Modeling

Prerequisites

This chapter requires an understanding of simple 3-dimensional geometry, knowledge of how to represent points in 3-space, enough programming experience to be comfortable writing code that calls API functions to do required tasks, ability to design a program in terms of simple data structures such as stacks, and an ability to organize things in 3D space.

Introduction

Modeling is the process of defining the geometry that makes up a scene and implementing that definition with the tools of your graphics API. This chapter is critical in developing your ability to create graphical images and takes us from quite simple modeling to fairly complex modeling based on hierarchical structures, and discusses how to implement each of these different stages of modeling in OpenGL. It is fairly comprehensive for the kinds of modeling one would want to do with a basic graphics API, but there are other kinds of modeling used in advanced API work and some areas of computer graphics that involve more sophisticated kinds of constructions than we include here, so we cannot call this a genuinely comprehensive discussion. It is, however, a good enough introduction to give you the tools to start creating interesting images.

The chapter has four distinct parts because there are four distinct levels of modeling that you can use to create images. We begin with simple geometric modeling: modeling where you define the coordinates of each vertex of each component you will use at the point where that component will reside in the final scene. This is straightforward but can be very time-consuming to do for complex scenes, so we will also discuss importing models from various kinds of modeling tools that can allow you to create parts of a scene more easily.

The second section describes the next step in modeling. Here we extend the utility of your simple modeling by defining the primitive transformations you can use for computer graphics operations and by discussing how you can start with simple modeling and use transformations to create more general model components in your scene. This is a very important part of the modeling process because it allows you to build standard templates for many different graphic objects and then place them in your scene with the appropriate transformations. These transformations are also critical to the ability to define and implement motion in your scenes because it is typical to move objects, lights, and the eyepoint with transformations that are controlled by parameters that change with time. This can allow you to extend your modeling to define animations that can represent such concepts as changes over time.

In the third section of the chapter we introduce the concept of the scene graph, a modeling tool that gives you a unified approach to defining all the objects and transformations that are to make up a scene and to specifying how they are related and presented. We then describe how you work from the scene graph to write the code that implements your model. This concept is new to the introductory graphics course but has been used in some more advanced graphics tools, and we believe you will find it to make the modeling process much more straightforward for anything beyond a very simple scene. In the second level of modeling discussed in this section, we introduce hierarchical modeling in which objects are designed by assembling other objects to make more complex structures. These structures can allow you to simulate actual physical assemblies and develop models of structures like physical machines. Here we develop the basic ideas of scene graphs introduced earlier to get a structure that allows individual components to move relative to each other in ways that would be difficult to define from first principles.

Finally, the fourth section of the chapter covers the implementation of modeling in the OpenGL API. This includes the set of operations that implement polygons, as well as those that provide the
geometry compression that we describe in the first section. This section also describes the use of OpenGL’s pre-defined geometric components that you can use directly in your images to let you use more complex objects without having to determine all the vertices directly, but that are defined only in standard positions so you must use transformations to place them correctly in your scenes. It also includes a discussion of transformations and how they are used in OpenGL, and describes how to implement a scene graph with this API.

Simple Geometric Modeling

Introduction

The subject of computer graphics deals with geometry and its representation in ways that allow it to be manipulated and displayed by a computer. Because these notes are intended for a first course in the subject, the geometry will be simple and familiar representations of 3-dimensional space, although there has been excellent work in the field that uses sophisticated and unfamiliar ways to deal with geometry, and there has been work that goes well beyond the familiar 3-space. When you work with a graphics API, however, you must first model the image in a way that is understood by the graphics API you are using. For most APIs, this means using only a few simple graphics primitives, such as points, line segments, and polygons.

The space we will use for our modeling is simple Euclidean 3-space with standard coordinates, which we will call the X-, Y-, and Z-coordinates. Figure 2.1 below illustrates a point, a line segment, a polygon, and a polyhedron—the basic elements of the computer graphics world that you will use for most of your graphics. In this space a point is simply a single location in 3-space, specified by its coordinates and often seen as a triple of real numbers such as \((px, py, pz)\). A point is drawn on the screen by lighting a single pixel at the screen location that best represents the location of that point in space. To draw the point you will specify that you want to draw points and specify the point’s coordinates, usually in 3-space, and the graphics API will calculate the coordinates of the point on the screen that best represents that point and will light that pixel. Note that a point is usually presented as a square, not a dot, as indicated in the figure. A line segment is determined by its two specified endpoints, so to draw the line you indicate that you want to draw lines and define the points that are the two endpoints. Again, these endpoints are specified in 3-space and the graphics API calculates their representations on the screen, and draws the line segment between them. A polygon is a region of space that lies in a plane and is bounded in the plane by a collection of line segments. It is determined by a sequence of points (called the vertices of the polygon) that specify a set of line segments that form its boundary, so to draw the polygon you indicate that you want to draw polygons and specify the sequence of vertex points. A polyhedron is a region of 3-space bounded by polygons, called the faces of the polyhedron. A polyhedron is defined by specifying a sequence of faces, each of which is a polygon. 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. As we will see when we discuss lighting and shading in subsequent chapters, the direction in which we go around the vertices of each face of a polygon is
very important, and whenever you design a polyhedron, you should plan your polygons so that their vertices are ordered in a sequence that is counterclockwise as seen from outside the polyhedron (or, to put it another way, that the angle to each vertex as seen from a point inside the face is increasing rather than decreasing as you go around each face).

Before you can create an image, you must define the objects that are to appear in that image through some kind of modeling process. Perhaps the most difficult—or at least the most time-consuming—part of beginning graphics programming is creating the models that are part of the image you want to create. Part of the difficulty is in designing the objects themselves, which may require you to sketch parts of your image by hand 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. Another part of the difficulty is actually entering the data for the points in an appropriate kind of data structure and writing the code that will interpret this data as points, line segments, and polygons for the model. But until you get the points and their relationships right, you will not be able to get the image right.

Definitions

We need to have some common terminology as we talk about modeling. We will think of modeling as the process of 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. However, for much graphics programming, and certainly as you are beginning to learn about this field, you will probably want to do your modeling by defining your geometry in terms of relatively simple primitive terms so you may be fully in control of the modeling process.

Besides defining a single point, line segment, or polygon, graphics APIs provide modeling support 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 involve connected sets of points, such as line segments, quad strips, triangle strips, or triangle fans. This allows you to assemble simpler components into more complex groupings and is often the only way you can define polyhedra for your scene. Some of these modeling techniques involve a concept called geometry compression, which allow you to define a geometric object using fewer vertices than would normally be needed. The OpenGL support for geometry compression will be discussed as part of the general discussion of OpenGL modeling processes. The discussions and examples below will show you how to build your repertoire of techniques you can use for your modeling.

Before going forward, however, we need to mention another way to specify points for your models. In some cases, it can be helpful to think of your 3-dimensional space as embedded as an affine subspace of 4-dimensional space. If we think of 4-dimensional space as having X, Y, Z, and W components, this embedding identifies the three-dimensional space with the subspace \( \mathbb{R}^3 \) of the four-dimensional space, so the point \((x, y, z)\) is identified with the four-dimensional point \((x, y, z, 1)\). Conversely, the four-dimensional point \((x, y, z, w)\) is identified with the three-dimensional point \((x/w, y/w, z/w)\) whenever \(w \neq 0\). The four-dimensional representation of points with a non-zero \(w\) component is called homogeneous coordinates, and calculating the three-dimensional equivalent for a homogeneous representation by dividing by \(w\) is called homogenizing the point. The four-dimensional point \((x, y, z, 0)\) is not identified with a point in 3-space because it cannot be homogenized, but it is identified with a direction, or a “point at infinity.” This has an application in the chapter below on lighting when we discuss directional instead of positional lights, but in general we will not encounter homogeneous coordinates often in these notes.
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 a set of points and will then show the function call that draws the object we have defined.

