, mat);
```

In this example, the `lmdef()` call stores the material properties in the *mat* array as material number `39`. This creates a material definition that will be referenced later by its integer index, `39`.

Each property in the array expects a fixed number of floats to follow it. `DIFFUSE` is followed by three floats that are the red, green, and blue diffuse reflectance coefficients. Similarly, the `AMBIENT` and `SPECULAR` properties are each followed by three floats that represent the red, green and blue ambient and specular reflectance coefficients, respectively. All color components should be in the range 0.0 to 1.0. The system *clamps* (limits) the color components to a maximum value of 1.0, and scales the components by 255 before loading them in the framebuffer.

Specular reflectance also depends on the viewpoint. You specify the viewpoint in the lighting model. (See the description of the `LOCALVIEWER` property under Defining a Lighting Model in this Section for more details.)

`SHININESS` is followed by one float that controls the size and apparent brightness of a specular highlight. The shininess value can range from 0 to 128. Higher values result in smaller, more focused specular highlights. A shininess value of 0 disables specular reflection entirely.

EMISSION is followed by three floats, in the range 0.0 to 1.0, that specify the red, green and blue components of the emitted light. This example defines a material that has only an emission property:

```
static float mat[] = {
    EMISSION, 0, .369, .165,
    LMNULL
};
lmdef(DEFMATERIAL, 39, 2, mat);
```

Defining a Light Source

There are two types of light sources that you can use: a *point light source* and an *infinite light source*. A point light source has a position and a direction. An infinite light source represents a light a great distance away (like the sun). Infinite light sources provide subtle overall lighting and have a performance advantage over point light sources.

The next example defines a slightly reddish-colored point light source:

```
static float lt[] = {
    LCOLOR, 1, .8, .8,
    POSITION, 0, 1.5, -.5, 1,
    LMNULL
};
lmdef(DEFLIGHT, 27, 0, lt);
```

This light source definition is associated with the integer index 27, by which it can later be referenced.

Although you use the same syntax for a light source as for a material, the actual properties are different. In this case, the LCOLOR property is followed by three floats that are the red, green, and blue components of the light source color. A light source is normally omnidirectional; that is, it emits light of equal intensity in all directions.

AMBIENT is followed by three floats, specifying the red, green and blue components of the ambient light associated with the light source.

POSITION is followed by four floats that are the *x*, *y*, *z*, and *w* coordinates of the light source. The *x*, *y*, and *z* coordinates represent the location of the light source in object coordinates. Note that the light source direction points from the vertex to the light source. The GL normalizes the light source direction.

Light source position is defined in homogeneous coordinates. If the *w* coordinate is zero, an infinite light source is defined; a non-zero *w* component defines a point light source and the *x*, *y*, and *z* coordinates represent the direction from the origin to the light source. This example defines a white light source that is positioned infinitely far away on the positive *z* axis:

```
static float lt[] = {
   LCOLOR, 1., 1., 1.,
   POSITION, 0., 0., 1., 0.,
   LMNULL
};
lmdef(DEFLIGHT, 27, 3, lt);
```

Defining a Lighting Model

The next example defines a typical lighting model:

```
static float lm[] = {
   AMBIENT, .1, .1, .1,
   LOCALVIEWER, 1,
   LMNULL
};
lmdef(DEFLMODEL, 14, 0, lm);
```

Once again, use the same syntax for a lighting model as for a material or light source. The `AMBIENT` property specifies the color of ambient light associated with the entire scene. This ambient light is nondirectional.

Specular reflectance from a point on a surface depends on the normal, the direction to the light source, and the direction to the viewpoint.

You can use a *local viewpoint* (placed at the origin) or an *infinite viewpoint* (placed at an infinite distance on the positive *z* axis).

Use `LOCALVIEWER` to indicate whether the viewpoint is local or infinite. `LOCALVIEWER` is followed by a single float, either 0.0 to indicate that the viewpoint is infinite, or 1.0 to indicate that the viewpoint is local. With a local viewpoint, the direction to the viewpoint needs to be recalculated at each vertex, which may cause a decrease in performance.

This example defines a lighting model with an infinite viewpoint:

```
static float lm[] = { LOCALVIEWER, 0, LMNULL };
lmdef(DEFLIGHT, 14, 2, lm);
```

When you use an infinite viewpoint with infinite lights, primitives with the same normal and material properties produce identical colors. In practice, this means that surfaces are lighted with constant color, which might appear slightly unrealistic.

9.3 Binding Lighting Definitions

After defining the three components of lighting, you must *bind* (activate) them. Before binding any lighting definitions, you must be in multimatrix mode:

```
mmode(MVIEWING);
```

The next example shows how to bind the previously defined material, light source, and lighting model.

