Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Clear the window to an arbitrary color
• Force any pending drawing to complete
• Draw with any geometric primitive—point, line, or polygon—in two or three dimensions
• Turn states on and off and query state variables
• Control the display of geometric primitives—for example, draw dashed lines or outlined polygons
• Specify normal vectors at appropriate points on the surfaces of solid objects
• Use vertex arrays and buffer objects to store and access geometric data with fewer function calls
• Save and restore several state variables at once
Although you can draw complex and interesting pictures using OpenGL, they’re all constructed from a small number of primitive graphical items. This shouldn’t be too surprising—look at what Leonardo da Vinci accomplished with just pencils and paintbrushes.

At the highest level of abstraction, there are three basic drawing operations: clearing the window, drawing a geometric object, and drawing a raster object. Raster objects, which include such things as two-dimensional images, bitmaps, and character fonts, are covered in Chapter 8. In this chapter, you learn how to clear the screen and draw geometric objects, including points, straight lines, and flat polygons.

You might think to yourself, “Wait a minute. I’ve seen lots of computer graphics in movies and on television, and there are plenty of beautifully shaded curved lines and surfaces. How are those drawn if OpenGL can draw only straight lines and flat polygons?” Even the image on the cover of this book includes a round table and objects on the table that have curved surfaces. It turns out that all the curved lines and surfaces you’ve seen are approximated by large numbers of little flat polygons or straight lines, in much the same way that the globe on the cover is constructed from a large set of rectangular blocks. The globe doesn’t appear to have a smooth surface because the blocks are relatively large compared with the globe. Later in this chapter, we show you how to construct curved lines and surfaces from lots of small geometric primitives.

This chapter has the following major sections:

• “A Drawing Survival Kit” explains how to clear the window and force drawing to be completed. It also gives you basic information about controlling the colors of geometric objects and describing a coordinate system.

• “Describing Points, Lines, and Polygons” shows you the set of primitive geometric objects and how to draw them.

• “Basic State Management” describes how to turn on and off some states (modes) and query state variables.

• “Displaying Points, Lines, and Polygons” explains what control you have over the details of how primitives are drawn—for example, what diameters points have, whether lines are solid or dashed, and whether polygons are outlined or filled.

• “Normal Vectors” discusses how to specify normal vectors for geometric objects and (briefly) what these vectors are for.
“Vertex Arrays” shows you how to put large amounts of geometric data into just a few arrays and how, with only a few function calls, to render the geometry it describes. Reducing function calls may increase the efficiency and performance of rendering.

“Buffer Objects” details how to use server-side memory buffers to store vertex array data for more efficient geometric rendering.

“Vertex-Array Objects” expands the discussions of vertex arrays and buffer objects by describing how to efficiently change among sets of vertex arrays.

“Attribute Groups” reveals how to query the current value of state variables and how to save and restore several related state values all at once.

“Some Hints for Building Polygonal Models of Surfaces” explores the issues and techniques involved in constructing polygonal approximations to surfaces.

One thing to keep in mind as you read the rest of this chapter is that with OpenGL, unless you specify otherwise, every time you issue a drawing command, the specified object is drawn. This might seem obvious, but in some systems, you first make a list of things to draw. When your list is complete, you tell the graphics hardware to draw the items in the list. The first style is called immediate-mode graphics and is the default OpenGL style. In addition to using immediate mode, you can choose to save some commands in a list (called a display list) for later drawing. Immediate-mode graphics are typically easier to program, but display lists are often more efficient. Chapter 7 tells you how to use display lists and why you might want to use them.

Version 1.1 of OpenGL introduced vertex arrays.

In Version 1.2, scaling of surface normals (GL_RESCALE_NORMAL) was added to OpenGL. Also, `glDrawRangeElements()` supplemented vertex arrays.

Version 1.3 marked the initial support for texture coordinates for multiple texture units in the OpenGL core feature set. Previously, multitexturing had been an optional OpenGL extension.

In Version 1.4, fog coordinates and secondary colors may be stored in vertex arrays, and the commands `glMultiDrawArrays()` and `glMultiDrawElements()` may be used to render primitives from vertex arrays.

In Version 1.5, vertex arrays may be stored in buffer objects that may be able to use server memory for storing arrays and potentially accelerating their rendering.
Version 3.0 added support for vertex array objects, allowing all of the state related to vertex arrays to be bundled and activated with a single call. This, in turn, makes switching between sets of vertex arrays simpler and faster.

Version 3.1 removed most of the immediate-mode routines and added the primitive restart index, which allows you to render multiple primitives (of the same type) with a single drawing call.

**A Drawing Survival Kit**

This section explains how to clear the window in preparation for drawing, set the colors of objects that are to be drawn, and force drawing to be completed. None of these subjects has anything to do with geometric objects in a direct way, but any program that draws geometric objects has to deal with these issues.

**Clearing the Window**

Drawing on a computer screen is different from drawing on paper in that the paper starts out white, and all you have to do is draw the picture. On a computer, the memory holding the picture is usually filled with the last picture you drew, so you typically need to clear it to some background color before you start to draw the new scene. The color you use for the background depends on the application. For a word processor, you might clear to white (the color of the paper) before you begin to draw the text. If you’re drawing a view from a spaceship, you clear to the black of space before beginning to draw the stars, planets, and alien spaceships. Sometimes you might not need to clear the screen at all; for example, if the image is the inside of a room, the entire graphics window is covered as you draw all the walls.

At this point, you might be wondering why we keep talking about clearing the window—why not just draw a rectangle of the appropriate color that’s large enough to cover the entire window? First, a special command to clear a window can be much more efficient than a general-purpose drawing command. In addition, as you’ll see in Chapter 3, OpenGL allows you to set the coordinate system, viewing position, and viewing direction arbitrarily, so it might be difficult to figure out an appropriate size and location for a window-clearing rectangle. Finally, on many machines, the graphics
hardware consists of multiple buffers in addition to the buffer containing colors of the pixels that are displayed. These other buffers must be cleared from time to time, and it’s convenient to have a single command that can clear any combination of them. (See Chapter 10 for a discussion of all the possible buffers.)

You must also know how the colors of pixels are stored in the graphics hardware known as bitplanes. There are two methods of storage. Either the red, green, blue, and alpha (RGBA) values of a pixel can be directly stored in the bitplanes, or a single index value that references a color lookup table is stored. RGBA color-display mode is more commonly used, so most of the examples in this book use it. (See Chapter 4 for more information about both display modes.) You can safely ignore all references to alpha values until Chapter 6.

As an example, these lines of code clear an RGBA mode window to black:

```cpp
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

The first line sets the clearing color to black, and the next command clears the entire window to the current clearing color. The single parameter to `glClear()` indicates which buffers are to be cleared. In this case, the program clears only the color buffer, where the image displayed on the screen is kept. Typically, you set the clearing color once, early in your application, and then you clear the buffers as often as necessary. OpenGL keeps track of the current clearing color as a state variable, rather than requiring you to specify it each time a buffer is cleared.

Chapter 4 and Chapter 10 discuss how other buffers are used. For now, all you need to know is that clearing them is simple. For example, to clear both the color buffer and the depth buffer, you would use the following sequence of commands:

```cpp
glClearColor(0.0, 0.0, 0.0, 0.0);
glClearDepth(1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

In this case, the call to `glClearColor()` is the same as before, the `glClearDepth()` command specifies the value to which every pixel of the depth buffer is to be set, and the parameter to the `glClear()` command now consists of the bitwise logical OR of all the buffers to be cleared. The following summary of `glClear()` includes a table that lists the buffers that can be cleared, their names, and the chapter in which each type of buffer is discussed.
Before issuing a command to clear multiple buffers, you have to set the values to which each buffer is to be cleared if you want something other than the default RGBA color, depth value, accumulation color, and stencil index. In addition to the `glClearColor()` and `glClearDepth()` commands that set the current values for clearing the color and depth buffers, `glClearIndex()`, `glClearAccum()`, and `glClearStencil()` specify the color index, accumulation color, and stencil index used to clear the corresponding buffers. (See Chapter 4 and Chapter 10 for descriptions of these buffers and their uses.)

OpenGL allows you to specify multiple buffers because clearing is generally a slow operation, as every pixel in the window (possibly millions) is touched, and some graphics hardware allows sets of buffers to be cleared simultaneously. Hardware that doesn’t support simultaneous clears performs them sequentially. The difference between

```c
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

void `glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);`

Sets the current clearing color for use in clearing color buffers in RGBA mode. (See Chapter 4 for more information on RGBA mode.) The red, green, blue, and alpha values are clamped if necessary to the range [0, 1]. The default clearing color is (0, 0, 0, 0), which is black.

void `glClear(GLbitfield mask);`

Clears the specified buffers to their current clearing values. The mask argument is a bitwise logical OR combination of the values listed in Table 2-1.

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color buffer</td>
<td>GL_COLOR_BUFFER_BIT</td>
<td>Chapter 4</td>
</tr>
<tr>
<td>Depth buffer</td>
<td>GL_DEPTH_BUFFER_BIT</td>
<td>Chapter 10</td>
</tr>
<tr>
<td>Accumulation buffer</td>
<td>GL_ACCUM_BUFFER_BIT</td>
<td>Chapter 10</td>
</tr>
<tr>
<td>Stencil buffer</td>
<td>GL_STENCIL_BUFFER_BIT</td>
<td>Chapter 10</td>
</tr>
</tbody>
</table>

Table 2-1 Clearing Buffers

Before issuing a command to clear multiple buffers, you have to set the values to which each buffer is to be cleared if you want something other than the default RGBA color, depth value, accumulation color, and stencil index. In addition to the `glClearColor()` and `glClearDepth()` commands that set the current values for clearing the color and depth buffers, `glClearIndex()`, `glClearAccum()`, and `glClearStencil()` specify the color index, accumulation color, and stencil index used to clear the corresponding buffers. (See Chapter 4 and Chapter 10 for descriptions of these buffers and their uses.)

OpenGL allows you to specify multiple buffers because clearing is generally a slow operation, as every pixel in the window (possibly millions) is touched, and some graphics hardware allows sets of buffers to be cleared simultaneously. Hardware that doesn’t support simultaneous clears performs them sequentially. The difference between

```c
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```
and

```gl
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT);
```

is that although both have the same final effect, the first example might run faster on many machines. It certainly won’t run more slowly.

### Specifying a Color

With OpenGL, the description of the shape of an object being drawn is independent of the description of its color. Whenever a particular geometric object is drawn, it’s drawn using the currently specified coloring scheme. The coloring scheme might be as simple as “draw everything in fire-engine red” or as complicated as “assume the object is made out of blue plastic, that there’s a yellow spotlight pointed in such and such a direction, and that there’s a general low-level reddish-brown light everywhere else.” In general, an OpenGL programmer first sets the color or coloring scheme and then draws the objects. Until the color or coloring scheme is changed, all objects are drawn in that color or using that coloring scheme. This method helps OpenGL achieve higher drawing performance than would result if it didn’t keep track of the current color.

For example, the pseudocode

```gl
set_current_color(red);
draw_object(A);
draw_object(B);
set_current_color(green);
set_current_color(blue);
draw_object(C);
```

draws objects A and B in red, and object C in blue. The command on the fourth line that sets the current color to green is wasted.

Coloring, lighting, and shading are all large topics with entire chapters or large sections devoted to them. To draw geometric primitives that can be seen, however, you need some basic knowledge of how to set the current color; this information is provided in the next few paragraphs. (See Chapter 4 and Chapter 5 for details on these topics.)

To set a color, use the command `glColor3f()`. It takes three parameters, all of which are floating-point numbers between 0.0 and 1.0. The parameters are, in order, the red, green, and blue components of the color. You can think of these three values as specifying a “mix” of colors: 0.0 means don’t use any
of that component, and 1.0 means use all you can of that component. Thus, the code

```c
glColor3f(1.0, 0.0, 0.0);
```

makes the brightest red the system can draw, with no green or blue components. All zeros makes black; in contrast, all ones makes white. Setting all three components to 0.5 yields gray (halfway between black and white). Here are eight commands and the colors they would set:

```c
setColor3f(0.0, 0.0, 0.0); /* black */
gColor3f(1.0, 0.0, 0.0);  /* red */
gColor3f(0.0, 1.0, 0.0);  /* green */
gColor3f(1.0, 1.0, 0.0);  /* yellow */
gColor3f(0.0, 0.0, 1.0);  /* blue */
gColor3f(1.0, 0.0, 1.0);  /* magenta */
gColor3f(0.0, 1.0, 1.0);  /* cyan */
gColor3f(1.0, 1.0, 1.0);  /* white */
```

You might have noticed earlier that the routine for setting the clearing color, `glClearColor()`, takes four parameters, the first three of which match the parameters for `glColor3f()`. The fourth parameter is the alpha value; it's covered in detail in “Blending” in Chapter 6. For now, set the fourth parameter of `glClearColor()` to 0.0, which is its default value.

### Forcing Completion of Drawing

As you saw in “OpenGL Rendering Pipeline” in Chapter 1, most modern graphics systems can be thought of as an assembly line. The main central processing unit (CPU) issues a drawing command. Perhaps other hardware does geometric transformations. Clipping is performed, followed by shading and/or texturing. Finally, the values are written into the bitplanes for display. In high-end architectures, each of these operations is performed by a different piece of hardware that's been designed to perform its particular task quickly. In such an architecture, there's no need for the CPU to wait for each drawing command to complete before issuing the next one. While the CPU is sending a vertex down the pipeline, the transformation hardware is working on transforming the last one sent, the one before that is being clipped, and so on. In such a system, if the CPU waited for each command to complete before issuing the next, there could be a huge performance penalty.
In addition, the application might be running on more than one machine. For example, suppose that the main program is running elsewhere (on a machine called the client) and that you’re viewing the results of the drawing on your workstation or terminal (the server), which is connected by a network to the client. In that case, it might be horribly inefficient to send each command over the network one at a time, as considerable overhead is often associated with each network transmission. Usually, the client gathers a collection of commands into a single network packet before sending it. Unfortunately, the network code on the client typically has no way of knowing that the graphics program is finished drawing a frame or scene. In the worst case, it waits forever for enough additional drawing commands to fill a packet, and you never see the completed drawing.

For this reason, OpenGL provides the command `glFlush()`, which forces the client to send the network packet even though it might not be full. Where there is no network and all commands are truly executed immediately on the server, `glFlush()` might have no effect. However, if you’re writing a program that you want to work properly both with and without a network, include a call to `glFlush()` at the end of each frame or scene. Note that `glFlush()` doesn’t wait for the drawing to complete—it just forces the drawing to begin execution, thereby guaranteeing that all previous commands execute in finite time even if no further rendering commands are executed.

There are other situations in which `glFlush()` is useful:

- Software renderers that build images in system memory and don’t want to constantly update the screen.
- Implementations that gather sets of rendering commands to amortize start-up costs. The aforementioned network transmission example is one instance of this.

```c
void glFlush(void);
```

Forces previously issued OpenGL commands to begin execution, thus guaranteeing that they complete in finite time.

A few commands—for example, commands that swap buffers in double-buffer mode—automatically flush pending commands onto the network before they can occur.
Chapter 2: State Management and Drawing Geometric Objects

Whenever you initially open a window or later move or resize that window, the window system will send an event to notify you. If you are using GLUT, the notification is automated; whatever routine has been registered to `glutReshapeFunc()` will be called. You must register a callback function that will

- Reestablish the rectangular region that will be the new rendering canvas
- Define the coordinate system to which objects will be drawn

In Chapter 3, you’ll see how to define three-dimensional coordinate systems, but right now just create a simple, basic two-dimensional coordinate system into which you can draw a few objects. Call `glutReshapeFunc(reshape)`, where `reshape()` is the following function shown in Example 2-1.

---

If `glFlush()` isn’t sufficient for you, try `glFinish()`. This command flushes the network as `glFlush()` does and then waits for notification from the graphics hardware or network indicating that the drawing is complete in the framebuffer. You might need to use `glFinish()` if you want to synchronize tasks—for example, to make sure that your three-dimensional rendering is on the screen before you use Display PostScript to draw labels on top of the rendering. Another example would be to ensure that the drawing is complete before it begins to accept user input. After you issue a `glFinish()` command, your graphics process is blocked until it receives notification from the graphics hardware that the drawing is complete. Keep in mind that excessive use of `glFinish()` can reduce the performance of your application, especially if you’re running over a network, because it requires round-trip communication. If `glFlush()` is sufficient for your needs, use it instead of `glFinish()`.

```c
void glFinish(void);
```

Forces all previously issued OpenGL commands to complete. This command doesn't return until all effects from previous commands are fully realized.

---

Coordinate System Survival Kit

Whenever you initially open a window or later move or resize that window, the window system will send an event to notify you. If you are using GLUT, the notification is automated; whatever routine has been registered to `glutReshapeFunc()` will be called. You must register a callback function that will

- Reestablish the rectangular region that will be the new rendering canvas
- Define the coordinate system to which objects will be drawn

In Chapter 3, you’ll see how to define three-dimensional coordinate systems, but right now just create a simple, basic two-dimensional coordinate system into which you can draw a few objects. Call `glutReshapeFunc(reshape)`, where `reshape()` is the following function shown in Example 2-1.
Example 2-1  Reshape Callback Function

```c
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
```

The kernel of GLUT will pass this function two arguments: the width and height, in pixels, of the new, moved, or resized window. `glViewport()` adjusts the pixel rectangle for drawing to be the entire new window. The next three routines adjust the coordinate system for drawing so that the lower left corner is (0, 0) and the upper right corner is (w, h) (see Figure 2-1).

To explain it another way, think about a piece of graphing paper. The `w` and `h` values in `reshape()` represent how many columns and rows of squares are on your graph paper. Then you have to put axes on the graph paper. The `gluOrtho2D()` routine puts the origin, (0, 0), in the lowest, leftmost square, and makes each square represent one unit. Now, when you render the points, lines, and polygons in the rest of this chapter, they will appear on this paper in easily predictable squares. (For now, keep all your objects two-dimensional.)

![Figure 2-1](image.png)  Coordinate System Defined by $w = 50$, $h = 50$
Describing Points, Lines, and Polygons

This section explains how to describe OpenGL geometric primitives. All geometric primitives are eventually described in terms of their vertices—coordinates that define the points themselves, the endpoints of line segments, or the corners of polygons. The next section discusses how these primitives are displayed and what control you have over their display.

What Are Points, Lines, and Polygons?

You probably have a fairly good idea of what a mathematician means by the terms point, line, and polygon. The OpenGL meanings are similar, but not quite the same.

One difference comes from the limitations of computer-based calculations. In any OpenGL implementation, floating-point calculations are of finite precision, and they have round-off errors. Consequently, the coordinates of OpenGL points, lines, and polygons suffer from the same problems.

A more important difference arises from the limitations of a raster graphics display. On such a display, the smallest displayable unit is a pixel, and although pixels might be less than 1/100 of an inch wide, they are still much larger than the mathematician’s concepts of infinitely small (for points) and infinitely thin (for lines). When OpenGL performs calculations, it assumes that points are represented as vectors of floating-point numbers. However, a point is typically (but not always) drawn as a single pixel, and many different points with slightly different coordinates could be drawn by OpenGL on the same pixel.

Points

A point is represented by a set of floating-point numbers called a vertex. All internal calculations are done as if vertices are three-dimensional. Vertices specified by the user as two-dimensional (that is, with only x- and y-coordinates) are assigned a z-coordinate equal to zero by OpenGL.

Advanced

OpenGL works in the homogeneous coordinates of three-dimensional projective geometry, so for internal calculations, all vertices are represented with four floating-point coordinates (x, y, z, w). If w is different from zero, these coordinates correspond to the Euclidean, three-dimensional point (x/w, y/w, z/w). You can specify the w-coordinate in OpenGL commands, but this is
rarely done. If the $w$-coordinate isn’t specified, it is understood to be 1.0. (See Appendix C for more information about homogeneous coordinate systems.)

**Lines**

In OpenGL, the term *line* refers to a *line segment*, not the mathematician’s version that extends to infinity in both directions. There are easy ways to specify a connected series of line segments, or even a closed, connected series of segments (see Figure 2-2). In all cases, though, the lines constituting the connected series are specified in terms of the vertices at their endpoints.

![Figure 2-2 Two Connected Series of Line Segments](image)

**Polygons**

Polygons are the areas enclosed by single closed loops of line segments, where the line segments are specified by the vertices at their endpoints. Polygons are typically drawn with the pixels in the interior filled in, but you can also draw them as outlines or a set of points. (See “Polygon Details” on page 60.)

In general, polygons can be complicated, so OpenGL imposes some strong restrictions on what constitutes a primitive polygon. First, the edges of OpenGL polygons can’t intersect (a mathematician would call a polygon satisfying this condition a *simple polygon*). Second, OpenGL polygons must be *convex*, meaning that they cannot have indentations. Stated precisely, a region is convex if, given any two points in the interior, the line segment joining them is also in the interior. See Figure 2-3 for some examples of valid and invalid polygons. OpenGL, however, doesn’t restrict the number of line segments making up the boundary of a convex polygon. Note that polygons with holes can’t be described. They are *nonconvex*, and they can’t be drawn with a boundary made up of a single closed loop. Be aware that if
you present OpenGL with a nonconvex filled polygon, it might not draw it as you expect. For instance, on most systems, no more than the *convex hull* of the polygon would be filled. On some systems, less than the convex hull might be filled.

![Valid and Invalid Polygons](image)

**Figure 2-3** Valid and Invalid Polygons

The reason for the OpenGL restrictions on valid polygon types is that it’s simpler to provide fast polygon-rendering hardware for that restricted class of polygons. Simple polygons can be rendered quickly. The difficult cases are hard to detect quickly, so for maximum performance, OpenGL crosses its fingers and assumes the polygons are simple.

Many real-world surfaces consist of nonsimple polygons, nonconvex polygons, or polygons with holes. Since all such polygons can be formed from unions of simple convex polygons, some routines to build more complex objects are provided in the GLU library. These routines take complex descriptions and tessellate them, or break them down into groups of the simpler OpenGL polygons that can then be rendered. (See “Polygon Tessellation” in Chapter 11 for more information about the *tessellation* routines.)

Since OpenGL vertices are always three-dimensional, the points forming the boundary of a particular polygon don’t necessarily lie on the same plane in space. (Of course, they do in many cases—if all the z-coordinates are zero, for example, or if the polygon is a *triangle.*) If a polygon’s vertices don’t lie in the same plane, then after various rotations in space, changes in the viewpoint, and projection onto the display screen, the points might no longer form a simple convex polygon. For example, imagine a four-point *quadrilateral* where the points are slightly out of plane, and look at it almost edge-on. You can get a nonsimple polygon that resembles a bow tie, as shown in Figure 2-4, which isn’t guaranteed to be rendered correctly. This situation isn’t all that unusual if you approximate curved surfaces by quadrilaterals made of points lying on the true surface. You can always avoid the problem by using triangles, as any three points always lie on a plane.
Rectangles

Since rectangles are so common in graphics applications, OpenGL provides a filled-rectangle drawing primitive, \texttt{glRect*()}. You can draw a rectangle as a polygon, as described in “OpenGL Geometric Drawing Primitives” on page 47, but your particular implementation of OpenGL might have optimized \texttt{glRect*()} for rectangles.

\begin{verbatim}
void glRect(sifd)(TYPE x1, TYPE y1, TYPE x2, TYPE y2);
void glRect(sifd)v(const TYPE *v1, const TYPE *v2);
\end{verbatim}

Draws the rectangle defined by the corner points \((x1, y1)\) and \((x2, y2)\). The rectangle lies in the plane \(z = 0\) and has sides parallel to the \(x\) and \(y\)-axes. If the vector form of the function is used, the corners are given by two pointers to arrays, each of which contains an \((x, y)\) pair.

Note that although the rectangle begins with a particular orientation in three-dimensional space (in the \(xy\)-plane and parallel to the axes), you can change this by applying rotations or other transformations. (See Chapter 3 for information about how to do this.)