Point and points

To draw a single point, we will simply define the coordinates of the point and give them to the graphics API function that draws points. Such a function can typically handle one point or a number of points, so if we want to draw only one point, we provide only one vertex; if we want to draw more points, we provide more vertices. Points are extremely fast to draw, and it is not unreasonable to draw tens of thousands of points if a problem merits that kind of modeling. On a very modest-speed machine without any significant graphics acceleration, a 50,000 point model can be re-drawn in a small fraction of a second.

Line segments

To draw a single line segment, we must simply supply two vertices to the graphics API function that draws lines. Again, this function will probably allow you to specify a number of line segments and will draw them all; for each segment you simply need to provide the two endpoints of the segment. Thus you will need to specify twice as many vertices as the number of line segments you wish to produce.

Connected lines

Connected lines—line segments that are joined “head to tail” to form a longer connected group—are shown in Figure 2.2. These are often called line strips, and your graphics API will probably provide a function for drawing them. The vertex list you use will define 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 the number of line segments drawn by the function will be one fewer than the number of vertices in the vertex list. This is a geometry compression technique because to define a line strip with N segments you only specify N+1 vertices instead of 2N vertices; instead of needing to define two points per line segment, each segment after the first only needs one vertex to be defined.

Triangle

To draw one or more unconnected triangles, your graphics API will provide a simple triangle-drawing function. With this function, each set of three vertices will define an individual triangle so that the number of triangles defined by a vertex list is one third the number of vertices in the list. The humble 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, and almost every kind of graphics tool knows how to manage objects defined by triangles. So treat this humblest of polygons well and learn how to think about polygons and polyhedra in terms of the triangles that make them up.
Sequence of triangles

Triangles are the foundation of most truly useful polygon-based graphics, and they have some very useful capabilities. Graphics APIs often provide two different geometry-compression techniques to assemble sequences of triangles into your image: triangle strips and triangle fans. These techniques can be very helpful if you are defining a large graphic object in terms of the triangles that make up its boundaries, when you can often find ways to include large parts of the object in triangle strips and/or fans. The behavior of each is shown in Figure 2.3 below.

![triangle strip and triangle fan](image)

Figure 2.3: triangle strip and triangle fan

Most graphics APIs support both techniques by interpreting the vertex list in different ways. 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. We will see in later chapters that the order of points around a polygon is important, and we must point out that these two techniques behave quite differently with respect to polygon order; for triangle fans, the orientation of all the triangles is the same (clockwise or counterclockwise), while for triangle strips, the orientation of alternate triangles is reversed. This may require some careful coding when lighting models are used. 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. In each case, the number of triangles defined by the vertex list is two less than the number of vertices in the list, so these are very efficient ways to specify triangles.

Quadrilateral

A convex quadrilateral, often called a “quad” to distinguish it from a general quadrilateral because the general quadrilateral need not be convex, is any convex 4-sided figure. The function in your graphics API that draws quads will probably allow you to draw a number of them. 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, so your vertex list will have four times as many points as there are quads. The sequence of vertices is that of the points as you go around the perimeter of the quadrilateral. In an example later in this chapter, we will use six quadrilaterals to define a cube that will be used in later examples.

Sequence of quads

You can frequently find large objects that contain a number of connected quads. Most graphics APIs have functions that allow you to define a sequence of quads. The vertices in the vertex list are taken as vertices of a sequence of quads that share common sides. For example, the first four vertices can define the first quad; the last two of these, together with the next two, define the next quad; and so on. The order in which the vertices are presented is shown in Figure 2.4. 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 its vertices were specified as if it were really a triangle strip — vertices 1/2/3 followed by 2/3/4 followed by 3/4/5 etc.

Figure 2.4: sequence of points in a quad strip

A good example of the use of quad strips and triangle fans would be creating your own model of a sphere. As we will see later in this chapter, there are pre-built sphere models from both the GLU and GLUT toolkits in OpenGL, but the sphere is a familiar object and it can be helpful to see how to create familiar things with new tools. There may also be times when you need to do things with a sphere that are difficult with the pre-built objects, so it is useful to have this example in your “bag of tricks.”

Recall the discussion of spherical coordinates in the discussion of mathematical fundamentals. We can use spherical coordinates to model our object at first, and then convert to Cartesian coordinates to present the model to the graphics system for actual drawing. Let’s think of creating a model of the sphere with N divisions around the equator and N/2 divisions along the prime meridian. In each case, then, the angular division will be theta = 360/N degrees. Let’s also think of the sphere as having a unit radius, so it will be easier to work with later when we have transformations. Then the basic structure would be:

```c
// create the two polar caps with triangle fans
doTriangleFan() // north pole
    set vertex at (1, 0, 90)
    for i = 0 to N
        set vertex at (1, 360/i, 90-180/N)
    endTriangleFan()
doTriangleFan() // south pole
    set vertex at (1, 0, -90)
    for i = 0 to N
        set vertex at (1, 360/i, -90+180/N)
    endTriangleFan()

// create the body of the sphere with quad strips
for j = -90+180/N to 90 - 180/2N
    // one quad strip per band around the sphere at a given latitude
    doQuadStrip()
        for i = 0 to 360
            set vertex at (1, i, j)
            set vertex at (1, i, j+180/N)
            set vertex at (1, i+360/N, j)
            set vertex at (1, i+360/N, j+180/N)
        endQuadStrip()
```

Note the order in which we set the points in the triangle fans and in the quad strips, as we noted when we introduced these concepts; this is not immediately an obvious order and you may want to think about it a bit. Because we’re working with a sphere, the quad strips as we have defined them are planar, so there is no need to divide each quad into two triangles to get planar surfaces.
General polygon

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. A graphics API will allow you to define and display a single polygon by specifying its vertices, and the vertices in the vertex list are taken as the vertices of the polygon in sequence order. As we noted in the earlier chapter on mathematical fundamentals, many APIs 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 good example of this kind of list is the STL graphics file format discussed in the chapter below on graphics hardcopy.

A more effective, though a bit more complex, approach is to create three 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 polygon or contains polygons, the second list is an edge list that contains an entry for each edge of the polygon; the entry is an ordered pair of numbers, each of which is an index of a point in the vertex list. If the object is a polyhedron, the third 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 edges that make up 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. So to draw the object, you loop across the face list to draw each face; for each face you loop across the edge list to determine each edge, and for each edge you get the vertices that determine the actual geometry.

As an example, let’s consider the classic cube, centered at the origin and with each side of length two. For the cube let’s define the vertex array, edge array, and face array that define the cube, and let’s outline how we could organize the actual drawing of the cube. We will return to this example later in this chapter and from time to time as we discuss other examples throughout the notes.

