Simple Geometric Modeling

Prerequisites

This module requires an understanding of simple 3-dimensional geometry, knowledge of how to represent points in 3-space, and enough programming experience to be comfortable writing code that calls functions to do the required tasks.

Introduction

In order to create any kind of image, one must first model the image in a way that can be understood by the computer and that is appropriate for the graphics API being used. For most graphics programming, particularly when beginning to learn the field, one models an image using a few simple graphics primitives — points, line segments, and polygons.

A point is simply a single location in 3-space, specified by its coordinates. It is drawn by lighting a single pixel at the screen location that best represents the location of that point in space. To draw the point you specify that you want to draw points and specify the point’s coordinates. A line segment is determined by its two endpoints, so to draw the line you indicate that you want to draw lines and specify the two endpoints. A polygon is determined by a sequence of points (called the vertices of the polygon), so to draw the polygon you indicate that you want to draw polygons and specify the sequence of vertex points. Figure 2.1 illustrates the distinction between a point, a line segment, and a polygon.

Perhaps the most difficult — or at least the most time-consuming — part of beginning graphics programming is creating the models of the image you want to create so that the image can be drawn. This may require you to create hand-drawn sketches of your image so you can determine the correct values for the points used in defining it, for example, or it may be possible to determine the values for points from some other technique. But until you get the points right, you will not be able to get the image right.

Definitions

There are some terms we need to talk about modeling. Modeling is defining the objects that are part of the scene you want to view in an image. There are many ways to model a scene for an image; in fact, there are a number of commercial programs you can buy that let you model scenes with very high-level tools. For much graphics programming, and certainly as you are beginning in this field, you will do your modeling by defining your geometry in terms of relatively simple primitive terms.

While it is possible to think about graphics in two dimensions, these modules consider graphics entirely in three-dimensional terms. Because of this, we will think about 3D graphics primitives. Probably the most important of these is the concept of a polyhedron — a 3D object which is bounded by polygons. In turn, a polygon is a plane region bounded by edges, and an edge is a line segment determined by vertices. Finally, a vertex is a point in 3-space. Because figures in 3-space determined by more than three vertices cannot be guaranteed to lie in a plane, polyhedra...
are often defined to have triangular faces; a triangle always lies in a plane (because three points in 3-space determine a plane.

All modeling requires that you know the coordinates of the points you need to create your image. Once you have chosen the simple components that the image will have, and you know the coordinates of all the vertex points for these components, you will use functions such as those below to create the image:

```c
    glBegin(mode);
    // vertex list: point data to create a primitive object in
    // the drawing mode you have indicated
    glEnd();
```

The vertex list is interpreted differently 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.

Besides defining a single point, line segment, or polygon, there are techniques for defining larger objects that are made up of several simple objects. These can involve disconnected sets of objects, such as points, line segments, quads, or triangles, or can build images from connected objects, such as lines, quad strips, triangle strips, or triangle fans. Look at the discussions and examples below to build your repertoire of techniques you can use for your modeling.

Point (or vertex) information is presented to the computer through a set of functions that go under the general name of `glVertex(…)`. These functions transform raw numeric information into a form that the system can use. We say that `glVertex(…)` is a set of functions because there are many functions that differ only in the way they define their point data. You may want or need to specify your vertex coordinate data in any standard numeric type, and these functions allow the system to respond to your needs. For full details, see the manuals on OpenGL; here we will use only those versions of the `glVertex` function that we believe are most useful.

- If you want to specify your vertex data as three separate floats (we'll call them `x`, `y`, and `z`, though they could also be float constants), you can use `glVertex3f(x,y,z)` (note that the letter `f` is appended to the `glVertex` name; if you wanted to use three integers, you would use `glVertex3i` and if you wanted to use doubles you would use `glVertex3d`. Moreover if you wanted to do 2-dimensional graphics you would replace the 3 by 2.
- If you want to specify your data as an array you could declare your data as `glfloat x[3]` and then use `glVertex3fv(x)` to specify the vertex. This adds the letter `v` to say that the data is a vector (actually a pointer to an array, but an array’s name is really such a pointer). As before, you can use 2, 3, or 4 for the dimension and can use any of several letters besides `f` if your data is another type besides float.