Curves and Curved Surfaces

Any smoothly curved line or surface can be approximated—to any arbitrary degree of accuracy—by short line segments or small polygonal regions. Thus, subdividing curved lines and surfaces sufficiently and then approximating them with straight line segments or flat polygons makes them appear curved (see Figure 2-5). If you’re skeptical that this really works, imagine subdividing until each line segment or polygon is so tiny that it’s smaller than a pixel on the screen.
Even though curves aren’t geometric primitives, OpenGL provides some direct support for subdividing and drawing them. (See Chapter 12 for information about how to draw curves and curved surfaces.)

**Specifying Vertices**

With OpenGL, every geometric object is ultimately described as an ordered set of vertices. You use the `glVertex*()` command to specify a vertex.

```
void glVertex2s(TYPE x, TYPE y);
void glVertex3d(TYPE x, TYPE y, TYPE z);
void glVertex4f(TYPE x, TYPE y, TYPE z, TYPE w);
void glVertex3dv(const 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 effective only 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 \((0.0, 0.0, 3.1415926535898)\).
3.1415926535898) (double-precision floating-point numbers). The third example represents the vertex with three-dimensional coordinates (1.15, 0.5, –1.1) as a homogenous coordinate. (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 \( \text{glVertex*}() \) 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 70.)

### 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 \( \text{glBegin()} \) and a call to \( \text{glEnd()} \). The argument passed to \( \text{glBegin()} \) determines what sort of geometric primitive is constructed from the vertices. For instance, Example 2-3 specifies the vertices for the polygon shown in Figure 2-6.

**Example 2-3  Filled Polygon**

```c
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](image.png)  

**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 10 possible arguments and the corresponding types of primitives.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_POINTS</td>
<td>Individual points</td>
</tr>
<tr>
<td>GL_LINES</td>
<td>Pairs of vertices interpreted as individual line segments</td>
</tr>
<tr>
<td>GL_LINE_STRIP</td>
<td>Series of connected line segments</td>
</tr>
<tr>
<td>GL_LINE_LOOP</td>
<td>Same as above, with a segment added between last and first vertices</td>
</tr>
<tr>
<td>GL_TRIANGLES</td>
<td>Triples of vertices interpreted as triangles</td>
</tr>
<tr>
<td>GL_TRIANGLE_STRIP</td>
<td>Linked strip of triangles</td>
</tr>
<tr>
<td>GL_TRIANGLE_FAN</td>
<td>Linked fan of triangles</td>
</tr>
<tr>
<td>GL_QUADS</td>
<td>Quadruples of vertices interpreted as four-sided polygons</td>
</tr>
<tr>
<td>GL_QUAD_STRIP</td>
<td>Linked strip of quadrilaterals</td>
</tr>
<tr>
<td>GL_POLYGON</td>
<td>Boundary of a simple, convex polygon</td>
</tr>
</tbody>
</table>

**Table 2-2** Geometric Primitive Names and Meanings

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.

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_POINTS</td>
<td>Draws a point at each of the ( n ) vertices.</td>
</tr>
<tr>
<td>GL_LINES</td>
<td>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.</td>
</tr>
<tr>
<td>GL_LINE_STRIP</td>
<td>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.</td>
</tr>
<tr>
<td>GL_LINE_LOOP</td>
<td>Same as GL_LINE_STRIP, except that a final line segment is drawn from ( v_{n-1} ) to ( v_0 ), completing a loop.</td>
</tr>
<tr>
<td>GL_TRIANGLES</td>
<td>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 a multiple of 3, the final one or two vertices are ignored.</td>
</tr>
<tr>
<td>GL_TRIANGLE_STRIP</td>
<td>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 61). ( n ) must be at least 3 for anything to be drawn.</td>
</tr>
<tr>
<td>GL_TRIANGLE_FAN</td>
<td>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).</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose of Command</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>glVertex*()</td>
<td>set vertex coordinates</td>
<td>Chapter 2</td>
</tr>
<tr>
<td>glColor*()</td>
<td>set RGBA color</td>
<td>Chapter 4</td>
</tr>
<tr>
<td>glIndex*()</td>
<td>set color index</td>
<td>Chapter 4</td>
</tr>
<tr>
<td>glSecondaryColor*()</td>
<td>set secondary color for post-texturing application</td>
<td>Chapter 9</td>
</tr>
<tr>
<td>glNormal*()</td>
<td>set normal vector coordinates</td>
<td>Chapter 2</td>
</tr>
<tr>
<td>glMaterial*()</td>
<td>set material properties</td>
<td>Chapter 5</td>
</tr>
<tr>
<td>glFogCoord*()</td>
<td>set fog coordinates</td>
<td>Chapter 6</td>
</tr>
</tbody>
</table>

Table 2-3  Valid Commands between glBegin() and glEnd()
Chapter 2: State Management and Drawing Geometric Objects

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 instance, Example 2-4 draws an outlined circle.

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

```c
#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” in Chapter 11.) If you need to draw numerous 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 a 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:

```c
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 70.)

## Basic State Management

In the preceding section, you saw an example of a state variable, the current RGBA color, and how it can be associated with a primitive. OpenGL maintains many states and state variables. An object may be rendered with lighting, texturing, hidden surface removal, fog, and other states affecting its appearance.
By default, most of these states are initially inactive. These states may be costly to activate; for example, turning on texture mapping will almost certainly slow down the process of rendering a primitive. However, the image will improve in quality and will look more realistic, owing to the enhanced graphics capabilities.

To turn many of these states on and off, use these two simple commands:

```c
void glEnable(GLenum capability);
void glDisable(GLenum capability);
```

`glEnable()` turns on a capability, and `glDisable()` turns it off. More than 60 enumerated values can be passed as parameters to `glEnable()` or `glDisable()`. Some examples are `GL_BLEND` (which controls blending of RGBA values), `GL_DEPTH_TEST` (which controls depth comparisons and updates to the depth buffer), `GL_FOG` (which controls fog), `GL_LINE_STIPPLE` (patterned lines), and `GL_LIGHTING` (you get the idea).

You can also check whether a state is currently enabled or disabled.

```c
GLboolean glIsEnabled(GLenum capability)
```

Returns `GL_TRUE` or `GL_FALSE`, depending on whether or not the queried capability is currently activated.

The states you have just seen have two settings: on and off. However, most OpenGL routines set values for more complicated state variables. For example, the routine `glColor3f()` sets three values, which are part of the `GL_CURRENT_COLOR` state. There are five querying routines used to find out what values are set for many states:

```c
void glGetBooleanv(GLenum pname, GLboolean *params);
void glGetIntegerv(GLenum pname, GLint *params);
void glGetFloatv(GLenum pname, GLfloat *params);
void glGetDoublev(GLenum pname, GLdouble *params);
void glGetPointerv(GLenum pname, GLvoid **params);
```
Obtains Boolean, integer, floating-point, double-precision, or pointer state variables. The *pname* argument is a symbolic constant indicating the state variable to return, and *params* is a pointer to an array of the indicated type in which to place the returned data. See the tables in Appendix B for the possible values for *pname*. For example, to get the current RGBA color, a table in Appendix B suggests you use `glGetIntegerv(GL_CURRENT_COLOR, params)` or `glGetFloatv(GL_CURRENT_COLOR, params)`. A type conversion is performed, if necessary, to return the desired variable as the requested data type.

These querying routines handle most, but not all, requests for obtaining state information. (See “The Query Commands” in Appendix B for a list of all of the available OpenGL state querying routines.)

### Displaying Points, Lines, and Polygons

By default, a point is drawn as a single pixel on the screen, a line is drawn solid and 1 pixel wide, and polygons are drawn solidly filled in. The following paragraphs discuss the details of how to change these default display modes.

#### Point Details

To control the size of a rendered point, use `glPointSize()` and supply the desired size in pixels as the argument.

```c
void glPointSize(GLfloat size);
```

Sets the width in pixels for rendered points; *size* must be greater than 0.0 and by default is 1.0.

The actual collection of pixels on the screen that are drawn for various point widths depends on whether antialiasing is enabled. (Antialiasing is a technique for smoothing points and lines as they’re rendered; see “Antialiasing” and “Point Parameters” in Chapter 6 for more detail.) If antialiasing is disabled (the default), fractional widths are rounded to integer widths, and a screen-aligned square region of pixels is drawn. Thus, if the width is 1.0, the square is 1 pixel by 1 pixel; if the width is 2.0, the square is 2 pixels by 2 pixels; and so on.
With antialiasing or multisampling enabled, a circular group of pixels is drawn, and the pixels on the boundaries are typically drawn at less than full intensity to give the edge a smoother appearance. In this mode, noninteger widths aren’t rounded.

Most OpenGL implementations support very large point sizes. You can query the minimum and maximum sized for aliased points by using GL_ALIASED_POINT_SIZE_RANGE with `glGetFloatv()`. Likewise, you can obtain the range of supported sizes for antialiased points by passing GL_SMOOTH_POINT_SIZE_RANGE to `glGetFloatv()`. The sizes of supported antialiased points are evenly spaced between the minimum and maximum sizes for the range. Calling `glGetFloatv()` with the parameter GL_SMOOTH_POINT_SIZE_GRANULARITY will return how accurately a given antialiased point size is supported. For example, if you request `glPointSize(2.37)` and the granularity returned is 0.1, then the point size is rounded to 2.4.

**Line Details**

With OpenGL, you can specify lines with different widths and lines that are stippled in various ways—dotted, dashed, drawn with alternating dots and dashes, and so on.

**Wide Lines**

```c
void glLineWidth(GLfloat width);
```

Sets the width, in pixels, for rendered lines; `width` must be greater than 0.0 and by default is 1.0.

Version 3.1 does not support values greater than 1.0, and will generate a GL_INVALID_VALUE error if a value greater than 1.0 is specified.

The actual rendering of lines is affected if either antialiasing or multisampling is enabled. (See “Antialiasing Points or Lines” on page 269 and “Antialiasing Geometric Primitives with Multisampling” on page 275.) Without antialiasing, widths of 1, 2, and 3 draw lines 1, 2, and 3 pixels wide. With antialiasing enabled, noninteger line widths are possible, and pixels on the boundaries are typically drawn at less than full intensity. As with point sizes, a particular OpenGL implementation might limit the width of non-antialiased lines to its maximum antialiased line width, rounded to the nearest integer value. You can obtain the range of supported aliased line
widths by using GL_ALIASED_LINE_WIDTH_RANGE with glGetFloatv(). To determine the supported minimum and maximum sizes of antialiased line widths, and what granularity your implementation supports, call glGetFloatv(), with GL_SMOOTH_LINE_WIDTH_RANGE and GL_SMOOTH_LINE_WIDTH_GRANULARITY.

**Note:** Keep in mind that, by default, lines are 1 pixel wide, so they appear wider on lower-resolution screens. For computer displays, this isn't typically an issue, but if you're using OpenGL to render to a high-resolution plotter, 1-pixel lines might be nearly invisible. To obtain resolution-independent line widths, you need to take into account the physical dimensions of pixels.

**Advanced**

With non-antialiased wide lines, the line width isn't measured perpendicular to the line. Instead, it's measured in the y-direction if the absolute value of the slope is less than 1.0; otherwise, it's measured in the x-direction. The rendering of an antialiased line is exactly equivalent to the rendering of a filled rectangle of the given width, centered on the exact line.

**Stippled Lines**

To make stippled (dotted or dashed) lines, you use the command glLineStipple() to define the stipple pattern, and then you enable line stippling with glEnable().

```c
void glLineStipple(GLint factor, GLushort pattern);
```

Sets the current stippling pattern for lines. The pattern argument is a 16-bit series of 0s and 1s, and it's repeated as necessary to stipple a given line. A 1 indicates that drawing occurs, and a 0 that it does not, on a pixel-by-pixel basis, beginning with the low-order bit of the pattern. The pattern can be stretched out by using factor, which multiplies each subseries of consecutive 1s and 0s. Thus, if three consecutive 1s appear in the pattern, they're stretched to six if factor is 2. factor is clamped to lie between 1 and 256. Line stippling must be enabled by passing GL_LINE_STIPPLE to glEnable(); it's disabled by passing the same argument to glDisable().

With the preceding example and the pattern 0x3F07 (which translates to 0011111100000111 in binary), a line would be drawn with 3 pixels on, then 5 off, 6 on, and 2 off. (If this seems backward, remember that the
low-order bit is used first.) If \textit{factor} had been 2, the pattern would have been elongated: 6 pixels on, 10 off, 12 on, and 4 off. Figure 2-8 shows lines drawn with different patterns and repeat factors. If you don’t enable line stippling, drawing proceeds as if \textit{pattern} were 0xFFFF and \textit{factor} were 1. (Use \texttt{glDisable()} with \texttt{GL_LINE_STIPPLE} to disable stippling.) Note that stippling can be used in combination with wide lines to produce wide stippled lines.

\begin{center}
\begin{tabular}{|c|c|}
\hline
PATTERN & FACTOR \\
\hline
0x00FF & 1 \hfill \hfill \hfill \hfill \hfill \\
0x00FF & 2 \hfill \hfill \hfill \hfill \hfill \\
0x0C0F & 1 \hfill \hfill \hfill \hfill \hfill \\
0x0C0F & 3 \hfill \hfill \hfill \hfill \hfill \\
0xA AAA & 1 \hfill \hfill \hfill \hfill \hfill \\
0xA AAA & 2 \hfill \hfill \hfill \hfill \hfill \\
0xA AAA & 3 \hfill \hfill \hfill \hfill \hfill \\
0xA AAA & 4 \hfill \hfill \hfill \hfill \hfill \\
\hline
\end{tabular}
\end{center}

\textbf{Figure 2-8} Stippled Lines

One way to think of the stippling is that as the line is being drawn, the pattern is shifted by 1 bit each time a pixel is drawn (or \textit{factor} pixels are drawn, if \textit{factor} isn’t 1). When a series of connected line segments is drawn between a single \texttt{glBegin()} and \texttt{glEnd()}, the pattern continues to shift as one segment turns into the next. This way, a stippling pattern continues across a series of connected line segments. When \texttt{glEnd()} is executed, the pattern is reset, and if more lines are drawn before stippling is disabled the stippling restarts at the beginning of the pattern. If you’re drawing lines with \texttt{GL_LINES}, the pattern resets for each independent line.

Example 2-5 illustrates the results of drawing with a couple of different stipple patterns and line widths. It also illustrates what happens if the lines are drawn as a series of individual segments instead of a single connected line strip. The results of running the program appear in Figure 2-9.
Example 2-5  Line Stipple Patterns: lines.c

```c
#define drawOneLine(x1,y1,x2,y2)  glBegin(GL_LINES);  \
glVertex2f((x1),(y1)); glVertex2f((x2),(y2)); glEnd();

void init(void)
{
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glShadeModel(GL_FLAT);
}

void display(void)
{
  int i;

  glClearColor(GL_COLOR_BUFFER_BIT);
/* select white for all lines */
glColor3f(1.0, 1.0, 1.0);
/* in 1st row, 3 lines, each with a different stipple */
glEnable(GL_LINE_STIPPLE);
  glLineStipple(1, 0x0101);  /* dotted */
drawOneLine(50.0, 125.0, 150.0, 125.0);
  glLineStipple(1, 0x00FF);  /* dashed */
drawOneLine(150.0, 125.0, 250.0, 125.0);
  glLineStipple(1, 0x1C47);  /* dash/dot/dash */
drawOneLine(250.0, 125.0, 350.0, 125.0);
/* in 2nd row, 3 wide lines, each with different stipple */
gLineWidth(5.0);
  glLineStipple(1, 0x0101);  /* dotted */
drawOneLine(50.0, 100.0, 150.0, 100.0);
  glLineStipple(1, 0x00FF);  /* dashed */
drawOneLine(150.0, 100.0, 250.0, 100.0);
  glLineStipple(1, 0x1C47);  /* dash/dot/dash */
drawOneLine(250.0, 100.0, 350.0, 100.0);
  glLineWidth(1.0);
/* in 3rd row, 6 lines, with dash/dot/dash stipple */
/* as part of a single connected line strip */
gLineStipple(1, 0x1C47);  /* dash/dot/dash */
  glBegin(GL_LINE_STRIP);
  for (i = 0; i < 7; i++)
    glVertex2f(50.0 + ((GLfloat) i * 50.0), 75.0);
  glEnd();
```
Polygon Details

Polygons are typically drawn by filling in all the pixels enclosed within the boundary, but you can also draw them as outlined polygons or simply as points at the vertices. A filled polygon might be solidly filled or stippled with a certain pattern. Although the exact details are omitted here, filled polygons are drawn in such a way that if adjacent polygons share an edge or vertex, the pixels making up the edge or vertex are drawn exactly once—they’re
included in only one of the polygons. This is done so that partially transparent polygons don’t have their edges drawn twice, which would make those edges appear darker (or brighter, depending on what color you’re drawing with). Note that it might result in narrow polygons having no filled pixels in one or more rows or columns of pixels.

To antialias filled polygons, multisampling is highly recommended. For details, see “Antialiasing Geometric Primitives with Multisampling” in Chapter 6.

**Polygons as Points, Outlines, or Solids**

A polygon has two sides—front and back—and might be rendered differently depending on which side is facing the viewer. This allows you to have cutaway views of solid objects in which there is an obvious distinction between the parts that are inside and those that are outside. By default, both front and back faces are drawn in the same way. To change this, or to draw only outlines or vertices, use `glPolygonMode()`.

```c
void glPolygonMode(GLenum face, GLenum mode);
```

Controls the drawing mode for a polygon’s front and back faces. The parameter `face` can be GL_FRONT_AND_BACK, GL_FRONT, or GL_BACK; `mode` can be GL_POINT, GL_LINE, or GL_FILL to indicate whether the polygon should be drawn as points, outlined, or filled. By default, both the front and back faces are drawn filled.

Version 3.1 only accepts GL_FRONT_AND_BACK as a value for `face`, and renders polygons the same way regardless of whether they’re front- or back-facing.

For example, you can have the front faces filled and the back faces outlined with two calls to this routine:

```c
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
```

**Reversing and Culling Polygon Faces**

By convention, polygons whose vertices appear in counterclockwise order on the screen are called front-facing. You can construct the surface of any “reasonable” solid—a mathematician would call such a surface an orientable manifold (spheres, donuts, and teapots are orientable; Klein bottles and Möbius strips aren’t)—from polygons of consistent orientation. In other words, you can use all clockwise polygons or all counterclockwise polygons. (This is essentially the mathematical definition of orientable.)
Suppose you’ve consistently described a model of an orientable surface but happen to have the clockwise orientation on the outside. You can swap what OpenGL considers the back face by using the function `glFrontFace()`, supplying the desired orientation for front-facing polygons.

```c
void glFrontFace(GLenum mode);
```

Controls how front-facing polygons are determined. By default, `mode` is GL_CCW, which corresponds to a counterclockwise orientation of the ordered vertices of a projected polygon in window coordinates. If `mode` is GL_CW, faces with a clockwise orientation are considered front-facing.

**Note:** The orientation (clockwise or counterclockwise) of the vertices is also known as its *winding*.

In a completely enclosed surface constructed from opaque polygons with a consistent orientation, none of the back-facing polygons are ever visible—they’re always obscured by the front-facing polygons. If you are outside this surface, you might enable *culling* to discard polygons that OpenGL determines are back-facing. Similarly, if you are inside the object, only back-facing polygons are visible. To instruct OpenGL to discard front- or back-facing polygons, use the command `glCullFace()` and enable culling with `glEnable()`.

```c
void glCullFace(GLenum mode);
```

Indicates which polygons should be discarded (culled) before they’re converted to screen coordinates. The mode is either GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate front-facing, back-facing, or all polygons. To take effect, culling must be enabled using `glEnable()` with GL_CULL_FACE; it can be disabled with `glDisable()` and the same argument.

**Advanced**

In more technical terms, deciding whether a face of a polygon is front- or back-facing depends on the sign of the polygon’s area computed in window coordinates. One way to compute this area is

\[
a = \frac{1}{2} \sum_{i=0}^{n-1} x_i y_{i+1} - x_{i+1} y_i
\]
where \( x_i \) and \( y_i \) are the \( x \) and \( y \) window coordinates of the \( i \)th vertex of the \( n \)-vertex polygon and

\[
i \oplus 1 \text{ is } (i+1) \mod n.
\]

Assuming that GL_CCW has been specified, if \( a > 0 \), the polygon corresponding to that vertex is considered to be front-facing; otherwise, it's back-facing. If GL_CW is specified and if \( a < 0 \), then the corresponding polygon is front-facing; otherwise, it's back-facing.

**Try This**

Modify Example 2-5 by adding some filled polygons. Experiment with different colors. Try different polygon modes. Also, enable culling to see its effect.

**Stippling Polygons**

By default, filled polygons are drawn with a solid pattern. They can also be filled with a 32-bit by 32-bit window-aligned stipple pattern, which you specify with `glPolygonStipple()`.

```c
void glPolygonStipple(const GLubyte *mask);
```

Defines the current stipple pattern for filled polygons. The argument `mask` is a pointer to a 32 \times 32 bitmap that's interpreted as a mask of 0s and 1s. Where a 1 appears, the corresponding pixel in the polygon is drawn, and where a 0 appears, nothing is drawn. Figure 2-10 shows how a stipple pattern is constructed from the characters in `mask`. Polygon stippling is enabled and disabled by using `glEnable()` and `glDisable()` with GL_POLYGON_STIPPLE as the argument. The interpretation of the `mask` data is affected by the `glPixelStore*()` GL_UNPACK* modes. (See “Controlling Pixel-Storage Modes” in Chapter 8.)

In addition to defining the current polygon stippling pattern, you must enable stippling:

```c
glEnable(GL_POLYGON_STIPPLE);
```

Use `glDisable()` with the same argument to disable polygon stippling.

Figure 2-11 shows the results of polygons drawn unstippled and then with two different stippling patterns. The program is shown in Example 2-6. The reversal of white to black (from Figure 2-10 to Figure 2-11) occurs because the program draws in white over a black background, using the pattern in Figure 2-10 as a stencil.
By default, for each byte the most significant bit is first. Bit ordering can be changed by calling `glPixelStorei()`.

**Figure 2-10**  Constructing a Polygon Stipple Pattern
Example 2-6  Polygon Stipple Patterns: polys.c

```c
void display(void)
{
    GLubyte fly[] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
        0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
        0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
        0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
        0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
        0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
        0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
        0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
        0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
        0x07, 0xe1, 0x87, 0xe0, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
        0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
        0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
        0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
        0x10, 0x63, 0xc6, 0x08, 0x10, 0x30, 0x0c, 0x08,
        0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};

    GLubyte halftone[] = {
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    };
}
```

Figure 2-11  Stippled Polygons
Chapter 2: State Management and Drawing Geometric Objects

```c
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55);

glClearColor(0.0, 0.0, 0.0, 0.0);
glShadeModel(GL_FLAT);

void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(350, 150);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
```
glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}

You might want to use display lists to store polygon stipple patterns to maximize efficiency. (See “Display List Design Philosophy” in Chapter 7.)

Marking Polygon Boundary Edges

Advanced

OpenGL can render only convex polygons, but many nonconvex polygons arise in practice. To draw these nonconvex polygons, you typically subdivide them into convex polygons—usually triangles, as shown in Figure 2-12—and then draw the triangles. Unfortunately, if you decompose a general polygon into triangles and draw the triangles, you can’t really use glPolygonMode() to draw the polygon’s outline, as you get all the triangle outlines inside it. To solve this problem, you can tell OpenGL whether a particular vertex precedes a boundary edge; OpenGL keeps track of this information by passing along with each vertex a bit indicating whether that vertex is followed by a boundary edge. Then, when a polygon is drawn in GL_LINE mode, the nonboundary edges aren’t drawn. In Figure 2-12, the dashed lines represent added edges.

By default, all vertices are marked as preceding a boundary edge, but you can manually control the setting of the edge flag with the command glEdgeFlag*(). This command is used between glBegin() and glEnd() pairs, and it affects all the vertices specified after it until the next glEdgeFlag() call is made. It applies only to vertices specified for polygons, triangles, and quads, not to those specified for strips of triangles or quads.
For instance, Example 2-7 draws the outline shown in Figure 2-13.