We begin by defining the data and data types for the cube. The vertices are points, which are arrays of three points, while the edges are pairs of indices of points in the point list and the faces are quadruples of indices of faces in the face list. In C, these would be given as follows:

typedef float point3[3];
typedef int   edge[2];
typedef int   face[4];   // assumes a face has four edges for this example

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}  };

edge   edges[24]   = { { 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                      { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                      ..., // more entries
                      { 7, 6 }, { 6, 4 }, { 4, 0 }, { 0, 7 }  };
Notice that in our edge list, each edge is actually listed twice—once for each direction the in which the edge can be drawn. We need this distinction to allow us to be sure our faces are oriented properly, as we will describe in the discussion on lighting and shading in later chapters. For now, we simply ensure that each face is drawn with edges in a counterclockwise direction as seen from outside that face of the cube. Drawing the cube, then, proceeds by working our way through the face list and determining the actual points that make up the cube so they may be sent to the generic (and fictitious) \texttt{setVertex(...)} function. In a real application we would have to work with the details of a graphics API, but here we sketch how this would work in a pseudocode approach. In this pseudocode, we assume that there is no automatic closure of the edges of a polygon so we must list both the vertex at both the beginning and the end of the face when we define the face; if this is not neede by your API, then you may omit the first \texttt{setVertex} call in the pseudocode for the function \texttt{cube()} below.

```c
void cube(void) {
  for faces 1 to 6
    start face
      setVertex(vertices[edges[cube[face][0]][0]]);
    for each edge in the face
      setVertex(vertices[edges[cube[face][edge]][1]]);
    end face
}
```

Neither the simple triangle list nor the more complex structure of vertex, edge, and face lists takes into account the very significant savings in memory you can get by using geometry compression techniques. There are a number of these techniques, but we will only discuss line strips, triangle strips, triangle fans, and quad strips because these are more often supported by a graphics API. 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, when a model is stored in a file, it is difficult to have the model contain all the information you would need to set up these compressed forms, so we will not pursue these compression techniques 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 (with URL http://www.pdb.bnl.gov) has a wide range of structure models available at no charge. If you want models of all kinds of different things, try the site avalon.viewpoint.com; this contains a large number of public-domain models contributed to the community by generous 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. The digitizing equipment will probably come with tools that capture the points and store the geometry in a standard format, which may or may not be easy to use for your particular graphics API. If it happens to be one that your API does not support, you may need to convert that format to one you use or to find a tool that does that conversion.

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. This has the same issue as digitizing models in terms of the format of the file that the tools produce, but a good tool should be able to save the models in several formats, one of which you could use fairly easily with your graphics API. 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. This will be discussed in the chapter on interpolation and spline modeling, for example.

If you get models from various sources, you will probably find that they come in a number of different kinds of data format. There are a large number of widely used formats for storing graphics information, and it sometimes seems as though every graphics tool uses a file format of its own. Some available tools will open models with many formats and allow you to save them in a different format, essentially serving as format converters as well as modeling tools. In any case, you are likely to end up needing to understand some model file formats and writing your own tools to read these formats and produce the kind of internal data that you need for your models, 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. An excellent resource on file formats is the Encyclopedia of Graphics File Formats, published by O’Reilly Associates, and we refer you to that book for details on particular formats.

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.
Transformations and Modeling

This section requires some understanding of 3D geometry, particularly a sense of how objects can be moved around in 3-space. You should also have some sense of how the general concept of stacks works.

Introduction

Transformations are probably the key point in creating significant images in any graphics system. It is extremely difficult to model everything in a scene in the place where it is to be placed, and it is even worse if you want to move things around in real time through animation and user control. Transformations let you define each object in a scene in any space that makes sense for that object, and then place it in the world space of a scene as the scene is actually viewed. Transformations can also allow you to place your eyepoint and move it around in the scene.

There are several kinds of transformations in computer graphics: projection transformations, viewing transformations, and modeling transformations. Your graphics API should support all of these, because all will be needed to create your images. Projection transformations are those that specify how your scene in 3-space is mapped to the 2D screen space, and are defined by the system when you choose perspective or orthogonal projections; viewing transformations are those that allow you to view your scene from any point in space, and are set up when you define your view environment, and modeling transformations are those you use to create the items in your scene and are set up as you define the position and relationships of those items. Together these make up the graphics pipeline that we discussed in the first chapter of these notes.

Among the modeling transformations, there are three fundamental kinds: rotations, translations, and scaling. These all maintain the basic geometry of any object to which they may be applied, and are fundamental tools to build more general models than you can create with only simple modeling techniques. Later in this chapter we will describe the relationship between objects in a scene and how you can build and maintain these relationships in your programs.

The real power of modeling transformation, though, does not come from using these simple transformations on their own. It comes from combining them to achieve complete control over your modeled objects. The individual simple transformations are combined into a composite modeling transformation that is applied to your geometry at any point where the geometry is specified. The modeling transformation can be saved at any state and later restored to that state to allow you to build up transformations that locate groups of objects consistently. As we go through the chapter we will see several examples of modeling through composite transformations.

Finally, the use of simple modeling and transformations together allows you to generate more complex graphical objects, but these objects can take significant time to display. You may want to store these objects in pre-compiled display lists that can execute much more quickly.

Definitions

In this section we outline the concept of a geometric transformation and describe the fundamental transformations used in computer graphics, and describe how these can be used to build very general graphical object models for your scenes.

Transformation

A transformation is a function that takes geometry and produces new geometry. The geometry can be anything a computer graphics systems works with — a projection, a view, or an object to be
displayed. We have already talked about projections and views, so in this section we will talk about projections as modeling tools. In this case, the transformation needs to preserve the geometry of the objects we apply them to, so the basic transformations we work with are those that maintain geometry, which are the three we mentioned earlier: rotations, translations, and scaling. Below we look at each of these transformations individually and together to see how we can use transformations to create the images we need.

Our vehicle for looking at transformations will be the creation and movement of a rugby ball. This ball is basically an ellipsoid (an object that is formed by rotating an ellipse around its major axis), so it is easy to create from a sphere using scaling. Because the ellipsoid is different along one axis from its shape on the other axes, it will also be easy to see its rotations, and of course it will be easy to see it move around with translations. So we will first discuss scaling and show how it is used to create the ball, then discuss rotation and show how the ball can be rotated around one of its short axes, then discuss translations and show how the ball can be moved to any location we wish, and finally will show how the transformations can work together to create a rotating, moving ball like we might see if the ball were kicked. The ball is shown with some simple lighting and shading as described in the chapters below on these topics.

Scaling changes the entire coordinate system in space by multiplying each of the coordinates of each point by a fixed value. Each time it is applied, this changes each dimension of everything in the space. A scaling transformation requires three values, each of which controls the amount by which one of the three coordinates is changed, and a graphics API function to apply a scaling transformation will take three real values as its parameters. Thus if we have a point \((x, y, z)\) and specify the three scaling values as \(S_x\), \(S_y\), and \(S_z\), then the point is changed to \((x*S_x, y*S_y, z*S_z)\) when the scaling transformation is applied. If we take a simple sphere that is centered at the origin and scale it by 2.0 in one direction (in our case, the y-coordinate or vertical direction), we get the rugby ball that is shown in Figure 2.4 next to the original sphere. It is important to note that this scaling operates on everything in the space, so if we happen to also have a unit sphere at position farther out along the axis, scaling will move the sphere farther away from the origin and will also multiply each of its coordinates by the scaling amount, possibly distorting its shape. This shows that it is most useful to apply scaling to an object defined at the origin so only the dimensions of the object will be changed.