```
lmbind(MATERIAL, 39);
lmbind(LIGHT0, 27);
lmbind(LMODEL, 14);
```

Material index `39` represents the material definition from our previous example. You activate this material definition by binding it to the target `MATERIAL`. Only one material can be active at any time. Light source `27` from the previous example is activated by binding it to the target `LIGHT0`.

There are at least eight light source targets available, named `LIGHT0`, `LIGHT1`, `LIGHT2`, and so on. The actual number of these targets is defined in *gl.h* by the constant `MAXLIGHTS`. Any number of the light source targets can be active at any time. You can bind a light definition to only one light source target.

The current ModelView matrix transforms the position of the light source when you call `lmbind()` (see Chapter 7, “Coordinate Transformations” for more information on transformations). In this example, the position of the light source has a non-zero *w* component. This makes it a point light source. Its position is transformed in the same way that a point is transformed.

Lighting model index `14` represents the lighting model definition from a previous example. You activate it by binding it to the target `LMODEL`. Only one lighting model can be active at any time.

At this point, lighting is enabled. More specifically, lighting is enabled when both a material and a lighting model are bound. The indexes `39`, `27`, and `14`

were chosen for this example and are arbitrary. You could use index 1 for all three components because indexes for materials are separate from those of lights, and lighting models. Any index between 1 and 65535 is legal. Index 0 is used to turn lighting off, as described in Section 9.4, “Changing Lighting Settings.”

9.4 Changing Lighting Settings

Use the `lmbind()` subroutine to change or deactivate (unbind) currently bound lighting definitions. Specify index 0 with `lmbind()` to unbind a material, light, or lighting model.

This example turns off the light source bound to `LIGHT0`:

```
lmbind(LIGHT0, 0);
```

Note: After this `lmbind()` call, lighting remains on because the presence of active lights is not necessary for lighting.

Recall that lighting is enabled when both a material and a lighting model are bound. To turn off lighting, you must unbind either the *material* or the *lighting model*. This example turns lighting off:

```
lmbind(LMODEL, 0);
```

Suppose you have defined more than one material for geometry you are drawing. When you bind the second material definition, it becomes the active material, overriding the previously bound material.

You can also use `lmdef()` to change the properties of an existing definition instead of creating a new definition for each variation to that definition.

Recall the point light source definition 27 from the previous example. This example changes this light source definition to produce a greenish color:

```
static float lt[] = {
    LCOLOR, .8, 1, .8,
    LMNULL
};
lmdef(DEFLIGHT, 27, 2, lt);
```

If light source definition 27 is currently bound to a light source, such as LIGHT0, the light color change takes effect immediately. Because LCOLOR is the only property in this array, only this property is changed. Changes made to bound definitions are effective immediately. Only the specified properties are changed, the other properties remain the same.

A light source position is transformed by the ModelView matrix at the time of the lmbind() call. It can be retransformed by binding it again. This is often used to make a light move from frame to frame. A light source that was previously bound can only be rebound to its original light target.

9.5 Default Settings

When you create a definition with lmdef(), the properties are first set to their default values. Properties specified in the array override these defaults. Later, when you change this definition with lmdef(), properties not specified in your array are left unchanged.

These definitions contain the default settings for material, light source, and lighting model:

```
static float mat[] = {
   ALPHA, 1.0,
   AMBIENT, .2, .2, .2,
   COLORINDEXES, 0, 127.5, 255,
   DIFFUSE, .8, .8, .8,
   EMISSION, 0.0, 0.0, 0.0,
   SPECULAR, 0.0, 0.0, 0.0,
   SHININESS, 0.0,
   LMNULL
};

static float lt[] = {
   AMBIENT, 0.0, 0.0, 0.0,
   LCOLOR, 1.0, 1.0, 1.0,
   POSITION, 0.0, 0.0, 1.0, 0.0,
   SPOTLIGHT, 0.0, 180.0,
   SPOTDIRECTION, 0.0, 0.0, -1.0,
   LMNULL
};
```

```
static float lm[] = {
   AMBIENT, .2, .2, .2,
   ATTENUATION, 1.0, 0.0,
   ATTENUATION2, 0.0,
   LOCALVIEWER, 0.0,
   TWOSIDE, 0.0,
   LMNULL
};
```

Passing `NULL` as the fourth parameter of an `lmdef()` call creates a definition of a default set of properties. It is also equivalent to passing an array of type `float` that has one element, the special constant `LMNULL`.

The following example sets defaults for the example lighting model:

```
lmdef(DEFMATERIAL, 39, 0, NULL);
lmdef(DEFLIGHT, 27, 0, NULL);
lmdef(DEFLMODEL, 14, 0, NULL);
```

9.6 Advanced Lighting Features

This section covers advanced lighting features. Not all lighting features are supported on all systems. Consult the `lmdef()` man page to determine which features are available on your system.

9.6.1 Attenuation

In reality, the effect of a light source on a surface diminishes as the distance between the light source and the surface increases. You can simulate this effect with the *attenuation* feature of GL lighting. Attenuation is defined in the lighting model and applies to all point light sources. Attenuation does not apply to infinite or ambient light sources. Attenuation is not supported on all systems; it is ignored on systems that do not support it.

Attenuation is a function of the distance between a point light source and the surface it illuminates. You can specify *constant*, *linear* and *distance-squared* attenuation factors.

The formula for attenuation is:

attenuation factor = 1 / (k0 + k1∗dist + k2∗dist∗dist) (EQ 9-1)

where:

dist	is the distance between the vertex and the point light source. This distance is never negative.
k0	controls constant attenuation.
k1	controls linear attenuation.
k2	controls distance-squared attenuation.

The attenuation formula is calculated for each lighted vertex. The following example specifies constant and linear attenuation factors for lighting model `14`.

```
static float atten[] = {
   ATTENUATION, .1, 1,
   LMNULL
};
lmdef(DEFLMODEL, 14, 2, atten);
```

The `ATTENUATION` property is followed by two non-negative floats. These floats specify *k0* and *k1* of the attenuation formula.

This example adds distance-squared attenuation to lighting model `14`:

```
static float atten[] = {
   ATTENUATION, .1, 0,
   ATTENUATION2, 1,
   LMNULL
};
    lmdef(DEFLMODEL, 14, 3, atten);
```

The `ATTENUATION2` property is followed by one non-negative float, specifying *k2*.

The attenuation factor defaults are *k0* = 1, *k1* = 0, and *k2* = 0. These values define a lighting model without attenuation. You can disable attenuation by restoring these default values.

Both of these examples use a small, but non-zero value for the constant term *k0*. As *dist* approaches zero, a non-zero *k0* bounds the maximum value of the attenuation formula; otherwise, it would approach infinity.

9.6.2 Spotlights

A point light source is omnidirectional by default. You can make a point light source into a directional *spotlight* using the SPOTLIGHT property. Spotlights are available only on certain systems. See the `lmdef()` man page to determine whether your system supports spotlights.

A spotlight emits a cone of light that is centered along the spotlight direction. The intensity of a spotlight is a function of the angle between the spotlight direction and the direction to the vertex being illuminated. Typically, the intensity falls off as this direction angle increases.

You can control the shape of a spotlight's intensity falloff with two values: an *exponent* and a *spread angle*. An exponent of 1 produces a gradual falloff that is actually the cosine of the direction angle. An exponent of 128 gives the sharpest possible falloff; an exponent of 0 gives a constant intensity. The spread angle defines a cone outside which no light is emitted. The intensity falloff as controlled by the exponent is cut off by this cone. The cone defined by the spread angle is independent of the intensity falloff controlled by the exponent.

This example defines and binds a white spotlight:

```
static float spot[] = {
    LCOLOR, 1, 1, 1,
    POSITION, 0, 2, 0, 1,
    SPOTDIRECTION, 0, -1, 0,
    SPOTLIGHT, 100, 45,
    LMNULL
};
lmdef(DEFLIGHT, 11, 5, spot);
lmbind(LIGHT0, 11);
```

The SPOTLIGHT property is followed by two floats specifying the exponent and spread angle of the light cone. The exponent can range from 0 to 128; an exponent of 100 specifies a sharp falloff. The spread angle can range from 0 to 90 degrees; a spread angle of 45 defines a cone with a radius angle of 45 degrees. A special value of 180 degrees is also permitted, and is used in the next example.

The SPOTDIRECTION property is followed by three floats, the *x*, *y*, and *z* coordinates of the spotlight direction vector. The direction vector is automatically normalized. This example points the spotlight in the direction of

the negative *y* axis. The spotlight direction vector is transformed by the current ModelView matrix in the same manner as a normal.

Notice that the POSITION property specifies a point light source. This is necessary for a spotlight. The SPOTLIGHT property is ignored for an infinite light source.

This example turns off the spotlight effect:

```
static float spotoff[] = { SPOTLIGHT, 0.0, 180.0, LMNULL };
lmdef(DEFLIGHT, 11, 2, spotoff);
```

The combination of setting the exponent to 0.0 and the spread angle to 180.0 turns off the spotlight effect. By default, a point light source is not a spotlight. The SPOTDIRECTION property is ignored when the light source is not a spotlight.

You can combine spotlights with attenuation to yield an effect that is reminiscent of a real spotlight. The spotlight effect can create a highly non-linear intensity gradient across a surface. To make the approximation of the gradient as accurate as possible, place the vertices of the surface illuminated by a spotlight very close to one another.

9.6.3 Two-Sided Lighting

In general, lighting calculations are correct only when you view the side of a polygon where the normal faces toward you. If you use the right-hand rule to define the polygons and their normals, lighting calculations are correct for the front faces of those polygons. This is called one-sided lighting. With *two-sided lighting*, the lighting for backfacing polygons is also correct. Two-sided lighting is available only on certain systems. Use getgdesc(GD_LIGHTING_TWOSIDE) to determine if two-sided lighting is available.

This example adds two-sided lighting to a lighting model definition:

```
static float two[] = { TWOSIDE, 1, LMNULL };
    lmdef(DEFLMODEL, 14, 3, two);
```

The TWOSIDE property is followed by one float, either 1.0 or 0.0, that specifies whether the two-sided lighting feature should be enabled or disabled, respectively. It is disabled by default.

Two-sided lighting applies only to primitives with facets, such as polygons or triangle meshes. Two-sided lighting is ignored for other lighted primitives, such as points or lines.

With two-sided lighting, the material properties of the front and back faces are normally identical; that is, the active material is used for both the front and the back faces. You can also specify independent front and back material properties. Independent front and back materials can be useful to distinguish between the inside and the outside of an object. This feature is quite effective when combined with user-defined clipping planes. See Chapter 8, "Hidden-Surface Removal", for more information about two-sided polygons.

This example binds material definition `40` to the back material as follows:

```
lmbind(BACKMATERIAL, 40);
```

You unbind the back material by binding it to 0:

```
lmbind(BACKMATERIAL, 0);
```

By default, the back material is bound to `0`. Whether a back material is bound or not has no effect on whether lighting is on or off.

9.6.4 Fast Updates to Material Properties

When you change the properties of an active (currently bound) definition, those changes take effect immediately.

Assume that material definition `39` exists and is currently bound. The following example changes its `DIFFUSE` property immediately:

```
static float mat[] = {
    DIFFUSE, .369, 0, .165,
    LMNULL
};
    lmdef(DEFMATERIAL, 39, 2, mat);
```

This mechanism provides a general method for updating the properties of a material, a light source, or a lighting model.

It is useful to have an even more efficient method of changing material properties. For example, it is reasonable to change a specific material property

at each vertex of many polygons. For this reason, the GL provides a *fast update* mechanism for material properties.

You use the `lmcolor()` subroutine to set a mode where the current color updates a specific material property. The current color should be set explicitly after the call to `lmcolor()` and before a vertex or normal is issued. The current color can be set with `c`, `cpack()`, or `RGBcolor()`.

Note: For higher performance, call `lmcolor()` only once, prior to drawing a large number of primitives. Do not call `lmcolor()` within a primitive (between `bgn` and `end` calls). If you must change more than one material property per vertex, excluding related `LMC_AD` ambient or diffuse changes, it is probably better to use `lmdef()` than to mix `lmcolor()` and `color()` calls within a single primitive.

Some of the color commands specify red, green, blue, and alpha as integers in the range of 0 to 255. This range is mapped to a 0.0 to 1.0 range when used to update material properties.

This example illustrates using `lmcolor()` to update the `DIFFUSE` property of the current material.

```
lmcolor(LMC_DIFFUSE);
RGBcolor(94, 0, 42);
lmcolor(LMC_COLOR);
```

The first call to `lmcolor()` sets `LMC_DIFFUSE` mode. In this mode, the `RGBcolor()` call directly updates the diffuse property. The second call to `lmcolor()` restores the default mode.

This example sets the diffuse property of the active material to roughly the same values as the previous example, but there is an important difference. When you use `lmdef()` to change an active material, the material *definition also changes*.

When you use `lmcolor()` to change an active material, the *change has no effect on the material definition*. Actually, any changes made to the active material using `lmcolor()` are lost when you use `lmbind()` to bind another material.

This example updates the ambient and diffuse properties of the current material at each vertex of a polygon.

```
lmcolor(LMC_AD);
bgnpolygon();
   cpack(0x800000ff);
   n3f(np[0]);
   v3f(vp[0]);
   cpack(0x8000ff00);
   n3f(np[1]);
   v3f(vp[1]);
   cpack(0x80ff0000);
   n3f(np[2]);
   v3f(vp[2]);
endpolygon();
lmcolor(LMC_COLOR);
```

This example uses a normal array, *np*, and a vertex array, *vp*, in the same manner as the triangle example in Section 9.2.1, “Defining Surface Normals.” The call to `lmcolor(LMC_AD)` sets a mode where the calls to `cpack()` directly update the ambient and diffuse material properties simultaneously.

The modes `LMC_AD`, which updates both the ambient and diffuse properties, and `LMC_DIFFUSE`, which updates only the diffuse property, also update the `ALPHA` material property with the alpha component of the current color. See Section 9.6.5, “Transparency,” for alpha information. Note that the alpha component in this example is set to `0x80` (roughly 0.5) at each vertex.

The default mode, `LMC_COLOR`, has an interesting property. If no normals are present for a primitive, that primitive is not lighted. More exactly, if a color command follows the last normal before a primitive is drawn, that primitive is not lighted.

`LMC_EMISSION`, `LMC_AMBIENT`, and `LMC_SPECULAR` update their corresponding material properties. In `LMC_NULL` mode, color commands are ignored while lighting is enabled.

If two-sided lighting is enabled and no `BACKMATERIAL` is bound, then fast updates to material properties affect both the front and back faces. If a `BACKMATERIAL` is bound, changes to material properties affect only the front face.

9.6.5 Transparency

Normally, materials are opaque. You can control the transparency of a material with the *alpha* component. Not all systems support alpha. Alpha information is ignored by systems that do not support it. In lighting, alpha is specified in the material definition.

