Chapter 3: Implementing Modeling in OpenGL

This chapter discusses the way OpenGL implements the general modeling discussion of the last chapter. This includes functions for specifying geometry, specifying points for that geometry in model space, specifying normals for these vertices, and specifying and managing transformations that move these objects from model space into the world coordinate system. It also includes the set of operations that implement polygons, including those that provide the geometry compression that was described in the previous chapter. Finally, it discusses some pre-built geometric models that are provided by the OpenGL and GLUT environments to help you create your scenes more easily. When you have finished this chapter, you should be able to write graphics programs with the OpenGL graphics API that implement the modeling you saw in the previous chapter, though the full set of appearance information that makes graphics scenes interesting will not be covered until later chapters.

The OpenGL model for specifying geometry

In defining your model for your program, you will use a single function to specify the geometry of your model to OpenGL. This function specifies that geometry is to follow, and its parameter defines the way in which that geometry is to be interpreted for display:

```
glBegin(mode);
// vertex list: point data to create a primitive object in
// the drawing mode you have indicated
// appearance information such as normals and texture
// coordinates may also be specified here
glEnd();
```

The vertex list is interpreted as needed for each drawing mode, and both the drawing modes and the interpretation of the vertex list are described in the discussions below. This pattern of `glBegin(mode) - vertex list - glEnd` uses different values of the `mode` to establish the way the vertex list is used in creating the image. Because you may use a number of different kinds of components in an image, you may use this pattern several times for different kinds of drawing. We will see a number of examples of this pattern in this module.

In OpenGL, point (or vertex) information is presented to the computer through a set of functions that go under the general name of `glVertex*(…)`. These functions enter the numeric value of the vertex coordinates into the OpenGL pipeline for the processing to convert them into image information. We say that `glVertex*(…)` is a *set* of functions because there are many functions that differ only in the way they define their vertex coordinate data. You may want or need to specify your coordinate data in any standard numeric type, and these functions allow the system to respond to your needs.

- If you want to specify your vertex data as three separate real numbers, or floats (we'll use the variable names *x*, *y*, and *z*, though they could also be float constants), you can use `glVertex3f(x,y,z)`. Here the character `f` in the name indicates that the arguments are floating-point; we will see below that other kinds of data formats may also be specified for vertices.
- If you want to define your coordinate data in an array, you could declare your data in a form such as `glFloat x[3]` and then use `glVertex3fv(x)` to specify the vertex. Adding the letter `v` to the function name specifies that the data is in vector form (actually a pointer to the memory that contains the data, but an array's name is really such a pointer). Other dimensions besides 3 are also possible, as noted below.

Additional versions of the functions allow you to specify the coordinates of your point in two dimensions (`glVertex2*`); in three dimensions specified as integers (`glVertex3i`), doubles (`glVertex3d`), or shorts (`glVertex3s`); or as four-dimensional points (`glVertex4*`). The

four-dimensional version uses homogeneous coordinates, as described earlier in this chapter. You will see some of these used in the code examples later in this chapter.

One of the most important things to realize about modeling in OpenGL is that you can call your own functions between a `glBegin(mode)` and `glEnd()` pair to determine vertices for your vertex list. Any vertices these functions define by making a `glVertex*(…)` function call will be added to the vertex list for this drawing mode. This allows you to do whatever computation you need to calculate vertex coordinates instead of creating them by hand, saving yourself significant effort and possibly allowing you to create images that you could not generate by hand. For example, you may include various kind of loops to calculate a sequence of vertices, or you may include logic to decide which vertices to generate. An example of this way to generate vertices is given among the first of the code examples toward the end of this module.

Another important point about modeling is that a great deal of other information can go between a `glBegin(mode)` and `glEnd()` pair. We will see the importance of including information about vertex normals in the chapters on lighting and shading, and of including information on texture coordinates in the chapter on texture mapping. So this simple construct can be used to do much more than just specify vertices. Although you may carry out whatever processing you need within the `glBegin(mode)` and `glEnd()` pair, there are a limited number of OpenGL operations that are permitted here. In general, the available OpenGL operations here are `glVertex`, `glColor`, `glNormal`, `glTexCoord`, `glEvalCoord`, `glEvalPoint`, `glMaterial`, `glCallList`, and `glCallLists`, although this is not a complete list. Your OpenGL manual will give you additional information if needed.

Point and points mode

The mode for drawing points with the `glBegin` function is named `GL_POINTS`, and any vertex data between `glBegin` and `glEnd` is interpreted as the coordinates of a point we wish to draw. If we want to draw only one point, we provide only one vertex between `glBegin` and `glEnd`; if we want to draw more points, we provide more vertices between them. If you use points and want to make each point more visible, the function `glPointSize(float size)` allows you to set the size of each point, where `size` is any nonnegative real value and the default size is 1.0.

The code below draws a sequence of points in a straight line. This code takes advantage of fact that we can use ordinary programming processes to define our models, showing we need not hand-calculate points when we can determine them by an algorithmic approach. We specify the vertices of a point through a function `pointAt()` that calculates the coordinates and calls the `glVertex*()` function itself, and then we call that function within the `glBegin/glEnd` pair. The function calculates points on a spiral along the *z*-axis with *x*- and *y*-coordinates determined by functions of the parameter *t* that drives the entire spiral.

```
void pointAt(int i) {
   glVertex3f(fx(t)*cos(g(t)),fy(t)*sin(g(t)),0.2*(float)(5-i));
}

void pointSet( void ) {
   int i;

   glBegin(GL_POINTS);
      for ( i=0; i<10; i++ )
      pointAt(i);
   glEnd();
}
```

Some functions that drive the *x*- and *y*-coordinates may be familiar to you through studies of functions of polar coordinates in previous mathematics classes, and you are encouraged to try out some possibilities on your own.

Line segments

To draw line segments, we use the `GL_LINES` mode for `glBegin/glEnd`. For each segment we wish to draw, we define the vertices for the two endpoints of the segment. Thus between `glBegin` and `glEnd` each pair of vertices in the vertex list defines a separate line segment.

Line strips

Connected lines are called *line strips* in OpenGL, and you can specify them by using the mode `GL_LINE_STRIP` for `glBegin/glEnd`. The vertex list defines the line segments as noted in the general discussion of connected lines above, so if you have *N* vertices, you will have *N*-1 line segments. With either line segments or connected lines, we can set the line width to emphasize (or de-emphasize) a line. Heavier line widths tend to attract more attention and give more emphasis than lighter line widths. The line width is set with the `glLineWidth(float width)` function. The default value of `width` is 1.0 but any nonnegative width can be used.