You will see this used in the code examples at the end of this module.

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 
\texttt{glBegin(mode)} and \texttt{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.

\textit{Some examples}

We will begin with very simple objects and proceed to more useful ones. With each kind of 
primitive object, we will describe how that object is specified, and in later examples, we will create 
as a set of points and will then show the function call that draws the object we have defined.

Point and points

This is perhaps the simplest of all the drawings. In the \texttt{glBegin} function, the mode we use is 
\texttt{GL_POINTS}, and the data between \texttt{glBegin} and \texttt{glEnd} is the set of coordinates of each point 
we wish to draw. As noted above, each point’s coordinates are presented using some form of the 
\texttt{glVertex* (...) function}, and the number of points drawn is the number of vertices that are 
specified. If we want to draw only one point, we provide only one vertex between \texttt{glBegin} and 
\texttt{glEnd}; if we want to draw more points, we provide more vertices between them.

Line segments

To draw a line segment, we use the \texttt{GL_LINES mode} for \texttt{glBegin}. For each segment we wish 
to draw, the data we need to provide is the two endpoints of the segment. Thus between 
\texttt{glBegin} and \texttt{glEnd} we must provide two vertices in the vertex list for each segment, using 
some form of the \texttt{glVertex* (...) function}.

Connected lines

Connected lines — line segments that are joined “head to tail” to form a longer connected group — 
are available by using the mode \texttt{GL_LINE_STRIP} for \texttt{glBegin}. The vertex list defines the line 
segments by using the first two vertices for the first line segment, and then by using each new 
vertex and its predecessor to define each additional segment. Thus if you have N vertices, you will 
have \(N-1\) line segments.

Triangle

If you want to draw one or more unconnected triangles, you can use \texttt{glBegin} with the mode 
\texttt{GL_TRIANGLES}. In this mode, the vertex list is taken three at a time, and each set of three 
vertices defines an individual triangle. The triangle may seem to be the most simple of the 
polygons, but as we noted earlier, it is probably the most important because no matter how you use 
it, and no matter what points form its vertices, it always lies in a plane. Because of this, most 
polygon-based modeling really comes down to triangle-based modeling in the end. So treat this 
humblest of polygons well.

Sequence of triangles

Triangles are the foundation of most truly useful polygon-based graphics, and they have some very 
useful capabilities. OpenGL provides two different techniques to assemble sequences of triangles 
into your image: triangle strips and triangle fans. Each has its own mode for \texttt{glBegin}:
GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN respectively. The behavior of each is shown in Figure 2.2 below.

![triangle strip and triangle fan](image)

Figure 2.2: triangle strip and triangle fan

To create a triangle strip, the first three vertices in the vertex list create the first triangle, and each vertex after that creates a new triangle with the two vertices immediately before it. To create a triangle fan, the first three vertices create the first triangle and each vertex after that creates a new triangle with the point immediately before it and the first point in the list.

Quadrilateral

A quadrilateral (actually a “quad”, because a general quadrilateral need not be convex) is any convex 4-sided figure, and to create a set of one or more distinct quadrilaterals you can use the GL_QUADS mode with `glBegin(...)` - `glEnd`. Each quadrilateral requires four vertices in the vertex list, so the first four vertices define the first quadrilateral, the next four the second quadrilateral, and so on. The sequence of vertices is that of the points in the quadrilateral. In an example below, we use six quadrilaterals to define a cube that will be used very often (with modifications and expansion as needed) in later examples.