![Outlined Polygon Drawn Using Edge Flags](image)

**Figure 2-13** Outlined Polygon Drawn Using Edge Flags

**Example 2-7** Marking Polygon Boundary Edges

```c
void glPolygonMode(GLenum mode, GLenum surfstyle);
void glEdgeFlag(GLboolean flag);
void glEdgeFlagv(const GLboolean *flag);
```

Indicates whether a vertex should be considered as initializing a boundary edge of a polygon. If `flag` is GL_TRUE, the edge flag is set to TRUE (the default), and any vertices created are considered to precede boundary edges until this function is called again with `flag` being GL_FALSE.

Normal Vectors

A *normal vector* (or *normal*, for short) is a vector that points in a direction that’s perpendicular to a surface. For a flat surface, one perpendicular direction is the same for every point on the surface, but for a general curved surface, the normal direction might be different at each point on the surface. With OpenGL, you can specify a normal for each polygon or for each vertex. Vertices of the same polygon might share the same normal (for a flat surface) or have different normals (for a curved surface). You can’t assign normals anywhere other than at the vertices.
An object’s normal vectors define the orientation of its surface in space—in particular, its orientation relative to light sources. These vectors are used by OpenGL to determine how much light the object receives at its vertices. Lighting—a large topic by itself—is the subject of Chapter 5, and you might want to review the following information after you’ve read that chapter. Normal vectors are discussed briefly here because you define normal vectors for an object at the same time you define the object’s geometry.

You use `glNormal*()` to set the current normal to the value of the argument passed in. Subsequent calls to `glVertex*()` cause the specified vertices to be assigned the current normal. Often, each vertex has a different normal, which necessitates a series of alternating calls, as in Example 2-8.

**Example 2-8**  
*Surface Normals at Vertices*

```c
glBegin (GL_POLYGON);
  glNormal3fv(n0);
  glVertex3fv(v0);
  glNormal3fv(n1);
  glVertex3fv(v1);
  glNormal3fv(n2);
  glVertex3fv(v2);
  glNormal3fv(n3);
  glVertex3fv(v3);
glEnd();
```

There’s no magic to finding the normals for an object—most likely, you have to perform some calculations that might include taking derivatives—but there are several techniques and tricks you can use to achieve certain effects. Appendix H, “Calculating Normal Vectors,” explains how to find normal vectors for surfaces. If you already know how to do this, if you can count on always being supplied with normal vectors, or if you don’t want to use the OpenGL lighting facilities, you don’t need to read this appendix.

---

1 This appendix is available online at [http://www.opengl-redbook.com/appendices/](http://www.opengl-redbook.com/appendices/).
Note that at a given point on a surface, two vectors are perpendicular to the surface, and they point in opposite directions. By convention, the normal is the one that points to the outside of the surface being modeled. (If you get inside and outside reversed in your model, just change every normal vector from \((x, y, z)\) to \((-x, -y, -z)\).

Also, keep in mind that since normal vectors indicate direction only, their lengths are mostly irrelevant. You can specify normals of any length, but eventually they have to be converted to a length of 1 before lighting calculations are performed. (A vector that has a length of 1 is said to be of unit length, or normalized.) In general, you should supply normalized normal vectors. To make a normal vector of unit length, divide each of its \(x\), \(y\), \(z\)-components by the length of the normal:

\[
\frac{x^2 + y^2 + z^2}{\sqrt{x^2 + y^2 + z^2}}
\]

Normal vectors remain normalized as long as your model transformations include only rotations and translations. (See Chapter 3 for a discussion of transformations.) If you perform irregular transformations (such as scaling or multiplying by a shear matrix), or if you specify nonunit-length normals, then you should have OpenGL automatically normalize your normal vectors after the transformations. To do this, call \texttt{glEnable(GL\_NORMALIZE)}.

If you supply unit-length normals, and you perform only uniform scaling (that is, the same scaling value for \(x\), \(y\), and \(z\)), you can use \texttt{glEnable(GL\_RESCALE\_NORMAL)} to scale the normals by a constant factor, derived from the modelview transformation matrix, to return them to unit length after transformation.

Note that automatic normalization or rescaling typically requires additional calculations that might reduce the performance of your application. Rescaling normals uniformly with \texttt{GL\_RESCALE\_NORMAL} is usually less expensive than performing full-fledged normalization with \texttt{GL\_NORMALIZE}. By default, both automatic normalizing and rescaling operations are disabled.

**Vertex Arrays**

You may have noticed that OpenGL requires many function calls to render geometric primitives. Drawing a 20-sided polygon requires at least 22 function calls: one call to \texttt{glBegin()}, one call for each of the vertices, and a final call to \texttt{glEnd()}. In the two previous code examples, additional information (polygon boundary edge flags or surface normals) added function calls for
each vertex. This can quickly double or triple the number of function calls required for one geometric object. For some systems, function calls have a great deal of overhead and can hinder performance.

An additional problem is the redundant processing of vertices that are shared between adjacent polygons. For example, the cube in Figure 2-14 has six faces and eight shared vertices. Unfortunately, if the standard method of describing this object is used, each vertex has to be specified three times: once for every face that uses it. Therefore, 24 vertices are processed, even though eight would be enough.

OpenCL has vertex array routines that allow you to specify a lot of vertex-related data with just a few arrays and to access that data with equally few function calls. Using vertex array routines, all 20 vertices in a 20-sided polygon can be put into one array and called with one function. If each vertex also has a surface normal, all 20 surface normals can be put into another array and also called with one function.

Arranging data in vertex arrays may increase the performance of your application. Using vertex arrays reduces the number of function calls, which improves performance. Also, using vertex arrays may allow reuse of already processed shared vertices.

**Note:** Vertex arrays became standard in Version 1.1 of OpenGL. Version 1.4 added support for storing fog coordinates and secondary colors in vertex arrays.

There are three steps to using vertex arrays to render geometry:

1. Activate (enable) the appropriate arrays, with each storing a different type of data: vertex coordinates, surface normals, RGBA colors, secondary colors, color indices, fog coordinates, texture coordinates, polygon edge flags, or vertex attributes for use in a vertex shader.
2. Put data into the array or arrays. The arrays are accessed by the addresses of (that is, pointers to) their memory locations. In the client-server model, this data is stored in the client’s address space, unless you choose to use buffer objects (see “Buffer Objects” on page 91), for which the arrays are stored in server memory.

3. Draw geometry with the data. OpenGL obtains the data from all activated arrays by dereferencing the pointers. In the client-server model, the data is transferred to the server’s address space. There are three ways to do this:
   - Accessing individual array elements (randomly hopping around)
   - Creating a list of individual array elements (methodically hopping around)
   - Processing sequential array elements

   The dereferencing method you choose may depend on the type of problem you encounter. Version 1.4 added support for multiple array access from a single function call.

Interleaved vertex array data is another common method of organization. Instead of several different arrays, each maintaining a different type of data (color, surface normal, coordinate, and so on), you may have the different types of data mixed into a single array. (See “Interleaved Arrays” on page 88.)

### Step 1: Enabling Arrays

The first step is to call `glEnableClientState()` with an enumerated parameter, which activates the chosen array. In theory, you may need to call this up to eight times to activate the eight available arrays. In practice, you’ll probably activate up to six arrays. For example, it is unlikely that you would activate both GL_COLOR_ARRAY and GL_INDEX_ARRAY, as your program’s display mode supports either RGBA mode or color-index mode, but probably not both simultaneously.

```c
void glEnableClientState(GLenum array)
```

Specifies the array to enable. The symbolic constants GL_VERTEX_ARRAY, GL_COLOR_ARRAY, GL_SECONDARY_COLOR_ARRAY, GL_INDEX_ARRAY, GL_NORMAL_ARRAY, GL_FOG_COORD_ARRAY, GL_TEXTURE_COORD_ARRAY, and GL_EDGE_FLAG_ARRAY are acceptable parameters.
Note: Version 3.1 supports only vertex array data stored in buffer objects (see “Buffer Objects” on page 91 for details).

If you use lighting, you may want to define a surface normal for every vertex. (See “Normal Vectors” on page 68.) To use vertex arrays for that case, you activate both the surface normal and vertex coordinate arrays:

```c
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

Suppose that you want to turn off lighting at some point and just draw the geometry using a single color. You want to call `glDisable()` to turn off lighting states (see Chapter 5). Now that lighting has been deactivated, you also want to stop changing the values of the surface normal state, which is wasted effort. To do this, you call

```c
glDisableClientState(GL_NORMAL_ARRAY);
```

You might be asking yourself why the architects of OpenGL created these new (and long) command names, like `gl*ClientState()`, for example. Why can’t you just call `glEnable()` and `glDisable()`? One reason is that `glEnable()` and `glDisable()` can be stored in a display list, but the specification of vertex arrays cannot, because the data remains on the client’s side.

If multitexturing is enabled, enabling and disabling client arrays affects only the active texturing unit. See “Multitexturing” on page 467 for more details.

**Step 2: Specifying Data for the Arrays**

There is a straightforward way by which a single command specifies a single array in the client space. There are eight different routines for specifying arrays—one routine for each kind of array. There is also a command that can specify several client-space arrays at once, all originating from a single interleaved array.
Chapter 2: State Management and Drawing Geometric Objects

To access the other seven arrays, there are seven similar routines:

void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);
void glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);
void glSecondaryColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);
void glIndexPointer(GLenum type, GLsizei stride, const GLvoid *pointer);
void glNormalPointer(GLenum type, GLsizei stride, const GLvoid *pointer);
void glFogCoordPointer(GLenum type, GLsizei stride, const GLvoid *pointer);
void glTexCoordPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);
void glEdgeFlagPointer(GLsizei stride, const GLvoid *pointer);

Note: Additional vertex attributes, used by programmable shaders, can be stored in vertex arrays. Because of their association with shaders, they are discussed in Chapter 15, “The OpenGL Shading Language,” on page 720. For Version 3.1, only generic vertex arrays are supported for storing vertex data.

The main difference among the routines is whether size and type are unique or must be specified. For example, a surface normal always has three components, so it is redundant to specify its size. An edge flag is always a single Boolean, so neither size nor type needs to be mentioned. Table 2-4 displays legal values for size and data types.

For OpenGL implementations that support multitexturing, specifying a texture coordinate array with glVertexPointer() only affects the currently active texture unit. See “Multitexturing” on page 467 for more information.
Example 2-9 uses vertex arrays for both RGBA colors and vertex coordinates. RGB floating-point values and their corresponding \((x, y)\) integer coordinates are loaded into the GL_COLOR_ARRAY and GL_VERTEX_ARRAY.

**Example 2-9** Enabling and Loading Vertex Arrays: varray.c

```c
static GLint vertices[] = {25, 25, 
                          100, 325, 
                          175, 25,  
                          175, 325, 
                          250, 25,  
                          325, 325};

static GLfloat colors[] = {1.0, 0.2, 0.2,  
                           0.2, 0.2, 1.0,  
                           0.8, 1.0, 0.2, 
                           0.75, 0.75, 0.75, 
                           0.35, 0.35, 0.35, 
                           0.5, 0.5, 0.5};
```

<table>
<thead>
<tr>
<th>Command</th>
<th>Sizes</th>
<th>Values for type Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>glVertexPointer</td>
<td>2, 3, 4</td>
<td>GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE</td>
</tr>
<tr>
<td>glColorPointer</td>
<td>3, 4</td>
<td>GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE</td>
</tr>
<tr>
<td>glSecondaryColorPointer</td>
<td>3</td>
<td>GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE</td>
</tr>
<tr>
<td>glIndexPointer</td>
<td>1</td>
<td>GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE</td>
</tr>
<tr>
<td>glNormalPointer</td>
<td>3</td>
<td>GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE</td>
</tr>
<tr>
<td>glFogCoordPointer</td>
<td>1</td>
<td>GL_FLOAT, GL_DOUBLE</td>
</tr>
<tr>
<td>glTexCoordPointer</td>
<td>1, 2, 3, 4</td>
<td>GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE</td>
</tr>
<tr>
<td>glEdgeFlagPointer</td>
<td>1</td>
<td>no type argument (type of data must be GLboolean)</td>
</tr>
</tbody>
</table>

Table 2-4 Vertex Array Sizes (Values per Vertex) and Data Types
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3, GL_FLOAT, 0, colors);
glVertexPointer(2, GL_INT, 0, vertices);

**Stride**

The *stride* parameter for the **gl*Pointer()** routines tells OpenGL how to access the data you provide in your pointer arrays. Its value should be the number of bytes between the starts of two successive pointer elements, or zero, which is a special case. For example, suppose you stored both your vertex’s RGB and \((x, y, z)\) coordinates in a single array, such as the following:

```c
static GLfloat intertwined[] =
{1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
  1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
  1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
  0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
  0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
  0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

To reference only the color values in the *intertwined* array, the following call starts from the beginning of the array (which could also be passed as \&intertwined[0]) and jumps ahead \(6 \times \text{sizeof(GLfloat)}\) bytes, which is the size of both the color and vertex coordinate values. This jump is enough to get to the beginning of the data for the next vertex:

```c
glColorPointer(3, GL_FLOAT, 6*\text{sizeof(GLfloat)}, \&intertwined[0]);
```

For the vertex coordinate pointer, you need to start from further in the array, at the fourth element of *intertwined* (remember that C programmers start counting at zero):

```c
glVertexPointer(3, GL_FLOAT, 6*\text{sizeof(GLfloat)}, \&intertwined[3]);
```

If your data is stored similar to the *intertwined* array above, you may find the approach described in “Interleaved Arrays” on page 88 more convenient for storing your data.

With a stride of zero, each type of vertex array (RGB color, color index, vertex coordinate, and so on) must be tightly packed. The data in the array must be homogeneous; that is, the data must be all RGB color values, all vertex coordinates, or all some other data similar in some fashion.
Step 3: Dereferencing and Rendering

Until the contents of the vertex arrays are dereferenced, the arrays remain on the client side, and their contents are easily changed. In Step 3, contents of the arrays are obtained, sent to the server, and then sent down the graphics processing pipeline for rendering.

You can obtain data from a single array element (indexed location), from an ordered list of array elements (which may be limited to a subset of the entire vertex array data), or from a sequence of array elements.

Dereferencing a Single Array Element

```c
void glArrayElement(GLint ith)
```

Obtains the data of one (the `ith`) vertex for all currently enabled arrays. For the vertex coordinate array, the corresponding command would be `glVertex[size][type]v()`, where size is one of [2, 3, 4], and type is one of [s,i,f,d] for GLshort, GLint, GLfloat, and GLdouble, respectively. Both size and type were defined by `glVertexPointer()`. For other enabled arrays, `glArrayElement()` calls `glEdgeFlagv()`, `glTexCoord[size][type]v()`, `gIColor[size][type]v()`, `glSecondaryColor3[type]v()`, `glIndex[type]v()`, `glNormal3[type]v()`, and `glFogCoord[type]v()`. If the vertex coordinate array is enabled, the `glVertex*v()` routine is executed last, after the execution (if enabled) of up to seven corresponding array values.

`glArrayElement()` is usually called between `glBegin()` and `glEnd()`. (If called outside, `glArrayElement()` sets the current state for all enabled arrays, except for vertex, which has no current state.) In Example 2-10, a triangle is drawn using the third, fourth, and sixth vertices from enabled vertex arrays. (Again, remember that C programmers begin counting array locations with zero.)

**Example 2-10 Using glArrayElement() to Define Colors and Vertices**

```c
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
gIColorPointer(3, GL_FLOAT, 0, colors);
glVertexPointer(2, GL_INT, 0, vertices);

glBegin(GL_TRIANGLES);
glArrayElement(2);
glArrayElement(3);
glArrayElement(5);
glEnd();
```
When executed, the latter five lines of code have the same effect as

```c
glBegin(GL_TRIANGLES);
glColor3fv(colors + (2 * 3));
glVertex2iv(vertices + (2 * 2));
glColor3fv(colors + (3 * 3));
glVertex2iv(vertices + (3 * 2));
glColor3fv(colors + (5 * 3));
glVertex2iv(vertices + (5 * 2));
glEnd();
```

Since `glArrayElement()` is only a single function call per vertex, it may reduce the number of function calls, which increases overall performance.

Be warned that if the contents of the array are changed between `glBegin()` and `glEnd()`, there is no guarantee that you will receive original data or changed data for your requested element. To be safe, don’t change the contents of any array element that might be accessed until the primitive is completed.

**Dereferencing a List of Array Elements**

`glArrayElement()` is good for randomly “hopping around” your data arrays. Similar routines, `glDrawElements()`, `glMultiDrawElements()`, and `glDrawRangeElements()`, are good for hopping around your data arrays in a more orderly manner.

```c
void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid *indices);
```

Defines a sequence of geometric primitives using `count` number of elements, whose indices are stored in the array `indices`. `type` must be one of GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT, indicating the data type of the `indices` array. `mode` specifies what kind of primitives are constructed and is one of the same values that is accepted by `glBegin()`; for example, GL_POLYGON, GL_LINE_LOOP, GL_LINES, GL_POINTS, and so on.

The effect of `glDrawElements()` is almost the same as this command sequence:

```c
glBegin(mode);
for (i = 0; i < count; i++)
    glArrayElement(indices[i]);
glEnd();
```
**glDrawElements()** additionally checks to make sure *mode*, *count*, and *type* are valid. Also, unlike the preceding sequence, executing `glDrawElements()` leaves several states indeterminate. After execution of `glDrawElements()`, current RGB color, secondary color, color index, normal coordinates, fog coordinates, texture coordinates, and edge flag are indeterminate if the corresponding array has been enabled.

With `glDrawElements()`, the vertices for each face of the cube can be placed in an array of indices. Example 2-11 shows two ways to use `glDrawElements()` to render the cube. Figure 2-15 shows the numbering of the vertices used in Example 2-11.

![Figure 2-15 Cube with Numbered Vertices](image)

### Example 2-11  Using `glDrawElements()` to Dereference Several Array Elements

```c
static GLubyte frontIndices[] = {4, 5, 6, 7};
static GLubyte rightIndices[] = {1, 2, 6, 5};
static GLubyte bottomIndices[] = {0, 1, 5, 4};
static GLubyte backIndices[] = {0, 3, 2, 1};
static GLubyte leftIndices[] = {0, 4, 7, 3};
static GLubyte topIndices[] = {2, 3, 7, 6};

glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, frontIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, rightIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, backIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, topIndices);
```

**Note:** It is an error to encapsulate `glDrawElements()` between a `glBegin()`/`glEnd()` pair.

With several primitive types (such as GL_QUADS, GL_TRIANGLES, and GL_LINES), you may be able to compact several lists of indices together into a single array. Since the GL_QUADS primitive interprets each group of four
vertices as a single polygon, you may compact all the indices used in Example 2-11 into a single array, as shown in Example 2-12:

**Example 2-12**  Compacting Several `glDrawElements()` Calls into One

```c
static GLubyte allIndices[] = {4, 5, 6, 7, 1, 2, 6, 5,
                              0, 1, 5, 4, 0, 3, 2, 1,
                              0, 4, 7, 3, 2, 3, 7, 6};

glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, allIndices);
```

For other primitive types, compacting indices from several arrays into a single array renders a different result. In Example 2-13, two calls to `glDrawElements()` with the primitive `GL_LINE_STRIP` render two line strips. You cannot simply combine these two arrays and use a single call to `glDrawElements()` without concatenating the lines into a single strip that would connect vertices #6 and #7. (Note that vertex #1 is being used in both line strips just to show that this is legal.)

**Example 2-13**  Two `glDrawElements()` Calls That Render Two Line Strips

```c
static GLubyte oneIndices[] = {0, 1, 2, 3, 4, 5, 6};
static GLubyte twoIndices[] = {7, 1, 8, 9, 10, 11};

glDrawElements(GL_LINE_STRIP, 7, GL_UNSIGNED_BYTE, oneIndices);
glDrawElements(GL_LINE_STRIP, 6, GL_UNSIGNED_BYTE, twoIndices);
```

The routine `glMultiDrawElements()` was introduced in OpenGL Version 1.4 to enable combining the effects of several `glDrawElements()` calls into a single call.

```
void glMultiDrawElements(GLenum mode, GLsizei *count,
                        GLenum type, const GLvoid **indices,
                        GLsizei primcount);
```

Calls a sequence of `primcount` (a number of) `glDrawElements()` commands. `indices` is an array of pointers to lists of array elements. `count` is an array of how many vertices are found in each respective array element list. `mode` (primitive type) and `type` (data type) are the same as they are in `glDrawElements()`.

The effect of `glMultiDrawElements()` is the same as

```c
for (i = 0; i < primcount; i++) {
    if (count[i] > 0)
        glDrawElements(mode, count[i], type, indices[i]);
}
```
The calls to `glDrawElements()` in Example 2-13 can be combined into a single call of `glMultiDrawElements()`, as shown in Example 2-14:

**Example 2-14**  Use of `glMultiDrawElements()`: mvarray.c

```c
static GLubyte oneIndices[] = {0, 1, 2, 3, 4, 5, 6};
static GLubyte twoIndices[] = {7, 1, 8, 9, 10, 11};
static GLsizei count[] = {7, 6};
static GLvoid * indices[2] = {oneIndices, twoIndices};

glMultiDrawElements(GL_LINE_STRIP, count, GL_UNSIGNED_BYTE, 
    indices, 2);
```

Like `glDrawElements()` or `glMultiDrawElements()`, `glDrawRangeElements()` is also good for hopping around data arrays and rendering their contents. `glDrawRangeElements()` also introduces the added restriction of a range of legal values for its indices, which may increase program performance. For optimal performance, some OpenGL implementations may be able to prefetch (obtain prior to rendering) a limited amount of vertex array data. `glDrawRangeElements()` allows you to specify the range of vertices to be prefetched.

```c
void glDrawRangeElements(GLenum mode, GLuint start, 
    GLuint end, GLsizei count, GLenum type, const GLvoid *indices);
```

It is a mistake for vertices in the array `indices` to reference outside the range `[start, end]`. However, OpenGL implementations are not required to find or report this mistake. Therefore, illegal index values may or may not generate an OpenGL error condition, and it is entirely up to the implementation to decide what to do.

You can use `glGetIntegerv()` with `GL_MAX_ELEMENTS_VERTICES` and `GL_MAX_ELEMENTS_INDICES` to find out, respectively, the recommended maximum number of vertices to be prefetched and the maximum number
of indices (indicating the number of vertices to be rendered) to be referenced. If \( end - start + 1 \) is greater than the recommended maximum of preloaded vertices, or if \( count \) is greater than the recommended maximum of indices, \texttt{glDrawRangeElements()} should still render correctly, but performance may be reduced.

Not all vertices in the range \([start, end]\) have to be referenced. However, on some implementations, if you specify a sparsely used range, you may unnecessarily process many vertices that go unused.

With \texttt{glArrayElement()}, \texttt{glDrawElements()}, \texttt{glMultiDrawElements()}, and \texttt{glDrawRangeElements()}, it is possible that your OpenGL implementation caches recently processed (meaning transformed, lit) vertices, allowing your application to “reuse” them by not sending them down the transformation pipeline additional times. Take the aforementioned cube, for example, which has six faces (polygons) but only eight vertices. Each vertex is used by exactly three faces. Without \texttt{gl*Elements()}, rendering all six faces would require processing 24 vertices, even though 16 vertices are redundant. Your implementation of OpenGL may be able to minimize redundancy and process as few as eight vertices. (Reuse of vertices may be limited to all vertices within a single \texttt{glDrawElements()} or \texttt{glDrawRangeElements()} call, a single index array for \texttt{glMultiDrawElements()}, or, for \texttt{glArrayElement()}, within one \texttt{glBegin()}/\texttt{glEnd()} pair.)