As an example of a line strip, let's consider a parametric curve. Such curves in 3-space are often interesting objects of study. The code below define a rough spiral in 3-space that is a good (though simple) example of using a single parameter to define points on a parametric curve so it can be drawn for study.

```
glBegin(GL_LINE_STRIP);
   for ( i=0; i<=10; i++ )
      glVertex3f(2.0*cos(3.14159*(float)i/5.0),
         2.0*sin(3.14159*(float)i/5.0),0.5*(float)(i-5));
glEnd();
```

This can be made much more sophisticated by increasing the number of line segments, and the code can be cleaned up a bit as described in the code fragment below. Simple experiments with the *step* and *zstep* variables will let you create other versions of the spiral as experiments.

```
#define PI 3.14159
#define N 100
step = 2.0*PI/(float)N;
zstep = 2.0/(float)N;
glBegin(GL_LINE_STRIP);
   for ( i=0; i<=N; i++)
      glVertex3f(2.0*sin(step*(float)i),2.0*cos(step*(float)i),
      -1.0+zstep*(float)i);
glEnd();
```

If this spiral is presented in a program that includes some simple rotations, you can see the spiral from many points in 3-space. Among the things you will be able to see are the simple sine and cosine curves, as well as one period of the generic shifted sine curve.

Line loops

A line loop is just like a line strip except that an additional line segment is drawn from the last vertex in the list to the first vertex in the list, creating a closed loop. There is little more to be said about line loops; they are specified by using the mode `GL_LINE_LOOP`.

Triangle

To draw unconnected triangles, you use `glBegin/glEnd` with the mode `GL_TRIANGLES`. This is treated exactly as discussed in the previous chapter and produces a collection of triangles, one for each three vertices specified.

Sequence of triangles

OpenGL provides both of the standard geometry-compression techniques to assemble sequences of triangles: triangle strips and triangle fans. Each has its own mode for `glBegin/glEnd`: `GL_TRIANGLE_STRIP` and `GL_TRIANGLE_FAN` respectively. These behave exactly as described in the general section above.

Because there are two different modes for drawing sequences of triangles, we'll consider two examples in this section. The first is a triangle fan, used to define an object whose vertices can be seen as radiating from a central point. An example of this might be the top and bottom of a sphere, where a triangle fan can be created whose first point is the north or south pole of the sphere. The second is a triangle strip, which is often used to define very general kinds of surfaces, because most surfaces seem to have the kind of curvature that keeps rectangles of points on the surface from being planar. In this case, triangle strips are much better than quad strips as a basis for creating curved surfaces that will show their surface properties when lighted.

The triangle fan (that defines a cone, in this case) is organized with its vertex at point (0.0, 1.0, 0.0) and with a circular base of radius 0.5 in the X–Z plane. Thus the cone is oriented towards the y-direction and is centered on the y-axis. This provides a surface with unit diameter and height, as shown in Figure 3.1. When the cone is used in creating a scene, it can easily be defined to have whatever size, orientation, and location you need by applying appropriate modeling transformations in an appropriate sequence. Here we have also added normals and flat shading to emphasize the geometry of the triangle fan, although the code does not reflect this.

```
glBegin(GL_TRIANGLE_FAN);
   glVertex3f(0., 1.0, 0.);  // the point of the cone
   for (i=0; i < numStrips; i++) {
      angle = 2. * (float)i * PI / (float)numStrips;
      glVertex3f(0.5*cos(angle), 0.0, 0.5*sin(angle));
      // code to calculate normals would go here
   }
glEnd();
```

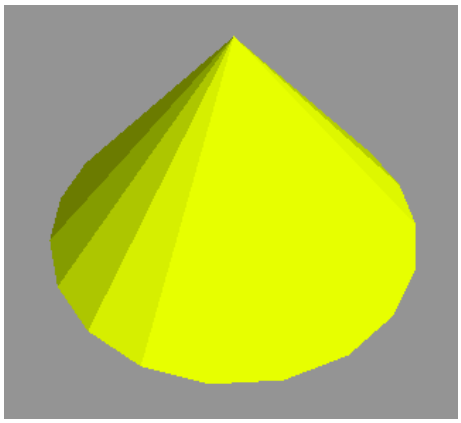

Figure 3.1: the cone produced by the triangle fan

The triangle strip example is based on an example of a function surface defined on a grid. Here we describe a function whose domain is in the X–Z plane and whose values are shown as the Y-value of each vertex. The grid points in the X–Z plane are given by functions `XX(i)` and `ZZ(j)`, and the values of the function are held in an array, with `vertices[i][j]` giving the value of the function at the grid point `(XX(i),ZZ(j))` as defined in the short example code fragment below.

```
for ( i=0; i<XSIZE; i++ )
   for ( j=0; j<ZSIZE; j++ ) {
      x = XX(i);
      z = ZZ(j);
      vertices[i][j] = (x*x+2.0*z*z)/exp(x*x+2.0*z*z+t);
   }
```

The surface rendering can then be organized as a nested loop, where each iteration of the loop draws a triangle strip that presents one section of the surface. Each section is one unit in the X direction that extends across the domain in the Z direction. The code for such a strip is shown below, and the resulting surface is shown in Figure 3.2. Again, the code that calculates the normals is omitted; this example is discussed further and the normals are developed in the later chapter on shading. This kind of surface is explored in more detail in the chapters on scientific applications of graphics.

```
for ( i=0; i<XSIZE-1; i++ )
   for ( j=0; j<ZSIZE-1; j++ ) {
      glBegin(GL_TRIANGLE_STRIP);
         glVertex3f(XX(i),vertices[i][j],ZZ(j));
         glVertex3f(XX(i+1),vertices[i+1][j],ZZ(j));
         glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
         glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
      glEnd();
   }
```

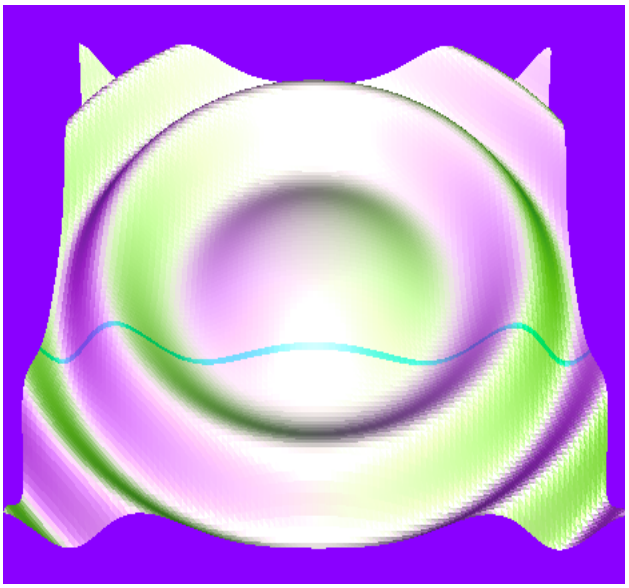

Figure 3.2: the full surface created by triangle strips, with a single strip highlighted in cyan

This example is a white surface lighted by three lights of different colors, a technique we describe in the chapter on lighting. This surface example is also briefly revisited in the quads discussion

below. Note that the sequence of points here is slightly different here than it is in the example below because of the way quads are specified. In this example instead of one quad, we will have two triangles—and if you rework the example below to use quad strips instead of simple quads to display the mathematical surface, it is simple to make the change noted here and do the surface with extended triangle strips.