Sequence of quadrilaterals

To create a sequence of quadrilaterals, the mode for `glBegin(...)` is GL_QUAD_STRIP. This specifies that the vertices in the vertex list are to be taken as vertices of a sequence of quadrilaterals that share some common sides — the first four vertices define the first quadrilateral and the last two of these, together with the next two, define the next vertex. However, the order in which the vertices are presented is different from that in the GL_QUADS mode, as noted in Figure 2.3.

![sequence of points in a quad strip](image)

Figure 2.3: sequence of points in a quad strip

Note the order of the vertices; instead of the expected sequence around the quads, the points in each pair have the same order. Thus the sequence 3-4 is the opposite order than would be expected, and this same sequence goes on in each additional pair of extra points. This difference is critical to note when you are implementing quad strip constructions. It might be helpful to think of this in terms of triangles, because a quad strip acts as though it were really a triangle strip.
Some images need to include more general kinds of polygons. While these can be created by constructing them manually as collections of triangles and/or quads, it might be easier to define and display a single polygon. The GL_POLYGON mode for glBegin allows you to define and display a single convex polygon. The vertices in the vertex list are taken as the vertices of the polygon in sequence order. As we noted in the early mathematical discussions, OpenGL can only handle convex polygons — polygons for which any two points in the polygon also have the entire line segment between them in the polygon. We refer you to that earlier discussion for more details.

Data structures to hold objects

There are many ways to hold the information that describes a graphic object. One of the simplest is the triangle list — an array of triples, and each set of three triples represents a separate triangle. Drawing the object is then a simple matter of reading three triples from the list and drawing the triangle.

A more effective, though a bit more complex, approach is to create two lists. The first is a vertex list, and it is simply an array of triples that contains all the vertices that would appear in the object. If the object is a polyhedron, the second is a face list, containing information on each of the faces in the polyhedron. Each face is indicated by listing the indices of all the vertices of the face, in the order needed by the orientation of the face. You can then draw the face by using the indices as an indirect reference to the actual vertices. This is the basic approach taken by the discussion of the classic cube below; there is a vertex array, and the actual drawing takes place by referring to the vertices by their indices, not their values. (We should point out quickly, however, that the code in that example does not use actual face lists but rather is organized rather informally.) It is also possible to include one more step and include an edge list, a list of all the edges of the object, each stored as a pair of indices into the vertex list.

Neither of these approaches takes into account the very significant savings in memory you can get by using triangle strips, triangle fans, or quad strips as discussed below. These approaches not only save space, but they are more effective for the graphics system as well because they allow the system to retain some of the information it generates in rendering one triangle or quad when it goes to generate the next one. However, it is difficult to have a model stored in a file contain all the information you would need to set up these compressed forms, so we will not pursue these further here.

Additional sources of graphic objects

Interesting and complex graphic objects can be difficult to create, because it can take a lot of work to measure or calculate the detailed coordinates of each vertex needed. There are more automatic techniques being developed, including 3D scanning techniques and detailed laser rangefinding to measure careful distances and angles to points on an object that is being measured, but they are out of the reach of most college classrooms. So what do we do to get interesting objects? There are four approaches.

The first way to get models is to buy them: to go is to the commercial providers of 3D models. There is a serious market for some kinds of models, such as medical models of human structures, from the medical and legal worlds. This can be expensive, but it avoids having to develop the expertise to do professional modeling and then putting in the time to create the actual models. If you are interested, an excellent source is viewpoint.com; they can be found on the Web.
A second way to get models is to find them in places where people make them available to the public. If you have friends in some area of graphics, you can ask them about any models they know of. If you are interested in molecular models, the protein data bank (URL http://www.pdb.bnl.gov) has a wide range of structure models available at no charge. If you want models of “stuff”, try avalon.viewpoint.com; this site contains a large number of public-domain models contributed to the community by friendly people over the years.

A third way to get models is to digitize them yourself with appropriate kinds of digitizing devices. There are a number of these available with their accuracy often depending on their cost, so if you need to digitize some physical objects you can compare the cost and accuracy of a number of possible kinds of equipment.