Dereferencing a Sequence of Array Elements

While \texttt{glArrayElement()}, \texttt{glDrawElements()}, and \texttt{glDrawRangeElements()} “hop around” your data arrays, \texttt{glDrawArrays()} plows straight through them.

```c
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Constructs a sequence of geometric primitives using array elements starting at \( first \) and ending at \( first + count - 1 \) of each enabled array. \( mode \) specifies what kinds of primitives are constructed and is one of the same values accepted by \texttt{glBegin()}; for example, GL_POLYGON, GL_LINE_LOOP, GL_LINES, GL_POINTS, and so on.

The effect of \texttt{glDrawArrays()} is almost the same as this command sequence:

```c
    glBegin (mode);
    for (i = 0; i < count; i++)
        glArrayElement(first + i);
    glEnd();
```
As is the case with `glDrawElements()`, `glDrawArrays()` also performs error checking on its parameter values and leaves the current RGB color, secondary color, color index, normal coordinates, fog coordinates, texture coordinates, and edge flag with indeterminate values if the corresponding array has been enabled.

**Try This**

Change the icosahedron drawing routine in Example 2-19 on page 115 to use vertex arrays.

Similar to `glMultiDrawElements()`, the routine `glMultiDrawArrays()` was introduced in OpenGL Version 1.4 to combine several `glDrawArrays()` calls into a single call.

```c
void glMultiDrawArrays(GLenum mode, GLint *first, GLsizei *count GLsizei primcount);
```

Calls a sequence of `primcount` (a number of) `glDrawArrays()` commands. `mode` specifies the primitive type with the same values as accepted by `glBegin()`. `first` and `count` contain lists of array locations indicating where to process each list of array elements. Therefore, for the `i`th list of array elements, a geometric primitive is constructed starting at `first[i]` and ending at `first[i] + count[i] – 1`.

The effect of `glMultiDrawArrays()` is the same as

```c
for (i = 0; i < primcount; i++) {
    if (count[i] > 0)
        glDrawArrays(mode, first[i], count[i]);
}
```

**Restarting Primitives**

As you start working with larger sets of vertex data, you are likely to find that you need to make numerous calls to the OpenGL drawing routines, usually rendering the same type of primitive (such as `GL_TRIANGLES_STRIP`, for example) that you used in the previous drawing call. Of course, you can use the `glMultiDraw*()` routines, but they require the overhead of maintaining the arrays for the starting index and length of each primitive.

OpenGL Version 3.1 added the ability to restart primitives within the same drawing call by specifying a special value, the `primitive restart index`, which
is specially processed by OpenGL. When the primitive restart index is encountered in a draw call, a new rendering primitive of the same type is started with the vertex following the index. The primitive restart index is specified by the `glPrimitiveRestartIndex()` routine.

```c
void glPrimitiveRestartIndex(GLuint index);
```

Specifies the vertex array element index used to indicate that a new primitive should be started during rendering. When processing of vertex array element indices encounters a value that matches `index`, no vertex data is processed, the current graphics primitive is terminated, and a new one of the identical type is started.

Primitive restarting is controlled by calling `glEnable()` or `glDisable()` and specifying `GL_PRIMITIVE_RESTART`, as demonstrated in Example 2-15.

**Example 2-15** Using `glPrimitiveRestartIndex()` to Render Multiple Triangle Strips: `primrestart.c`.

```c
#define BUFFER_OFFSET(offset) ((GLvoid *) NULL + offset)
#define XStart      -0.8
#define XEnd         0.8
#define YStart      -0.8
#define YEnd         0.8
#define NumXPoints          11
#define NumYPoints          11
#define NumPoints           (NumXPoints * NumYPoints)
#define NumPointsPerStrip   (2*NumXPoints)
#define NumStrips           (NumYPoints-1)
#define RestartIndex        0xffff

void init()
{
    GLuint    vbo, ebo;
    GLfloat   *vertices;
    GLushort  *indices;

    /* Set up vertex data */
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, 2*NumPoints*sizeof(GLfloat),
                 NULL, GL_STATIC_DRAW);
```
vertices = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

if (vertices == NULL) {
    fprintf(stderr, "Unable to map vertex buffer\n");
    exit(EXIT_FAILURE);
} else {
    int i, j;
    GLfloat dx = (XEnd - XStart) / (NumXPoints - 1);
    GLfloat dy = (YEnd - YStart) / (NumYPoints - 1);
    GLfloat *tmp = vertices;
    int n = 0;

    for (j = 0; j < NumYPoints; ++j) {
        GLfloat y = YStart + j*dy;

        for (i = 0; i < NumXPoints; ++i) {
            GLfloat x = XStart + i*dx;
            *tmp++ = x;
            *tmp++ = y;
        }
    }

    glUnmapBuffer(GL_ARRAY_BUFFER);
    glVertexPointer(2, GL_FLOAT, 0, BUFFER_OFFSET(0));
    glEnableClientState(GL_VERTEX_ARRAY);
}

/* Set up index data */
glGenBuffers(1, &ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);

/* We allocate an extra restart index because it simplifies */
/* the element-array loop logic */
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             NumStrips*(NumPointsPerStrip+1)*sizeof(GLushort),
             NULL, GL_STATIC_DRAW);
indices = glMapBuffer(GL_ELEMENT_ARRAY_BUFFER,
                      GL_WRITE_ONLY);

if (indices == NULL) {
    fprintf(stderr, "Unable to map index buffer\n");
    exit(EXIT_FAILURE);
} else {
    int i, j;
    GLushort *index = indices;
for (j = 0; j < NumStrips; ++j) {
    GLushort bottomRow = j*NumYPoints;
    GLushort topRow = bottomRow + NumYPoints;

    for (i = 0; i < NumXPoints; ++i) {
        *index++ = topRow + i;
        *index++ = bottomRow + i;
    }
    *index++ = RestartIndex;
}

glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);
}

glPrimitiveRestartIndex(RestartIndex);
glEnable(GL_PRIMITIVE_RESTART);
}

void display()
{
    int i, start;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1, 1, 1);
    glDrawElements(GL_TRIANGLE_STRIP, NumStrips*(NumPointsPerStrip + 1),
                   GL_UNSIGNED_SHORT, BUFFER_OFFSET(0));

    glutSwapBuffers();
}

Instanced Drawing

Advanced
OpenGL Version 3.1 (specifically, GLSL version 1.40) added support for
instanced drawing, which provides an additional value—gl_InstanceID,
called the instance ID, and accessible only in a vertex shader—that is
monotonically incremented for each group of primitives specified.
**glDrawArraysInstanced()** operates similarly to **glMultiDrawArrays()**, except that the starting index and vertex count (as specified by *first* and *count*, respectively) are the same for each call to **glDrawArrays()**.

```c
void glDrawArraysInstanced(GLenum mode, GLint first, GLsizei count, GLsizei primcount);
```

Effectively calls **glDrawArrays()** *primcount* times, setting the GLSL vertex shader value `gl_InstanceID` before each call. *mode* specifies the primitive type. *first* and *count* specify the range of array elements that are passed to **glDrawArrays()**.

**glDrawArraysInstanced()** has the same effect as this call sequence (except that your application cannot manually update `gl_InstanceID`):

```c
for (i = 0; i < primcount; i++) {
    gl_InstanceID = i;
    glDrawArrays(mode, first, count);
}
gl_InstanceID = 0;
```

Likewise, **glDrawElementsInstanced()** performs the same operation, but allows random-access to the data in the vertex array:

```c
void glDrawElementsInstanced(GLenum mode, GLsizei count, GLenum type, const void *indicies, GLsizei primcount);
```

Effectively calls **glDrawElements()** *primcount* times, setting the GLSL vertex shader value `gl_InstanceID` before each call. *mode* specifies the primitive type. *type* indicates the data type of the array *indices* and must be one of the following: GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT. *indicies* and *count* specify the range of array elements that are passed to **glDrawElements()**.

The implementation of **glDrawElementsInstanced()** is shown here:

```c
for (i = 0; i < primcount; i++) {
    gl_InstanceID = i;
    glDrawElements(mode, count, type, indicies);
}
gl_InstanceID = 0;
```
**Interleaved Arrays**

*Advanced*

Earlier in this chapter (see “Stride” on page 76), the special case of interleaved arrays was examined. In that section, the array *intertwined*, which interleaves RGB color and 3D vertex coordinates, was accessed by calls to `glColorPointer()` and `glVertexPointer()`. Careful use of *stride* helped properly specify the arrays:

```c
static GLfloat intertwined[] =
{1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
  1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
  1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
  0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
  0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
  0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

There is also a behemoth routine, `glInterleavedArrays()`, that can specify several vertex arrays at once. `glInterleavedArrays()` also enables and disables the appropriate arrays (so it combines “Step 1: Enabling Arrays” on page 72 and “Step 2: Specifying Data for the Arrays” on page 73). The array *intertwined* exactly fits one of the 14 data-interleaving configurations supported by `glInterleavedArrays()`. Therefore, to specify the contents of the array *intertwined* into the RGB color and vertex arrays and enable both arrays, call

```c
glInterleavedArrays(GL_C3F_V3F, 0, intertwined);
```

This call to `glInterleavedArrays()` enables GL_COLOR_ARRAY and GL_VERTEX_ARRAY. It disables GL_SECONDARY_COLOR_ARRAY, GL_INDEX_ARRAY, GL_NORMAL_ARRAY, GL_FOG_COORD_ARRAY, GL_TEXTURE_COORD_ARRAY, and GL_EDGE_FLAG_ARRAY.

This call also has the same effect as calling `glColorPointer()` and `glVertexPointer()` to specify the values for six vertices in each array. Now you are ready for Step 3: calling `glArrayElement()`, `glDrawElements()`, `glDrawRangeElements()`, or `glDrawArrays()` to dereference array elements.

Note that `glInterleavedArrays()` does not support edge flags.

The mechanics of `glInterleavedArrays()` are intricate and require reference to Example 2-16 and Table 2-5. In that example and table, you’ll see `e_t`, `e_c`, and `e_n`, which are the Boolean values for the enabled or disabled texture coordinate, color, and normal arrays; and you’ll see `s_t`, `s_c`, and `s_v`, which are the sizes (numbers of components) for the texture coordinate, color, and
vertex arrays. t_c is the data type for RGBA color, which is the only array that can have nonfloating-point interleaved values. p_c, p_n, and p_v are the calculated strides for jumping into individual color, normal, and vertex values; and s is the stride (if one is not specified by the user) to jump from one array element to the next.

The effect of \texttt{glInterleavedArrays()} is the same as calling the command sequence in Example 2-16 with many values defined in Table 2-5. All pointer arithmetic is performed in units of \texttt{sizeof(GLubyte)}.

\begin{verbatim}
Example 2-16  Effect of \texttt{glInterleavedArrays(format, stride, pointer)}

int str;
/*  set e_t, e_c, e_n, s_t, s_c, s_v, t_c, p_c, p_n, p_v, and s
    *  as a function of Table 2-5 and the value of format
    */

str = stride;
if (str == 0)
    str = s;

glDisableClientState(GL_EDGE_FLAG_ARRAY);
glDisableClientState(GL_INDEX_ARRAY);
glDisableClientState(GL_SECONDARY_COLOR_ARRAY);
if (et) {
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    glTexCoordPointer(st, GL_FLOAT, str, pointer);
} else
\end{verbatim}

\begin{center}
\textbf{Compatibility}
\begin{tabular}{|c|}
\hline
\texttt{glInterleavedArrays} \\
\hline
\end{tabular}
\end{center}

\texttt{glInterleavedArrays} initializes all eight arrays, disabling arrays that are not specified in \textit{format}, and enabling the arrays that are specified. \textit{format} is one of 14 symbolic constants, which represent 14 data configurations; Table 2-5 displays \textit{format} values. \textit{stride} specifies the byte offset between consecutive vertices. If \textit{stride} is 0, the vertices are understood to be tightly packed in the array. \textit{pointer} is the memory address of the first coordinate of the first vertex in the array.

If multitexturing is enabled, \texttt{glInterleavedArrays()} affects only the active texture unit. See “Multitexturing” on page 467 for details.
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
if (ec) {
    glEnableClientState(GL_COLOR_ARRAY);
    glColorPointer(sc, tc, str, pointer+pc);
} else
    glDisableClientState(GL_COLOR_ARRAY);

if (en) {
    glEnableClientState(GL_NORMAL_ARRAY);
    glNormalPointer(GL_FLOAT, str, pointer+pn);
} else
    glDisableClientState(GL_NORMAL_ARRAY);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(sv, GL_FLOAT, str, pointer+pv);

In Table 2-5, T and F are True and False. f is sizeof(GLfloat). c is 4 times sizeof(GLubyte), rounded up to the nearest multiple of f.

<table>
<thead>
<tr>
<th>Format</th>
<th>e_t</th>
<th>e_c</th>
<th>e_n</th>
<th>s_t</th>
<th>s_c</th>
<th>s_v</th>
<th>t_c</th>
<th>p_c</th>
<th>p_n</th>
<th>p_v</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_V2F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>2</td>
<td>0</td>
<td>2f</td>
<td></td>
<td>0</td>
<td>2f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_V3F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>0</td>
<td>3f</td>
<td></td>
<td>0</td>
<td>3f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_C4UB_V2F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>4</td>
<td>2</td>
<td>GL_UNSIGNED_BYTE 0</td>
<td>c</td>
<td>c+2f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_C4UB_V3F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>4</td>
<td>3</td>
<td>GL_UNSIGNED_BYTE 0</td>
<td>c</td>
<td>c+3f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_C3F_V3F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>3</td>
<td>3</td>
<td>GL_FLOAT         0</td>
<td>3f</td>
<td>6f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_N3F_V3F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>3</td>
<td>0</td>
<td>3f</td>
<td>GL_FLOAT         0</td>
<td>3f</td>
<td>6f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_C4F_N3F_V3F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>4</td>
<td>3</td>
<td>GL_FLOAT         0</td>
<td>4f</td>
<td>7f</td>
<td>10f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_T2F_V3F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>2</td>
<td>3</td>
<td>2f</td>
<td>GL_FLOAT         0</td>
<td>2f</td>
<td>5f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_T4F_V4F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>4</td>
<td>4</td>
<td>4f</td>
<td>GL_FLOAT         0</td>
<td>4f</td>
<td>8f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_T2F_C4UB_V3F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>GL_UNSIGNED_BYTE 2f</td>
<td>c+2f</td>
<td>c+5f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_T2F_C3F_V3F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>GL_FLOAT         2f</td>
<td>5f</td>
<td>8f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_T2F_N3F_V3F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>2</td>
<td>3</td>
<td>2f</td>
<td>GL_FLOAT         2f</td>
<td>5f</td>
<td>8f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL_T2F_C4F_N3F_V3F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>GL_FLOAT         2f</td>
<td>6f</td>
<td>9f</td>
<td>12f</td>
<td></td>
</tr>
<tr>
<td>GL_T4F_C4F_N3F_V4F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>GL_FLOAT         4f</td>
<td>8f</td>
<td>11f</td>
<td>15f</td>
<td></td>
</tr>
</tbody>
</table>

Table 2-5 Variables That Direct glInterleavedArrays()

Start by learning the simpler formats, GL_V2F, GL_V3F, and GL_C3F_V3F. If you use any of the formats with C4UB, you may have to use a struct data
type or do some delicate type casting and pointer math to pack four unsigned bytes into a single 32-bit word.

For some OpenGL implementations, use of interleaved arrays may increase application performance. With an interleaved array, the exact layout of your data is known. You know your data is tightly packed and may be accessed in one chunk. If interleaved arrays are not used, the *stride* and size information has to be examined to detect whether data is tightly packed.

**Note:** `glInterleavedArrays()` only enables and disables vertex arrays and specifies values for the vertex-array data. It does not render anything. You must still complete “Step 3: Dereferencing and Rendering” on page 77 and call `glArrayElement()`, `glDrawElements()`, `glDrawRangeElements()`, or `glDrawArrays()` to dereference the pointers and render graphics.

---

**Buffer Objects**

**Advanced**

There are many operations in OpenGL where you send a large block of data to OpenGL, such as passing vertex array data for processing. Transferring that data may be as simple as copying from your system’s memory down to your graphics card. However, because OpenGL was designed as a client-server model, any time that OpenGL needs data, it will have to be transferred from the client’s memory. If that data doesn’t change, or if the client and server reside on different computers (distributed rendering), that data transfer may be slow, or redundant.

*Buffer objects* were added to OpenGL Version 1.5 to allow an application to explicitly specify which data it would like to be stored in the graphics server.

Many different types of buffer objects are used in the current versions of OpenGL:

- Vertex data in arrays can be stored in server-side buffer objects starting with OpenGL Version 1.5. They are described in “Using Buffer Objects with Vertex-Array Data” on page 102 of this chapter.

- Support for storing pixel data, such as texture maps or blocks of pixels, in buffer objects was added into OpenGL Version 2.1 It is described in “Using Buffer Objects with Pixel Rectangle Data” in Chapter 8.
• Version 3.1 added *uniform buffer objects* for storing blocks of uniform-variable data for use with shaders.

You will find many other features in OpenGL that use the term “objects,” but not all apply to storing blocks of data. For example, texture objects (introduced in OpenGL Version 1.1) merely encapsulate various state settings associated with texture maps (See “Texture Objects” on page 437). Likewise, vertex-array objects, added in Version 3.0, encapsulate the state parameters associated with using vertex arrays. These types of objects allow you to alter numerous state settings with many fewer function calls. For maximum performance, you should try to use them whenever possible, once you’re comfortable with their operation.

**Note:** An object is referred to by its name, which is an unsigned integer identifier. Starting with Version 3.1, all names must be generated by OpenGL using one of the `glGen*()` routines; user-defined names are no longer accepted.

### Creating Buffer Objects

In OpenGL Version 3.0, any nonzero unsigned integer may used as a buffer object identifier. You may either arbitrarily select representative values or let OpenGL allocate and manage those identifiers for you. Why the difference? By having OpenGL allocate identifiers, you are guaranteed to avoid an already used buffer object identifier. This helps to eliminate the risk of modifying data unintentionally. In fact, OpenGL Version 3.1 requires that all object identifiers be generated, disallowing user-defined names.

To have OpenGL allocate buffer objects identifiers, call `glGenBuffers()`.

```c
void glGenBuffers(GLsizei n, GLuint *buffers);
```

Returns *n* currently unused names for buffer objects in the array *buffers*. The names returned in *buffers* do not have to be a contiguous set of integers.

The names returned are marked as used for the purposes of allocating additional buffer objects, but only acquire a valid state once they have been bound.

Zero is a reserved buffer object name and is never returned as a buffer object by `glGenBuffers()`.
You can also determine whether an identifier is a currently used buffer object identifier by calling `glIsBuffer()`.

```c
GLboolean glIsBuffer(GLuint buffer);
```

Returns GL_TRUE if `buffer` is the name of a buffer object that has been bound, but has not been subsequently deleted. Returns GL_FALSE if `buffer` is zero or if `buffer` is a nonzero value that is not the name of a buffer object.

### Making a Buffer Object Active

To make a buffer object active, it needs to be bound. Binding selects which buffer object future operations will affect, either for initializing data or using that buffer for rendering. That is, if you have more than one buffer object in your application, you’ll likely call `glBindBuffer()` multiple times: once to initialize the object and its data, and then subsequent times either to select that object for use in rendering or to update its data.

To disable use of buffer objects, call `glBindBuffer()` with zero as the buffer identifier. This switches OpenGL to the default mode of not using buffer objects.

```c
void glBindBuffer(GlEnum target, GLuint buffer);
```

Specifies the current active buffer object. `target` must be set to one of GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER, or GL_UNIFORM_BUFFER. `buffer` specifies the buffer object to be bound to.

`glBindBuffer()` does three things: 1. When using `buffer` of an unsigned integer other than zero for the first time, a new buffer object is created and assigned that name. 2. When binding to a previously created buffer object, that buffer object becomes the active buffer object. 3. When binding to a `buffer` value of zero, OpenGL stops using buffer objects.

### Allocating and Initializing Buffer Objects with Data

Once you’ve bound a buffer object, you need to reserve space for storing your data. This is done by calling `glBufferData()`.
glBufferData() first allocates memory in the OpenGL server for storing your data. If you request too much memory, a GL_OUT_OF_MEMORY error will be set. Once the storage has been reserved, and if the data parameter is not NULL, size units of storage (usually bytes) are copied from the client’s memory into the buffer object. However, if you need to dynamically load the data at some point after the buffer is created, pass NULL in for the data pointer. This will reserve the appropriate storage for your data, but leave it uninitialized.
The final parameter to `glBufferData()`, `usage`, is a performance hint to OpenGL. Based upon the value you specify for `usage`, OpenGL may be able to optimize the data for better performance, or it can choose to ignore the hint. There are three operations that can be done to buffer object data:

1. **Drawing**—the client specifies data that is used for rendering.
2. **Reading**—data values are read from an OpenGL buffer (such as the framebuffer) and used in the application in various computations not immediately related to rendering.
3. **Copying**—data values are read from an OpenGL buffer and then used as data for rendering.

Additionally, depending upon how often you intend to update the data, there are various operational hints for describing how often the data will be read or used in rendering:

- **Stream mode**—you specify the data once, and use it only a few times in drawing or other operations.
- **Static mode**—you specify the data once, but use the values often.
- **Dynamic mode**—you may update the data often and use the data values in the buffer object many times as well.

Possible values for `usage` are described in Table 2-6.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_STREAM_DRAW</td>
<td>Data is specified once and used at most a few times as the source of drawing and image specification commands.</td>
</tr>
<tr>
<td>GL_STREAM_READ</td>
<td>Data is copied once from an OpenGL buffer and is used at most a few times by the application as data values.</td>
</tr>
<tr>
<td>GL_STREAM_COPY</td>
<td>Data is copied once from an OpenGL buffer and is used at most a few times as the source for drawing or image specification commands.</td>
</tr>
<tr>
<td>GL_STATIC_DRAW</td>
<td>Data is specified once and used many times as the source of drawing or image specification commands.</td>
</tr>
<tr>
<td>GL_STATIC_READ</td>
<td>Data is copied once from an OpenGL buffer and is used many times by the application as data values.</td>
</tr>
</tbody>
</table>

*Table 2-6* Values for `usage` Parameter of `glBufferData()`
Chapter 2: State Management and Drawing Geometric Objects

Updating Data Values in Buffer Objects

There are two methods for updating data stored in a buffer object. The first method assumes that you have data of the same type prepared in a buffer in your application. `glBufferSubData()` will replace some subset of the data in the bound buffer object with the data you provide.

```c
void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid *data);
```

Update `size` bytes starting at `offset` (also measured in bytes) in the currently bound buffer object associated with `target` using the data pointed to by `data`. `target` must be one of `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`.

`glBufferSubData()` will generate a `GL_INVALID_VALUE` error if `size` is less than zero or if `size + offset` is greater than the original size specified when the buffer object was created.

The second method allows you more control over which data values are updated in the buffer. `glMapBuffer()` and `glMapBufferRange()` return a pointer to the buffer object memory, into which you can write new values...
(or simply read the data, depending on your choice of memory access permissions), just as if you were assigning values to an array. When you’ve completed updating the values in the buffer, you call `glUnmapBuffer()` to signify that you’ve completed updating the data.

`glMapBuffer()` provides access to the entire set of data contained in the buffer object. This approach is useful if you need to modify much of the data in buffer, but may be inefficient if you have a large buffer and need to update only a small portion of the values.

When you’ve completed accessing the storage, you can unmap the buffer by calling `glUnmapBuffer()`.

As a simple example of how you might selectively update elements of your data, we’ll use `glMapBuffer()` to obtain a pointer to the data in a buffer object containing three-dimensional positional coordinates, and then update only the z-coordinates.
GLfloat* data;

data = (GLfloat*) glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);

if (data != (GLfloat*) NULL) {
    for (i = 0; i < 8; ++i )
        data[3*i+2] *= 2.0;  /* Modify Z values */
    glUnmapBuffer(GL_ARRAY_BUFFER);
} else {
    /* Handle not being able to update data */
}

If you need to update only a relatively small number of values in the buffer (as compared to its total size), or small contiguous ranges of values in a very large buffer object, it may be more efficient to use glMapBufferRange(). It allows you to map only the range of data values you need.