Quads

To create a set of one or more distinct quads you use `glBegin/glEnd` with the `GL_QUADS` mode. As described earlier, this will take four vertices for each quad. An example of an object based on quadrilaterals would be the function surface discussed in the triangle strip above. For quads, the code for the surface looks like this:

```
for ( i=0; i<XSIZE-1; i++ )
   for ( j=0; j<ZSIZE-1; j++ ) {
      // quad sequence: points (i,j),(i+1,j),(i+1,j+1),(i,j+1)
      glBegin(GL_QUADS);
         glVertex3f(XX(i),vertices[i][j],ZZ(j));
         glVertex3f(XX(i+1),vertices[i+1][j],ZZ(j));
         glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
         glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
      glEnd();
   }
```

Note that neither this surface nor the one composed from triangles is going to look very good yet because it does not yet contain any lighting or color information. These will be added in later chapters as this concept of function surfaces is re-visited when we discuss lighting and color.

Quad strips

To create a sequence of quads, the mode for `glBegin/glEnd` is `GL_QUAD_STRIP`. This operates in the way we described at the beginning of the chapter, and as we noted there, the order in which the vertices are presented is different from that in the GL_QUADS mode because a quad strip can be implemented as a triangle strip. Be careful of this when you define your geometry or you may get a very unusual kind of display!

In a fairly common application, we can create long, narrow tubes with square cross-section. This can be used as the basis for drawing 3-D coordinate axes or for any other application where you might want to have, say, a beam in a structure. The quad strip defined below creates the tube oriented along the *Z*-axis with the cross-section centered on that axis. The dimensions given make a unit tube—a tube that is one unit in each dimension, making it actually a cube. These dimensions will make it easy to scale to fit any particular use.

```
#define RAD 0.5
#define LEN 1.0
glBegin(GL_QUAD_STRIP);
   glVertex3f( RAD, RAD, LEN ); // start of first side
   glVertex3f( RAD, RAD, 0.0 );
   glVertex3f(-RAD, RAD, LEN );
   glVertex3f(-RAD, RAD, 0.0 );
   glVertex3f(-RAD,-RAD, LEN ); // start of second side
   glVertex3f(-RAD,-RAD, 0.0 );
   glVertex3f( RAD,-RAD, LEN ); // start of third side
   glVertex3f( RAD,-RAD, 0.0 );
```

```
   glVertex3f( RAD, RAD, LEN ); // start of fourth side
   glVertex3f( RAD, RAD, 0.0 );
glEnd();
```

You can also get the same object by using the GLUT cube that is discussed below and applying appropriate transformations to center it on the *Z*-axis.

General polygon

The GL_POLYGON mode for glBegin/glEnd is used to allow you to display a single convex polygon. The vertices in the vertex list are taken as the vertices of the polygon in sequence order, and we remind you that the polygon needs to be convex. It is not possible to display more than one polygon with this operation because the function will always assume that whatever points it receives go in the same polygon.

The definition of GL_POLYGON mode is that it displays a convex polygon, but what if you give it a non-convex polygon? (Examples of convex and non-convex polygons are given in Figure 3.3, repeated from the previous chapter.) As we saw in the previous chapter, a convex polygon can be represented by a triangle fan so OpenGL tries to draw the polygon using a triangle fan. This will cause very strange-looking figures if the original polygon is not complex!

Figure 3.3: A convex polygon (left) and a non-convex polygon (right)

Probably the simplest kind of multi-sided convex polygon is the regular *N*-gon, an *N*-sided figure with all edges of equal length and all interior angles between edges of equal size. This is simply created (in this case, for *N*=7), again using trigonometric functions to determine the vertices.

```
#define PI 3.14159
#define N 7
step = 2.0*PI/(float)N;
glBegin(GL_POLYGON);
   for ( i=0; i<=N; i++)
      glVertex3f(2.0*sin(step*(float)i),
                 2.0*cos(step*(float)i),0.0);
glEnd();
```

Note that this polygon lives in the *X*–*Y* plane; all the *Z*-values are zero. This polygon is also in the default color (white) because we have not specified the color to be anything else. This is an example of a "canonical" object—an object defined not primarily for its own sake, but as a template that can be used as the basis of building another object as noted later, when transformations and object color are available. An interesting application of regular polygons is to create regular polyhedra—closed solids whose faces are all regular *N*-gons. These polyhedra are created by writing a function to draw a simple *N*-gon and then using transformations to place these properly in 3-space to be the boundaries of the polyhedron.

Antialiasing

As we saw in the previous chapter, geometry drawn with antialiasing is smoother and less "jaggy" than geometry drawn in the usual "all-or-nothing" pixel mode. OpenGL provides some capabilities for antialiasing by allowing you to enable point smoothing, line smoothing, and/or polygon smoothing. These are straightforward to specify, but they operate with color blending and there may be some issues around the order in which you draw your geometry. OpenGL calculates the coverage factor for antialiasing based on computing the proportion of a pixel that is covered by the geometry being presented, as described in the previous chapter. For more on RGBA blending and the order in which drawing is done, see the later chapter on color and blending.

To use the built-in OpenGL antialiasing, choose the various kinds of point, line, or polygon smoothing with the `glEnable(...)` function. Each implementation of OpenGL will define a default behavior for smoothing, so you may want to override that default by defining your choice with the `glHint(...)` function. The appropriate pairs of enable/hint are shown here:

```
glEnable(GL_LINE_SMOOTH);
glHint(GL_LINE_SMOOTH_HINT,GL_NICEST);
glEnable(GL_POINT_SMOOTH);
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);
glEnable(GL_POLYGON_SMOOTH);
glHint(GL_POLYGON_SMOOTH_HINT,GL_NICEST);
```

There is a more sophisticated kind of polygon smoothing involving entire scene antialiasing, done by drawing the scene into the accumulation buffer with slight offsets so that boundary pixels will be chosen differently for each version. This is a time-consuming process and is generally considered a more advanced use of OpenGL than we are assuming in this book. We do discuss the accumulation buffer in a later chapter when we discuss motion blur, but we will not go into more detail here.