A fourth way to get models is to create them yourself. There are a number of tools that support high-quality interactive 3D modeling, and it is no shame to create your models with such tools. It is also possible to create interesting models analytically, using mathematical approaches to generate the vertices. This is perhaps slower than getting them from other sources, but you have final control over the form and quality of the model, so perhaps it might be worth the effort.

One problem may present itself if you find models from a public-domain source or from some kind of modeling software. These models are usually stored in a file format that is specific to the tools used to create them, and it may take some work to write filters that will read these formats into the kind of data structures you want for your program. Perhaps things that are “free” might cost more than things you buy if you can save the work of the conversion — but that’s up to you to decide.

Additional objects

Most OpenGL systems also include the OpenGL Utility Library (GLU), and you can take advantage of this to use additional primitive objects in your images. Among these are general quadric objects: objects defined by quadric equations (polynomial equations in three variables with degree no higher than two in any term). Spheres (gluSphere), cylinders (gluCylinder), and disks (gluDisk) are pre-defined in the GLU system. There are a number of additional details involved in using GLU objects that are more complex than we believe belong in a “simple modeling” module, and we refer you to your system’s GLU manual for these details.

A word to the wise...

As we said above, modeling can be the most time-consuming part of creating an image, but you simply aren’t going to create a useful or interesting image unless the modeling is done carefully and well. If you are concerned about the programming part of the modeling for your image, it might be best to create a simple version of your model and get the programming (or other parts that we haven’t talked about yet) done for that simple version. Once you are satisfied that the programming works and that you have gotten the other parts right, you can replace the simple model — the one with just a few polygons in it — with the one that represents what you really want to present. And don’t forget that you can use all your programming tools to determine the vertices for your models, working between the glBegin (...) and glEnd () functions and setting vertices with the glVertex*( ...) functions.

Code examples

The examples below illustrate how to define objects using the glBegin (...) - glEnd delimiters and vertex lists defined by various kinds of glVertex*** (...) functions. These examples are anything but exhaustive of all the possible combinations, but they should be good starting points.
In addition, this module is accompanied by a set of short example programs that use the kinds of object definitions below to show you how the modeling is used in context.

You will probably be surprised to see how primitive the images are that the example programs create. This is because these simple programs do not include any kind of lighting information. For better images, look at the examples of these same kinds of modeling that use lighting.

A set of points

While it would be interesting to create a 3D space with points scattered in it as a model of a starfield, it’s probably more useful to define an orderly set of points that is easier to see in a sample program. We will define a set of points that lie on the line $x = y = -z$ using a simple for loop as kind of a third-grade graphing exercise. These points are initially defined by specifying the point coordinates directly.

```c
void pointAt(int i)
{
    glVertex3f(0.2*(float)(i-5),0.2*(float)(i-5),0.2*(float)(5-i));
}

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

As we noted above, however, we can take advantage of the computational capabilities of ordinary programming to define our models, so it is absolutely not necessary to hand-calculate points when we can determine them by an algorithmic approach. So this example can be re-phrased as follows. and this is the version in the sample program with this module.

```c
void pointAt(int i)
{
    glVertex3f(0.2*(float)(i-5),0.2*(float)(i-5),0.2*(float)(5-i));
}

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

A line loop

Parametric curves in 3-space are often interesting objects of study. The `glBegin(...)` ... `glEnd()` pair and the vertex list 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.

```c
void pointAt(int i)
{
    glVertex3f(0.2*(float)(i-5),0.2*(float)(i-5),0.2*(float)(5-i));
}

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

```c
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.

```c
#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();
```

The example program that draws this spiral includes more than just drawing the spiral; it also includes some simple keyboard-driven rotations that let you see the spiral from many points in 3-space. Among the things you can see are the simple sine and cosine curves, as well as one period of the generic shifted sine curve.