```
static float mat[] = {
    ALPHA, .5,
    LMNULL
};
```

The ALPHA property is followed by one float. When used in conjunction with `blendfunction()`, a transparent effect can be achieved. The use of LMC_DIFFUSE or LMC_AD mode overrides the ALPHA material property with the alpha of the current color.

The ALPHA property works with two-sided lighting. You can achieve different front and back transparencies by binding material definitions with different ALPHA properties to MATERIAL and BACKMATERIAL. See Chapter 4 for more information on transparency.

9.6.6 Lighting Multiple GL Windows

You can use lighting in more than one GL window. The definitions that `lmdef()` creates and modifies are shared among all GL windows of a process. On the other hand, the material, light sources, and lighting model that `lmbind()` activates are specific to the GL window that was active at the time of the `lmbind()` call.

9.7 Lighting Performance

This section gives a general feeling for the performance implications of specific lighting features. The performance of a given feature might vary among the different graphics products as well as among different software releases on the same product. Nevertheless, there are some guidelines you can follow.

9.7.1 Restrictions on ModelView, Projection, and User-Defined Matrices

Many GL programs change the ModelView matrix using only `rot()`, `rotate()`, `scale()`, and `translate()`. They change the Projection matrix using only `ortho2()`, `ortho()`, `perspective()`, and `window()`. These calls all work with lighting; however, there are certain restrictions to be aware of.

These restrictions apply to the ModelView, Projection and user-defined transformation matrices when lighting is used:

- In some cases, normals transformed by the ModelView matrix do not maintain their unit length. The GL detects this condition of the matrix and automatically renormalizes the transformed normals. To avoid this extra calculation, use uniform scale factors; `scale(x,y,z)` only if $x = y = z$.
- Ensure that you use an *orthonormal* (see the `lmdef()` man page for a definition of orthonormal) matrix with `loadmatrix()` or `multmatrix()`. If you use `loadmatrix()` or `multmatrix()` to specify a non-orthonormal matrix in the ModelView matrix, you must use `nmode(NNORMALIZE)` to renormalize the normals properly.
- No projection components are allowed in the ModelView matrix. More specifically, the right-most column of the matrix must be `[0 0 0 1]`.
- Two restrictions apply to the Projection matrix:
  - No rotation is allowed.
  - The top two elements of the right column must be 0.

  IRIS-4D / VGX, VGXT and SkyWriter systems permit a general 4×4 Projection matrix, so the Projection matrix restrictions do not apply to these systems.

9.7.2 Computational Considerations

Certain lighting features and operations take longer to calculate than others. The following are areas where there are performance tradeoffs:

- Two-sided lighting takes extra computation, but its performance should be better than half the performance of one-sided lighting.
- Each additional light source takes extra computation. The more you use, the longer it takes to compute the color for a vertex.
- A time-consuming calculation that can occur in lighting is the square root operation. It is used for a local viewpoint, a point light source, and in normalizing non-unit-length normals. Any of these features adversely affects performance.
- GL calls other than `n`, `v`, `c`, `cpack()`, and `RGBcolor()` between `bgn*` and `end*` calls might incur a performance penalty. In fact, only a limited set of calls are allowed between a `bgn*` and `end*` call. See Appendix A for subroutines that can be called between a `bgn/end` sequence.
- Calling `lmcolor()` with an argument other than `LMC_COLOR` or `LMC_NULL` might incur a slight performance penalty.

Following are suggestions for getting the highest possible performance from lighting:

- Use an infinite viewpoint by setting `LOCALVIEWER` to 0, the default.
- Use a single infinite light source.
- Use the default `lmcolor(LMC_COLOR)` or use `lmcolor(LMC_NULL)`.
- Use the default `nmode(NAUTO)`.
- Take advantage of drawing primitives that share vertices for lines or polygons; for instance, use `bgntmesh()` or `bgnqstrip` for drawing polygons.
- On IRIS-4D / VGX and SkyWriter systems, using less than one normal per vertex (such as one normal per polygon) does not reduce the lighting calculation specifically. However, it does reduce the amount of data transferred to the Geometry Engines.

9.8 Color Map Lighting

You can do lighting in color map mode, but that method is generally designed for systems without enough bitplanes to support RGB mode. On graphics systems with enough bitplanes, RGB mode lighting is recommended.

Color map lighting generates a pseudo-intensity, which is a function of the direction to the light source and the direction to the viewpoint. This pseudo-intensity is mapped to a color map value. A well-chosen range of color map values gives a reasonable lighting effect. You can represent multiple materials by creating a color map range for each material. Color map lighting is enabled when both lighting and color map mode are enabled.

Color map lighting has inherent limitations, and many of the advanced lighting features are not supported. Color map lighting recognizes only a limited set of properties, described here.

A material definition in color map mode uses the `COLORINDEXES` and `SHININESS` properties:

```
static float mat[] = {
    COLORINDEXES, 512, 576, 639,
    SHININESS, 5,
    LMNULL
};
```

This property is followed by three floats, representing an ambient index, a diffuse index, and a specular index. These indices should correspond to appropriate values in the color map. Lighting produces values that range from the ambient index to the specular index. Lighting generates the ambient index when there is no diffuse or specular reflection. It generates the diffuse index when the diffuse reflection is at a maximum, but there is no specular reflection. It generates the specular index when the specular reflection is at a maximum. The specular index must be greater than or equal to the diffuse index, which must in turn be greater than or equal to the ambient index. All other material properties are ignored.

A light source definition in color map mode uses only the `POSITION` property:

```
static float lt[] = {
    POSITION, 0, 0, 1, 0,
    LMNULL
};
```

Each light source contributes an intensity proportional to its LCOLOR values, so that the scene doesn't get washed out when more that one light source is used.

IRIS Indigo systems allow local lights in color map mode. On other systems, only infinite light source positions are allowed. This requires that you set the *w* component of `POSITION` to 0. All other light source properties are ignored. You can specify zero or more light sources.

A lighting model definition in color map mode uses only the `LOCALVIEWER` property:

```
static float lm[] = {
    LOCALVIEWER, 0,
    LMNULL
};
```

Only an infinite viewpoint is allowed. `LOCALVIEWER` can be set only to 0, which is the default.

9.9 Sample Lighting Program

The following sample program, *cylinder2.c*, demonstrates GL lighting.

```
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>

Matrix Identity = { 1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1 };

float mat[] = {
    AMBIENT, .1, .1, .1,
    DIFFUSE, 0, .369, .165,
    SPECULAR, .5, .5, .5,
    SHININESS, 10,
    LMNULL,
};

static float lm[] = {
    AMBIENT, .1, .1, .1,
    LOCALVIEWER, 1,
    LMNULL
};
```

```
static float lt[] = {
   LCOLOR, 1, 1, 1,
   POSITION, 0, 0, 1, 0,
   LMNULL
};

main()
{
long xorigin, yorigin, xsize, ysize;
float rx, ry;
short val;

   winopen("cylinder");
   qdevice(ESCKEY);
   getorigin(&xorigin, &yorigin);
   getsize(&xsize, &ysize);
   RGBmode();
   doublebuffer();
   gconfig();
   lsetdepth(getgdesc(GD_ZMIN), getgdesc(GD_ZMAX));
   zbuffer(1);
   mmode(MVIEWING);
   loadmatrix(Identity);
   perspective(600, xsize/(float)ysize, .25, 15.0);
   lmdef(DEFMATERIAL, 1, 0, mat);
   lmdef(DEFLIGHT, 1, 0, lt);
   lmdef(DEFLMODEL, 1, 0, lm);
   lmbind(MATERIAL, 1);
   lmbind(LMODEL, 1);
   lmbind(LIGHT0, 1);
   translate(0, 0, -4);

   while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
      ry = 300 *
(2.0*(getvaluator(MOUSEX)-xorigin)/xsize-1.0);
      rx = -300 *
(2.0*(getvaluator(MOUSEY)-yorigin)/ysize-1.0);
      czclear(0x404040, getgdesc(GD_ZMAX));
      pushmatrix();
      rot(ry, 'y');
      rot(rx, 'x');
      drawcyl();
      popmatrix();
      swapbuffers();
   }
}
```

```
drawcyl()
{
double dy = .2;
double theta, dtheta = 2*M_PI/20;
double x, y, z;
float n[3], v[3];
int i, j;

   for (i = 0, y = -1; i < 10; i++, y += dy) {
      bgntmesh();
      for (j = 0, theta = 0; j <= 20; j++, theta += dtheta) {
         if (j == 20) theta = 0;
         x = cos(theta);
         z = sin(theta);
         n[0] = x; n[1] = 0; n[2] = z;
         n3f(n);
         v[0] = x; v[1] = y; v[2] = z;
         v3f(v);
         v[1] = y + dy;
         v3f(v);
      }
      endtmesh();
   }
}
```