The cube we will use in many examples

Because a cube is made up of six square faces, it is very tempting to try to make the cube from a single quad strip. Looking at the geometry, though, it is impossible to make a single quad strip go around the cube; in fact, the largest quad strip you can create from a cube's faces has only four quads. It is possible to create two quad strips of three faces each for the cube (think of how a baseball is stitched together), but here we will only use a set of six quads whose vertices are the eight vertex points of the cube. Below we repeat the declarations of the vertices, normals, edges, and faces of the cube from the previous chapter. We will use the `glVertex3fv(…)` vertex specification function within the specification of the quads for the faces.

```
typedef float point3[3];
typedef int   edge[2];
typedef int   face[4];  // each face of a cube has four edges

   point3 vertices[8] = {{-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} };

   point3 normals[6]  = {{ 0.0, 0.0, 1.0},
                         {-1.0, 0.0, 0.0},
                         { 0.0, 0.0,-1.0},
                         { 1.0, 0.0, 0.0},
```

```
                        { 0.0,-1.0, 0.0},
                        { 0.0, 1.0, 0.0} };

edge   edges[24]   = {{ 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                      { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                      { 4, 5 }, { 5, 7 }, { 7, 6 }, { 6, 4 },
                      { 1, 0 }, { 3, 1 }, { 2, 3 }, { 0, 2 },
                      { 4, 0 }, { 5, 1 }, { 7, 3 }, { 6, 2 },
                      { 5, 4 }, { 7, 5 }, { 6, 7 }, { 4, 6 }};

face   cube[6]     = {{  0,  1,  2,  3 }, {  5,  9, 18, 13 },
                      { 14,  6, 10, 19 }, {  7, 11, 16, 15 },
                      {  4,  8, 17, 12 }, { 22, 21, 20, 23 }};
```

As we said before, drawing the cube proceeds by working our way through the face list and determining the actual points that make up the cube. We will expand the function we gave earlier to write the actual OpenGL code below. Each face is presented individually in a loop within the `glBegin-glEnd` pair, and with each face we include the normal for that face. Note that only the first vertex of the first edge of each face is identified, because the GL_QUADS drawing mode takes each set of four vertices as the vertices of a quad; it is not necessary to close the quad by including the first point twice.

```
void cube(void) {
   int face, edge;
   glBegin(GL_QUADS);
      for (face = 0; face < 6; face++) {
         glNormal3fv(normals[face];
         for (edge = 0; edge < 4; edge++)
            glVertex3fv(vertices[edges[cube[face][edge]][0]]);
      }
   glEnd();
}
```

This cube is shown in Figure 3.4, presented through the six steps of adding individual faces (the faces are colored in the typical RGBCMY sequence so you may see each added in turn). This approach to defining the geometry is actually a fairly elegant way to define a cube, and takes very little coding to carry out. However, this is not the only approach we could take to defining a cube. Because the cube is a regular polyhedron with six faces that are squares, it is possible to define the cube by defining a standard square and then using transformations to create the faces from this master square. Carrying this out is left as an exercise for the student.

This approach to modeling an object includes the important feature of specifying the normals (the vectors perpendicular to each face) for the object. We will see in the chapters on lighting and shading that in order to get the added realism of lighting on an object, we must provide information on the object's normals, and it was straightforward to define an array that contains a normal for each face. Another approach would be to provide an array that contains a normal for each vertex if you would want smooth shading for your model; see the chapter on shading for more details. We will not pursue these ideas here, but you should be thinking about them when you consider modeling issues with lighting.

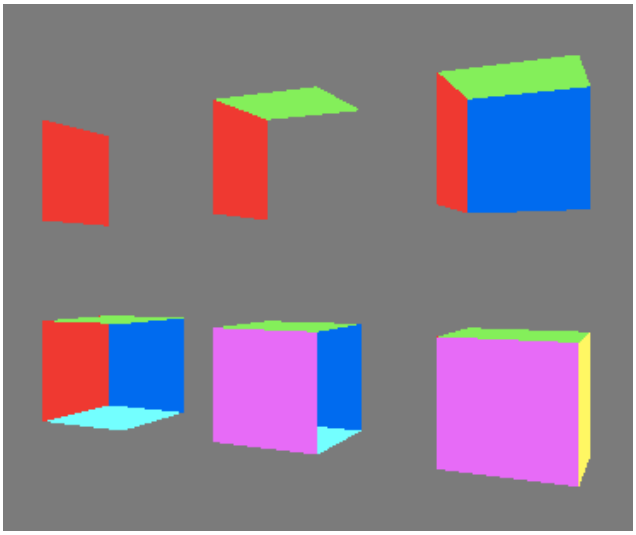

Figure 3.4: the cube as a sequence of quads

Additional objects with the OpenGL toolkits

Modeling with polygons alone would require you to write many standard graphics elements that are so common, any reasonable graphics system should include them. OpenGL includes the OpenGL Utility Library, GLU, with many useful functions, and most releases of OpenGL also include the OpenGL Utility Toolkit, GLUT. We saw in the first chapter that GLUT includes window management functions, and both GLU and GLUT include a number of built-in graphical elements that you can use. This chapter describes a number of these elements.

The objects that these toolkits provide are defined with several parameters that define the details, such as the resolution in each dimension of the object with which the object is to be presented. Many of these details are specific to the particular object and will be described in more detail when we describe each of these.

GLU quadric objects

The GLU toolkit provides several general quadric objects, which are objects defined by quadric equations (polynomial equations in three variables with degree no higher than two in any term), including Spheres (`gluSphere`), cylinders (`gluCylinder`), and disks (`gluDisk`). Each GLU primitive is declared as a GLUquadric and is allocated with the function

```
GLUquadric* gluNewQuadric( void )
```

Each quadric object is a surface of revolution around the *Z*-axis. Each is modeled in terms of subdivisions around the *Z*-axis, called slices, and subdivisions along the *Z*-axis, called stacks. Figure 3.5 shows an example of a typical pre-built quadric object, a GLUT wireframe sphere, modeled with a small number of slices and stacks so you can see the basis of this definition.

The GLU quadrics are very useful in many modeling circumstances because you can use scaling and other transformations to create many common objects from them. The GLU quadrics are also useful because they have capabilities that support many of the OpenGL rendering capabilities that support creating interesting images. You can determine the drawing style with the `gluQuadricDrawStyle()` function that lets you select whether you want the object filled, wireframe, silhouette, or drawn as points. You can get normal vectors to the surface for lighting models and smooth shading with the `gluQuadricNormals()` function that lets you choose whether you want no normals, or normals for flat or smooth shading. Finally, with the

`gluQuadricTexture()` function you can specify whether you want to apply texture maps to the GLU quadrics in order to create objects with visual interest. See later chapters on lighting and on texture mapping for the details.

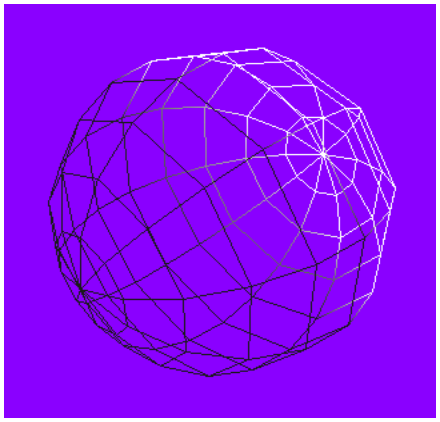

Figure 3.5: A GLUT wireframe sphere with 10 slices and 10 stacks

Below we describe each of the GLU primitives by listing its function prototype; more details may be found in the GLU section of your OpenGL manual.

GLU cylinder:

```
void gluCylinder(GLUquadric* quad, GLdouble base, GLdouble top,
GLdouble height, GLint slices, GLint stacks)
```

quad identifies the quadrics object you previously created with `gluNewQuadric`
base is the radius of the cylinder at $z = 0$, the base of the cylinder
top is the radius of the cylinder at z = height, and
height is the height of the cylinder.