GLvoid *glMapBufferRange(GLenum target, GLintptr offset,
                          GLsizeiptr length, GLbitfield access);

Returns a pointer into the data storage for the currently bound buffer object associated with target, which must be one of GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER, or GL_UNIFORM_BUFFER. offset and length specify the range to be mapped. access is a bitmask composed of GL_MAP_READ_BIT, GL_MAP_WRITE_BIT, which indicate the operations that a client may do on the data, and optionally GL_MAP_INVALIDATE_RANGE_BIT, GL_MAP_INVALIDATE_BUFFER_BIT, GL_MAP_FLUSH_EXPLICIT_BIT, or GL_MAP_UNSYNCHRONIZED_BIT, which provide hints on how OpenGL should manage the data in the buffer.

glMapBufferRange() will return NULL if an error occurs. GL_INVALID_VALUE is generated if offset or length are negative, or offset+length is greater than the buffer size. GL_OUT_OF_MEMORY error is generated if adequate memory cannot be obtained to map the buffer. GL_INVALID_OPERATION is generated if any of the following occur: The buffer is already mapped; access does not have either GL_MAP_READ_BIT or GL_MAP_WRITE_BIT set; access has GL_MAP_READ_BIT set and any of GL_MAP_INVALIDATE_RANGE_BIT, GL_MAP_INVALIDATE_BUFFER_BIT, or GL_MAP_UNSYNCHRONIZED_BIT is also set; or both GL_MAP_WRITE_BIT and GL_MAP_FLUSH_EXPLICIT_BIT are set in access.
Using `glMapBufferRange()`, you can specify optional hints by setting additional bits within `access`. These flags describe how the OpenGL server needs to preserve data that was originally in the buffer before you mapped it. The hints are meant to aid the OpenGL implementation in determining which data values it needs to retain, or for how long, to keep any internal copies of the data correct and consistent.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_MAP_INVALIDATE_RANGE_BIT</td>
<td>Specify that the previous values in the mapped range may be discarded, but preserve the other values within the buffer. Data within this range are undefined unless explicitly written. No OpenGL error is generated if later OpenGL calls access undefined data, and the results of such calls are undefined (but may cause application or system errors). This flag may not be used in conjunction with the GL_READ_BIT.</td>
</tr>
<tr>
<td>GL_MAP_INVALIDATE_BUFFER_BIT</td>
<td>Specify that the previous values of the entire buffer may be discarded, and all values with the buffer are undefined unless explicitly written. No OpenGL error is generated if later OpenGL calls access undefined data, and the results of such calls are undefined (but may cause application or system errors). This flag may not be used in conjunction with the GL_READ_BIT.</td>
</tr>
<tr>
<td>GL_MAP_FLUSH_EXPLICIT_BIT</td>
<td>Indicate that discrete ranges of the mapped region may be updated, that the application will signal when modifications to a range should be considered completed by calling <code>glFlushMappedBufferRange()</code>. No OpenGL error is generated if a range of the mapped buffer is updated but not flushed, however, the values are undefined until flushed. Using this option will require any modified ranges to be explicitly flushed to the OpenGL server—<code>glUnmapBuffer()</code> will not automatically flush the buffer's data.</td>
</tr>
</tbody>
</table>

*Table 2-7*  Values for the *access* Parameter of `glMapBufferRange()`
Chapter 2: State Management and Drawing Geometric Objects

As described in Table 2-7, specifying GL_MAP_FLUSH_EXPLICIT_BIT in the access flags when mapping a buffer region with glMapBufferRange() requires ranges modified within the mapped buffer to be indicated to the OpenGL by a call to glFlushMappedBufferRange(). No OpenGL errors are generated for the pending operations that access or modify the mapped region, but the results of those operations is undefined.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_MAP_UNSYNCHRONIZED_BIT</td>
<td>Specify that OpenGL should not attempt to synchronize pending operations on a buffer (e.g., updating data with a call to glBufferData(), or the application is trying to use the data in the buffer for rendering) until the call to glMapBufferRange() has completed. No OpenGL errors are generated for the pending operations that access or modify the mapped region, but the results of those operations is undefined.</td>
</tr>
</tbody>
</table>

Table 2-7 (continued) Values for the access Parameter of glMapBufferRange()

As described in Table 2-7, specifying GL_MAP_FLUSH_EXPLICIT_BIT in the access flags when mapping a buffer region with glMapBufferRange() requires ranges modified within the mapped buffer to be indicated to the OpenGL by a call to glFlushMappedBufferRange().

GLvoid glFlushMappedBufferRange(GLenum target, GLintptr offset, GLsizei length);

Signal that values within a mapped buffer range have been modified, which may cause the OpenGL server to update cached copies of the buffer object. target must be one of the following: GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER, or GL_UNIFORM_BUFFER. offset and length specify the range of the mapped buffer region, relative to the beginning of the mapped range of the buffer.

A GL_INVALID_VALUE error is generated if offset or length is negative or if offset+length is greater than the size of the mapped region. A GL_INVALID_OPERATION error is generated if there is no buffer bound to target (i.e., zero was specified as the buffer to be bound in a call to glBindBuffer() for target), or if the buffer bound to target is not mapped, or if it is mapped without having set the GL_MAP_FLUSH_EXPLICIT_BIT.
Copying Data Between Buffer Objects

On some occasions, you may need to copy data from one buffer object to another. In versions of OpenGL prior to Version 3.1, this would be a two-step process:

1. Copy the data from the buffer object into memory in your application. You would do this either by mapping the buffer and copying it into a local memory buffer, or by calling `glGetBufferSubData()` to copy the data from the server.

2. Update the data in another buffer object by binding to the new object and then sending the new data using `glBufferData()` (or `glBufferSubData()` if you’re replacing only a subset). Alternatively, you could map the buffer, and then copy the data from a local memory buffer into the mapped buffer.

In OpenGL Version 3.1, the `glCopyBufferSubData()` command copies data without forcing it to make a temporary stop in your application’s memory.

```c
void glCopyBufferSubData(GLenum readbuffer, GLenum writebuffer,
                        GLintptr readoffset, GLintptr writeoffset,
                        GLsizeiptr size);
```

Copy data from the buffer object associated with `readbuffer` to the buffer object bound to `writebuffer`. `readbuffer` and `writebuffer` must be one of `GL_ARRAY_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TEXTURE_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`.

`readoffset` and `size` specify the amount of data copied into the destination buffer object, replacing the same size of data starting at `writeoffset`.

Numerous situations will cause a GL_INVALID_VALUE error to be generated: `readoffset`, `writeoffset`, or `size` being negative; `readoffset + size` exceeding the extent of the buffer object bound to `readbuffer`; `writeoffset + size` exceeding the extent of the buffer object bound to `writebuffer`; or if `readbuffer` and `writebuffer` are bound to the same object, and the regions specified by `readoffset` and `size` overlap the region defined by `writeoffset` and `size`.

A GL_INVALID_OPERATION error is generated if either `readbuffer` or `writebuffer` is bound to zero, or either buffer is currently mapped.
Cleaning Up Buffer Objects

When you’re finished with a buffer object, you can release its resources and make its identifier available by calling `glDeleteBuffers()`. Any bindings to currently bound objects that are deleted are reset to zero.

```c
void glDeleteBuffers(GLsizei n, const GLuint *buffers);
```

Deletes `n` buffer objects, named by elements in the array `buffers`. The freed buffer objects may now be reused (for example, by `glGenBuffers()`).

If a buffer object is deleted while bound, all bindings to that object are reset to the default buffer object, as if `glBindBuffer()` had been called with zero as the specified buffer object. Attempts to delete nonexistent buffer objects or the buffer object named zero are ignored without generating an error.

Using Buffer Objects with Vertex-Array Data

To store your vertex-array data in buffer objects, you will need to add a few steps to your application.

1. (Optional) Generate buffer object identifiers.
2. Bind a buffer object, specifying that it will be used for either storing vertex data or indices.
3. Request storage for your data, and optionally initialize those data elements.
4. Specify offsets relative to the start of the buffer object to initialize the vertex-array functions, such as `glVertexPointer()`.
5. Bind the appropriate buffer object to be utilized in rendering.
6. Render using an appropriate vertex-array rendering function, such as `glDrawArrays()` or `glDrawElements()`.

If you need to initialize multiple buffer objects, you will repeat steps 2 through 4 for each buffer object.

Both “formats” of vertex-array data are available for use in buffer objects. As described in “Step 2: Specifying Data for the Arrays,” vertex, color, lighting normal, or any other type of associated vertex data can be stored in a buffer.
object. Additionally, interleaved vertex array data, as described in “Interleaved Arrays,” can also be stored in a buffer object. In either case, you would create a single buffer object to hold all of the data to be used as vertex arrays.

As compared to specifying a memory address in the client’s memory where OpenGL should access the vertex-array data, you specify the offset in machine units (usually bytes) to the data in the buffer. To help illustrate computing the offset, and to frustrate the purists in the audience, we’ll use the following macro to simplify expressing the offset:

```
#define BUFFER_OFFSET(bytes)  ((GLubyte*) NULL + (bytes))
```

For example, if you had floating-point color and position data for each vertex, perhaps represented as the following array

```
GLfloat vertexData[][6] = {
    { R_0, G_0, B_0, X_0, Y_0, Z_0 },
    { R_1, G_1, B_1, X_1, Y_1, Z_1 },
    ...
    { R_n, G_n, B_n, X_n, Y_n, Z_n }
};
```

that were used to initialize the buffer object, you could specify the data as two separate vertex array calls, one for colors and one for vertices:

```
glColorPointer(3, GL_FLOAT, 6*sizeof(GLfloat), BUFFER_OFFSET(0));
glVertexPointer(3, GL_FLOAT, 6*sizeof(GLfloat),
                   BUFFER_OFFSET(3*sizeof(GLfloat));
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

Conversely, since the data in `vertexData` matches a format for an interleaved vertex array, you could use `glInterleavedArrays()` for specifying the vertex-array data:

```
glInterleavedArrays(GL_C3F_V3F, 0, BUFFER_OFFSET(0));
```

Putting this all together, Example 2-17 demonstrates how buffer objects of vertex data might be used. The example creates two buffer objects, one containing vertex data and the other containing index data.

**Example 2-17**  Using Buffer Objects with Vertex Data

```
#define VERTICES     0
#define INDICES      1
#define NUM_BUFFERS  2
```
GLuint buffers[NUM_BUFFERS];

GLfloat vertices[][3] = {
    { -1.0, -1.0, -1.0 },
    {  1.0, -1.0, -1.0 },
    {  1.0,  1.0, -1.0 },
    { -1.0,  1.0, -1.0 },
    { -1.0, -1.0,  1.0 },
    {  1.0, -1.0,  1.0 },
    {  1.0,  1.0,  1.0 },
    { -1.0,  1.0,  1.0 }
};

GLubyte indices[][4] = {
    { 0, 1, 2, 3 },
    { 4, 7, 6, 5 },
    { 0, 4, 5, 1 },
    { 3, 2, 6, 7 },
    { 0, 3, 7, 4 },
    { 1, 5, 6, 2 }
};

glGenBuffers(NUM_BUFFERS, buffers);

glBindBuffer(GL_ARRAY_BUFFER, buffers[VERTICES]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glEnableClientState(GL_VERTEX_ARRAY);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[INDICES]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices
GL_STATIC_DRAW);

glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE,
BUFFER_OFFSET(0));

**Vertex-Array Objects**

As your programs grow larger and use more models, you will probably find that you switch between multiple sets of vertex arrays each frame. Depending on how many vertex attributes you’re using for each vertex, the number of calls—such as to `glVertexPointer()`—may start to become large.
Vertex-array objects bundle collections of calls for setting the vertex array’s state. After being initialized, you can quickly change between different sets of vertex arrays with a single call.

To create a vertex-array object, first call `glGenVertexArrays()`, which will create the requested number of uninitialized objects:

```c
void glGenVertexArrays(GLsizei n, GLuint *arrays);
```

Returns \(n\) currently unused names for use as vertex-array objects in the array `arrays`. The names returned are marked as used for the purposes of allocating additional buffer objects, and initialized with values representing the default state of the collection of uninitialized vertex arrays.

After creating your vertex-array objects, you’ll need to initialize the new objects, and associate the set of vertex-array data that you want to enable with the individual allocated objects. You do this with the `glBindVertexArray()` routine. Once you initialize all of your vertex-array objects, you can use `glBindVertexArray()` to switch between the different sets of vertex arrays that you’ve set up.

```c
GLvoid glBindVertexArray(GLuint array);
```

`glBindVertexArray()` does three things. When using the value `array` that is other than zero and was returned from `glGenVertexArrays()`, a new vertex-array object is created and assigned that name. When binding to a previously created vertex-array object, that vertex array object becomes active, which additionally affects the vertex array state stored in the object. When binding to an `array` value of zero, OpenGL stops using vertex-array objects and returns to the default state for vertex arrays.

A GL_INVALID_OPERATION error is generated if `array` is not a value previously returned from `glGenVertexArrays()`, or if it is a value that has been released by `glDeleteVertexArrays()`, or if any of the `gl*Pointer()` routines are called to specify a vertex array that is not associated with a buffer object while a non-zero vertex-array object is bound (i.e., using a client-side vertex array storage).

Example 2-18 demonstrates switching between two sets of vertex arrays using vertex-arrays objects.
Example 2-18  Using Vertex-Array Objects: vao.c

```c
#define BUFFER_OFFSET(offset)   ((GLvoid*) NULL + offset)
#define NumberOf(array)         (sizeof(array)/sizeof(array[0]))

typedef struct {  
    GLfloat x, y, z;
} vec3;

typedef struct {  
    vec3    xlate;   /* Translation */  
    GLfloat angle;  
    vec3    axis;
} XForm;

eenum { Cube, Cone, NumVAOs };  
GLuint    VAO[NumVAOs];  
GLenum   PrimType[NumVAOs];  
GLsizei  NumElements[NumVAOs];  
XForm    Xform[NumVAOs] = {  
    { { -2.0, 0.0, 0.0 }, 0.0, { 0.0, 1.0, 0.0 } },  
    { { 0.0, 0.0, 2.0 }, 0.0, { 1.0, 0.0, 0.0 } }  
};  
GLfloat  Angle = 0.0;

void  
init()  
{
    enum { Vertices, Colors, Elements, NumVBOs };  
    GLuint   buffers[NumVBOs];  

    glGenVertexArrays(NumVAOs, VAO);

    {
        GLfloat  cubeVerts[][3] = {
            { -1.0, -1.0, -1.0 },  
            { -1.0, -1.0, 1.0 },  
            { -1.0, 1.0, -1.0 },  
            { -1.0, 1.0, 1.0 },  
            { 1.0, -1.0, -1.0 },  
            { 1.0, -1.0, 1.0 },  
            { 1.0, 1.0, -1.0 },  
            { 1.0, 1.0, 1.0 },
        };
    }

    glGenBuffers(NumVBOs, buffers);
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    glBufferData(GL_ARRAY_BUFFER, len, cubeVerts, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glUseProgram(program);
    glUniformMatrix4fv(loc_trans, 1, GL_FALSE, &Xform[0].xlate[0][0]);
}
```
GLfloat cubeColors[][3] = {
    { 0.0,  0.0,  0.0 },
    { 0.0,  0.0,  1.0 },
    { 0.0,  1.0,  0.0 },
    { 0.0,  1.0,  1.0 },
    { 1.0,  0.0,  0.0 },
    { 1.0,  0.0,  1.0 },
    { 1.0,  1.0,  0.0 },
    { 1.0,  1.0,  1.0 },
};

GLubyte cubeIndices[] = {
    0, 1, 3, 2,
    4, 6, 7, 5,
    2, 3, 7, 6,
    0, 4, 5, 1,
    0, 2, 6, 4,
    1, 5, 7, 3
};

glBindVertexArray(VAO[Cube]);
glGenBuffers(NumVBOs, buffers);
glBindBuffer(GL_ARRAY_BUFFER, buffers[Vertices]);
glBufferData(GL_ARRAY_BUFFER, sizeof(cubeVerts),
              cubeVerts, GL_STATIC_DRAW);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glEnableClientState(GL_VERTEX_ARRAY);

glBindBuffer(GL_ARRAY_BUFFER, buffers[Colors]);
glBufferData(GL_ARRAY_BUFFER, sizeof(cubeColors),
              cubeColors, GL_STATIC_DRAW);
glColorPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glEnableClientState(GL_COLOR_ARRAY);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[Elements]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(cubeIndices),
             cubeIndices, GL_STATIC_DRAW);

PrimType[Cube] = GL_QUADS;
NumElements[Cube] = NumberOf(cubeIndices);
}

{ int i, idx;
  float dTheta;
#define NumConePoints 36
/* We add one more vertex for the cone's apex */
GLfloat coneVerts[NumConePoints+1][3] = {
    {0.0, 0.0, 1.0}
};
GLfloat coneColors[NumConePoints+1][3] = {
    {1.0, 1.0, 1.0}
};
GLubyte coneIndices[NumConePoints+1];
dTheta = 2*M_PI / (NumConePoints - 1);
idx = 1;
for (i = 0; i < NumConePoints; ++i, ++idx) {
    float theta = i*dTheta;
    coneVerts[idx][0] = cos(theta);
    coneVerts[idx][1] = sin(theta);
    coneVerts[idx][2] = 0.0;
    coneColors[idx][0] = cos(theta);
    coneColors[idx][1] = sin(theta);
    coneColors[idx][2] = 0.0;
    coneIndices[idx] = idx;
}

glBindVertexArray(VAO[Cone]);
glGenBuffers(NumVBOs, buffers);
glBindBuffer(GL_ARRAY_BUFFER, buffers[Vertices]);
glBufferData(GL_ARRAY_BUFFER, sizeof(coneVerts), coneVerts, GL_STATIC_DRAW);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glEnableClientState(GL_VERTEX_ARRAY);

glBindBuffer(GL_ARRAY_BUFFER, buffers[Colors]);
glBufferData(GL_ARRAY_BUFFER, sizeof(coneColors), coneColors, GL_STATIC_DRAW);
glColorPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glEnableClientState(GL_COLOR_ARRAY);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[Elements]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(coneIndices), coneIndices, GL_STATIC_DRAW);

PrimType[Cone] = GL_TRIANGLE_FAN;
NumElements[Cone] = Numberof(coneIndices);
}

glEnable(GL_DEPTH_TEST);
void display()
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glRotatef(Angle, 0.0, 1.0, 0.0);

    for (i = 0; i < NumVAOs; ++i) {
        glPushMatrix();
        glTranslatef(Xform[i].xlate.x, Xform[i].xlate.y, Xform[i].xlate.z);
        glRotatef(Xform[i].angle, Xform[i].axis.x, Xform[i].axis.y, Xform[i].axis.z);
        glBindVertexArray(VAO[i]);
        glDrawElements(PrimType[i], NumElements[i], GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));
        glPopMatrix();
    }

    glPopMatrix();
    glutSwapBuffers();
}

To delete vertex-array objects and release their names for reuse, call `glDeleteVertexArrays()`. If you're using buffer objects for storing data, they are not deleted when the vertex-array object referencing them is deleted. They continue to exist (until you delete them). The only change that occurs is if the buffer objects were currently bound when you deleted the vertex-array object, they become unbound.

```c
void glDeleteVertexArrays(GLsizei n, GLuint *arrays);
```

Deletes the `n` vertex-arrays objects specified in `arrays`, enabling the names for reuse as vertex arrays later. If a bound vertex array is deleted, the bindings for that vertex array become zero (as if you had called `glBindBuffer()` with a value of zero), and the default vertex array becomes the current one. Unused names in arrays are released, but no changes to the current vertex array state are made.
Finally, if you need to determine whether a particular value might represent an allocated (but not necessarily initialized) vertex-array object, you can check by calling `glIsVertexArray()`.

```c
GLboolean glIsVertexArray(GLuint array);
```

Returns GL_TRUE if `array` is the name of a vertex-array object that was previously generated with `glGenVertexArrays()`, but has not been subsequently deleted. Returns GL_FALSE if `array` is zero or a nonzero value that is not the name of a vertex-array object.

### Attribute Groups

In “Basic State Management” you saw how to set or query an individual state or state variable. You can also save and restore the values of a collection of related state variables with a single command.

OpenGL groups related state variables into an attribute group. For example, the GL_LINE_BIT attribute consists of five state variables: the line width, the GL_LINE_STIPPLE enable status, the line stipple pattern, the line stipple repeat counter, and the GL_LINE_SMOOTH enable status. (See “Antialiasing” in Chapter 6.) With the commands `glPushAttrib()` and `glPopAttrib()`, you can save and restore all five state variables at once.

Some state variables are in more than one attribute group. For example, the state variable GL_CULL_FACE is part of both the polygon and the enable attribute groups.

In OpenGL Version 1.1, there are now two different attribute stacks. In addition to the original attribute stack (which saves the values of server state variables), there is also a client attribute stack, accessible by the commands `glPushClientAttrib()` and `glPopClientAttrib()`.

In general, it’s faster to use these commands than to get, save, and restore the values yourself. Some values might be maintained in the hardware, and getting them might be expensive. Also, if you’re operating on a remote client, all the attribute data has to be transferred across the network connection and back as it is obtained, saved, and restored. However, your OpenGL implementation keeps the attribute stack on the server, avoiding unnecessary network delays.
There are about 20 different attribute groups, which can be saved and restored by `glPushAttrib()` and `glPopAttrib()`. There are two client attribute groups, which can be saved and restored by `glPushClientAttrib()` and `glPopClientAttrib()`. For both server and client, the attributes are stored on a stack, which has a depth of at least 16 saved attribute groups. (The actual stack depths for your implementation can be obtained using `GL_MAX_ATTRIB_STACK_DEPTH` and `GL_MAX_CLIENT_ATTRIB_STACK_DEPTH` with `glGetIntegerv()`.) Pushing a full stack or popping an empty one generates an error.

(See the tables in Appendix B to find out exactly which attributes are saved for particular mask values—that is, which attributes are in a particular attribute group.)