A single polygon with many sides

Probably the simplest kind of 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, again using trigonometric functions to determine the vertices.

```c
#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 XY-plane; all the Z-values are zero. This polygon is also in the default color (white) for simple models. 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. We will see this later in the module on transformations.

Two objects made of triangles

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 approximation to a cone that can be used for such applications as a pointer in 3-space. The second is a triangle strip, often used to define surfaces when the surface has the kind of curvature that keeps rectangles from being planar. Triangle strips are a better basis for curved surfaces that need to show their surface properties when lighted.

The triangle fan is defined in a canonical way, with a circular base of radius 1.0 in the XZ-plane centered at the origin and a vertex at point (0.0,2.0,0.0) so the cone points upwards in space. When the cone is actually used, it will be transformed to have the size, orientation, and location you need.
glBegin(GL_TRIANGLE_FAN);
glVertex3f(0,2.0,0);
for (i=0; i < numStrips-1;i++) {
    angle = 2 * i * PI / numStrips;
    glVertex3f(cos(angle),0.0,sin(angle));
}       
glVertex3f(cos(0.0),0.0,sin(0.0));
glEnd();

The triangle strip example is probably easier to see if you refer first to the quadrilaterals section. There we show how to define a mathematical surface as a set of quads. But for many such surfaces, quads are not planar and so do not show the surface properties correctly. Instead of taking the quad approach, the glBegin(GL_QUADS) ... glEnd() section could be replaced by a small quad strip:

```
  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();
```

Note that the sequence of points is slightly different here than it is in the example below; we have a triangle with the points with indices \((i,j), (i+1,j), (i,j+1)\) and another with the points with indices \((i+1,j), (i,j+1), (i+1,j+1)\). Thus instead of one quad, we will have two triangles — and if you are clever enough to use quad strips instead of simple quads to re-do the mathematical surface below, it is simple to make the change noted here and do the surface with triangle strips.

**An object made from a sequence of quadrilaterals**

Quadrilaterals (usually called *quads* in OpenGL) are sometimes a more natural basis to build models on than are triangles. In one of the example programs for this module we create long, narrow tubes with rectangular cross-section as the basis for drawing a 3-D coordinate system. The quad strip defined below creates the tube for the Z-axis, and the other two axes are created by using transformations with this tube.

```c
#define RAD 0.03
  glBegin(GL_QUAD_STRIP);
  glVertex3f( RAD, RAD,  3.0 ); // start of first side
  glVertex3f( RAD, RAD, -3.0 );
  glVertex3f(-RAD, RAD,  3.0 );
  glVertex3f(-RAD, RAD, -3.0 );
  glVertex3f(-RAD,-RAD,  3.0 ); // start of second side
  glVertex3f(-RAD,-RAD, -3.0 );
  glVertex3f( RAD,-RAD,  3.0 ); // start of third side
  glVertex3f( RAD,-RAD, -3.0 );
  glVertex3f( RAD, RAD,  3.0 ); // start of fourth side
  glVertex3f( RAD, RAD, -3.0 );
  glEnd();
```

Another example of an object based on quadrilaterals would be a parametric surface, a surface defined by two parameters. In the same way a single parameter defined a curve in the spiral example above, two parameters can define a surface that is approximated by a set of of quads. The
code below defines such an approach to a mathematical surface defined by an equation in two variables. The student is encouraged to try this same kind of surface based on quad strips.

```c
// This example computes Y as a function of X and Z, but the principle
// of parametric surfaces is supported by viewing the surface as points
// (X,Y,Z) with two parameters, i and j
#define XSIZE 125  // larger than this for best results
#define ZSIZE XSIZE
#define MINX -3.0  // define the range of each parameter
#define MAXX 3.0
#define MINZ -3.0
#define MAXZ 3.0
// functions return X and Z values for parameters i and j respectively
GLfloat XX(int i) {
  return (MINX+((MAXX-MINX)/(float)(XSIZE-1))*(float)(i));
}
GLfloat ZZ(int j) {
  return (MINZ+((MAXZ-MINZ)/(float)(ZSIZE-1))*(float)(j));
}
// function defines the parametric surface based on the function
// cos(x*x+z*z+t). This is actually a family of surfaces parametrized
// by the value of t, which needs to be defined before this is actually
// drawn. Substituting other functions is encouraged!
void surface(void) {
  point3 vec1, vec2, triNormal;
  int  i, j;
  float x, z;

  #define EPSILON .001;
  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);
    }
  // define the surface */
  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 this surface is not 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 surface is re-visited when we discuss lighting and color.

