Chapter 8

OpenGL – A 3D Graphics API

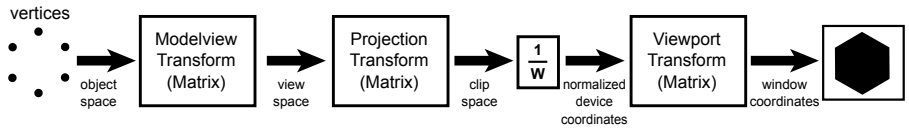
8.1 Pipeline architecture

You can think of OpenGL as a pipeline, with processing stations along the way, that takes in the vertices of your model (i.e points in 3D object space) and transforms them into screen coordinates. The basic pipeline is organized as shown below.

The 3D coordinates of vertices enter on the left, and undergo a series of transformations and operations. The operations include:

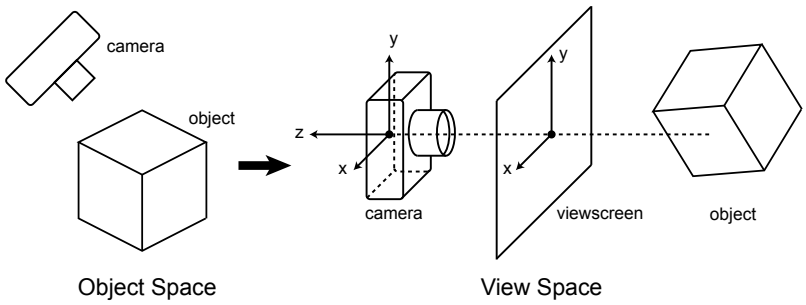
1. Clipping away vertices outside of the view volume,
2. Determining which surfaces are visible and which are hidden (occluded) behind other surfaces,
3. *Rasterizing* polygons (i.e. filling in the space between polygon edges), and drawing lines,
4. Shading and texturing of the pixels making up the polygonal faces,
5. Performing a number of image processing operations such as bluring and compositing.

The transformations are described below.

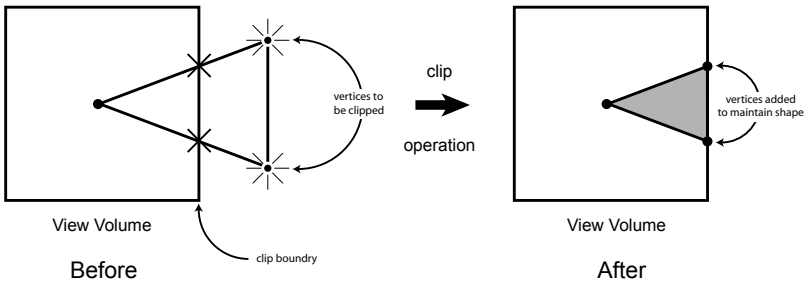


8.2 Transformations in the OpenGL pipeline

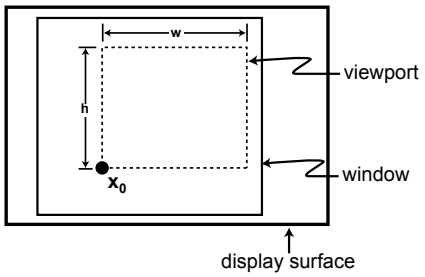
The *Modelview Matrix* transforms object coordinates into the viewing (or camera) coordinate system, where the camera viewpoint is at the origin $(0, 0, 0)$, the view direction is aligned with the negative z axis, and the view screen is perpendicular to the view direction, so that the x and y axes are parallel to the plane of the view screen. An example Modelview transform is shown in the figure below. The steps implied in the transform are first translating the camera viewpoint to the origin and then rotating the camera to properly align it with the x, y, and z coordinate axes. Note: This is equivalent to translating the entire scene in the opposite direction, and then rotating the scene about the origin in the opposite direction, which is what is actually done in OpenGL (i.e. the vertices of the model are transformed).