![Figure 2.4: a sphere a scaled by 2.0 in the y-direction to make a rugby ball (left) and the same sphere is shown unscaled (right)](image-url)
**Rotation** takes everything in your space and changes each coordinate by rotating it around the origin of the geometry in which the object is defined. The rotation will always leave a line through the origin in the space fixed, that is, will not change the coordinates of any point on that line. To define a rotation transformation, you need to specify the amount of the rotation (in degrees or radians, as needed) and the line about which the rotation is done. A graphics API function to apply a rotation transformation, then, will take the angle and the line as its parameters; remember that a line through the origin can be specified by three real numbers that are the coordinates of the direction vector for that line. It is most useful to apply rotations to objects centered at the origin in order to change only the orientation with the transformation.

**Translation** takes everything in your space and changes each point’s coordinates by adding a fixed value to each coordinate. The effect is to move everything that is defined in the space by the same amount. To define a translation transformation, you need to specify the three values that are to be added to the three coordinates of each point. A graphics API function to apply a translation, then, will take these three values as its parameters. A translation shows a very consistent treatment of everything in the space, so a translation is usually applied after any scaling or rotation in order to take an object with the right size and right orientation and place it correctly in space.

Finally, we put these three kinds of transformations together to create a sequence of images of the rugby ball as it moves through space, rotating as it goes, shown in Figure 2.5. This sequence was created by first defining the rugby ball with a scaling transformation and a translation putting it on the ground appropriately, creating a composite transformation as discussed in the next section. Then rotation and translation values were computed for several times in the flight of the ball, allowing us to rotate the ball by slowly-increasing amounts and placing it as if it were in a standard gravity field. Each separate image was created with a set of transformations that can be generically described by:

```plaintext
translate( Tx, Ty, Tz )
rotate( angle, x-axis )
scale( 1., 2., 1. )
drawBall()
```
where the operation drawBall() was defined as

```plaintext
translate( Tx, Ty, Tz )
scale( 1., 2., 1. )
drawSphere()
```

Notice that the ball rotates in a slow counterclockwise direction as it travel from left to right. The position of the ball describes a parabola as it moves to the right, modeling the effect of gravity on the ball’s flight. This kind of composite transformation constructions is described in the next section, and as we note there, the order of these transformation calls is critical in order to achieve the effect we need.

**Composite transformations**

In order to achieve the image you want, you may need to apply more than one simple transformation to achieve what is called a composite transformation. For example, if you want to create a rectangular box with height $A$, width $B$, and depth $C$, with center at $(C_1, C_2, C_3)$, and oriented at an angle $A$ relative to the Z-axis, you could start with a cube one unit on a side and with center at the origin, and get the box you want by applying the following sequence of operations:

1. first, scale the cube to the right size to create the rectangular box with dimensions $A$, $B$, $C$,
2. second, rotate the cube by the amount $A$ to the right orientation, and
3. third, translate the cube to the position $C_1$, $C_2$, $C_3$.

This sequence is critical because of the way transformations work in the whole space as illustrated above. For example, if we rotated first and then scaled with different scale factors in each dimension, we would introduce distortions in the box. If we translated first and then rotated, the rotation would move the box to an entirely different place. Because the order is very important, we find that there are certain sequences of operations that give predictable, workable results, and the order above is the one that works best: apply scaling first, apply rotation second, and apply translation last.

The order of transformations is important in ways that go well beyond the translation and rotation example above. In general, transformations are an example of noncommutative operations, operations for which $f\circ g \neq g\circ f$ (that is, $f(g(x)) \neq g(f(x))$). Most students have little experience with noncommutative operations until you get to a linear algebra course, so this may be a new idea. But let’s look at the operations we described above: if we take the point $(1, 1, 0)$ and apply a rotation by 90° around the Z-axis, we get the point $(-1, 1, 0)$. If we then apply a translation by $(2, 0, 0)$ we get the point $(1, 1, 0)$ again. However, if we start with $(1, 1, 0)$ and first apply the translation, we get $(3, 1, 0)$ and if then apply the rotation, we get the point $(-1, 3, 0)$ which is certainly not the same as $(1, 1, 0)$. That is, using some pseudocode for rotations, translations, and point setting, the two code sequences

```plaintext
rotate(90, 0, 0, 1)  translate(2, 0, 0)
translate (2, 0, 0)  rotate(90, 0, 0, 1)
setPoint(1, 1, 0)  setPoint(1, 1, 0)
```

produce very different results; that is, the rotate and translate operations are not commutative.

This behavior is not limited to different kinds of transformations. Different sequences of rotations can result in different images as well. Again, if you consider the sequence of rotations

(sequenc ehere)

and the same rotations in a different sequence

(different sequence here)

then the results are quite different, as is shown in Figure 2.7 below.
Figure 2.7: the results from two different orderings of the same rotations

Mathematical notation can be applied in many ways, so your previous mathematical experience may not help you very much in deciding how you want to approach this problem. However, we want to define the sequence of transformations as last-specified, first-applied, or in another way of thinking about it, we want to apply our functions so that the function nearest to the geometry is applied first. Another way to think about this is in terms of building composite functions by multiplying the pieces, and in this case we want to compose each new function by multiplying it on the right of the previous functions. So the standard operation sequence we see above would be achieved by the following algebraic sequence of operations:

\[
\text{translate} \ast \text{rotate} \ast \text{scale} \ast \text{geometry} \quad \text{or, looking at function composition, as}
\]

\[
\text{translate(rotate(scale(geometry)))}
\]

which could be implemented by the following sequence of function calls that is not intended to represent any particular API:

```plaintext
translate(C1, C2, C3); // translate to the desired point
rotate(A, Z); // rotate by A around the Z-axis
scale(A, B, C); // scale by the desired amounts
cube(); // define the geometry of the cube
```

This sequence would seem to be exactly the opposite of the sequence noted above, but in fact we readily see that the scaling operation is the function closest to the geometry (which is expressed in the function `cube()` ) because of the last-specified, first-applied nature of transformations. In Figure 2.8 we see the sequence of operations as we proceed from the plain cube (at the left), to the scaled cube next, then to the scaled and rotated cube, and finally to the cube that uses all the transformations (at the right). The application is to create a long, thin, rectangular bar that is oriented at a 45° angle upwards and lies above the definition plane.

In general, the overall sequence of transformations that are applied to a model by considering the total sequence of transformations in the order in which they are specified, as well as the geometry on which they work:

\[
P \quad V \quad T_0 \quad T_1 \quad T_2 \quad \ldots \quad T_n \quad T_{n+1} \quad \ldots \quad T_{\text{last}} \quad \ldots \quad \text{geometry}
\]

Here, $P$ is the projection transformation, $V$ is the viewing transformation, and $T_0$, $T_1$, $\ldots$ $T_{\text{last}}$ are the transformations specified in the program to model the scene, in order ($T_1$ is first, $T_{\text{last}}$ is last). The projection transformation is defined in the `reshape` function; the viewing transformation is defined in the `init` function or at the beginning of the `display` function so it is defined at the beginning of the modeling process. But the sequence is actually applied in reverse: $T_{\text{last}}$ is actually applied first, and $V$ and finally $P$ are applied last. The code would then have the definition of $P$ first, the definition of $V$ second, the definitions of $T_0$, $T_1$, $\ldots$ $T_{\text{last}}$ next in order, and the definition of the geometry last. You need to understand this sequence very well, because it’s critical to understand how you build complex hierarchical models.
Transformation stacks and their manipulation

In defining a scene, we often want to define some standard pieces and then assemble them in standard ways, and then use the combined pieces to create additional parts, and go on to use these parts in additional ways. To do this, we need to create individual parts through functions that do not pay any attention to ways the parts will be used later, and then be able to assemble them into a whole. Eventually, we can see that the entire image will be a single whole that is composed of its various parts.