The cube we will use in many later examples

5/29/00
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 show the vertex declarations (we use an array of arrays; it may be new to you but it’s a very effective way to define an object) and show how the cube is specified. With this option, we use the glVertex3fv(…) vertex specification function. In an example program we will show how this looks without any lighting specifications, but it will need lights and lighting in order to really look like a cube.

```c
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) };

glBegin(GL_QUADS);
// glNormal3fv(normals[0]); // normals will be used for later work
  glVertex3fv(vertices[1]); // first quad: positive Z face
  glVertex3fv(vertices[5]);
  glVertex3fv(vertices[7]);
  glVertex3fv(vertices[3]);

  // glNormal3fv(normals[1]);
  glVertex3fv(vertices[7]); // second quad: positive Y face
  glVertex3fv(vertices[6]);
  glVertex3fv(vertices[2]);
  glVertex3fv(vertices[3]);

  // glNormal3fv(normals[2]);
  glVertex3fv(vertices[2]); // third quad: negative Z face
  glVertex3fv(vertices[6]);
  glVertex3fv(vertices[4]);
  glVertex3fv(vertices[0]);

  // glNormal3fv(normals[3]);
  glVertex3fv(vertices[5]); // fourth quad: positive X face
  glVertex3fv(vertices[4]);
  glVertex3fv(vertices[6]);
  glVertex3fv(vertices[7]);

  // glNormal3fv(normals[4]);
  glVertex3fv(vertices[4]); // fifth quad: negative Y face
  glVertex3fv(vertices[5]);
  glVertex3fv(vertices[1]);
  glVertex3fv(vertices[0]);

  // glNormal3fv(normals[1]);
  glVertex3fv(vertices[0]); // sixth quad: negative X face
  glVertex3fv(vertices[1]);
  glVertex3fv(vertices[3]);
  glVertex3fv(vertices[2]);
  glEnd();
```

This approach specifically identifies all the vertices of each face of the cube, which is the way a simple modeling approach would work. But as we noted earlier, the cube is a regular polyhedron with four edges and 90 degree interior angles for each face, and it is possible to define the cube by
defining a standard square and then using transformations to create the faces from this master square. We will see this approach later on when we work with transformations.

Source codes

The source codes for these examples may be found in the files

- **points.c**: code that includes the drawing of a set of 10 points on a line in 3-space
- **poly.c**: code that includes the drawing of a simple regular N-gon in the XZ-plane
- **spiral.c**: code that includes the drawing of a spiral as 100 connected line segments

These examples also include other features that make it easier to see the nature of the object that is being modeled and displayed. The main additions are a set of coordinate axes to give you an orientation in the space (Z-axis towards you, Y-axis up, X-axis to the right) and a user-controlled set of rotations based on the Q/W, A/S, and Z/X keys that will allow you to move the object (and its space) around so you can see the objects from all sides.
Modeling with GLU and GLUT

Prerequisites
An understanding of the simple modeling with polygon primitives.

Introduction
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 module that GLUT includes window management functions, and both GLU and GLUT include a number of built-in graphical elements that you can use. This module describes a number of these elements.

Definitions
The primitives that GLU provides to you are defined with some parameters that define the primitive and some that define the resolution with which the primitive is modeled. The former are element-specific and will be described when we describe each primitive, but the latter are general and are described here.