The *Projection Matrix* transforms the vertices in view coordinates into the canonical view volume (a cube of sides $2 \times 2 \times 2$, centered at the origin, and aligned with the 3 coordinate axes). Typically, this will be either by an orthographic projection or a perspective projection. This transform includes multiplication by the projection transformation matrix followed by a normalization of each vertex, calculated by dividing each vertex by its own w coordinate. We say that the vertices thus transformed are in *Normalized Device Coordinates* as they are now independent of any display or camera parameters. During this process, OpenGL discards any vertices not in the view volume by clipping them to the sides of the view volume, as shown below. The integrity of edges and polygon faces is maintained by adding a new vertex wherever the view volume clips an edge. Thus, after conversion to Normalized Device Coordinates, and clipping, all remaining vertices have coordinates in the range $-1 \leq x, y, z \leq 1$.



Finally, the *Viewport Matrix* transforms vertices into window coordinates. After this transformation, everything is in pixel coordinates relative to the lower left corner of the window on the display. As shown to the right, the viewport is a rectangular region on the window defined by the position $\mathbf{x}_0$ of its lower left corner, its width w and its height h. The Viewport Matrix performs a scale of the canonical view volume (in Normalized Device Coordinates) by $w/2$ in the horizontal direction, and by $h/2$ in the vertical direction, transforming the front face of the volume from a 2×2 square to a $w \times h$ rectangle. This is followed by a translation of the front lower-lefthand corner $(-w/2, -h/2, -1)$ to the position $\mathbf{x}_0$. In this space, all the rasterization and shading operations are performed, and each resulting pixel is drawn to the display.



8.3 OpenGL API calls affecting the pipeline transforms

The OpenGL API defines a set of procedure calls that are used to update the three matrices in the pipeline. These are described below.

8.3.1 Modelview Matrix

One selects which matrix is to be affected by matrix operations in OpenGL, by a call to the procedure `glMatrixMode()`. This procedure takes one parameter, which is a symbol indicating which matrix to affect. Once this call is made,

the indicated matrix will remain the active matrix until another call is made to `glMatrixMode()`.

The Modelview Matrix is selected via a call to

```
glMatrixMode(GL_MODELVIEW);
```

The Modelview Matrix is used both to transform the vertices of the various objects in the scene from their own object coordinates into scene (world) coordinates, and to position the camera within the scene. The commands typically used to affect the Modelview matrix are:

`glLoadIdentity();` – to initialize the matrix to the identity matrix

`glTranslatef(`$\triangle x$`,` $\triangle y$`,` $\triangle z$`);` – to translate by the vector $\begin{bmatrix} \triangle x \\ \triangle y \\ \triangle z \end{bmatrix}$,

`glScalef(`s_x`,` s_y`,` s_z`);` – to scale by the scale factors s_x, s_y, and s_z,

`glRotatef(`θ`,` u_x`,` u_y`,` u_z`);` – to rotate θ degrees about the axis $\mathbf{u} = \begin{bmatrix} u_x \\ u_y \\ u_x \end{bmatrix}$.

Note that the `f` following `glTranslate_()`, `glScale_()` or `glRotate_()` indicates that parameters are floats. To use doubles, use `d` or to use integers use `i` instead.

The translate, scale and rotate calls each create a matrix to do the indicated transform, and multiply it onto the right hand side of the currently selected matrix (usually the Modelview Matrix). Therefore, the sequence:

`glLoadIdentity();` $\Longrightarrow M = I$
`glRotatef(30, 1, 0, 0);` $\Longrightarrow M = IR_{30,x}$
`glTranslatef(-5, -5, -5);` $\Longrightarrow M = IR_{30,x}T_{-5,-5,-5}$

results in a Modelview matrix M, which when multiplied by a vertex $\mathbf{x}$, yields a transformed vertex

$$\mathbf{x}' = M\mathbf{x}$$

that is first translated by -5 in each direction, and then rotated by 30° about the x axis.