The key issue is that there is some kind of transformation in place when you start to define the object. When we begin to put the simple parts of a composite object in place, we will use some transformations but we need to undo the effect of those transformations when we put the next part in place. In effect, we need to save the state of the transformations when we begin to place a new part, and then to return to that transformation state (discarding any transformations we may have added past that mark) to begin to place the next part. Note that we are always adding and discarding at the end of the list; this tells us that this operation has the computational properties of a stack. We may define a stack of transformations and use it to manage this process as follows:

• as transformations are defined, they are multiplied into the current transformation in the order noted in the discussion of composite transformations above, and
• when we want to save the state of the transformation, we make a copy of the current version of the transformation and push that copy onto the stack, and apply all the subsequent transformations to the copy at the top of the stack. When we want to return to the original transformation state, we can pop the stack and throw away the copy that was removed, which gives us the original transformation so we can begin to work again at that point. Because all transformations are applied to the one at the top of the stack, when we pop the stack we return to the original context.

Designing a scene that has a large number of pieces of geometry as well as the transformations that define them can be difficult. In the next section we introduce the concept of the scene graph as a design tool to help you create complex and dynamic models both efficiently and effectively.

Compiling geometry
Rendering complex graphics objects can be rather slow if their geometry must be re-computed fully each time they are displayed. To save time, most graphics APIs allow you to compile geometry into a form that permits much faster rendering. Such compiled geometry will contain the final geometry after all the calculations and transformations to create the object have been done. In most graphics APIs, compiled geometry cannot be changed, so if you need to make changes to an object frequently it is probably not a good candidate for compilation. If changes are needed, you will need to re-compile the object. We will discuss how OpenGL compiles geometry later in this chapter.

Scene Graphs and Modeling Graphs

Introduction

In this chapter, we define modeling as the process of defining and organizing a set of geometry that represents a particular scene. While modern graphics APIs can provide you with a great deal of assistance in rendering your images, modeling is usually supported less well and causes programmers considerable difficulty when they begin to work in computer graphics. Organizing a scene with transformations, particularly when that scene involves hierarchies of components and when some of those components are moving, involves relatively complex concepts that need to be organized very systematically to create a successful scene. Hierarchical modeling has long been done by using trees or tree-like structures to organize the components of the model.

Recent graphics systems, such as Java3D and VRML 2, have formalized the concept of a scene graph as a powerful tool for both modeling scenes and organizing the rendering process for those scenes. By understanding and adapting the structure of the scene graph, we can organize a careful and formal tree approach to both the design and the implementation of hierarchical models. This can give us tools to manage not only modeling the geometry of such models, but also animation and interactive control of these models and their components.

In this section of the chapter we will describe the scene graph structure and will adapt it to a modeling graph that you can use to design scenes, and we will identify how this modeling graph gives us the three key transformations that go into creating a scene: the projection transformation, the viewing transformation, and the modeling transformation(s) for the scene’s content. This structure is very general and lets us manage all the fundamental principles in defining a scene and translating it into a graphics API. Our terminology is based on the scene graph of Java3D and should help anyone who uses that system understand the way scene graphs work there.

A brief summary of scene graphs

The fully-developed scene graph of the Java3D API has many different aspects and can be complex to understand fully, but we can abstract it somewhat to get an excellent model to help us think about scenes that we can use in developing the code to implement our modeling. A brief outline of the Java3D scene graph in Figure 2.9 will give us a basis to discuss the general approach to graph-structured modeling as it can be applied to beginning computer graphics.

A virtual universe holds one or more (usually one) locales, which are essentially positions in the universe to put scene graphs. Each scene graph has two kinds of branches: content branches, which are to contain shapes, lights, and other content, and view branches, which are to contain viewing information. This division is somewhat flexible, but we will use this standard approach to build a framework to support our modeling work.
The *content branch* of the scene graph is organized as a collection of nodes that contains group nodes, transform groups, and shape nodes. A *group node* is a grouping structure that can have any number of children; besides simply organizing its children, a group can include a switch that selects which children to present in a scene. A *transform group* is a collection of modeling transformations that affect all the geometry that lies below it. The transformations will be applied to any of the transform group’s children with the convention that transforms “closer” to the geometry (geometry that is defined in shape nodes lower in the graph) are applied first. A *shape node* includes both geometry and appearance data for an individual graphic unit. The geometry data includes standard 3D coordinates, normals, and texture coordinates, and can include points, lines, triangles, and quadrilaterals, as well as triangle strips, triangle fans, and quadrilateral strips. The appearance data includes color, shading, or texture information. Lights and eye points are included in the content branch as a particular kind of geometry, having position, direction, and other appropriate parameters. Scene graphs also include shared groups, or groups that are included in more than one branch of the graph, which are groups of shapes that are included indirectly in the graph, and any change to a shared group affects all references to that group. This allows scene graphs to include the kind of template-based modeling that is common in graphics applications.

![Diagram of the structure of the scene graph as defined in Java3D](image)

The *view branch* of the scene graph includes the specification of the display device, and thus the projection appropriate for that device. It also specifies the user’s position and orientation in the scene and includes a wide range of abstractions of the different kinds of viewing devices that can be used by the viewer. It is intended to permit viewing the same scene on any kind of display device, including sophisticated virtual reality devices. This is a much more sophisticated approach than we need for our relatively simple modeling. We will simply consider the eye point as part of the geometry of the scene, so we set the view by including the eye point in the content branch and get the transformation information for the eye point in order to create the view transformations in the view branch.

In addition to the modeling aspect of the scene graph, Java3D also uses it to organize the processing as the scene is rendered. Because the scene graph is processed from the bottom up, the content branch is processed first, followed by the viewing transformation and then the projection.
transformation. However, the system does not guarantee any particular sequence in processing the node’s branches, so it can optimize processing by selecting a processing order for efficiency, or can distribute the computations over a networked or multiprocessor system. Thus the Java3D programmer must be careful to make no assumptions about the state of the system when any shape node is processed. We will not ask the system to process the scene graph itself, however, because we will only use the scene graph to develop our modeling code.

An example of modeling with a scene graph

We will develop a scene graph to design the modeling for an example scene to show how this process can work. To begin, we present an already-completed scene so we can analyze how it can be created, and we will take that analysis and show how the scene graph can give us other ways to present the scene. Consider the scene as shown in Figure 2.10, where a helicopter is flying above a landscape and the scene is viewed from a fixed eye point. (The helicopter is the small green object toward the top of the scene, about 3/4 of the way across the scene toward the right.)