Each GLU primitive is declared as a GLUquadric and is allocated with the function

\[
\text{GLUquadric* gluNewQuadric( void )}
\]

Each is a surface of revolution defined with the z-axis as the rotation axis. Each is modeled in terms of subdivisions around the z-axis, called slices, and subdivisions along the z-axis, called stacks. Figure 2.4 shows an example of a typical pre-built quadric, a GLUT wireframe sphere, modeled with a small number of slices and stacks so you can see the basis of this definition.

![Figure 2.4: A GLUT wireframe sphere with 10 slices and 10 stacks](image)

Below we describe the GLU primitives by listing the function prototype for each; 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 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 isthe 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 primitives are more complex than the GLU primitives, including both solid and wireframe versions of many interesting solids. They include a cone, a cube, a dodecahedron (12-sided regular polyhedron), an icosahedron (20-sided regular polyhedron), an octahedron (8-sided regular polyhedron), a sphere, a teapot (the canonical Utah teapot, an icon of computer graphics), a tetrahedron (4-sided regular polyhedron), and a torus (doughnut). Each has a canonical position and orientation, typically centered at the origin and within a standard volume and, if it has an axis, the axis is aligned with the z-axis. As with the GLU standard primitives, some of the GLUT primitives allow you to specify the granularity of the primitive’s modeling, but some do not; the example program will show you the differences. If you have GLUT with your OpenGL, you should check the GLUT manuals for the details.

Some examples

Our example for this module is quite simple, initially displaying a medium-quality GLU sphere and using a menu to let you select another GLU or GLUT primitive, to zoom in or out on that primitive, and to increase or decrease the quality of the modeling when that is possible. Menus are provided by the GLUT toolkit and will be discussed later in the module on event-driven programming, but the example is much more interesting if we allow you to see most of the primitives in one function.

The heart of the example is the following code that responds to a menu selection to change the primitive you are displaying, change the resolution on your primitive, or to zoom in or zoom out on your primitive. This code depends on the menu, so the fundamentals of the menu definition are also shown, though this is getting somewhat ahead of ourselves.
// menu functions in main()
    glutCreateMenu(options_menu);
    glutAddMenuEntry("GLU sphere", 1);
    glutAddMenuEntry("GLU cylinder", 2);
    glutAddMenuEntry("GLUT dodecahedron", 3);
    glutAddMenuEntry("GLUT torus", 4);
    glutAddMenuEntry("GLUT teapot", 5);
    glutAddMenuEntry("More detail", 6);
    glutAddMenuEntry("Less detail", 7);
    glutAddMenuEntry("Move in", 8);
    glutAddMenuEntry("Move out", 9);
    glutAttachMenuName(GLUT_RIGHT_BUTTON, "Selections");

    // function to be called to handle mouse inputs
    void options_menu(int input)
    {
        if ((input >= 1) && (input <=5 ))
        {
            selectedObject = input;
        }
        else
        {
            if ( input == 6 ) resolution *= 2;  // higher resolution
            if ( input == 7 )                   // lower resolution
                if (resolution > 2 ) resolution = (resolution+1)/2;
            if ( input == 8 ) distance -= 1.0;  // move in
            if ( input == 9 ) distance += 1.0;  // move out
        }
        glutPostRedisplay();
    }

    // function to manage the display that responds to the menu options
    void display( void )
    {
        GLUquadric *myQuad;
        GLdouble radius = 1.0;
        GLint slices, stacks;
        GLint nsides, rings;

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(60.0,1.0,1.0,30.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        //                eye point        center of view    up
        gluLookAt(distance,distance,distance,0.0,0.0,0.0,0.0,1.0,0.0);

        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;
            }
        }
    }
```c
    case (3): {
        glutSolidDodecahedron(); break;
    }
    case (4): {
        nsides = rings = resolution;
        glutSolidTorus(1.0, 2.0, nsides, rings);
        break;
    }
    case (5): {
        glutSolidTeapot(2.0); break;
    }
    glutSwapBuffers();
```

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.

Code examples

* graphicsPrimitivesMenu.c