8.3.2 Projection Matrix

The call

```
glMatrixMode(GL_PROJECTION);
```

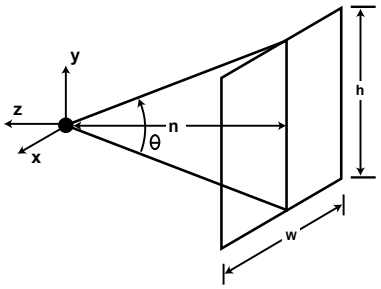
tells OpenGL that calls that affect matrices should be applied to the Projection Matrix. The calls that are especially applicable to the Projection Matrix are:

`GlLoadIdentity();` – initializes the matrix to the identity matrix.

`GlOrtho(l, r, b, t, n, f);` – multiplies an orthographic projection matrix onto the right side of the matrix.

`GlFrustum(l, r, b, t, n, f);` – multiplies a perspective projection matrix onto the right side of the matrix.

In these calls, the parameters `l`, `r`, `b`, `t` specify the left, right, bottom, and top coordinates of the viewscreen; while `n` and `f` are the distances, along the view direction, of the near and far clipping planes. The near clipping plane corresponds with the viewplane, and `n` corresponds with the focal length of the camera. The far clipping plane is a distance, beyond which, the camera ignores anything in the scene.



Alternatively, the call

```
gluPerspective(theta, aspect, n, f);
```

can be used in place of `glFrustum()` to specify a perspective projection. The parameter `theta` is the vertical camera viewing angle in degrees, and `aspect` is the aspect ratio of the window (i.e. its width divided by its height). The figure to the left shows the relationship between the parameters and the perspective view volume. If we let α stand for `aspect` and θ stand for `theta` the relationships between the width w and height h of the window are given by

$$h = 2n \sin \theta/2,$$
$$w = \alpha h.$$

.

For example, a camera with view screen centered along the view direction would have its projection transform set up by:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
glOrtho(-w/2, w/2, -h/2, h/2, n.  f);
```

for orthographic projection, or

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
glFrustum(-w/2, w/2, -h/2, h/2, n, f);
```

for perspective projection.

If OpenGL is being used strictly for 2D viewing, the call

```
gluOrtho2d(l, r, b, t);
```

can be used to create an orthographic projection, whose view plane is at the origin and that is of zero depth, so that all z coordinates must be 0 for vertices to be visible (i.e only the x, y coordinates matter).

8.3.3 Viewport

The viewport is set by a single call to

```
glViewport(x0, y0, w, h);
```

This causes the viewport matrix to be constructed, scaling the normalized device coordinates so that the width becomes `w` and the height becomes `h`, and translating the lower left corner to (`x0`, `y0`, 0).

Notes from
OpenGL Programming Guide, 2nd Edition, OpenGL v1.1
OpenGL Architecture Review Board, Woo, Neider, Davis
Addison Wesley 1997

Specifying Vertices

With OpenGL, all geometric objects are ultimately described as an ordered set of vertices. You use the **glVertex*()** command to specify a vertex.

void **glVertex**{234}{sifd}[v](*TYPE coords*);

Specifies a vertex for use in describing a geometric object. You can supply up to four coordinates (*x*, *y*, *z*, *w*) for a particular vertex or as few as two (*x*, *y*) by selecting the appropriate version of the command. If you use a version that doesn't explicitly specify *z* or *w*, *z* is understood to be 0 and *w* is understood to be 1. Calls to **glVertex*()** are only effective between a **glBegin()** and **glEnd()** pair.

Example 2-2 provides some examples of using **glVertex*()**.

Example 2-2 Legal Uses of glVertex*()

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);