GLU disk:

The GLU disk is different from the other GLU primitives because it is only two-dimensional, lying entirely within the X–Y plane. Thus instead of being defined in terms of stacks, the second granularity parameter is loops, the number of concentric rings that define the disk.

```
void gluDisk(GLUquadric* quad, GLdouble inner, GLdouble outer,
GLint slices, GLint loops)
```

quad identifies the quadrics object you previously created with `gluNewQuadric`
inner is the inner radius of the disk (may be 0).
outer is the outer radius of the disk.

GLU sphere:

```
void gluSphere(GLUquadric* quad, GLdouble radius, GLint slices,
GLint stacks)
```

quad identifies the quadrics object you previously created with `gluNewQuadric`
radius is the radius of the sphere.

The GLUT objects

Models provided by GLUT are more oriented to geometric solids, except for the teapot object. They do not have as wide a usage in general situations because they are of fixed shape and many cannot be modeled with varying degrees of complexity. They also do not include shapes that can readily be adapted to general modeling situations. Finally, there is no general way to create a texture map for these objects, so it is more difficult to make scenes using them have stronger visual interest. The GLUT models include a cone (`glutSolidCone`), cube (`glutSolidCube`), dodecahedron (12-sided regular polyhedron, `glutSolidDodecahedron`), icosahedron (20-sided regular polyhedron, `glutSolidIcosahedron`), octahedron (8-sided regular polyhedron, `glutSolidOctahedron`), a sphere (`glutSolidSphere`), a teapot (the Utah teapot, an icon of computer graphics sometimes called the "teapotahedron", `glutSolidTeapot`), a tetrahedron (4-sided regular polyhedron, `glutSolidTetrahedron`), and a torus (`glutSolidTorus`).

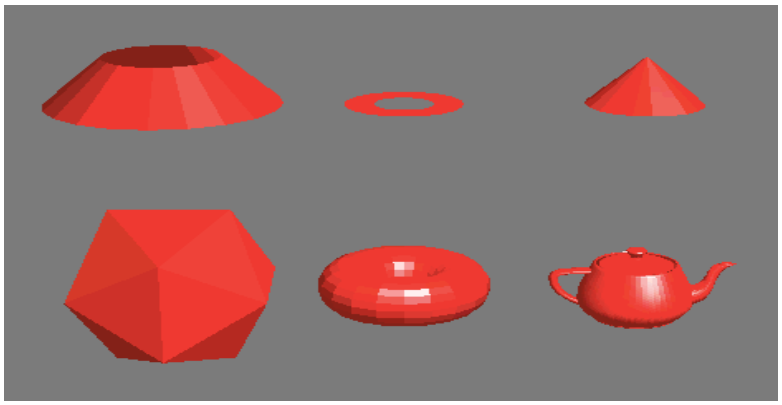

Figure 3.6: several GLU and GLUT objects as described in the text

While we only listed the "solid" versions of the GLUT primitives, they include both solid and wireframe versions. Each object has a canonical position and orientation, typically being centered at the origin and lying within a standard volume and, if it has an axis of symmetry, that axis is aligned with the z-axis. As with the GLU standard primitives, the GLUT cone, sphere, and torus allow you to specify the granularity of the primitive's modeling, but the others do not. You should not take the term "solid" for the GLUT objects too seriously; they are not actually solid but are simply bounded by polygons. "Solid" merely means that the shapes are filled in, in contrast with the `glutWireSphere` and similar constructs. If you clip the "solid" objects you will find that they are, in fact, hollow.

If you have GLUT with your OpenGL, you should check the GLUT manuals for the details on these solids and on many other important capabilities that GLUT will add to your OpenGL system. If you do not already have it, you can download the GLUT code from the OpenGL Web site for many different systems and install it in your OpenGL area so you may use it readily with your system.

Selections from the overall collection of GLU and GLUT objects are shown in Figure 3.6 to show the range of items you can create with these tools. From top left and moving clockwise, we see a gluCylinder, a gluDisk, a glutSolidCone, a glutSolidIcosahedron, a glutSolidTorus, and a glutSolidTeapot. You should think about how you might use various transformations to create other figures from these basic parts.

An example

Our example for this module is quite simple. It is the heart of the `display()` function for a simple application that displays the built-in sphere, cylinder, dodecahedron, torus, and teapot provided by OpenGL and the GLU and GLUT toolkits. In the full example, there are operations that allow the user to choose the object and to control its display in several ways, but for this example we will only focus on the models themselves, as provided through a switch() statement such as might be used to implement a menu selection. This function is not complete, but would need the addition of viewing and similar functionality that is described in the chapter on viewing and projection.

```
void display( void )
{
   GLUquadric *myQuad;
   GLdouble radius = 1.0;
   GLint slices, stacks;
   GLint nsides, rings;

   ...

   switch (selectedObject) {
      case (1): {
         myQuad=gluNewQuadric();
         slices = stacks = resolution;
         gluSphere( myQuad , radius , slices , stacks  );
         break;
         }
      case (2): {
         myQuad=gluNewQuadric();
         slices = stacks = resolution;
         gluCylinder( myQuad, 1.0, 1.0, 1.0, slices, stacks );
         break;
         }
      case (3): {
         glutSolidDodecahedron(); break;
         }
      case (4): {
         nsides = rings = resolution;
         glutSolidTorus( 1.0, 2.0, nsides, rings);
         break;
         }
      case (5): {
         glutSolidTeapot(2.0); break;
         }
      }
   ...
}
```

A word to the wise...

One of the differences between student programming and professional programming is that students are often asked to create applications or tools for the sake of learning creation, not for the sake of creating working, useful things. The graphics primitives that are the subject of the first section of this module are the kind of tools that students are often asked to use, because they require more analysis of fundamental geometry and are good learning tools. However, working programmers developing real applications will often find it useful to use pre-constructed templates and tools such as the GLU or GLUT graphics primitives. You are encouraged to use the GLU and

GLUT primitives whenever they can save you time and effort in your work, and when you cannot use them, you are encouraged to create your own primitives in a way that will let you re-use them as your own library and will let you share them with others.

Transformations in OpenGL

In OpenGL, there are only two kinds of transformations: projection transformations and modelview transformations. The latter includes both the viewing and modeling transformations. We have already discussed projections and viewing, so here we will focus on the transformations used in modeling.

Among the modeling transformations, there are three fundamental kinds: rotations, translations, and scaling. In OpenGL, these are applied with the built-in functions (actually function sets) `glRotate`, `glTranslate`, and `glScale`, respectively. As we have found with other OpenGL function sets, there are different versions of each of these, varying only in the kind of parameters they take.

The `glRotate` function is defined as

```
glRotatef(angle, x, y, z)
```

where `angle` specifies the angle of rotation, in degrees, and *x*, *y*, and *z* specify the coordinates of a vector, all as floats (`f`). There is another rotation function `glRotated` that operates in exactly the same way but the arguments must all be doubles (`d`). The vector specified in the parameters defines the fixed line for the rotation. This function can be applied to any matrix set in `glMatrixMode`, allowing you to define a rotated projection if you are in projection mode or to rotate objects in model space if you are in modelview mode. You can use `glPushMatrix` and `glPopMatrix` to save and restore the unrotated coordinate system.