```c
void glPushAttrib(GLbitfield mask);
void glPopAttrib(void);
```

`glPushAttrib()` saves all the attributes indicated by bits in `mask` by pushing them onto the attribute stack. `glPopAttrib()` restores the values of those state variables that were saved with the last `glPushAttrib()`. Table 2-8 lists the possible mask bits that can be logically ORed together to save any combination of attributes. Each bit corresponds to a collection of individual state variables. For example, `GL_LIGHTING_BIT` refers to all the state variables related to lighting, which include the current material color; the ambient, diffuse, specular, and emitted light; a list of the lights that are enabled; and the directions of the spotlights. When `glPopAttrib()` is called, all these variables are restored.

The special mask `GL_ALL_ATTRIB_BITS` is used to save and restore all the state variables in all the attribute groups.

<table>
<thead>
<tr>
<th>Mask Bit</th>
<th>Attribute Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_ACCUM_BUFFER_BIT</td>
<td>accum-buffer</td>
</tr>
<tr>
<td>GL_ALL_ATTRIB_BITS</td>
<td>—</td>
</tr>
<tr>
<td>GL_COLOR_BUFFER_BIT</td>
<td>color-buffer</td>
</tr>
<tr>
<td>GL_CURRENT_BIT</td>
<td>current</td>
</tr>
<tr>
<td>GL_DEPTH_BUFFER_BIT</td>
<td>depth-buffer</td>
</tr>
</tbody>
</table>

Table 2-8  Attribute Groups
### Table 2-8 (continued)  Attribute Groups

<table>
<thead>
<tr>
<th>Mask Bit</th>
<th>Attribute Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_Enable_BIT</td>
<td>enable</td>
</tr>
<tr>
<td>GL_Eval_BIT</td>
<td>eval</td>
</tr>
<tr>
<td>GL_Fog_BIT</td>
<td>fog</td>
</tr>
<tr>
<td>GL_Hint_BIT</td>
<td>hint</td>
</tr>
<tr>
<td>GL_Lighting_BIT</td>
<td>lighting</td>
</tr>
<tr>
<td>GL_Line_BIT</td>
<td>line</td>
</tr>
<tr>
<td>GL_List_BIT</td>
<td>list</td>
</tr>
<tr>
<td>GL_Multisample_BIT</td>
<td>multisample</td>
</tr>
<tr>
<td>GL_Pixel_Mode_BIT</td>
<td>pixel</td>
</tr>
<tr>
<td>GL_Point_BIT</td>
<td>point</td>
</tr>
<tr>
<td>GL_Polygon_BIT</td>
<td>polygon</td>
</tr>
<tr>
<td>GL_Polygon_Stipple_BIT</td>
<td>polygon-stipple</td>
</tr>
<tr>
<td>GL_Scissor_BIT</td>
<td>scissor</td>
</tr>
<tr>
<td>GL_Stencil_Buffer_BIT</td>
<td>stencil-buffer</td>
</tr>
<tr>
<td>GL_Texture_BIT</td>
<td>texture</td>
</tr>
<tr>
<td>GL_Transform_BIT</td>
<td>transform</td>
</tr>
<tr>
<td>GL_Viewport_BIT</td>
<td>viewport</td>
</tr>
</tbody>
</table>

void glPushClientAttrib(GLbitfield mask);
void glPopClientAttrib(void);

`glPushClientAttrib()` saves all the attributes indicated by bits in `mask` by pushing them onto the client attribute stack. `glPopClientAttrib()` restores the values of those state variables that were saved with the last `glPushClientAttrib()`. Table 2-9 lists the possible mask bits that can be logically ORed together to save any combination of client attributes.

Two client attribute groups, feedback and select, cannot be saved or restored with the stack mechanism.
Some Hints for Building Polygonal Models of Surfaces

Following are some techniques that you can use as you build polygonal approximations of surfaces. You might want to review this section after you’ve read Chapter 5 on lighting and Chapter 7 on display lists. The lighting conditions affect how models look once they’re drawn, and some of the following techniques are much more efficient when used in conjunction with display lists. As you read these techniques, keep in mind that when lighting calculations are enabled, normal vectors must be specified to get proper results.

Constructing polygonal approximations to surfaces is an art, and there is no substitute for experience. This section, however, lists a few pointers that might make it a bit easier to get started.

- Keep polygon orientations (windings) consistent. Make sure that when viewed from the outside, all the polygons on the surface are oriented in the same direction (all clockwise or all counterclockwise). Consistent orientation is important for polygon culling and two-sided lighting. Try to get this right the first time, as it’s excruciatingly painful to fix the problem later. (If you use `glScale*()` to reflect geometry around some axis of symmetry, you might change the orientation with `glFrontFace()` to keep the orientations consistent.)

- When you subdivide a surface, watch out for any nontriangular polygons. The three vertices of a triangle are guaranteed to lie on a plane; any polygon with four or more vertices might not. Nonplanar

<table>
<thead>
<tr>
<th>Mask Bit</th>
<th>Attribute Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_CLIENT_PIXEL_STORE_BIT</td>
<td>pixel-store</td>
</tr>
<tr>
<td>GL_CLIENT_VERTEX_ARRAY_BIT</td>
<td>vertex-array</td>
</tr>
<tr>
<td>GL_CLIENT_ALL_ATTRIB_BITS</td>
<td>--</td>
</tr>
<tr>
<td>can’t be pushed or popped</td>
<td>feedback</td>
</tr>
<tr>
<td>can’t be pushed or popped</td>
<td>select</td>
</tr>
</tbody>
</table>

Table 2-9  Client Attribute Groups
polygons can be viewed from some orientation such that the edges cross each other, and OpenGL might not render such polygons correctly.

- There’s always a trade-off between the display speed and the quality of the image. If you subdivide a surface into a small number of polygons, it renders quickly but might have a jagged appearance; if you subdivide it into millions of tiny polygons, it probably looks good but might take a long time to render. Ideally, you can provide a parameter to the subdivision routines that indicates how fine a subdivision you want, and if the object is farther from the eye, you can use a coarser subdivision. Also, when you subdivide, use large polygons where the surface is relatively flat, and small polygons in regions of high curvature.

- For high-quality images, it’s a good idea to subdivide more on the silhouette edges than in the interior. If the surface is to be rotated relative to the eye, this is tougher to do, as the silhouette edges keep moving. Silhouette edges occur where the normal vectors are perpendicular to the vector from the surface to the viewpoint—that is, when their vector dot product is zero. Your subdivision algorithm might choose to subdivide more if this dot product is near zero.

- Try to avoid T-intersections in your models (see Figure 2-16). As shown, there’s no guarantee that the line segments AB and BC lie on exactly the same pixels as the segment AC. Sometimes they do, and sometimes they don’t, depending on the transformations and orientation. This can cause cracks to appear intermittently in the surface.

![Figure 2-16](image)  Modifying an Undesirable T-Intersection
• If you’re constructing a closed surface, be sure to use exactly the same numbers for coordinates at the beginning and end of a closed loop, or you can get gaps and cracks due to numerical round-off. Here’s an example of bad code for a two-dimensional circle:

```c
/* don’t use this code */
#define PI 3.14159265
#define EDGES 30

/* draw a circle */
gBegin(GL_LINE_STRIP);
for (i = 0; i <= EDGES; i++)
    glVertex2f(cos((2*PI*i)/EDGES), sin((2*PI*i)/EDGES));
gEnd();
The edges meet exactly only if your machine manages to calculate exactly the same values for the sine and cosine of 0 and of (2*PI*EDGES/EDGES). If you trust the floating-point unit on your machine to do this right, the authors have a bridge they’d like to sell you... To correct the code, make sure that when i == EDGES, you use 0 for the sine and cosine, not 2*PI*EDGES/EDGES. (Or simpler still, use GL_LINE_LOOP instead of GL_LINE_STRIP, and change the loop termination condition to i < EDGES.)

An Example: Building an Icosahedron

To illustrate some of the considerations that arise in approximating a surface, let’s look at some example code sequences. This code concerns the vertices of a regular icosahedron (which is a Platonic solid composed of 20 faces that span 12 vertices, the face of each being an equilateral triangle). An icosahedron can be considered a rough approximation of a sphere. Example 2-19 defines the vertices and triangles making up an icosahedron and then draws the icosahedron.

Example 2-19  Drawing an Icosahedron

```c
#define X .525731112119133606
#define Z .850650808352039932

static GLfloat vdata[12][3] = {
    {-X, 0.0, Z}, {X, 0.0, Z}, {-X, 0.0, -Z}, {X, 0.0, -Z},
    {0.0, Z, X}, {0.0, Z, -X}, {0.0, -Z, X}, {0.0, -Z, -X},
    {Z, X, 0.0}, {-Z, X, 0.0}, {Z, -X, 0.0}, {-Z, -X, 0.0}
};
```
static GLuint tindices[20][3] = {
    {1, 4, 0}, {4, 9, 0}, {4, 5, 9}, {8, 5, 4}, {1, 8, 4},
    {1, 10, 8}, {10, 3, 8}, {8, 3, 5}, {3, 2, 5}, {3, 7, 2},
    {3, 10, 7}, {10, 6, 7}, {6, 11, 7}, {6, 0, 11}, {6, 1, 0},
    {10, 1, 6}, {11, 0, 9}, {2, 11, 9}, {5, 2, 9}, {11, 2, 7}
};

int i;

glBegin(GL_TRIANGLES);
for (i = 0; i < 20; i++) {
    /* color information here */
    glVertex3fv(&vdata[tindices[i][0]][0]);
    glVertex3fv(&vdata[tindices[i][1]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();

The strange numbers X and Z are chosen so that the distance from the origin to any of the vertices of the icosahedron is 1.0. The coordinates of the 12 vertices are given in the array vdata[][], where the zeroth vertex is \( \{X, 0.0, Z\} \), the first is \( \{X, 0.0, Z\} \), and so on. The array tindices[][] tells how to link the vertices to make triangles. For example, the first triangle is made from the zeroth, fourth, and first vertices. If you take the vertices for triangles in the order given, all the triangles have the same orientation.

The line that mentions color information should be replaced by a command that sets the color of the \( i \)th face. If no code appears here, all faces are drawn in the same color, and it will be impossible to discern the three-dimensional quality of the object. An alternative to explicitly specifying colors is to define surface normals and use lighting, as described in the next subsection.

**Note:** In all the examples described in this section, unless the surface is to be drawn only once, you should probably save the calculated vertex and normal coordinates so that the calculations don’t need to be repeated each time the surface is drawn. This can be done using your own data structures or by constructing display lists (see Chapter 7).

**Calculating Normal Vectors for a Surface**

If a surface is to be lit, you need to supply the vector normal to the surface. Calculating the normalized cross product of two vectors on that surface provides their normal vector. With the flat surfaces of an icosahedron, all
three vertices defining a surface have the same normal vector. In this case, the normal needs to be specified only once for each set of three vertices. The code in Example 2-20 can replace the “color information here” line in Example 2-19 for drawing the icosahedron.

Example 2-20  Generating Normal Vectors for a Surface

```c
GLfloat d1[3], d2[3], norm[3];
for (j = 0; j < 3; j++) {
    d1[j] = vdata[tindices[i][0]][j] - vdata[tindices[i][1]][j];
    d2[j] = vdata[tindices[i][1]][j] - vdata[tindices[i][2]][j];
}
normcrossprod(d1, d2, norm);
glNormal3fv(norm);
```

The function `normcrossprod()` produces the normalized cross product of two vectors, as shown in Example 2-21.

Example 2-21  Calculating the Normalized Cross Product of Two Vectors

```c
void normalize(float v[3])
{
    GLfloat d = sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
    if (d == 0.0) {
        error(“zero length vector”);
        return;
    }
    v[0] /= d;
    v[1] /= d;
    v[2] /= d;
}

void normcrossprod(float v1[3], float v2[3], float out[3])
{
    out[0] = v1[1]*v2[2] - v1[2]*v2[1];
    out[1] = v1[2]*v2[0] - v1[0]*v2[2];
    out[2] = v1[0]*v2[1] - v1[1]*v2[0];
    normalize(out);
}
```

If you’re using an icosahedron as an approximation for a shaded sphere, you’ll want to use normal vectors that are perpendicular to the true surface of the sphere, rather than perpendicular to the faces. For a sphere, the normal vectors are simple; each points in the same direction as the vector from
the origin to the corresponding vertex. Since the icosahedron vertex data is for an icosahedron of radius 1, the normal data and vertex data are identical. Here is the code that would draw an icosahedral approximation of a smoothly shaded sphere (assuming that lighting is enabled, as described in Chapter 5):

```c
glBegin(GL_TRIANGLES);
for (i = 0; i < 20; i++) {
    glNormal3fv(&vdata[tindices[i][0]][0]);
    glVertex3fv(&vdata[tindices[i][0]][0]);
    glNormal3fv(&vdata[tindices[i][1]][0]);
    glVertex3fv(&vdata[tindices[i][1]][0]);
    glNormal3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
}
 glEnd();
```

**Improving the Model**

A 20-sided approximation to a sphere doesn’t look good unless the image of the sphere on the screen is quite small, but there’s an easy way to increase the accuracy of the approximation. Imagine the icosahedron inscribed in a sphere, and subdivide the triangles as shown in Figure 2-17. The newly introduced vertices lie slightly inside the sphere, so push them to the surface by normalizing them (dividing them by a factor to make them have length 1). This subdivision process can be repeated for arbitrary accuracy. The three objects shown in Figure 2-17 use 20, 80, and 320 approximating triangles, respectively.

![Figure 2-17](image-url)  Subdividing to Improve a Polygonal Approximation to a Surface
Example 2-22 performs a single subdivision, creating an 80-sided spherical approximation.

**Example 2-22  Single Subdivision**

```c
void drawtriangle(float *v1, float *v2, float *v3)
{
    glBegin(GL_TRIANGLES);
    glNormal3fv(v1);
    glVertex3fv(v1);
    glNormal3fv(v2);
    glVertex3fv(v2);
    glNormal3fv(v3);
    glVertex3fv(v3);
    glEnd();
}
```

```c
void subdivide(float *v1, float *v2, float *v3)
{
    GLfloat v12[3], v23[3], v31[3];
    GLint i;

    for (i = 0; i < 3; i++) {
        v12[i] = (v1[i]+v2[i])/2.0;
        v23[i] = (v2[i]+v3[i])/2.0;
        v31[i] = (v3[i]+v1[i])/2.0;
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    drawtriangle(v1, v12, v31);
    drawtriangle(v2, v23, v12);
    drawtriangle(v3, v31, v23);
    drawtriangle(v12, v23, v31);
}
```

```c
for (i = 0; i < 20; i++) {
    subdivide(&vdata[tindices[i][0]][0],
        &vdata[tindices[i][1]][0],
        &vdata[tindices[i][2]][0]);
}
```

Example 2-23 is a slight modification of Example 2-22 that recursively subdivides the triangles to the proper depth. If the depth value is 0, no
subdivisions are performed, and the triangle is drawn as is. If the depth is 1, a single subdivision is performed, and so on.

**Example 2-23  Recursive Subdivision**

```c
void subdivide(float *v1, float *v2, float *v3, long depth)
{
    GLfloat v12[3], v23[3], v31[3];
    GLint i;

    if (depth == 0) {
        drawtriangle(v1, v2, v3);
        return;
    }
    for (i = 0; i < 3; i++) {
        v12[i] = (v1[i]+v2[i])/2.0;
        v23[i] = (v2[i]+v3[i])/2.0;
        v31[i] = (v3[i]+v1[i])/2.0;
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    subdivide(v1, v12, v31, depth-1);
    subdivide(v2, v23, v12, depth-1);
    subdivide(v3, v31, v23, depth-1);
    subdivide(v12, v23, v31, depth-1);
}
```

**Generalized Subdivision**

A recursive subdivision technique such as the one described in Example 2-23 can be used for other types of surfaces. Typically, the recursion ends if either a certain depth is reached or some condition on the curvature is satisfied (highly curved parts of surfaces look better with more subdivision).

To look at a more general solution to the problem of subdivision, consider an arbitrary surface parameterized by two variables, \( u[0] \) and \( u[1] \). Suppose that two routines are provided:

```c
void surf(GLfloat u[2], GLfloat vertex[3], GLfloat normal[3]);
float curv(GLfloat u[2]);
```

If \( u[] \) is passed to `surf()`, the corresponding three-dimensional vertex and normal vectors (of length 1) are returned. If \( u[] \) is passed to `curv()`, the curvature of the surface at that point is calculated and returned. (See an
introductory textbook on differential geometry for more information about measuring surface curvature.)

Example 2-24 shows the recursive routine that subdivides a triangle until either the maximum depth is reached or the maximum curvature at the three vertices is less than some cutoff.

**Example 2-24**  Generalized Subdivision

```c
void subdivide(float u1[2], float u2[2], float u3[2],
               float cutoff, long depth)
{
    GLfloat v1[3], v2[3], v3[3], n1[3], n2[3], n3[3];
    GLfloat u12[2], u23[2], u32[2];
    GLint i;

    if (depth == maxdepth || (curv(u1) < cutoff &&
       curv(u2) < cutoff && curv(u3) < cutoff)) {
       surf(u1, v1, n1);
       surf(u2, v2, n2);
       surf(u3, v3, n3);
       glBegin(GL_POLYGON);
           glNormal3fv(n1); glVertex3fv(v1);
           glNormal3fv(n2); glVertex3fv(v2);
           glNormal3fv(n3); glVertex3fv(v3);
       glEnd();
       return;
    }

    for (i = 0; i < 2; i++) {
       u12[i] = (u1[i] + u2[i])/2.0;
       u23[i] = (u2[i] + u3[i])/2.0;
       u31[i] = (u3[i] + u1[i])/2.0;
    }
    subdivide(u1, u12, u31, cutoff, depth+1);
    subdivide(u2, u23, u12, cutoff, depth+1);
    subdivide(u3, u31, u23, cutoff, depth+1);
    subdivide(u12, u23, u31, cutoff, depth+1);
}
```
Numbers

3D Computer Graphics, 390

A

accumulation buffer, 492, 494, 519–525
  clearing, 36, 495
  depth-of-field effect, use for, 521–525
  examples of use, 519
  motion blur, use for, 520
  sample program with depth-of-field effect, 522
AGL, 15
  aglChoosePixelFormat(), 820, 822
  aglConfigure(), 824
  aglCopyContext(), 820, 823
  aglCreateContext(), 820, 823
  aglDescribePixelFormat(), 822
  aglDescribeRenderer(), 823
  aglDestroyContext(), 820, 823
  aglDestroyPixelFormat(), 822
  aglDestroyRendererInfo(), 823
  aglDevicesOfPixelFormat(), 822
  aglDisable(), 824
  aglEnable(), 824
  aglErrorString(), 824
  aglGetCurrentContext(), 820, 823
  aglGetDrawable(), 820, 823
  aglGetError(), 824
  aglGetInteger(), 824
  aglGetVersion(), 820, 822
  aglGetVirtualScreen(), 823
  agIsEnabled(), 824
  aglNextPixelFormat(), 822
  aglNextRendererInfo(), 823
  aglQueryRendererInfo(), 823
  aglResetLibrary(), 824
  aglSetCurrentContext(), 820, 823
  aglSetDrawable(), 820, 821, 823
  aglSetFullScreen(), 821, 823
  aglSetInteger(), 821, 823
  aglSetOffScreen(), 821, 823
  aglSetVirtualScreen(), 823
  aglSwapBuffers(), 821, 824
  aglUpdateContext(), 821, 823
  aglUseFont(), 824
airbrushing, 666
Akeley, Kurt, 519
aliasing, see antialiasing
alpha, 251
  destination alpha, 280
  material properties, 233
  multisampling coverage, 279
texture image data type, 445
alpha blending, see blending
alpha test, 503
  querying current values, 503
  rendering pipeline stage, 14
ambient
  contribution to lighting equation, 243
global light, 228, 242
light, 207, 208, 216
  material properties, 209, 233
animation, 22–25
antialiasing, 267–280
  accumulation buffer used for, 520–??
  characters (by masking), 650
  characters (by texturing), 661
  color-index mode, 272
coverage values, 267
  enabling for points or lines, 269
  enabling for polygons, 280
texture, 270
  lines, 267, 269–275
antialiasing, *continued*
lines (by texturing), 661
points, 269–275, 653
polygons, 280
RGBA mode, 270
sample program in color-index mode, 272
sample program in RGBA mode, 270
Apple Interface to OpenGL, see AGL
ARB imaging subset, 367–388
architectural applications
  orthographic parallel projection, use of, 156
arcs, 559
array elements, see vertex arrays
aspect ratio
  perspective projection, 155
  viewport transformation, 159
atmospheric effects, see fog
attenuation of light, 217–218
attribute groups, 110–112
  client, 110
  list of, 743–800
  multitexturing, with, 470
  server, 110
  stack depth, obtaining, 111
  stacks, 110
auxiliary buffers, 493, 498
basis functions, 571, 572
Bernstein
  basis, 571
  polynomial, 575
Bézier
  basis, 571, 572
  curve, 575
  sample program using mesh for surface, 582
  sample program which draws curve, 573
  sample program which draws surface, 580
BGR and BGRA pixel formats, 336
billboarding, 259, 504
bitmaps, 322–329
  display lists cache bitmap data, 303
distorting, 648
drawing, 327
feedback mode, 629
fonts, used for, 324, 331
imaging pipeline operations, 344
ordering of data in, 325
origin of, 328
sample program, 324
sample program that creates a font, 331
size of, 325
bitplanes, 190, 490
  displayable colors, number of, 192
blending, 251–264, 516
  antialiasing polygons, 280
  controlling blending on a per-buffer basis, 516
  coverage calculations for antialiasing, 267
destination alpha, 280
differences among releases, 250
enabling, 255
enabling for antialiasing, 269
equation, 255
factors (source and destination), 252
images, 653
ordering polygons before drawing, 263
rendering pipeline stage, 14
sample program for three-dimensional, 264
sample program with blended polygons, 261
texture function, 448
three dimensions, in, 263
uses of, 258
buffer object, 833
buffer objects
  binding, 93
  creating, 92
  deleting, 102
  flushing a mapped buffer range, 100
  initializing with data, 94
  mapping a buffer, 97, 98
  partial copy, 101
  replacing data, 96
  unmapping a buffer, 97
buffer, see framebuffer

c
C programming language, 8
CAD/CAM, see computer-aided design
camera analogy, 126–127
  environment mapping, 463
  viewport transformations, 158
capping, see computational solid geometry
characters
  antialiasing, 661
circles, 559
clearing buffers, 36
clearing the color buffer, 35
clearing the framebuffer, 34–37, 495–496
  affected by scissoring, dithering, and masking, 496
client-server, see networked operation
clip coordinates, 128, 170
  feedback mode, 629
clip planes
  user defined, 169
clipping, 158
  interference regions found using clipping planes, 657
  overview, 125
  primitives in rendering pipeline, 12
  viewing volume, 153
clipping planes
  additional clipping planes, 128, 168–172
  far, 154–156, 162
  near, 154–156, 162
  querying number of additional, 170
  sample program with additional clipping planes, 170
color
  alpha values, 251
  background, 34
  cube showing blended RGB values, 189
  current raster color, 329
  human perception, 187
  RGBA values for, 38, 188
  specifying, 37
  specifying for tessellation, 548
  specifying in color-index mode, 199
  specifying in RGBA mode, 197
color buffer, 188, 190, 491, 492, 493
  clearing, 36
  masking, 499
color map, 188, 193
  loading for antialiasing, 272
  loading for smooth shading, 202
  loading, using GLUT, 735
  size of, 194
color matrix, 382–383
  example, 382
  post transform scale and bias, 383
  sample program, 382
color sum mode, 478
color tables, 369–371
proxies, 374
replacing part of, 373
sample program, 371
specifying, 369
color-index mode, 193–195
changing between RGBA mode and, 196
choosing between RGBA mode and, 195
coverage calculations for
antialiasing, 267

dithering, 517
layering with writemasks, 500
lighting, 246–247
lighting calculations in, 247
texturing limitations, 396, 404
vertex arrays, specifying values with, 74
combiner functions, 472–478
command syntax, 7–9
compositing images, 259
compositing transformations, 172–179
computational solid geometry
capping, 509
difference of several contours, 553
interference regions, 656
intersection of two contours, 553
union of several contours, 553
Convex, John, 664
cordinate systems
grand, fixed, 139, 148, 172
local, 139, 148, 172, 176
simple 2D, 40–41
cordinates
see clip coordinates, depth coordinates, eye
cordinates, homogeneous
cordinates, normalized device
cordinates, object cordinates, q
texture coordinates, texture
cordinates, w coordinates, or window
cordinates
coverage, pixel, 267
Coxeter, H. S. M., 803
cross product, 151
CSG, see computational solid geometry
cube maps, 465
culling, 61–62
enabling, 62
rendering pipeline stage, 12
selection mode, 610
curves and curved surfaces, 45
also see evaluators or NURBS
Curves and Surfaces for Computer-Aided
Geometric Design, 571
cylinders, 559

D
data types
RGB color conversion, 197
special OpenGL, 8
texture data, 403
decals, 503, 654
  polygon offset used for, 294
texture function, 447
deprecation mechanism, 27
depth buffer, 206, 492, 494
  also see hidden-surface removal
background, using masking for a common, 500
blending, use for three-dimensional, 263
clearing, 36, 206, 495
decals, for, 654
Dirichlet domains, for, 663
drawing static backgrounds, 662
masking, 499
pixel data, 342, 353
depth coordinates, 129, 161
perspective division, 161
picking use, 621
polygon offset, 293–296
rendering pipeline stage for depth-range operations, 12
  sample program with picking, 621
selection hit records, 610
depth test, 510
  also see depth buffer
  rendering pipeline stage, 14
depth-cuing, see fog
depth-of-field effect, 521–525
  sample program, 522
destination factor, see blending
determining object coordinates from window coordinates, 180, 182
diffuse
  contribution to lighting equation, 243
light, 208, 216
  material properties, 209, 233
directional light source, 217
Dirichlet domains, 663
disks, 559
display lists, 33, 299
  changing mode settings, 319
compiling, 307
creating, 305
deleting, 311
disadvantages, 304, 310
error handling, 306
executing, 305, 310
executing multiple, 312
font creation, 313, 329
hierarchical, 310
immediate mode, mixing with, 310
indices for, obtaining, 306
naming, 306
nesting, 310
nesting limit, querying, 310
networked operation, 309
querying use of an index, 311
rendering use of an index, 311
rendering pipeline stage, 11
sample program creating a font, 314
sample program for creating, 299, 305
sharing among rendering contexts, 814, 826
state variables saved and restored, 319
tessellation, use with, 557
uses for, 303, 319
vertex-array data, 309
what can be stored in, 308
distorted images, 647
texture images, 451
dithering, 192–193, 516
  and clearing, 496
  rendering pipeline stage, 14
dot product
  lighting calculations, use in, 243
texture combiner function, 475
double-buffering, 23–25
  automatic glFlush(), 39
changing between single-buffering and, 196
object selection using the back buffer, 646
querying its presence, 494