GLdouble dvect[3] = {5.0, 9.0, 1992.0};
glVertex3dv(dvect);
```

The first example represents a vertex with three-dimensional coordinates (2, 3, 0). (Remember that if it isn't specified, the *z* coordinate is understood to be 0.) The coordinates in the second example are (0.0, 0.0, 3.1415926535898) (double-precision floating-point numbers). The third example represents the vertex with three-dimensional coordinates (1.15, 0.5, –1.1). (Remember that the *x*, *y*, and *z* coordinates are eventually divided

by the *w* coordinate.) In the final example, *dvect* is a pointer to an array of three double-precision floating-point numbers.

On some machines, the vector form of **glVertex*()** is more efficient, since only a single parameter needs to be passed to the graphics subsystem. Special hardware might be able to send a whole series of coordinates in a single batch. If your machine is like this, it's to your advantage to arrange your data so that the vertex coordinates are packed sequentially in memory. In this case, there may be some gain in performance by using the vertex array operations of OpenGL. (See "Vertex Arrays" on page 65.)

OpenGL Geometric Drawing Primitives

Now that you've seen how to specify vertices, you still need to know how to tell OpenGL to create a set of points, a line, or a polygon from those vertices. To do this, you bracket each set of vertices between a call to **glBegin()** and a call to **glEnd()**. The argument passed to **glBegin()** determines what sort of geometric primitive is constructed from the vertices. For example, Example 2-3 specifies the vertices for the polygon shown in Figure 2-6.

Example 2-3 Filled Polygon