This rotation follows the right-hand rule, so the rotation will be counterclockwise as viewed from the direction of the vector (*x*, *y*, *z*). The simplest rotations are those around the three coordinate axes, so that `glRotatef(angle, 1., 0., 0.)` will rotate the model space around the *X*-axis.

The `glTranslate` function is defined as

```
glTranslatef(Tx, Ty, Tz)
```

where *Tx*, *Ty*, and *Tz* specify the coordinates of a translation vector as floats (`f`). Again, there is a translation function `glTranslated` that operates exactly the same but has doubles (`d`) as arguments. As with `glRotate`, this function can be applied to any matrix set in `glMatrixMode`, so you may define a translated projection if you are in projection mode or translated objects in model space if you are in modelview mode. You can again use `glPushMatrix` and `glPopMatrix` to save and restore the untranslated coordinate system.

The `glScale` function is defined as

```
glScalef(Sx, Sy, Sz)
```

where *Sx*, *Sy*, and *Sz* specify the coordinates of a scaling vector as floats (`f`). Again, there is a translation function `glScaled` that operates exactly the same but has doubles (`d`) as arguments. As above, this function can be applied to any matrix set in `glMatrixMode`, so you may define a scaled projection if you are in projection mode or scaled objects in model space if you are in modelview mode. You can again use `glPushMatrix` and `glPopMatrix` to save and restore the unscaled coordinate system. Because scaling changes geometry in non-uniform ways, a scaling transformation may change the normals of an object. If scale factors other than 1.0 are applied in modelview mode and lighting is enabled, automatic normalization of normals should probably also be enabled. See the chapter on lighting for details.

OpenGL provides a number of facilities for manipulating transformations. As we will see in more detail in the chapter on mathematical fundamentals, a transformation for 3D computer graphics is represented by a 4x4 array, which in turn is stored as an array of 16 real numbers. You may save the current modelview matrix with the function `glGetFloatv(GL_MODELVIEW_MATRIX, params)` where params is an array `GLfloat params[16]`. You do not restore the modelview matrix directly, but if your transformation mode is set to modelview with the function `glMatrixMode(GL_MODELVIEW)`, you can multiply the current modelview matrix by `myMatrix`, saved as a 16-element array, with the function `glMultMatrix(myMatrix)`. You can do similar manipulations of the OpenGL projection matrix. This kind of operation requires you to be comfortable with expressing and manipulating transformations as matrices, but OpenGL provides enough transformation tools that it's rare to need to handle transformations this way.

As we saw earlier in the chapter, there are many transformations that go into defining exactly how a piece of geometry is presented in a graphics scene. When we consider the overall order of transformations for the entire model, we must consider not only the modeling transformations but also the projection and viewing transformations. If we consider the total sequence of transformations in the order in which they are specified, we will have the sequence:

$$P \quad V \quad T_0 \quad T_1 \quad \dots \quad T_n \quad T_{n+1} \quad \dots \quad T_{last}$$

with P being the projection transformation, V the viewing transformation, and T_0, T_1, … T_{last} the transformations specified in the program to model the scene, in order (T_0 is first, T_{last} is last and is closest to the actual geometry). The projection transformation is defined in the `reshape()` function; the viewing transformation is defined in the `init()` function, in the `reshape` function, or at the beginning of the `display()` function so it is defined at the beginning of the modeling process. But the sequence in which the transformations are applied is actually the reverse of the sequence above: T_{last} is actually applied first, and V and finally P are applied last. You need to understand this sequence very well, because it's critical to understand how you build complex, hierarchical models.

Code examples for transformations

Simple transformations:
All the code examples use a standard set of axes, which are not included here, and the following definition of the simple square:

```
void square (void)
{
   typedef GLfloat point [3];
   point v[8] = {{12.0, -1.0, -1.0},
                 {12.0, -1.0,  1.0},
                 {12.0,  1.0,  1.0},
                 {12.0,  1.0, -1.0} };

   glBegin (GL_QUADS);
      glVertex3fv(v[0]);
      glVertex3fv(v[1]);
      glVertex3fv(v[2]);
      glVertex3fv(v[3]);
   glEnd();
}
```

To display the simple rotations example, we use the following display function:

```
void display( void )
```

```
{  int i;
   float theta = 0.0;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   axes(10.0);
   for (i=0; i<8; i++) {
      glPushMatrix();
      glRotatef(theta, 0.0, 0.0, 1.0);
      if (i==0)
         glColor3f(1.0, 0.0, 0.0);
      else
         glColor3f(1.0, 1.0, 1.0);
      square();
      theta += 45.0;
      glPopMatrix();
   }
   glutSwapBuffers();
}
```

To display the simple translations example, we use the following display function:

```
void display( void )
{  int i;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   axes(10.0);
   for (i=0; i<=12; i++) {
      glPushMatrix();
      glTranslatef(-2.0*(float)i, 0.0, 0.0);
      if (i==0) glColor3f(1.0, 0.0, 0.0);
         else  glColor3f(1.0, 1.0, 1.0);
      square();
      glPopMatrix();
   }
   glutSwapBuffers();
}
```

To display the simple scaling example, we use the following display function:

```
void display( void )
{  int i;
   float s;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   axes(10.0);
   for (i=0; i<6; i++) {
      glPushMatrix();
      s = (6.0-(float)i)/6.0;
      glScalef( s, s, s );
      if (i==0)
         glColor3f(1.0, 0.0, 0.0);
      else
         glColor3f(1.0, 1.0, 1.0);
      square();
      glPopMatrix();
   }
   glutSwapBuffers();
}
```