![Figure 2.10: a scene that we will describe with a scene graph](image)

This scene contains two principal objects: a helicopter and a ground plane. The helicopter is made up of a body and two rotors, and the ground plane is modeled as a single set of geometry with a texture map. There is some hierarchy to the scene because the helicopter is made up of smaller components, and the scene graph can help us identify this hierarchy so we can work with it in rendering the scene. In addition, the scene contains a light and an eye point, both at fixed locations. The first task in modeling such a scene is now complete: to identify all the parts of the scene, organize the parts into a hierarchical set of objects, and put this set of objects into a viewing context. We must next identify the relationship among the parts of the landscape way so we may create the tree that represents the scene. Here we note the relationship among the ground and the parts of the helicopter. Finally, we must put this information into a graph form.

The initial analysis of the scene in Figure 2.10, organized along the lines of view and content branches, leads to an initial (and partial) graph structure shown in Figure 2.11. The content branch of this graph captures the organization of the components for the modeling process. This describes how content is assembled to form the image, and the hierarchical structure of this branch helps us organize our modeling components. The view branch of this graph corresponds roughly to projection and viewing. It specifies the projection to be used and develops the projection transformation, as well as the eye position and orientation to develop the viewing transformation.
This initial structure is compatible with the simple OpenGL viewing approach we discussed in the previous chapter and the modeling approach earlier in this chapter, where the view is implemented by using built-in function that sets the viewpoint, and the modeling is built from relatively simple primitives. This approach only takes us so far, however, because it does not integrate the eye into the scene graph. It can be difficult to compute the parameters of the viewing function if the eye point is embedded in the scene and moves with the other content, and later we will address that part of the question of rendering the scene.

While we may have started to define our scene graph, we are not nearly finished. The initial scene graph of Figure 2.11 is incomplete because it merely includes the parts of the scene and describes which parts are associated with what other parts. To expand this first approximation to a more complete graph, we must add several things to the graph:

- the transformation information that describes the relationship among items in a group node, to be applied separately on each branch as indicated,
- the appearance information for each shape node, indicated by the shaded portion of those nodes,
- the light and eye position, either absolute (as used in Figure 2.10 and shown Figure 2.12) or relative to other components of the model, and
- the specification of the projection and view in the view branch.

These are all included in the expanded version of the scene graph with transformations, appearance, eyepoint, and light shown in Figure 2.121.

The content branch of this graph handles all the scene modeling and is very much like the content branch of the scene graph. It includes all the geometry nodes of the graph in Figure 2.11 as well as appearance information; includes explicit transformation nodes to place the geometry into correct sizes, positions, and orientations; includes group nodes to assemble content into logical groupings; and includes lights and the eye point, shown here in fixed positions without excluding the possibility that a light or the eye might be attached to a group instead of being positioned independently. In the example above, it identifies the geometry of the shape nodes such as the rotors or individual trees as shared. This might be implemented, for example, by defining the geometry of the shared shape node in a function and calling that from each of the rotor or tree nodes that uses it.
The view branch of this graph is similar to the view branch of the scene graph but is treated much more simply, containing only projection and view components. The projection component includes the definition of the projection (orthogonal or perspective) for the scene and the definition of the window and viewport for the viewing. The view component includes the information needed to create the viewing transformation, and because the eye point is placed in the content branch, this is simply a copy of the set of transformations that position the eye point in the scene as represented in the content branch.

The appearance part of the shape node is built from color, lighting, shading, texture mapping, and similar operations that are not developed here because we are focusing now on only the geometry and transformations in modeling. However, much of the rest of these notes is devoted to building the appearance properties of the shape node, because the appearance component is perhaps the most important part of graphics for building high-quality images.

The scene graph for a particular image is not unique because there are many ways to organize a scene. When you have a well-defined set of transformation that place the eye point in a scene, we saw in the earlier chapter on viewing how you can take advantage of that information to organize the scene graph in a way that can define the viewing transformation explicitly and simply use the default view for the scene. As we noted there, the real effect of the viewing transformation is to be the inverse of the transformation that placed the eye. So we can explicitly compute the viewing transformation as the inverse of the placement transformation ourselves and place that at the top of the scene graph. Thus we can restructure the scene graph of Figure 2.12 as shown below in Figure 2.13 so it may take any arbitrary eye position. This will be the key point below as we discuss how to manage the eyepoint when it is a dynamic part of a scene.

It is very important to note that the scene graph need not describe a static geometry. Callbacks for user interaction and other events can affect the graph by controlling parameters of its components, as noted in the re-write guidelines in the next section. This can permit a single graph to describe an animated scene or even alternate views of the scene. The graph may thus be seen as having some components with external controllers, and the controllers are the event callback functions.
We need to extract the three key transformations from this graph in order to create the code that implements our modeling work. The projection transformation is straightforward and is built from the projection information in the view branch, and this is easily managed from tools in the graphics API. The viewing transformation is readily created from the transformation information in the view by analyzing the eye placement transformations as we saw above, and the modeling transformations for the various components are built by working with the various transformations in the content branch as the components are drawn. These operations are all straightforward; we begin with the viewing transformation and move on to coding the modeling transformations.

*The viewing transformation*

In a scene graph with no view specified, we assume that the default view puts the eye at the origin looking in the negative z-direction with the y-axis upward. If we use a set of transformations to position the eye differently, then the viewing transformation is built by inverting those transformations to restore the eye to the default position. This inversion takes the sequence of transformations that positioned the eye and inverts the primitive transformations in reverse order, so if $T_1T_2T_3...T_K$ is the original transformation sequence, the inverse is $T_K^u...T_3^uT_2^uT_1^u$ where the superscript $u$ indicates inversion, or “undo” as we might think of it.

Each of the primitive scaling, rotation, and translation transformations is easily inverted. For the scaling transformation $\text{scale}(Sx, Sy, Sz)$, we note that the three scale factors are used to multiply the values of the three coordinates when this is applied. So to invert this transformation, we must divide the values of the coordinates by the same scale factors, getting the inverse as $\text{scale}(1/Sx, 1/Sy, 1/Sz)$. Of course, this tells us quickly that the scaling function can only be inverted if none of the scaling factors are zero.
For the rotation transformation \texttt{rotate}(\texttt{angle}, \texttt{line}) that rotates space by the value \texttt{angle} around the fixed line \texttt{line}, we must simply rotate the space by the same angle in the reverse direction. Thus the inverse of the rotation transformation is \texttt{rotate}(-\texttt{angle}, \texttt{line}).

For the translation transformation \texttt{translate}(\texttt{Tx, Ty, Tz}) that adds the three translation values to the three coordinates of any point, we must simply subtract those same three translation values when we invert the transformation. Thus the inverse of the translation transformation is \texttt{translate}(-\texttt{Tx, -Ty, -Tz}).

Putting this together with the information on the order of operations for the inverse of a composite transformation above, we can see that, for example, the inverse of the set of operations (written as if they were in your code)

\begin{verbatim}
  translate(Tx, Ty, Tz)
  rotate(\texttt{angle}, \texttt{line})
  scale(Sx, Sy, Sz)
\end{verbatim}

is the set of operations

\begin{verbatim}
  scale(1/Sx, 1/Sy, 1/Sz)
  rotate(-\texttt{angle}, \texttt{line})
  translate(-\texttt{Tx, -Ty, -Tz})
\end{verbatim}

Now let us apply this process to the viewing transformation. Deriving the eye transformations from the tree is straightforward. Because we suggest that the eye be considered one of the content components of the scene, we can place the eye at any position relative to other components of the scene. When we do so, we can follow the path from the root of the content branch to the eye to obtain the sequence of transformations that lead to the eye point. That sequence of transformations is the eye transformation that we may record in the view branch.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{image.png}
\caption{the same scene as in Figure 2.10 but with the eye point following directly behind the helicopter}
\end{figure}

In Figure 2.14 we show the change that results in the view of Figure 2.10 when we define the eye to be immediately behind the helicopter, and in Figure 2.15 we show the change in the scene graph of Figure 2.12 that implements the changed eye point. The eye transform consists of the transforms that places the helicopter in the scene, followed by the transforms that place the eye relative to the helicopter. Then as we noted earlier, the viewing transformation is the inverse of the
eye positioning transformation, which in this case is the inverse of the transformations that placed the eye relative to the helicopter, followed by the inverse of the transformations that placed the helicopter in the scene.