```
glBegin(GL_POLYGON);
   glVertex2f(0.0, 0.0);
   glVertex2f(0.0, 3.0);
   glVertex2f(4.0, 3.0);
   glVertex2f(6.0, 1.5);
   glVertex2f(4.0, 0.0);
glEnd();
```

Figure 2-6 Drawing a Polygon or a Set of Points

If you had used GL_POINTS instead of GL_POLYGON, the primitive would have been simply the five points shown in Figure 2-6. Table 2-2 in the following function summary for **glBegin()** lists the ten possible arguments and the corresponding type of primitive.

void **glBegin**(GLenum *mode*);

Marks the beginning of a vertex-data list that describes a geometric primitive. The type of primitive is indicated by *mode*, which can be any of the values shown in Table 2-2.

Value	Meaning
GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of a simple, convex polygon

Table 2-2 Geometric Primitive Names and Meanings

void **glEnd**(void);

Marks the end of a vertex-data list.

Figure 2-7 shows examples of all the geometric primitives listed in Table 2-2. The paragraphs that follow the figure describe the pixels that are drawn for each of the objects. Note that in addition to points, several types of lines and polygons are defined. Obviously, you can find many ways to draw the same primitive. The method you choose depends on your vertex data.

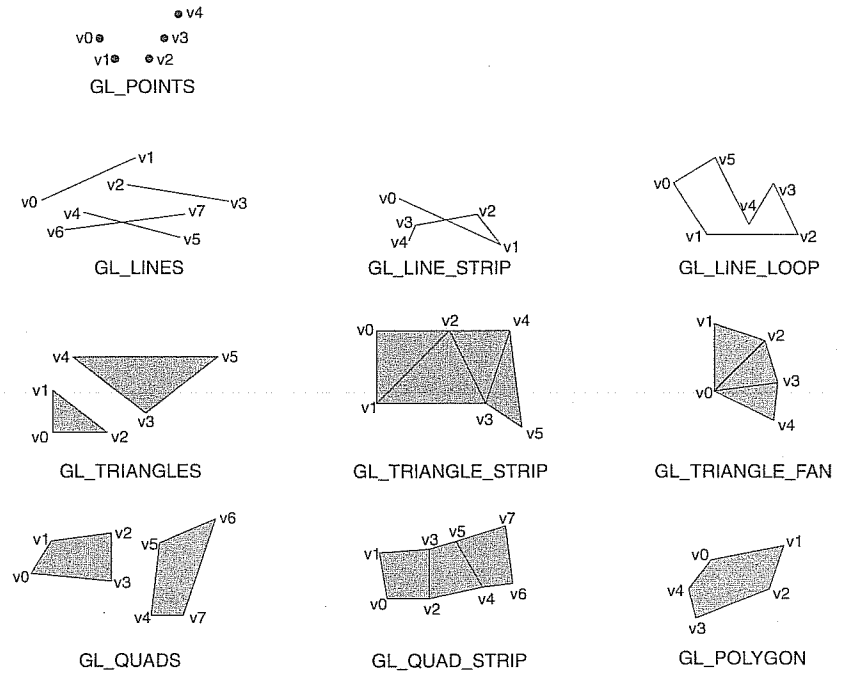


Figure 2-7 Geometric Primitive Types

As you read the following descriptions, assume that n vertices (v_0, v_1, v_2, ... , v_{n-1}) are described between a **glBegin()** and **glEnd()** pair.

GL_POINTS	Draws a point at each of the n vertices.
GL_LINES	Draws a series of unconnected line segments. Segments are drawn between v_0 and v_1, between v_2 and v_3, and so on. If n is odd, the last segment is drawn between v_{n-3} and v_{n-2}, and v_{n-1} is ignored.
GL_LINE_STRIP	Draws a line segment from v_0 to v_1, then from v_1 to v_2, and so on, finally drawing the segment from v_{n-2} to v_{n-1}. Thus, a total of $n-1$ line segments are drawn. Nothing is drawn unless n is larger than 1. There are no restrictions on the vertices describing a line strip (or a line loop); the lines can intersect arbitrarily.
GL_LINE_LOOP	Same as GL_LINE_STRIP, except that a final line segment is drawn from v_{n-1} to v_0, completing a loop.

GL_TRIANGLES Draws a series of triangles (three-sided polygons) using vertices v_0, v_1, v_2, then v_3, v_4, v_5, and so on. If *n* isn't an exact multiple of 3, the final one or two vertices are ignored.

GL_TRIANGLE_STRIP
Draws a series of triangles (three-sided polygons) using vertices v_0, v_1, v_2, then v_2, v_1, v_3 (note the order), then v_2, v_3, v_4, and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface. Preserving the orientation is important for some operations, such as culling. (See "Reversing and Culling Polygon Faces" on page 56) *n* must be at least 3 for anything to be drawn.

GL_TRIANGLE_FAN
Same as GL_TRIANGLE_STRIP, except that the vertices are v_0, v_1, v_2, then v_0, v_2, v_3, then v_0, v_3, v_4, and so on (see Figure 2-7).

GL_QUADS Draws a series of quadrilaterals (four-sided polygons) using vertices v_0, v_1, v_2, v_3, then v_4, v_5, v_6, v_7, and so on. If *n* isn't a multiple of 4, the final one, two, or three vertices are ignored.

GL_QUAD_STRIP Draws a series of quadrilaterals (four-sided polygons) beginning with v_0, v_1, v_3, v_2, then v_2, v_3, v_5, v_4, then v_4, v_5, v_7, v_6, and so on (see Figure 2-7). *n* must be at least 4 before anything is drawn. If *n* is odd, the final vertex is ignored.

GL_POLYGON Draws a polygon using the points v_0, ... , v_{n-1} as vertices. *n* must be at least 3, or nothing is drawn. In addition, the polygon specified must not intersect itself and must be convex. If the vertices don't satisfy these conditions, the results are unpredictable.

Restrictions on Using glBegin() and glEnd()

The most important information about vertices is their coordinates, which are specified by the **glVertex***() command. You can also supply additional vertex-specific data for each vertex—a color, a normal vector, texture coordinates, or any combination of these—using special commands. In

addition, a few other commands are valid between a **glBegin()** and **glEnd()** pair. Table 2-3 contains a complete list of such valid commands.

Command	Purpose of Command	Reference
glVertex*()	set vertex coordinates	Chapter 2
glColor*()	set current color	Chapter 4
glIndex*()	set current color index	Chapter 4
glNormal*()	set normal vector coordinates	Chapter 2
glTexCoord*()	set texture coordinates	Chapter 9
glEdgeFlag*()	control drawing of edges	Chapter 2
glMaterial*()	set material properties	Chapter 5
glArrayElement()	extract vertex array data	Chapter 2
glEvalCoord*(), glEvalPoint*()	generate coordinates	Chapter 12
glCallList(), glCallLists()	execute display list(s)	Chapter 7

Table 2-3 Valid Commands between glBegin() and glEnd()

No other OpenGL commands are valid between a **glBegin()** and **glEnd()** pair, and making most other OpenGL calls generates an error. Some vertex array commands, such as **glEnableClientState()** and **glVertexPointer()**, when called between **glBegin()** and **glEnd()**, have undefined behavior but do not necessarily generate an error. (Also, routines related to OpenGL, such as **glX*()** routines have undefined behavior between **glBegin()** and **glEnd()**.) These cases should be avoided, and debugging them may be more difficult.

Note, however, that only OpenGL commands are restricted; you can certainly include other programming-language constructs (except for calls, such as the aforementioned **glX*()** routines). For example, Example 2-4 draws an outlined circle.

Example 2-4 Other Constructs between glBegin() and glEnd()