Transformation stacks: The OpenGL functions that are used to manage the transformation stack are `glPushMatrix()` and `glPopMatrix()`. Technically, they apply to the stack of whatever transformation is the current matrix mode, and the `glMatrixMode` function with parameters `GL_PROJECTION` and `GL_MODELVIEW` sets that mode. We only rarely want to use a stack of projection transformations (and in fact the stack of projections can only hold two transformations) so we will almost always work with the stack of modeling/viewing transformation. The rabbit head example was created with the display function given below. This function makes the stack operations more visible by using indentations; this is intended for emphasis in the example only and is not standard programming practice in graphics. Note that we have defined only very simple display properties (just a simple color) for each of the parts; we could in fact have defined a much more complex set of properties and have made the parts much more visually interesting. We could also have used a much more complex object than a simple `gluSphere` to make the parts much more structurally interesting. The sky's the limit…

```
void display( void )
{
// Indentation level shows the level of the transformation stack
// The basis for this example is the unit gluSphere; everything
// else is done by explicit transformations

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
      // model the head
      glColor3f(0.4, 0.4, 0.4);    //   dark gray head
      glScalef(3.0, 1.0, 1.0);
      myQuad = gluNewQuadric();
      gluSphere(myQuad, 1.0, 10, 10);
   glPopMatrix();
   glPushMatrix();
      // model the left eye
      glColor3f(0.0, 0.0, 0.0);    //   black eyes
      glTranslatef(1.0, -0.7, 0.7);
      glScalef(0.2, 0.2, 0.2);
      myQuad = gluNewQuadric();
      gluSphere(myQuad, 1.0, 10, 10);
   glPopMatrix();
   glPushMatrix();
      // model the right eye
      glTranslatef(1.0, 0.7, 0.7);
      glScalef(0.2, 0.2, 0.2);
      myQuad = gluNewQuadric();
      gluSphere(myQuad, 1.0, 10, 10);
   glPopMatrix();
   glPushMatrix();
      // model the left ear
      glColor3f(1.0, 0.6, 0.6);    //   pink ears
      glTranslatef(-1.0, -1.0, 1.0);
      glRotatef(-45.0, 1.0, 0.0, 0.0);
      glScalef(0.5, 2.0, 0.5);
      myQuad = gluNewQuadric();
      gluSphere(myQuad, 1.0, 10, 10);
   glPopMatrix();
   glPushMatrix();
      // model the right ear
      glColor3f(1.0, 0.6, 0.6);    //   pink ears
      glTranslatef(-1.0, 1.0, 1.0);
      glRotatef(45.0, 1.0, 0.0, 0.0);
```

```
        glScalef(0.5, 2.0, 0.5);
        myQuad = gluNewQuadric();
        gluSphere(myQuad, 1.0, 10, 10);
    glPopMatrix();
    glutSwapBuffers();
}
```

In OpenGL, the stack for the modelview matrix is to be at least 32 deep, but this can be inadequate to handle some complex models if the hierarchy is more than 32 layers deep. In this case, as we mentioned in the previous chapter, you need to know that a transformation is a 4x4 matrix of `GLfloat` values that is stored in a single array of 16 elements. You can create your own stack of these arrays that can have any depth you want, and then push and pop transformations as you wish on that stack. To deal with the modelview transformation itself, there are functions that allow you to save and to set the modelview transformation as you wish. You can capture the current value of the transformation with the function

```
glGetFloatv(GL_MODELVIEW_MATRIX, viewProj);
```

(here we have declared `GLfloat viewProj[16]`), and you can use the functions

```
glLoadIdentity();
glMultMatrixf( viewProj );
```

to set the current modelview matrix to the value of the matrix `viewProj`, assuming that you were in modelview mode when you execute these functions.

Inverting the eyepoint transformation

In an example somewhat like the more complex eye-following-helicopter example above, we built a small program in which the eye follows a red sphere at a distance of 4 units as the sphere flies in a circle above some geometry. In this case, the geometry is a cyan plane on which are placed several cylinders at the same distance from the center as the sphere flies, along with some coordinate axes. A snapshot from this very simple model is shown in Figure 3.7. The display

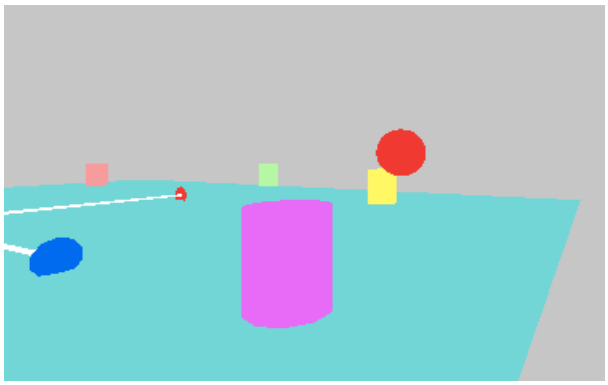

Figure 3.7 the eye following a sphere flying over some cylinders on a plane

function code that implements this viewing is shown after the figure; you will note that the display function begins with the default view and is followed by the transformations

translate by –4 in *Z*
translate by –5 in *X* and –.75 in *Y*
rotate by *-theta* around *Y*

that are the inverse of the cylinder placement and eye placement, first for the sphere

rotate by theta around *Y*
translate by 5 in *X* and .75 in *Y*

and then for the eye to lie 4 units in *Z* from the sphere. Note that this means, for example, that when the sphere is on the *X*-axis, for example, the eye is 4 units from the sphere in the *Z*-direction. There is no explicit transformation in the code, however; the inverse is from the position of the eyepoint in the scene graph, because the eye point is never set relative to the sphere in the code.

```
void display( void )
{
   ...
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   // Define eye position relative to the sphere it is to follow
   // place eye in scene with default definition
   gluLookAt(0.,0.,0., 0.,0.,-1., 0.,1.,0.);

   // invert transformations that place the eye
   glTranslatef(0.,0.,-4.);
   glTranslatef(-5.,-0.75,0.);
   glRotatef(-theta,0.,1.,0.);

   // draw the sphere we're following around...
   glPushMatrix();
   glRotatef(theta, 0., 1., 0.);
   glTranslatef(5., 0.75, 0.);
   glColor3f(1., 0., 0.);
   myQuad = gluNewQuadric();
   gluSphere(myQuad, .25, 20, 20);
   glPopMatrix();

   // draw whatever geometry the sphere is flying over...
   ...

    glutSwapBuffers();
}
```

Creating display lists

In the previous chapter we discussed the idea of compiling geometry in order to make display operations more efficient. In OpenGL, graphics objects can be compiled into what is called a *display list*, which will contain the final geometry of the object as it is ready for display. Sample code and an explanation of display lists is given below.

Display lists are relatively easy to create in OpenGL. First, choose an unsigned integer (often you will just use small integer constants, such as 1, 2, ...) to serve as the name of your list. Then before you create the geometry for your list, call the function `glNewList`. Code whatever geometry you want into the list, and at the end, call the function `glEndList`. Everything between the new list and the end list functions will be executed whenever you call `glCallList` with a valid list name as parameter. All the operations between `glNewList` and `glEndList` will be carried out, and only the actual set of instructions to the drawing portion of the OpenGL system will be saved. When the display list is executed, then, those instructions are simply sent to the drawing system; any operations needed to generate these instructions are omitted.

Because display lists are generally intended to be defined only once but used often, you do not want to create a list in a function such as the `display()` function that is called often. Thus it is common to create them in the `init()` function or in a function called from within `init()`.

Some sample code is given below, with most of the content taken out and only the display list operations left.

```
void Build_lists(void) {
   glNewList(1, GL_COMPILE);
      glBegin(GL_TRIANGLE_STRIP);
         glNormal3fv(...); glVertex3fv(...);
         ...
      glEnd();
   glEndList();
}

static void Init(void) {
   ...
   Build_lists();
   ...
}

void Display(void) {
   ...
   glCallList(1);
   ...
}
```

You will note that the display list was created in `GL_COMPILE` mode, and it was not executed (the object was not displayed) until the list was called. It is also possible to have the list displayed as it is created if you create the list in `GL_COMPILE_AND_EXECUTE` mode.