This change in the position of the eye means that the set of transformations that lead to the eye point in the view branch must be changed, but the mechanism of writing the inverse of these transformations before beginning to write the definition of the scene graph still applies; only the actual transformations to be inverted will change. This is how the scene graph will help you to organize the viewing process that was described in the earlier chapter on viewing.

The process of placing the eye point can readily be generalized. For example, if you should want to design a scene with several possible eye points and allow a user to choose among them, you can design the view branch by creating one view for each eye point and using the set of transformations leading to each eye point as the transformation for the corresponding view. You can then invert each of these sets of transformations to create the viewing transformation for each of the eye points. The choice of eye point will then create a choice of view, and the viewing transformation for that view can then be chosen to implement the user choice.

Because the viewing transformation is performed before the modeling transformations, we see from Figure 2.13 that the inverse transformations for the eye must be applied before the content branch is analyzed and its operations are placed in the code. This means that the display operation must begin with the inverse of the eye placement transformations, which has the effect of moving the eye to the top of the content branch and placing the inverse of the eye path at the front of each set of transformations for each shape node.

Using the modeling graph for coding

Let us use the name “modeling graph” for the analogue of the scene graph we illustrated in the previous section. Because the modeling graph is intended as a learning tool, we will resist the temptation to formalize its definition beyond the terms we used there:

- shape node containing two components
  - geometry content
  - appearance content
- transformation node
- group node
- projection node
- view node
Because we do not want to look at any kind of automatic parsing of the modeling graph to create the scene, we will merely use the graph to help organize the structure and the relationships in the model to help you organize your code to implement your simple or hierarchical modeling. This is quite straightforward and is described in detail below.

Once you know how to organize all the components of the model in the modeling graph, you next need to write the code to implement the model. This turns out to be straightforward, and you can use a simple set of re-write guidelines that allow you to re-write the graph as code. In this set of rules, we assume that transformations are applied in the reverse of the order they are declared, as they are in OpenGL, for example. This is consistent with your experience with tree handling in your programming courses, because you have usually discussed an expression tree which is parsed in leaf-first order. It is also consistent with the Java3D convention that transforms that are “closer” to the geometry (nested more deeply in the scene graph) are applied first.

The informal re-write guidelines are as follows, including the re-writes for the view branch as well as the content branch:

• Nodes in the view branch involve only the window, viewport, projection, and viewing transformations. The window, viewport, and projection are handled by simple functions in the API and should be at the top of the display function.

• The viewing transformation is built from the transformations of the eye point within the content branch by copying those transformations and undoing them to place the eye effectively at the top of the content branch. This sequence should be next in the display function.

• The content branch of the modeling graph is usually maintained fully within the display function, but parts of it may be handled by other functions called from within the display, depending on the design of the scene. A function that defines the geometry of an object may be used by one or more shape nodes. The modeling may be affected by parameters set by event callbacks, including selections of the eye point, lights, or objects to be displayed in the view.

• Group nodes are points where several elements are assembled into a single object. Each separate object is a different branch from the group node. Before writing the code for a branch that includes a transformation group, the student should push the modelview matrix; when returning from the branch, the student should pop the modelview matrix.

• Transformation nodes include the familiar translations, rotations, and scaling that are used in the normal ways, including any transformations that are part of animation or user control. In writing code from the modeling graph, students can write the transformations in the same sequence as they appear in the tree, because the bottom-up nature of the design work corresponds to the last-defined, first-used order of transformations.

• As you work your way through the modeling graph, you will need to save the state of the modeling transformation before you go down any branch of the graph from which you will need to return as the graph is traversed. Because of the simple nature of each transformation primitive, it is straightforward to undo each as needed to create the viewing transformation. This can be handled through a transformation stack that allows you to save the current transformation by pushing it onto the stack, and then restore that transformation by popping the stack.

• Shape nodes involve both geometry and appearance, and the appearance must be done first because the current appearance is applied when geometry is defined.
  - An appearance node can contain texture, color, blending, or material information that will make control how the geometry is rendered and thus how it will appear in the scene.
  - A geometry node will contain vertex information, normal information, and geometry structure information such as strip or fan organization.

• Most of the nodes in the content branch can be affected by any interaction or other event-driven activity. This can be done by defining the content by parameters that are modified
by the event callbacks. These parameters can control location (by parametrizing rotations or translations), size (by parametrizing scaling), appearance (by parametrizing appearance details), or even content (by parametrizing switches in the group nodes).

We will give some examples of writing graphics code from a modeling graph in the sections below, so look for these principles as they are applied there.