Index

861
Index

Index

drawing
  clearing the window, 34
  forcing completion of, 38
  icosahedron, 115
  points, 48
  polygons, 48, 61
  preparing for, 34
  rectangles, 45
  spheres, cylinders, and disks, 559–567
drawing pixel data, see pixel data
Duff, Tom, 259

E
edge flags, 67–68
tessellated polygons generate, 546
  vertex arrays, specifying values with, 74
emission, 208, 234, 241
enabling
  alpha test, 502
  antialiasing of points or lines, 269
  antialiasing polygons, 280
  blending, 255
  color material properties mode, 237
  color sum mode, 478
culling, 62
depth test, 510
dithering, 193, 517
evaluators, 575, 579
fog, 281
lighting, 231
line stippling, 57
logical operations, 517
multisampling, 276
normal vectors for evaluated surfaces,
  automatic generation of, 579, 587
polygon offset, 294
polygon stippling, 63
rescaling normals, 70, 212
stencil test, 504
texture coordinate generation, 462
texturing, 396, 399
  unit length normal vectors ensured, 70,
  212
endianness, 349
environment mapping, 463, 465
erata, xlii
error handling, 637–639
  error string description, 639
evaluators, 572–586
  basis functions, 571, 575
  evenly spaced values, 577, 581
  one-dimensional, 572
  rendering pipeline stage, 11
  sample program using mesh for 2D Bézier
    surface, 582
  sample program which draws 1D Bézier
    curve, 573
  sample program which draws 2D Bézier
    surface, 580
  sample program which generates texture
    coordinates, 584
texture coordinates, generating, 584
texture coordinates, two-dimensional, 580
  event management, using GLUT, 21
extensions
  Microsoft Windows and
    wglGetProcAddress(), 643
  vendor-specific, 641
eye coordinates, 128, 170
texture coordinate generation, 458, 462

F
fade effect, 645
Farin, Gerald E., 571
feedback, 627–634
  array contents, 633
  pass-through markers, 630
  querying current rendering mode, 608
  returned data, 629
sample program, 631
steps to perform, 628
tessellation, obtaining vertex data after, 557
Feiner, Steven K., xl, 803
field of view, 132
calculate, using trigonometry to, 163
filtering, 434–436
mipmapped textures, 424–429, 436
texture border colors, 452
flat shading, 200
flight simulation
fog, use of, 281
flushing, 38
fog, 280–291
blending factors, 284
color-index mode, 285
density, 285
enabling, 281
equations, 284
fog coordinates, 288
hints, 281
RGBA mode, 285
sample program in color-index mode, 286
sample program in RGBA mode, 281
sample program with fog coordinates, 289
Foley, James D., xl, 187, 191, 803
fonts, 329–332
antialiased characters (by masking), 650
antialiased characters (by texturing), 661
bitmapped, 331
creating with display lists, 313
drawing, 328
drawing as bitmaps, 324
multi-byte, 330
same program, 331
sample program using multiple display lists, 314
X fonts, using, 816
Foran, Jim, 482
foreshortening, perspective, 153
fragments, 14, 490
alpha test, 503
blending, 251
depth test, 510
rendering pipeline operations, 14
scissor test, 502
tests, 501–518
texture functions, 446
framebuffer, 190, 491
capacity per pixel, 492
clearing, 495–496
copying pixel data within, 333, 342, 344
enabling for reading, 497
enabling for writing, 497
minimum configuration with the X Window System, 492
querying color resolution, 190
reading pixel data from, 333, 335
writing pixel data to, 333, 341
framebuffer objects, 13
freeglut, xli, 15
front-facing polygons, 61
specifying material property for, 232
two-sided lighting, 229
frustum, 153
ftp (file-transfer protocol) site
GLX specification, 813
Fundamentals of Computer Aided Geometric Design, 571

G
Game of Life, 664
gamma correction, 191
Gardner, Martin, 664
geometric primitives, 42–53
rendering pipeline stage, 12
geosciences
use of texturing in applications, 458
giraffe, 194
GL_VERTEX_PROGRAM_POINT_SIZE, 721
GL_VERTEX_PROGRAM_TWO_SIDE, 722
glAccum(), 519
glActiveTexture(), 469
glAlphaFunc(), 503
glAreTexturesResident(), 442
glArrayElement(), 77
  legal between glBegin() and glEnd(), 52
Glassner, Andrew S., xl
glAttachShader(), 677
glBegin(), 47, 48, 546
  restrictions, 51
glBeginConditionalRender(), 515
glBeginQuery(), 513
glBeginTransformFeedback(), 724
glBindAttribLocation(), 718
glBindBuffer(), 93
  glBufferSubData(), 96
glBindBufferBase(), 696
glBindBufferRange(), 696
glBindFragDataLocation(), 730
  legal between glBegin() and glEnd(), 52
  vertex array objects, 105
glBitmap(), 324, 327, 816
  feedback mode, 629
  fonts, used for, 331
  imaging pipeline operations, 344
  pixel-storage modes effect, 347
  legal between glBegin() and glEnd(), 52
  multitexturing, 469
  vertex array objects, 105
glBindVertexArray(), 447, 48, 546
  legal between glBegin() and glEnd(), 52
  fonts, use for, 329
  sample program, 331
glCheckFramebufferStatus(), 538
glClampColor(), 198
glClear(), 35, 36, 496
  depth buffer, clearing the, 206
  legal between glBegin() and glEnd(), 52
  fog, use with, 286
glClearStencil(), 36, 495
glClientActiveTexture(), 472
glColorMask(), 496, 499
  glColorMaski(), 499
glColorMaterial(), 237
glColorPointer(), 74
glColorSubTable(), 373
  legal between glBegin() and glEnd(), 51
  glColorTable(), 369
  glColorTableParameter(), 371
  legal between glBegin() and glEnd(), 52
  glCompileShader(), 676
glCompressedTexImage1D(), 421
  legal between glBegin() and glEnd(), 52
glCompressedTexImage2D(), 421
  glCompressedTexImage3D(), 421
  glCompressedTexSubImage1D(), 422
  glCompressedTexSubImage2D(), 422
  glCompressedTexSubImage3D(), 422
glConvolutionFilter1D(), 380
glConvolutionFilter2D(), 375
glConvolutionParameter*, 381
glCopyBufferSubData(), 101
glCopyColorSubTable(), 373
glCopyColorTable(), 372
glCopyConvolutionFilter1D(), 380
glCopyConvolutionFilter2D(), 378
glCopyPixels(), 333, 342
  alternative uses, 665
  feedback mode, 629
  glReadBuffer() effect, 499
  imaging pipeline operations, 344
  pixel-transfer modes effect, 352
glCopyTexImage1D(), 413
glReadBuffer() effect, 499
  pixel-transfer modes effect, 352
glCopyTexImage2D(), 405
  glReadBuffer() effect, 499
  pixel-transfer modes effect, 352
glCopyTexSubImage1D(), 414
  glReadBuffer() effect, 499
  pixel-transfer modes effect, 352
glCopyTexSubImage2D(), 411
  glReadBuffer() effect, 499
  pixel-transfer modes effect, 352
glCopyTexSubImage3D(), 417
  pixel-transfer modes effect, 352
glCreateProgram(), 676
glCreateShader(), 675
glCullFace(), 62
glDeleteBuffers(), 102
glDeleteFramebuffers(), 528
glDeleteLists(), 312, 330
glDeleteProgram(), 680
glDeleteQueries(), 514
glDeleteRenderbuffers(), 529
glDeleteShader(), 680
glDeleteTextures(), 442
glDeleteVertexArrays(), 109
glDepthFunc(), 510
glDepthMask(), 499
  blending opaque and translucent objects, 264
  glDepthRange(), 161
  gluUnProject(), relationship to, 180
  glWindowPos*() effect, 326
glDetachShader(), 677
glDisable(), 10, 54
glDisableClientState(), 73
glDisablei(), 516
glDisableVertexAttribArray(), 720
glDrawArrays(), 82
glDrawBuffer(), 341, 343, 498
glDrawBuffers(), 498
glDrawElements(), 78
glDrawPixels(), 333, 340, 506, 662
  alternative uses, 665
  feedback mode, 629
  pixel-storage modes effect, 347
  pixel-transfer modes effect, 352
glDrawRangeElements(), 81
  version, 33
glEdgeFlag*, 68
  legal between glBegin() and glEnd(), 52
  glEdgeFlagPointer(), 74
glEnable(), 10, 54, 213
  also see enabling
  glEnableClientState(), 52, 72
  glEnablei(), 516
  glEnableVertexAttribArray(), 720
glEnd(), 47, 48, 546
  restrictions, 51
glEndConditionalRender(), 515
glEndList(), 301, 305, 307
glEndQuery(), 513
glEndTransformFeedback(), 724
glEvalCoord*, 577
  legal between glBegin() and glEnd(), 52
  used instead of glVertex*, 572, 575
glEvalCoord2*(), 579
glEvalMesh1(), 578
glEvalMesh2(), 581
glEvalPoint*()
  legal between glBegin() and glEnd(), 52
glext.h, 16
glFeedbackBuffer(), 628
  glRenderMode(), use with, 608
glFinish(), 40
glFlush(), 39, 816
glFlushMappedBufferRange(), 100
glFog*(), 284
glFogCoord*(), 288
glFogCoordPointer(), 74
glFramebufferRenderbuffer(), 533
glFramebufferTexture1D(), 535
glFramebufferTexture2D(), 535
glFramebufferTexture3D(), 535
glFramebufferTextureLayer(), 538
glFrontFace(), 62
glFrustum(), 133, 153, 154
glGenBuffers(), 92
glGenerateMipmap(), 428
glGenFramebuffers(), 527
glGenLists(), 301, 306
  fonts, use for, 330
glGenQueries(), 512
glGenRenderbuffers(), 529
glGenTextures(), 399, 438
glGenVertexArrays(), 105
glGetAttachedShaders(), 740
glGetAttribLocation(), 718
glGetBooleanv(), 10, 54, 743
  double-buffering support, querying, 494
  stereo support, querying, 494
glGetBufferParameteriv(), 740
glGetBufferPointerv(), 740
glGetBufferSubData(), 740
glGetClipPlane(), 740
  pixel-storage modes effect, 347
  get color table parameter*, 741
  get compressed texture image(), 741
  get convolution filter(), 741
  pixel-storage modes effect, 347
  get convolution parameter*, 741
  get double(), 10, 54, 743
  get error(), 10, 638, 741
  get floatv(), 10, 54, 743
  line width attributes, obtaining, 57
  get fragment data location(), 730
  get histogram(), 385, 741
  pixel-storage modes effect, 347
  get histogram parameter*, 741
  get integerv(), 10, 54, 743
  alpha test information, obtaining, 503
  attribute stack depth, obtaining, 111
  clipping planes, obtaining number of
  additional, 170
  color resolution, obtaining, 190
  display list nesting limit, obtaining, 310
  matrix stack depth, obtaining, 167
  maximum texture size, obtaining, 406
  name stack depth, obtaining, 406
  pixel map information, obtaining, 355
  rendering mode, obtaining current, 608
  stencil-related values, obtaining, 505
  vertex array range values, obtaining, 81
  get light*, 10, 741
  get map*, 741
  get material*, 741
  get minmax(), 387, 741
  pixel-storage modes effect, 347
  get minmax parameter*, 741
  get pixel map(), 741
  get pointer(), 10, 54, 743
  get pointerv(), 10, 54, 743
  get polygon stipple(), 10, 741
  get program info log(), 678
Index

867

Index

867

pixel-storage modes effect, 347

pixel-transfer modes effect, 352

glGetTexImage(), 742

pixel-storage modes effect, 347

pixel-transfer modes effect, 352

glGetTexParameter*, 742

texture residency, obtaining, 442

glGetUniform*, 742

fonts, use for, 330

sample program, 331

viewing transformations, use before, 131

glLoadIdentity(), 133, 135, 145

legal between glBegin() and glEnd(), 51

use with matrix stacks, 166

version, 33
glMultiDrawElements(), 80
   version, 33
glMultiTexCoord*(), 470
glMultMatrix*(), 134, 135
   glMultiTexCoordMatrix*(), 134, 136
   glNewList(), 301, 305, 307
   glNormal*()
      legal between glBegin() and glEnd(), 51
   glNormal3*(), 69
   glOrtho(), 157
      picking matrix use, 615
   glPassThrough(), 628, 630
   glPixelMap*(), 354
   glPixelStore*(), 347, 417
   cannot be stored in display lists, 309
      polygon stippling, 63
   texture image data, effect on, 403, 405,
      409, 411, 413, 416, 421, 422
   glPixelTransfer*(), 352, 662
      texture image data, effect on, 403, 405,
      409, 411, 413, 416, 421, 422
   glPixelZoom(), 356, 648
   glPointParameter*(), 292
   glPointSize(), 55, 721
   glPolygonMode(), 61
      antialiasing, effect on, 280
      polygon offset, use with, 294
   glPolygonOffset(), 294
   glPolygonStipple(), 63
      pixel-storage modes effect, 347
   glPopAttrib(), 10, 111, 318, 470, 743
   glPopClientAttrib(), 10, 112, 470, 743
   glPopMatrix(), 166, 176, 223, 318
      restore orientation of coordinate systems, 179
      selection, use with, 607
   glPopName(), 608, 609
   glPrimitiveRestartIndex(), 84
   glPrioritizeTextures(), 443
   glPushAttrib(), 10, 111, 318, 470, 743
   glPushClientAttrib(), 10, 112, 470, 743
   glPushMatrix(), 166, 176, 223, 318
      save orientation of coordinate systems, 179
      selection, use with, 607
   glPushName(), 607, 608, 609
   glRasterPos*(), 324, 325
      images, for positioning, 333
      multitexturing, with, 471
      selection hit, 610
   glReadBuffer(), 342, 499
   glReadPixels(), 333, 335
      glReadBuffer() effect, 499
      pixel-storage modes effect, 347
      pixel-transfer modes effect, 352
   glRect*(), 45
   glRenderbufferStorage(), 530
   glRenderbufferStorageMultisample(), 530
   glRenderMode(), 607, 608, 610, 628
   glResetHistogram(), 386
   glResetMinmax(), 388
   glRotate*(), 142, 172, 175
   glSampleCoverage(), 279
   glScale*(), 131, 142, 175
   glScissor(), 502
   glSecondaryColor*(), 479
   glSecondaryColorPointer(), 74
   glSelectBuffer(), 607, 608
      display lists, cannot be stored in, 309
   glSeparableFilter2D(), 379
   glShadeModel(), 200
   glShaderSource(), 675
   glStencilFunc(), 504
   glStencilFuncSeparate(), 504
   glStencilMask(), 499
   glStencilMaskSeparate(), 499
   glStencilOp(), 505
   glStencilOpSeparate(), 505
glTexBuffer(), 711

glTexCoord*(), 400, 449
legal between glBegin() and glEnd(), 52
texture unit 0, for, 471

glTexCoordPointer(), 74

glTexEnv*(), 399, 444, 473
level of detail bias, 430
multitexturing, 468

glTexGen*(), 457
cube maps, 466
environment mapping, 464
multitexturing, 468, 471

glTexImage1D(), 412
pixel-storage modes effect, 347
pixel-transfer modes effect, 352

glTexImage2D(), 399, 400
cube map textures, 465
pixel-storage modes effect, 347
pixel-transfer modes effect, 352
specifying mipmaps, 425

glTexImage3D(), 415
pixel-storage modes effect, 347
pixel-transfer modes effect, 352

glTexParameter*(), 399, 455
automatic mipmap regeneration, 429
mipmap level of detail, controlling, 433
mipmap levels, controlling base and maximum, 432
multitexturing, 468
specifying filtering methods, 435

Index

GLU, 3, 15, 542
drawing spheres, cylinders, and disks, 559–567
error string description, 639
obsolete routines
gluBeginPolygon(), 558
gluEndPolygon(), 558
gluNextContour(), 558
quadrics, 559–567
tessellation, 44, 542–559
version numbers, obtaining, 641
gluBeginCurve(), 587, 597
gluBeginSurface(), 587, 595
gluBeginTrim(), 601
gluBuild1DMipmapLevels(), 430
gluBuild1DMipmaps(), 429
gluBuild2DMipmapLevels(), 430
gluBuild2DMipmaps(), 429
gluBuild3DMipmapLevels(), 430
gluBuild3DMipmaps(), 429
gluCheckExtension(), 642
gluCylinder(), 560, 563
gluDeleteNurbsRenderer(), 591
gluDeleteQuadric(), 560, 561
gluDeleteTess(), 557, 558
gluDisk(), 560, 564
gluEndCurve(), 587, 597
gluEndSurface(), 587, 595
gluEndTrim(), 601
gluErrorString(), 561, 595, 639
polygon tessellation, 546
gluGetNurbsProperty(), 594, 742
gluGetString(), 641, 742
gluGetTessProperty(), 553, 742
gluLoadSamplingMatrices(), 594
gluLookAt(), 129, 131, 149, 172
gluNewNurbsRenderer(), 587, 591
gluNewQuadric(), 560, 561
gluNewTess(), 544, 558
gluUniform*(), 691

gluErrorString()
glUniformBlockBinding(), 696
glUniformMatrix*(), 691
glUnmapBuffer(), 97
gluNurbsCallback(), 587, 595, 598
gluNurbsCallbackData(), 599
gluNurbsCurve(), 587, 597
gluNurbsProperty(), 587, 592
  returning tessellated data, 597
gluNurbsSurface(), 587, 596
gluOrtho2D(), 158
  resized windows, use with, 41
gluPartialDisk(), 560, 564
gluPerspective(), 133, 156, 172
  picking matrix use, 615
gluPickMatrix(), 615
gluProject(), 183
gluPwlCurve(), 601
gluQuadricCallback(), 560, 561
gluQuadricDrawStyle(), 560, 561
gluQuadricNormals(), 560, 562
gluQuadricOrientation(), 560, 562
gluQuadricTexture(), 560, 562
gluScaleImage(), 405
glUseProgram(), 681
GLUT, 15, 731–737
  basic functions, 17–22
  event management, 21
    glutCreateWindow(), 19, 733
    glutDisplayFunc(), 19, 733
    glutIdleFunc(), 21, 736
    glutInit(), 18, 732
    glutInitContextFlags(), 19
    glutInitContextVersion(), 18
    glutInitDisplayMode(), 18, 732
    glutInitWindowPosition(), 18, 733
    glutInitWindowSize(), 18, 733
    glutKeyboardFunc(), 21, 733
    glutMainLoop(), 19, 737
    glutMotionFunc(), 21, 734
    glutMouseFunc(), 21, 734
    glutPostRedisplay(), 19, 302, 734
    glutReshapeFunc(), 21, 734
      simple example, 40
    glutSetColor(), 18, 199, 247, 735
      smooth shading, use for, 202
    glutSolidCone(), 736
    glutSolidCube(), 21, 735
    glutSolidDodecahedron(), 736
    glutSolidIcosahedron(), 736
    glutSolidOctahedron(), 736
    glutSolidSphere(), 21, 735
    glutSolidTeapot(), 736
    glutSolidTetrahedron(), 736
    glutSolidTorus(), 735
    glutSwapBuffers(), 25
    glutWireCone(), 736
    glutWireCube(), 21, 735
    glutWireDodecahedron(), 736
    glutWireIcosahedron(), 736
    glutWireOctahedron(), 736
    glutWireSphere(), 21, 172, 735
    glutWireTeapot(), 736
    glutWireTetrahedron(), 736
    glutWireTorus(), 735
    multisampling, 276
    window management, 40
    gluTessBeginContour(), 555
    gluTessBeginPolygon(), 554
    gluTessCallback(), 545, 555, 558
    gluTessEndContour(), 555
    gluTessEndPolygon(), 554
    gluTessNormal(), 553, 554, 557
    gluTessProperty(), 550, 555
    gluTessVertex(), 555, 558
    gluUnProject(), 180, 183
    gluUnProject4(), 182
    glValidateProgram(), 681
Index

871

glVertex*(), 46
  legal between glBegin() and glEnd(), 51
  using glEvalCoord*() instead, 572
glVertexAttrib*(), 719
glVertexAttrib4N*(), 719
glVertexAttrib*(), 719
glVertexAttribPointer(), 720
glVertexPointer(), 52, 74
glViewport(), 134, 159
  using with resized windows, 41
glWindowPos*(), 326
  multitexturing, with, 471
  selection hit, 610
GLX, 15, 813
  ftp site for GLX specification, 813
glxChooseFBConfig(), 813, 817
glxChooseVisual(), 814, 819
glxCopyContext(), 815, 817
glxCreateContext(), 815, 819
glxCreateGLXPixmap(), 814, 819
glxCreateNewContext(), 814, 817
glxCreatePbuffer(), 814, 817
glxCreatePixmap(), 814, 817
glxCreateWindow(), 814, 817
glxDestroyContext(), 815, 818
glxDestroyGLXPixmap(), 816, 819
glxDestroyPbuffer(), 816, 818
glxDestroyPixmap(), 816, 818
glxDestroyWindow(), 816, 818
glxGetClientString(), 813, 816
glxGetConfig(), 493, 814, 819
glxGetCurrentContext(), 815, 818
glxGetCurrentDisplay(), 815, 818
glxGetCurrentDrawable(), 815, 818
glxGetCurrentReadDrawable(), 815, 818
glxGetFBConfigAttrib(), 813, 817
glxGetFBConfigs(), 817
glxGetProcAddress(), 814, 817
glxGetSelectedEvent(), 815, 818
glxGetVersionString(), 814, 817
  Goursaud shading, see smooth shading

H
  Haeberli, Paul, 482, 519
  haze, see fog
  header files, 15
    Version 3.1, 17
  hidden-line removal, 659
    polygon offset used for, 294
  hidden-surface removal, 205–207, 510
  hierarchical models, 164, 310
    picking, 619–621
  highlights, see specular
  hints, 268
    fog, 281
      perspective correction, 268, 400
  histogram, 383–386
    resetting, 384, 386
    retrieving, 384
    sample program, 385
  hits (selection), see selection (hit records)
  holes in polygons, 43, 656
  homogeneous coordinates, 42, 804
  Hoschek, Josef, 571
  Hughes, John F., xl, 803
icosahedron, drawing, 115
identity matrix, 131, 135, 145
illumination, see lighting
images, 322, 333–343
also see pixel data
blending, 653
compositing, 252
distorted, 647
imaging pipeline, 343–359
interpolating between, 653
magnifying or reducing, 356
nonrectangular, 259
projecting, 661
sample code which draws an image, 341
sample program which draws, copies, and
zooms an image, 357
scaling and rotating, 661
sources of, 333
superimposing, 654
transposing, 666
warping, 661
imaging pipeline, see images (imaging pipeline)
imaging subset, 367–388
texture images, effect on, 404, 412
immediate mode, 33, 298
display lists, mixing with, 310
infinite light source, 217
input events
handling, using GLUT, 21
intensity
texture image data type, 445
Interactive Inspection of Solids: Cross-sections
and Interferences, 656
interference regions, 656
interleaved arrays, 88
interpolating
color values and texture coordinates, 268,
449
texture combiner function, 477
jaggies, 267
jittering, 525
Kilgard, Mark, xli, 15, 731, 813
Korobkin, Carl, 482
Lasser, Dieter, 571
layers, drawing, 649
Life, Game of, 664
light sources, 214–227
ambient light, 208, 216
contribution to lighting equation, 242
diffuse light, 208, 216
directional, 217
display lists cache values, 303
infinite light source, 217
local light source, 217
maximum number of sources, 213
moving along with the viewpoint, 225
moving light sources, 221–226
multiple light sources, 220
performance tips, 213
positional, 217
rendering pipeline stage, 12
RGBA values, 209
sample program that moves the light
source, 224
specifying a light source, 213
specular light, 208
spotlights, 219–220
stationary, 222
lighting
also see light sources, material properties
ambient light, 207
approximation of the real world, 207
attenuation, 217–218
calculations in color-index mode, 247
color-index mode, 246–247
default values, using, 214
display lists cache values, 303
enabling, 213, 214
enabling and disabling, 231
equation that calculates lighting, 241
global ambient light, 228, 242
lighting model, 227–230
lighting model, specifying a, 213
rendering pipeline stage, 12
sample program introducing lighting, 210
specular color separated, 230, 245, 479
steps to perform, 210
two-sided materials, 229
viewer, local or infinite, 229
line segment, 43
linear attenuation, 218
lines, 43
antialiasing, 269–275, 661
connected closed loop, specifying, 48, 50
connected strip, specifying, 48, 50
feedback mode, 629
querying line width, 57
sample program with wide, stippled lines, 59
specifying, 48, 50
stippling, 57
tessellated polygons decomposed into, 546
width, 56
local light source, 217
logical operations
rendering pipeline stage, 14
transposing images, using for, 666
lookup table, see color map
luminance, 336, 362
pixel data formats for, 338, 346
texture image data type, 445