OpenGL display lists are named by nonzero unsigned integer values (technically, `GLuint` values) and there are several tools available in OpenGL to manage these name values. We will assume in a first graphics course that you will not need many display lists and that you can manage a small number of list names yourself, but if you begin to use a number of display lists in a project, you should look into the `glGenLists`, `glIsList`, and `glDeleteLists` functions to help you manage the lists properly.

Summary

In this chapter we have presented the way the OpenGL graphics API allows you to define geometry and transformations to support the geometric modeling presented in the previous chapter. We have seen the OpenGL implementations of the point, line segment, triangle, quad, and polygon primitives as well as the line strip, triangle strip, triangle fan, and quad strip geometry compression techniques. We have also seen a number of geometric objects that are available through the GLU and GLUT graphics utilities and how they are used in creating scenes; while this chapter focuses only on geometry, we will see in later chapters that these GLU and GLUT primitives are also very easy to use with the OpenGL appearance tools.

We have also seen the way OpenGL implements the scaling, translation, and rotation transformations and how OpenGL manages the modelview transformation stack, allowing you to implement scene graphs easily with OpenGL tools. Finally, OpenGL allows you to compile parts of your geometry into display lists that you can re-use much more efficiently than would be done if you simply compute the geometry in immediate mode.

At this point you should have a good understanding of all the steps in the graphics pipeline for most polygon-based graphics, so you should be able to write complete graphics programs that include sound geometry. The only additional modeling presented in this book will be the surface

interpolation work for splines. The focus of most of the coming chapters, then, will be the appearance of graphics objects and creating more attractive or realistic graphics images.

Questions

1. The OpenGL primitives include quads as well as triangles, but is it really necessary to have a quad primitive? Is there anything you can do with quads that you couldn't do with triangles?

2. The GLU objects are handy, but they can be created pretty easily from the standard OpenGL triangle and quad primitives. Describe how to do this for as many of the objects as you can, including at least the gluSphere, gluCylinder, gluDisk, glutSolidCone, and glutCube.

3. The GLU objects and some GLUT objects use parameters, often called *slices* and *stacks* or *loops*, to define the granularity of the objects. For those objects that are defined with these parameters, can you think of another way you could define the granularity? What are the advantages of using small values for these parameters? What are the advantages of using large values for these parameters? Are there any GLU or GLUT objects that do not use slices and stacks? Why is this so?

4. Are the values for the parameters on the scaling and translation transformations in OpenGL in model space, world space, or 3D eye space? Give reasons for your answer.

5. The angle of the OpenGL rotation transformation is given in degrees around an arbitrary fixed line. What is the space in which the fixed line for the rotation is defined? In many application areas, angles are expressed in radians instead of degrees; what is the conversion formula that must be applied to radians to get degrees?

6. In the heat diffusion example in the Getting Started chapter, print the source code and highlight each OpenGL function that it uses. For each, explain what it does and why it's at this point in the code.

7. Consider the model in the `display()` function in the example of the previous question. Compare the number of operations needed to create and display this model, including all the transformations, with the number of `glVertex(...)` function calls that would be used in a display list. Draw conclusions about the relative efficiency of display lists and simple modeling.

8. The question above is actually misleading because the model in the heat diffusion example changes with each `idle()` callback. Why does it not make sense to try to use display lists for the heat diffusion problem?

9. In the carousel exercise in the previous chapter, place the eyepoint on one of the objects in the carousel and change the scene graph from the previous exercise to include this eye placement. Then write the scene graph that inverts the eyepoint to place it in standard position.

Exercises

Modeling is all about creating graphical objects, so the following sequence of exercises involves making some general graphical objects you can use.

10. Define a "unit cartoon dumbbell" as a thin cylinder on the x-axis with endpoints at 1.0 and –1.0, and with two spherical ends of modest size, each centered at one end of the cylinder. We

call this a cartoon dumbbell because early children's cartoons always seemed to use this shape when they involved weightlifters.

11. Let's make a more realistic set of weights with solid disk weights of various sizes. Define a set of standard disks with standard weights (5kg, 10kg, 20kg, say) with the weight of the disk as a parameter to determine the thickness and/or radius of the weight, assuming that the weight is proportional to the volume. Define a function that creates a barbell carrying a given weight that is a combination of the standard weights, placing the disks appropriately on the bar. (Note that we are not asking you to create realistic disks with a hole in the middle – yet.)

12. The 3D arrow that we used as an example in the previous chapter used general modeling concepts, not specific functions. Use the GLU and GLUT modeling tools to implement the 3D arrow as a working function you could use in an arbitrary scene.

13. Let's proceed to create an object that is a cylinder with a cylindrical hole, both having the same center line. Define the object to have unit length with inside and outside cylinders of defined radii and with disks at the ends to close the pipe. Show how you could use this object to create a more realistic kind of weight for the previous exercise.

14. All the cylinder-based objects we've defined so far have a standard orientation, but clearly we will want to have cylinders with any starting point and any ending point, so we will need to be able to give the cylinder any orientation. Let's start by considering a cylinder with one end at the origin and the other end at a point $P = (x, y, z)$ that is one unit from the origin. Write a function that rotates a unit cylinder with one end at the origin so the other end is at P. (Hint – you will need to use two rotations, and you should think about arctangent functions.)

15. With the orientation problem for a cylinder solved by the previous exercise, write a function to create a *tube strip* that connects points $p_0, p_1, p_2, \ldots p_n$ with tubes of radius r and with spheres of the same radius at the points in order to join the tubes to make a smooth transition from one tube to another. Use this to create a flexible bar between the two weights in the cartoon dumbbell in an earlier exercise, and show the bar supported in the middle with a bending bar in between.

16. Create a curve in the X–Y plane by defining a number of vertices and connecting them with a line strip. Then create a general surface of revolution by rotating this curve around the Y-axis. Make the surface model some object you know, such as a glass or a baseball bat, that has rotational symmetry. Can you use transformations to generalize this to rotate a curve around any line? How?

Experiments

17. OpenGL give you the ability to push and pop the transformation stack, but you can actually do a little better. Create a way to "mark" the stack, saving the top of the transformation stack when you do, and return to any marked stack point by restoring the transformation to the one that was saved when the stack was marked.

18. The previous experiment is nice, but it can easily lead to destroying the stack and leaving only the saved transformation on the stack. Can you find a way to return to a marked stack point and keep the stack as it was below that point?

Projects

19. Implement the carousel discussed in an earlier question, using some kind of standard object as the carousel horses—use a model from an outside source or make a very simple "horse" object. Create two views of the carousel, one from outside and one from the eye point on the horse.

20. (The small house) Design and create a simple house, with both exterior and interior walls, doors, windows, ceilings in the rooms, and a simple gable roof. Use different colors for various walls so they can be distinguished in any view. Set several different viewpoints around and within the house and show how the house looks from each.

21. (A scene graph parser) Implement the scene graph parser that was designed in this project in the previous chapter. Each transformation and geometry node is to contain the OpenGL function names and arguments needed to carry out the transformations and implement the geometry. The parser should be able to write the `display()` function for your scene.