In the example for Figure 2.14 above, we would use the tree to write code as shown in skeleton form in Figure 2.16. Most of the details, such as the inversion of the eye placement transformation, the parameters for the modeling transformations, and the details of the appearance of individual objects, have been omitted, but we have used indentation to show the pushing and popping of the modeling transformation stack so we can see the operations between these pairs easily. This is straightforward to understand and to organize.

```display()
    set the viewport and projection as needed
    initialize modelview matrix to identity
    define viewing transformation
        invert the transformations that set the eye location
    set eye through gluLookAt with default values
    define light position       // note absolute location
    push the transformation stack // ground
        translate
        rotate
        scale
        define ground appearance (texture)
        draw ground
    pop the transformation stack
    push the transformation stack // helicopter
        translate
        rotate
        scale
        push the transformation stack // top rotor
            translate
            rotate
            scale
            define top rotor appearance
            draw top rotor
        pop the transformation stack
    push the transformation stack // back rotor
        translate
        rotate
        scale
        define back rotor appearance
        draw back rotor
    pop the transformation stack
    // assume no transformation for the body
    define body appearance
    draw body
    pop the transformation stack
    swap buffers```

Figure 2.16: code sketch to implement the modeling in Figure 2.15

Animation is simple to add to this example. The rotors can be animated by adding an extra rotation in their definition plane immediately after they are scaled and before the transformations that orient them to be placed on the helicopter body, and by updating angle of the extra rotation each time the
idle event callback executes. The helicopter’s behavior itself can be animated by updating the parameters of transformations that are used to position it, again with the updates coming from the idle callback. The helicopter’s behavior may be controlled by the user if the positioning transformation parameters are updated by callbacks of user interaction events. So there are ample opportunities to have this graph represent a dynamic environment and to include the dynamics in creating the model from the beginning.

Other variations in this scene could be developed by changing the position of the light from its current absolute position to a position relative to the ground (by placing the light as a part of the branch group containing the ground) or to a position relative to the helicopter (by placing the light as a part of the branch group containing the helicopter). The eye point could similarly be placed relative to another part of the scene, or either or both could be placed with transformations that are controlled by user interaction with the interaction event callbacks setting the transformation parameters.

We emphasize that you should include appearance content with each shape node. Many of the appearance parameters involve a saved state in APIs such as OpenGL and so parameters set for one shape will be retained unless they are re-set for the new shape. It is possible to design your scene so that shared appearances will be generated consecutively in order to increase the efficiency of rendering the scene, but this is a specialized organization that is inconsistent with more advanced APIs such as Java3D. Thus it is very important to re-set the appearance with each shape to avoid accidentally retaining an appearance that you do not want for objects presented in in later parts of your scene.

Example

We want to further emphasize the transformation behavior in writing the code for a model from the modeling graph by considering another small example. Let us consider a very simple rabbit’s head as shown in Figure 2.17. This would have a large ellipsoidal head, two small spherical eyes, and two middle-sized ellipsoidal ears. So we will use the ellipsoid (actually a scaled sphere, as we saw earlier) as our basic part and will put it in various places with various orientations as needed.

The modeling graph for the rabbit’s head is shown in Figure 2.18. This figure includes all the transformations needed to assemble the various parts (eyes, ears, main part) into a unit. The fundamental geometry for all these parts is the sphere, as we suggested above. Note that the transformations for the left and right ears include rotations; these can easily be designed to use a parameter for the angle of the rotation so that you could make the rabbit’s ears wiggle back and forth.

![Figure 2.17: the rabbit’s head](image)
To write the code to implement the modeling graph for the rabbit’s head, then, we would apply the following sequence of actions on the modeling transformation stack:

- push the modeling transformation stack
- apply the transformations to create the head, and define the head:
  - scale
  - draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the left eye relative to the head, and define the eye:
  - translate
  - scale
  - draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the right eye relative to the head, and define the eye:
  - translate
  - scale
  - draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the left ear relative to the head, and define the ear:
  - translate
  - rotate
  - scale
  - draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the right ear relative to the head, and define the ear:
  - translate
  - rotate
  - scale
  - draw sphere
- pop the modeling transformation stack
You should trace this sequence of operations carefully and watch how the head is drawn. Note that if you were to want to put the rabbit’s head on a body, you could treat this whole set of operations as a single function `rabbitHead()` that is called between operations push and pop the transformation stack, with the code to place the head and move it around lying above the function call. This is the fundamental principle of hierarchical modeling — to create objects that are built of other objects, finally reducing the model to simple geometry at the lowest level. In the case of the modeling graph, that lowest level is the leaves of the tree, in the shape nodes.

The transformation stack we have used informally above is a very important consideration in using a scene graph structure. It may be provided by your graphics API or it may be something you need to create yourself; even if it provided by the API, there may be limits on the depth of the stack that will be inadequate for some projects and you may need to create your own. We will discuss this in terms of the OpenGL API later in this chapter.

**Using standard objects to create more complex scenes**

The example of transformation stacks is, in fact, a larger example — an example of using standard objects to define a larger object. In a program that defined a scene that needed rabbits, we would create the rabbit head with a function `rabbitHead()` that has the content of the code we used (and that is given below) and would apply whatever transformations would be needed to place a rabbit head properly on each rabbit body. The rabbits themselves could be part of a larger scene, and you could proceed in this way to create however complex a scene as you wish.

**Compiling geometry**

It can take a fair amount of time to calculate the various components of a piece of an image when that piece involves vertex lists and transformations. As a way to save time in displaying the image, many graphics APIs allow you to “compile” your geometry in a way that will allow it to be displayed much more quickly. Geometry that is to be compiled should be carefully chosen so that it is not changed between displays, and the modeling graph will help you identify parts of a scene that will not change from image to image. Once you have seen what parts you can compile, you can compile them and use the compiled versions to make the display faster. Look for details in the documentation of your own API.

*A word to the wise...*

As we noted above, you must take a great deal of care with transformation order. It can be very difficult to look at an image that has been created with mis-ordered transformations and understand just how that erroneous example happened. In fact, there is a skill in what we might call “visual debugging” — looking at an image and seeing that it is not correct, and figuring out what errors might have caused the image as it is seen. It is important that anyone working with images become skilled in this kind of debugging. However, obviously you cannot tell than an image is wrong unless you know what a correct image should be, so you must know in general what you should be seeing. As an obvious example, if you are doing scientific images, you must know the science well enough to know when an image makes no sense.

**Implementing Modeling in OpenGL**

*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:
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 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 common operations that are permitted 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.

```c
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.

```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();
```

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.

### Triangle

To draw unconnected triangles, you use `glBegin/glEnd` with the mode `GL_TRIANGLES`. This is treated exactly as discussed in the earlier section.

### 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 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 XZ-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 2.19. 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.

```c
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));
}
glEnd();
```

![Figure 2.19: the cone produced by the triangle fan](image)

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 domain 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.

```c
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 2.20. This kind of surface is explored in more
detail in the chapters on scientific applications of graphics.

```c
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 2.20: 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:

```c
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. 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.

```c
#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.

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, 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.

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, edges, and faces
of the cube from earlier in this chapter. We will use the glVertex3fv(...) vertex specification
function within the specification of the quads for the faces.

```c
typedef float point3[3];
typedef int edge[2];    // assumes a face has four edges for this example
typedef int face[4];

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} };

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 }, { 14,  6, 10, 19 },
{  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. 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.

```c
void cube(void) {
    for (face = 0; face < 6; face++) {
        glBegin(GL_QUADS);
        for (edge = 0; edge < 4; edge++)
            glVertex3fv(vertices[edges[cube[face][edge]][0]]);
    }
}
```
This cube is shown in Figure 2.21, and this approach 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.

![Figure 2.21: the cube as a sequence of quads](image)

This approach to modeling an object is missing one important feature — it does not deal with the normals (the vectors perpendicular to each face) of the object. We will see in the chapter on lighting that in order to get the added realism of lighting on an object, we must provide information on the object’s normals. It would be straightforward to define another array that contains a normal for each face, or another array that contains a normal for each vertex. We will not pursue these ideas here, but you should be thinking about them when you consider modeling issues with lighting.

*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 2.22 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.

![Figure 2.22: A GLUT wireframe sphere with 10 slices and 10 stacks](image)

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 styles 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.

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:**

```c
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.

```c
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). There are also wireframe versions of each of the GLUT solid objects.

The GLUT primitives 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.

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 2.23 to show the range of items you can create with these tools. You should think about how you might use various transformations to create other figures from these basic parts.
Figure 2.23: various GLU and GLUT objects: (... - list - ...)

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.

```c
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 `glRotate(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. Again, there is a translation function `glTranslated` that operates exactly the same but has doubles 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.

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:

\[
P \ V \ T_0 \ T_1 \ \ldots \ T_n \ T_{n+1} \ \ldots \ T_{\text{last}}
\]

with \( P \) being the projection transformation, \( V \) the viewing transformation, and \( T_0, T_1, \ldots T_{\text{last}} \) the transformations specified in the program to model the scene, in order (\( T_1 \) is first, \( T_{\text{last}} \) 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: \( T_{\text{last}} \) 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, heirarchical 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:

```c
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} };
```

```c
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:

```c
void display( void )
{
    int i;
    float theta = 0.0;
```
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…

```c
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, 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

```c
GLfloat viewProj[16];
```

and you can use the functions

```c
glLoadIdentity();
```

and

```c
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.

Creating display lists

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. 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. Sample code and a more complete explanation 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 often defined only once, 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.

```c
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.