M
magnifying images, 356
masking, 499
antialiasing characters, 652
layers, drawing, 649
rendering pipeline stage, 14
material properties, 214, 231–240
ambient, 209, 233
changing a single parameter with glColorMaterial(), 237
changing material properties, 235
diffuse, 209, 233
display lists cache values, 303
emission, 208, 234, 241
enabling color material properties mode, 237
rendering pipeline stage, 12
RGBA values, 210
sample program which changes material properties, 235
sample program which uses glColorMaterial(), 238
shininess, 234
specular, 209, 234
two-sided lighting, 229
matrix
choosing which matrix is current, 135
column-major ordering, 136
current, 131
display lists cache matrix operations, 303
identity, 131, 135, 145
loading, 135
loading transposed, 136
modelview, 128, 135
multiplying matrices, 135
multiplying transposed matrices, 136
NURBS, specifying for sampling, 593
orthographic parallel projection, 808
perspective projection, 807
projection, 133, 135
matrix, continued
rotation, 806
row-major ordering, 136
scaling, 806
texture, 481
transformation pipeline, 126
transformations of homogeneous coordinates, 804
translation, 806
matrix stack, 164–168
choosing which matrix stack is current, 166
modelview, 167
popping, 166
projection, 168
pushing, 166
querying stack depth, 167
texture, 481
Megahed, Abe, 656
Microsoft
callback functions on Windows, 547
Microsoft Win32, see Win32
Microsoft Windows 95/98/NT, 15, 824
Microsoft Windows to OpenGL interface, see WGL
minmax, 387–388
resetting, 387, 388
retrieving results, 387
sample program, 388
mipmapping, 424–429
automated generation, 429
base and maximum levels, 432
level of detail control, 431
minification filters, 436
texture objects for mipmap, 441
mirroring objects, see scaling
modeling transformations, 131, 137, 140–145
camera analogy, 126
connection to viewing transformations, 131
example, 143
rotation, 142
rotation matrix, 806
sample program, 145
scaling, 142
scaling matrix, 806
translation, 141
translation matrix, 806
models
rendering wireframe and solid, 21, 735
modelview matrix, 128, 135
arbitrary clipping planes, effect on, 169
stack, 167
mosaicing, 431
motion blur, 520
stippling, with, 646
motion, see animation
movie clips, 665
multiple layers
displaying with overlap, 649
multisampling, 275–279
fading point primitives, 292
sample program, 276
multitexturing, 467–472

name stack, 607–611
creating, 608
initializing, 608
loading, 608
multiple names, 619–621
popping, 608
pushing, 608
querying maximum depth, 609
networked operation, 39–40
attribute groups, saving and restoring, 110
display lists, 309
versions, 640

N
name stack, 607–611
Non-Uniform Rational B-Splines, see NURBS
nonplanar polygons, 44
normal vectors, 68–70, 212
calculating length, 70
cross product, calculating normalized, 117
enabling automatic unit length division, 70, 212
matrix transformations, 128
normalized, 70
NURBS, generating for, 596
quadrics, generated for, 562
rendering pipeline stage, 12
specifying, 69
tessellation, specifying for, 548
transformations, 805
uniform rescaling, 70
vertex arrays, specifying values with, 74
normal, see normal vectors
normalized device coordinates, 128
NURB Curves and Surfaces, 571
NURBS, 586–604
creating a NURBS curve or surface, 595–597
creating a NURBS object, 591
culling, 592
deleting a NURBS object, 591
display list use, 302
error handling, 594
method of display (lines or filled polygons), 592
normal vectors, generating, 596
properties, controlling NURBS, 591
querying property value, 594
references, 571
sample program which draws a lit NURBS surface, 588
sample program with a trimmed surface, 603
sampling precision, 592
source for matrices, 593
steps to use, 587
texture coordinate generation, 596
trimming, 601–604
NURBS Book, The, 571
NURBS for Curve and Surface Design, 571
NURBS tessellator
sample code, 599, 600

O
object coordinates, 128
texture coordinate generation, 458
objects, see models
occlusion queries
conditional rendering, 515
occlusion query, 511
opacity, 252
OpenGL contexts, 27
OpenGL Extension to the X Window System, see GLX
OpenGL Programming for the X Window System, xli, 813
OpenGL Reference Manual, xl, 813
OpenGL Utility Library, see GLU
OpenGL Utility Toolkit, see GLUT
orthographic parallel projection, 133, 156–157
matrix, 808
outer product, 378
outlined polygons, 61, 68
polygon offset solution, 293
overlapping objects, 656

P
packed pixel data, 338–339
painting, 252, 258, 666
partial disks, 559
pass-through markers, 630
performance tips
  clearing the window, 36
display lists, 302
flushing the pipeline, 38
fog, 281
hints, 268
light source attenuation, effect of, 218
light sources, effect of additional, 213
NURBS and display lists, 302
pixel data alignment, 350
pixel data, drawing, 366
polygon restrictions, 44
polygon subdivision, 114
removing hidden surfaces, 207
tessellation and display lists, 302
tessellation, use of, 557
texture images, internal format of, 403
texture objects, 437
two-sided lighting, 230
perspective projection, 153–156
correction hint, 268, 400
depth coordinates, effect on, 161
matrix, 807
perspective division, 128
picking, 614–624
back buffer for, using the, 646
depth coordinates, 621
hierarchical models, 619–621
projection matrix, special, 615
sample program, 616
sample program with depth coordinates, 621
strategies, 625
sweep selection, 626
Piegl, Les, 571
pipeline
  imaging, 343–359
  rendering, 10–14
  vertex transformation, 126
pixel
  coverage, 267
pixel data, 322, 333–343
  also see images
  BGR and BGRA formats, 336
  byte alignment, 350
  byte swapping, 348
  copying within the framebuffer, 13, 333, 342, 344
depth buffer pixel data, 342, 353
drawing or reading a subrectangle of, 349
drawing process in detail, 359–361
endianness, 349
feedback mode, 629
formats for reading or drawing, 335
formats for storing in memory, 338, 346
mapping, 13, 354–355
packed, 338–339
packing into processor memory, 13, 346–348
performance tips, 366
pipeline operations, 13, 343–359
pixel zoom, 356
querying pixel mapping information, 355
reading from the framebuffer, 333, 335
reading process in detail, 361–362
sample code which draws an image, 341
sample program that uses a pixel buffer
  object for storage, 364
sample program which draws, copies, and
  zooms pixel data, 357
stencil buffer pixel data, 338, 354
storage modes, 347, 417–419
transfer modes, 13, 352, 445
unpacking from processor memory, 13, 346–348
writing to the framebuffer, 333, 341
point light source, see positional light source
point parameters, 291
sample program, 292
points, 42
  antialiasing, 269–275, 653
drawing, 48
feedback mode, 629
point parameters, 291
round, 269–275, 653
size, 55
specifying, 48, 50
polygon offset, 293–296
  depth slope of a polygon, 295
  enabling, 294
  hidden-line removal, 659
  sample program, 296
  shadowing use, 484
polygonal approximations to surfaces, 113
polygons, 43
  boundary edges, 67–68
  concave, drawing filled, 542, 655
  convex, 43
  culling the faces, 61
drawing, 48
drawing as points, lines, or filled, 61
feedback mode, 629
front and back faces, 61
holes in, 43
non-convex, 44, 67
nonplanar, 44
polygon mode, 12, 61
reversing the faces, 61
self-intersecting, 549
simple, 43
specifying, 48, 51
stippling, 63
tessellation, specifying for, 554
Voronoi, 663
positional light source, 217
primitives
  geometric, 42–53
  raster, 322
priority of texture objects, 443

Procedural Elements for Computer Graphics, 560

programs
  aaindex.c, 272
  aargb.c, 270
  alpha3D.c, 264
  alpha.c, 261
  bezcurve.c, 573
  bezmesh.c, 582
  bezsurf.c, 580
  checker.c, 398
  clip.c, 170
  colormat.c, 238
  colormatrix.c, 382
  colortable.c, 371
  combiner.c, 477
  convolution.c, 376
cube.c, 130
cubemap.c, 466
dof.c, 522
drawf.c, 324
feedback.c, 631
fog.c, 281
fogcoord.c, 289
fogindex.c, 286
font.c, 331
histogram.c, 385
image.c, 357
light.c, 210
lines.c, 59
list.c, 305
material.c, 235
minmax.c, 388
mipmap.c, 426
model.c, 145
movelight.c, 224
multisamp.c, 276
multitex.c, 469
pboimage.c, 364
programs, continued
  pickdepth.c, 621
  picksquare.c, 616
  planet.c, 173
  pointp.c, 292
  polyoff.c, 296
  quadric.c, 565
  robot.c, 177
  select.c, 611
  shadowmap.c, 484–486
  smooth.c, 200
  sprite.c, 481
  stencil.c, 507
  stroke.c, 314
  surface.c, 588
  surfpoints.c, 599, 600
  tess.c, 548, 556
  texbind.c, 439
  texgen.c, 459
  texsub.c, 410
  texture3d.c, 415
  texturesurf.c, 584
  torus.c, using a display list, 299
  trim.c, 603
  unproject.c, 180
  projecting images, 661
  projection matrix, 133, 135
  matrix stack, 168
  orthographic parallel projection matrix, 808
  perspective projection matrix, 807
  shadows created with, 658
  projection transformation
    centered along view vector, 156
    off-axis perspective, 154
    parallel projection, 157
    perspective, 154, 156
    three-dimensional orthographic, 157
  projection transformations, 132, 152–158
  camera lens analogy, 126
  orthographic parallel, 133, 156–157
  perspective, 153–156
  picking, 615
  texturing effects, 482
  two-dimensional orthographic, 158
  proxies
    color table, see color table proxies, 374
  proxy textures, 406
    cube maps, 465

Q
  q texture coordinates, 482
  quadratic attenuation, 218
  quadrics, 559–567
    creating an object, 560
    destroying an object, 560
    drawing as points, lines, and filled polygons, 561
    error handling, 561
    normal vectors, generating, 562
    orientation, 562
    quadratic equation, 559
    sample program, 565
    steps to use, 560
    texture coordinates, generating, 562
  quadrilateral
    specifying, 48
  strip, specifying, 48, 51

R
  raster position, 325
    after drawing a bitmap, 327
    current, 325
    current raster color, 329
    current, obtaining the, 326
    selection hit, 610
    transformation of, 326
  rasterization, 190, 490
  rendering pipeline stage, 14
readImage(), 372
reading pixel data, see pixel data
Real Projective Plane, The, 803
rectangles
   specifying, 45
reducing images, 356
reflecting objects, see scaling
reflection, see material properties
reflective objects, see environment mapping
refresh, screen, 22
removing hidden surfaces, see hidden-surface removal
rescaling normals, 70, 212
resident textures, 408, 442
   management strategies, 443
   querying residence status, 442
RGBA mode, 191
   changing between color-index mode and, 196
   choosing between color-index mode and, 195
coverage calculations for antialiasing, 267
data type conversion, 197
light source colors, 209
lighting calculations in, 241
material property values, 210
vertex arrays, specifying values with, 74
Robins, Nate, xlii, 145, 151, 158, 179, 227, 236, 437, 448, 450, 457, 482
robot arm example, 175–179
Rogers, David, 560
Rossignac, Jarek, 656
rotating images, 661
rotation, 142
   matrix, 806

S
scaling, 142
   matrix, 806
scaling images, 661
Schneider, Bengt-Olaf, 656
Scientific American, 664
scissor test, 502
   and clearing, 496
   rendering pipeline stage, 14
secondary color, 478–479
   specular, 230, 245
Segal, Mark, 482
selection, 606–627
   back buffer for, using the, 646
   hit records, 610
   programming tips, 625
   querying current rendering mode, 608
   sample program, 611
   steps to perform, 607
   sweep selection, 626
shading
   flat, 200
   sample program with smooth shading, 200
   smooth, 200
   specifying shading model, 200
shadows, 241, 525, 658
shininess, 234
   also see environment mapping
silhouette edges, 114
smoke, see fog
smooth shading, 200
solar system example, 172–175
source factor, see blending
specifying the background color, 36
specifying the depth buffer clear value, 35
specular
   contribution to lighting equation, 244
   light, 208
   material properties, 209, 234
   secondary specular color, 230, 245, 479
sphere map, 463
spheres, 559, 735
split-screen
   multiple viewports, 159

Index 879
spotlights, see light sources
state attributes
  persevering, 111
  preserving vertex arrays, 112
  reverting, 111
  reverting vertex arrays, 112
state machine, 9–10
state variables, 53
  attribute groups, 110–112
  display list execution, effect of, 318
  enable and disable states, 54
  list of, 743–800
  querying, 54
stencil buffer, 492, 494
  clearing, 36, 495
  concave polygons, for drawing, 655
decals, for, 654
  Dirichlet domains, for, 663
Game of Life, for the, 664
hidden-line removal, 660
masking, 499
pixel data, 338, 354
stencil test, 504–510
  examples of using, 506
  interference regions found using clipping planes, 657
  querying stencil parameters, 505
  rendering pipeline stage, 14
  sample program, 507
stereo, 494, 498
  querying its presence, 494
stippling
  display lists cache stipple patterns, 303
  enabling line stippling, 57
  enabling polygon stippling, 63
  fade effect, use for, 645
  line pattern reset, 58, 629, 633
  lines, 57
  polygons, 63
sample program with line stipple, 59
stencil test, use of, 510
translucency, use to simulate, 644
stitching, 294
stretching objects, see scaling
stride
  vertex arrays, 76, 89
subdivision, 113–121
  generalized, 120
  icosahedron example, 118
  recursive, 119
subimages, 408–411, 413, 416
superimposing images, 654
surface normals, see normal vectors
surfaces, see evaluators or NURBS
swapping buffers, see double-buffering
syntax, see command syntax

T
Terminator 2, 463
tessellation, 44, 542–559
  backward compatibility with obsolete routines, 558
  begin and end callback routines, 546
  callback routines, 544–549
  combine callback routine, 546, 549
  contours, specifying, 554
  converting code to use the GLU 1.2 tessellator, 559
  creating an object, 544
  decomposition into geometric primitives, 546
deleting objects, 557
display list use, 302
edge flag generation, 546
error handling, 546
interior and exterior, determining, 550–553
intersecting contours combined, 546, 549
performance tips, 557
polygons, specifying, 554
properties, 550–554
reuse of objects, 544, 557
reversing winding direction, 554
sample code, 548, 556
user-specified data, 549
vertices, specifying, 547, 555
winding rules, 550–553
texels, 14, 391
text, see characters
texture coordinates, 400, 448–464
assigning manually, 448
clamping, 452–455
computing manually, 450
cube maps, 466
enabling automatic generation of, 462
environment mapping, automatic
generation for, 464
evaluators, generated by, 584
generating automatically, 457–464
multitexturing, special situations with, 471
NURBS, generating for, 596
q coordinate, 482
quadrics, generated for, 562
reference planes, specifying, 458
rendering pipeline stage, 12
repeating, 452–455
sample program with texture coordinate
generation, 459
tessellation, specifying for, 548
vertex arrays, specifying values with, 74
wrapping modes, 452–455
texture functions, 444–448
add, 448
blend, 448
blending color, 448
decal, 399, 447
fragment operations, 446
level of detail bias, 444
modulate, 448
pixel-transfer modes effect, 445
replace, 447
texture internal format, interaction with, 446
texture images
alpha data, 445
borders, 423, 452
components, 401
data types, 403
distorting, 451
framebuffer as a source of, 405, 411, 413, 417
imaging pipeline operations, 345
intensity data, 445
internal format, 401
luminance data, 445
mipmaps, 424–429
multitexturing, 468
one-dimensional, 412–414
performance affected by internal format, 403
power of 2 size restriction, 404
proxy textures, 406
querying maximum size, 406
residence status, 442
resident textures, 408, 442
resident textures, management strategies of, 443
sample program with mipmaps, 426
sample program with subimages, 410
specifying, 400–423
subimages, 408–411, 413, 416
three-dimensional, 414–419
working set of textures, 408, 437, 442
texture mapping
sample program using 3D textures, 415
texture mapping, see texturing
texture matrix, 481
texture objects, 399, 437–442
  binding, 438
  creating, 438
  data which can be stored in, 439
  deleting, 441
  fragmentation of texture memory, 444
  least-recently used (LRU) strategy, 444
  mipmap maps, 441
  naming, 438
  performance tips, 437
  priority, 443
  rendering pipeline, 13
  sample program, 398
  sample program with multiple texture objects, 439
  sharing among rendering contexts, 814, 826
  steps to perform, 437
  using, 438
  texturing
    also see texture coordinates, texture functions, texture images, texture matrix, and texture objects
  antialiasing characters, 661
  antialiasing lines, 661
  blending, 259
  border colors, treatment of, 452
  color-index mode limitations, 396, 404
  combiner functions, 472–478
  compressed textures, 420
  creating contours, 458
  cube maps, 465
  decals with alpha testing, 503
  differences among releases, 393
  enabling, 396, 399
  filtering, 434–436
  image transformations, 661
  mipmapping, 424–429, 436
  mosaic texture, 431
  multitexturing, 467–472
  perspective correction hint, 400
  popping visual artifact, 431
  rendering pipeline stage, 13
  sample code using point sprites, 481
  sample code with a depth texture, 484–486
  sample code with combiner functions, 477
  sample code with multitexturing, 469
  sample program, 398
  sample program with cube maps, 466
  sample program with evaluated, Bézier surface, 584
  sample program with mipmapping, 426
  sample program with texture coordinate generation, 459
  sample uses for, 661
  simulating shadows or spotlights, 482
  specular color separated, 230, 245, 479
  sphere map, 463
  steps to perform, 396
  using, 438


3D models, rendering, 21, 735
Tiller, Wayne, 571

tips, programming
  selection and picking, 625
  transformations, 162

transformations
  also see modeling transformations, projection transformations, viewing transformations, and viewport transformations
  combining multiple, 172–179
  display lists cache transformations, 303
  general-purpose commands, 134
  matrices, 805–808
  modeling, 137, 140–145
  ordering correctly, 137–140
  overview, 125
  projection, 132, 152–158
  reversing the geometric processing pipeline, 180
sample program, 130
sample program combining modeling transformations, 173, 177
sample program for modeling transformations, 145
sample program showing reversal of transformation pipeline, 180
troubleshooting, 162–164
units, 156
viewing, 137, 146–151
viewport, 134, 158–160
translation, 141
matrix, 806
translucent objects, 252, 644
  stencil test, creating with the, 510
transparent objects, 252
  creating with the alpha test, 503
transposing images, 666
triangle
  fan, specifying, 48
  specifying, 48, 50
  strip, specifying, 48, 50
  tessellated polygons decomposed into, 546
trimming
  curves and curved surfaces, 601–604
  sample program, 603
tutorials
  on-line, xlii
two-sided lighting, 229

U
up-vector, 131
Utility Library, OpenGL, see GLU
Utility Toolkit, OpenGL, see GLUT

V
van Dam, Andries, xl, 187, 191, 803
van Widenfelt, Rolf, 482
vendor-specific extensions, 641
versions, 639–641
  GLU, 641
vertex, 42
  also see vertex arrays
evaluators, generating with, 572
feedback mode, 629
per-vertex operations pipeline stage, 12
specifying, 46
tessellation, specifying for, 547, 555
transformation pipeline, 126
vertex arrays, 70–91
derference a list of array elements, 78, 80, 81
derference a sequence of array elements, 82, 83
derference a single element, 77
differences among releases, 33
disabling, 73
display list use, 309
enabling, 72
interleaved arrays, 88
interleaved arrays, specifying, 89
multitexturing texture coordinates, 471
querying, 743
querying range values, 81
restarting primitives, 83–??, 84, ??–86
reuse of vertices, 79
specifying data, 73
steps to use, 71
stride between data, 76, 89
vertex-array objects, 104–??, 109, ??–110
vertex-arrays objects, 105
video
  fake, 665
flipping an image with glPixelZoom(), 356
textured images, 408
viewing
camera analogy, 126–127
viewing transformations, 130, 137, 146–151
connection to modeling transformations, 131
default position, 131
different methods, 151
pilot view, 152
polar view, 152
tripod analogy, 126
up-vector, 131
viewing volume, 153
clipping, 158, 168
viewpoint
lighting, for, 229
viewport transformations, 129, 134, 158–160
photograph analogy, 126
rendering pipeline stage, 12
visual simulation
fog, use of, 281
Voronoi polygons, 663

W
w coordinates, 42, 129, 134
lighting, use with, 217
perspective division, 161
warping images, 661
Watt, Alan, 390
web sites, xl
errata list, xlii
Microsoft Developer Network, 825
WGL, 15, 825
wglCopyContext(), 826, 828
wglCreateContext(), 825, 828
wglCreateLayerContext(), 826, 828
wglDeleteContext(), 828
wglDescribeLayerPlane(), 825, 827
wglDestroyContext(), 826
wglGetCurrentContext(), 826, 828
wglGetCurrentDC(), 826, 828
wglGetLayerPaletteEntries(), 827, 829
wglGetProcAddress(), 828
wglMakeCurrent(), 826, 828
wglRealizeLayerPalette(), 827, 829
wglSetLayerPaletteEntries(), 829
wglShareLists(), 826, 828
wglSwapLayerBuffers(), 826, 828
wglUseFontBitmaps(), 827, 829
wglUseFontOutlines(), 827, 829
wglGetProcAddress(), 644
Williams, Lance, 424
Win32
ChoosePixelFormat(), 825, 827
CreateDIBitmap(), 826, 828
CreateDIBSection(), 826, 828
DeleteObject(), 826, 828
DescribePixelFormat(), 825, 827
GetVersion(), 825, 827
GetVersionEx(), 825, 827
SetPixelFormat(), 825, 827
SwapBuffers(), 826, 828
winding, 62
winding rules, 550–553
computational solid geometry, used for, 551
reversing winding direction, 554
window coordinates, 129, 158
feedback mode, 629
polygon offset, 294
raster position, 326
window management
GLenum() called, when window resized, 159
using GLUT, 40
working set of textures, 408, 437, 442
fragmentation of texture memory, 444
writemask, see masking (buffers)
writing pixel data, see pixel data (drawing)
www.opengl.org, xl
X
X Window System, 15, 813
  client-server rendering, 5
  minimum framebuffer configuration, 492
  X Visual, 196, 813

Z
  z buffer, see depth buffer
  z coordinates, see depth coordinates
  zooming images, 356
    filtered, 666