```
#define PI 3.1415926535898
GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
```

```
for (i = 0; i < circle_points; i++) {
   angle = 2*PI*i/circle_points;
   glVertex2f(cos(angle), sin(angle));
}
glEnd();
```

Note: This example isn't the most efficient way to draw a circle, especially if you intend to do it repeatedly. The graphics commands used are typically very fast, but this code calculates an angle and calls the **sin()** and **cos()** routines for each vertex; in addition, there's the loop overhead. (Another way to calculate the vertices of a circle is to use a GLU routine; see "Quadrics: Rendering Spheres, Cylinders, and Disks" on page 428.) If you need to draw lots of circles, calculate the coordinates of the vertices once and save them in an array and create a display list (see Chapter 7), or use vertex arrays to render them.

Unless they are being compiled into a display list, all **glVertex*()** commands should appear between some **glBegin()** and **glEnd()** combination. (If they appear elsewhere, they don't accomplish anything.) If they appear in a display list, they are executed only if they appear between a **glBegin()** and a **glEnd()**. (See Chapter 7 for more information about display lists.)

Although many commands are allowed between **glBegin()** and **glEnd()**, vertices are generated only when a **glVertex*()** command is issued. At the moment **glVertex*()** is called, OpenGL assigns the resulting vertex the current color, texture coordinates, normal vector information, and so on. To see this, look at the following code sequence. The first point is drawn in red, and the second and third ones in blue, despite the extra color commands.

