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

Specifying Vertices

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

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

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

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

Example 2-2 Legal Uses of glVertex*()

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

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

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

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

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

OpenGL Geometric Drawing Primitives

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

Example 2-3 Filled Polygon

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

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

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

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

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

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

Table 2-2 Geometric Primitive Names and Meanings

void **glEnd**(void);

Marks the end of a vertex-data list.

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

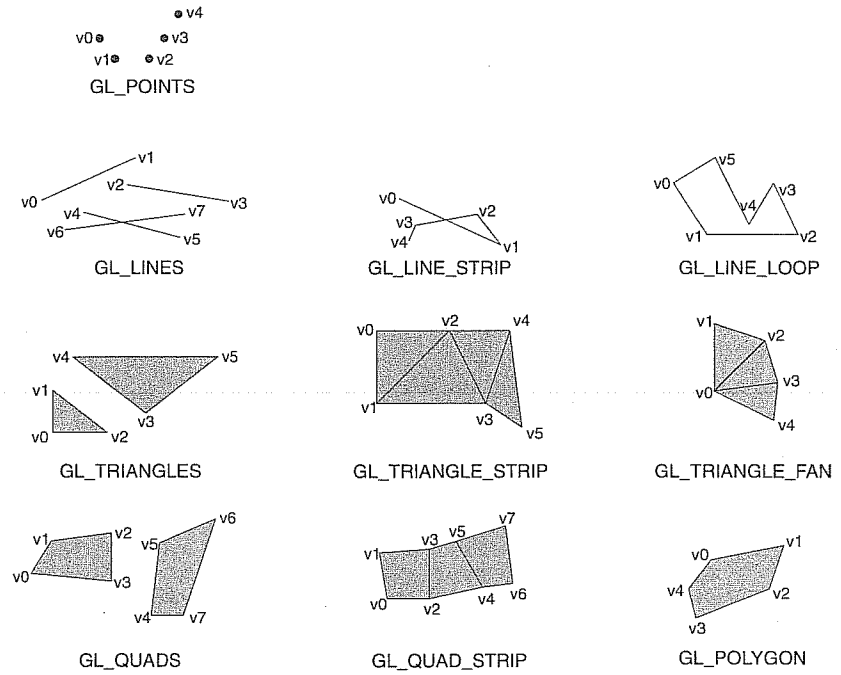

Figure 2-7 Geometric Primitive Types

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

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

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

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

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

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

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

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

Restrictions on Using glBegin() and glEnd()

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

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

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

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

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

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

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

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

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

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

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

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

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

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