```
glBegin(GL_POINTS);
   glColor3f(0.0, 1.0, 0.0);                    /* green */
   glColor3f(1.0, 0.0, 0.0);                    /* red */
   glVertex(...);
   glColor3f(1.0, 1.0, 0.0);                    /* yellow */
   glColor3f(0.0, 0.0, 1.0);                    /* blue */
   glVertex(...);
   glVertex(...);
glEnd();
```

You can use any combination of the 24 versions of the **glVertex*()** command between **glBegin()** and **glEnd()**, although in real applications all the calls in any particular instance tend to be of the same form. If your vertex-data specification is consistent and repetitive (for example, **glColor***, **glVertex***, **glColor***, **glVertex***,...), you may enhance your program's performance by using vertex arrays. (See "Vertex Arrays" on page 65.)

Creating Light Sources

Light sources have a number of properties, such as color, position, and direction. The following sections explain how to control these properties and what the resulting light looks like. The command used to specify all properties of lights is **glLight***(); it takes three arguments: to identify the light whose property is being specified, the property, and the desired value for that property.

void **glLight**{if}(GLenum *light*, GLenum *pname*, *TYPE param*);
void **glLight**{if}v(GLenum *light*, GLenum *pname*, *TYPE *param*);

Creates the light specified by *light*, which can be GL_LIGHT0, GL_LIGHT1, ... , or GL_LIGHT7. The characteristic of the light being set is defined by *pname*, which specifies a named parameter (see Table 5-1). *param* indicates the values to which the *pname* characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued light characteristics.

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA intensity of light
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w) position of light
GL_SPOT_DIRECTION	(0.0, 0.0, −1.0)	(x, y, z) direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight exponent
GL_SPOT_CUTOFF	180.0	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor

Table 5-1 Default Values for pname Parameter of glLight*()

OpenGL prog. guide pg 180

Parameter Name	Default Value	Meaning
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation factor

Table 5-1 Default Values for pname Parameter of glLight*() (continued)

Note: The default values listed for GL_DIFFUSE and GL_SPECULAR in Table 5-1 apply only to GL_LIGHT0. For other lights, the default value is (0.0, 0.0, 0.0, 1.0) for both GL_DIFFUSE and GL_SPECULAR.

Example 5-2 shows how to use **glLight*()**:

Example 5-2 Defining Colors and Position for a Light Source

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

As you can see, arrays are defined for the parameter values, and **glLightfv()** is called repeatedly to set the various parameters. In this example, the first three calls to **glLightfv()** are superfluous, since they're being used to specify the default values for the GL_AMBIENT, GL_DIFFUSE, and GL_SPECULAR parameters.

Note: Remember to turn on each light with **glEnable()**. (See "Enabling Lighting" on page 195 for more information about how to do this.)

All the parameters for **glLight*()** and their possible values are explained in the following sections. These parameters interact with those that define the overall lighting model for a particular scene and an object's material properties. (See "Selecting a Lighting Model" on page 192 and "Defining Material Properties" on page 195 for more information about these two topics. "The Mathematics of Lighting" on page 205 explains how all these parameters interact mathematically.)

OpenGL Prog. Guide

Selecting Lighting Model

- Whether lighting calculations should be performed differently for both the front and back faces of objects

This section explains how to specify a lighting model. It also discusses how to enable lighting—that is, how to tell OpenGL that you want lighting calculations performed.

The command used to specify all properties of the lighting model is **glLightModel*()**. **glLightModel*()** has two arguments: the lighting model property and the desired value for that property.

void **glLightModel**{if}(GLenum *pname*, *TYPE param*);
void **glLightModel**{if}**v**(GLenum *pname*, *TYPE *param*);

Sets properties of the lighting model. The characteristic of the lighting model being set is defined by *pname*, which specifies a named parameter (see Table 5-1). *param* indicates the values to which the *pname* characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued lighting model characteristics, not for GL_LIGHT_MODEL_AMBIENT.

Parameter Name	Default Value	Meaning
GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient RGBA intensity of the entire scene
GL_LIGHT_MODEL_LOCAL_VIEWER	0.0 or GL_FALSE	how specular reflection angles are computed
GL_LIGHT_MODEL_TWO_SIDE	0.0 or GL_FALSE	choose between one-sided or two-sided lighting

Table 5-2 Default Values for pname Parameter of glLightModel*()

Global Ambient Light

As discussed earlier, each light source can contribute ambient light to a scene. In addition, there can be other ambient light that's not from any particular source. To specify the RGBA intensity of such global ambient light, use the GL_LIGHT_MODEL_AMBIENT parameter as follows:

OpenGL Prog. Guide

Creating Materials in OpenGL

Mathematics of Lighting" on page 205 for the equations used in the lighting and material-property calculations.) Most of the material properties are conceptually similar to ones you've already used to create light sources. The mechanism for setting them is similar, except that the command used is called **glMaterial*()**.

void **glMaterial**{if}(GLenum *face*, GLenum *pname*, *TYPE param*);
void **glMaterial**{if}v(GLenum *face*, GLenum *pname*, *TYPE *param*);

Specifies a current material property for use in lighting calculations. *face* can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate which face of the object the material should be applied to. The particular material property being set is identified by *pname* and the desired values for that property are given by *param*, which is either a pointer to a group of values (if the vector version is used) or the actual value (if the nonvector version is used). The nonvector version works only for setting GL_SHININESS. The possible values for pname are shown in Table 5-3. Note that GL_AMBIENT_AND_DIFFUSE allows you to set both the ambient and diffuse material colors simultaneously to the same RGBA value.

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient color of material
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
GL_AMBIENT_AND_DIFFUSE		ambient and diffuse color of material
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	specular color of material
GL_SHININESS	0.0	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	emissive color of material
GL_COLOR_INDEXES	(0,1,1)	ambient, diffuse, and specular color indices

Table 5-3 Default Values for pname Parameter of glMaterial*()

As discussed in "Selecting a Lighting Model" on page 192, you can choose to have lighting calculations performed differently for the front- and back-facing polygons of objects. If the back faces might indeed be seen, you can supply different material properties for the front and the back surfaces

OpenGL Prog Guide