Chapter 2: Principles of Modeling

Modeling is the first step in the graphics pipeline that we saw in the previous two chapters. It is how you define the geometry that makes up a scene and implement that definition with the tools of your graphics API. This chapter is critical in understanding the standard polygon-based approach to modeling and 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. Many graphics APIs' modeling is based on polygons and this chapter is fairly comprehensive for that kind of modeling. However, there are several other kinds of modeling used in computer graphics and some areas of computer graphics involve more sophisticated kinds of constructions than we include here, so we cannot call this a genuinely comprehensive discussion. One of other approaches, ray tracing, is discussed in a later chapter. This chapter is, however, a good enough introduction to give you a good set of tools to start creating interesting images.

The chapter has three distinct parts because there are three distinct levels of modeling that you will use to create images. We begin with simple geometric modeling directly in world space: 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: creating your simple objects in standard positions in their own individual model space and using modeling transformations to place them in world space and give them any size, any orientation, and any position. This allows you to create a set of simple models and extends their value by allowing you to use them very generally. This involves a standard set of modeling transformations you can apply to your simple geometry in order to create more general model components in your scene. This is a very important part of the modeling process because it allows you to use appropriate transformations applied to standard template objects, allowing you to create modest number of graphic objects and then generalize them and place them in your scene as needed. These transformations are also critical to the ability to define and implement motion in your scenes because it is typical to move parts of your scene, such as 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 time-varying concepts. This second section is presented without any reference to a particular graphics API, but in the next chapter we will show how the concepts here are expressed in OpenGL.

In the third section of the chapter we give you an important tool to organize complex images by introducing 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.
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. When you have finished this chapter you should be able to understand how to organize the geometry for a scene based on simple model components and how to combine them with modeling transformations. You should also be able to understand how to create complex, hierarchical scenes with a scene graph and how to express such a scene graph in terms of graphics primitives.
Simple Geometric Modeling

Introduction

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, you will find that the geometry will be simple and will use familiar representations of 3-dimensional space. When you work with a graphics API, you will need to work with the kinds of object representations that API understands, so you must design your image or scene in ways that fit the API’s tools. For most APIs, this means using only a few simple graphics primitives, such as points, line segments, and polygons.

The application programmer starts by defining a particular object with respect to a coordinate system that includes a local origin lying somewhere in or around the object. This would naturally happen if the object was created with some sort of modeling or computer-aided design system or was defined by a mathematical function, and is described for some of these cases in subsequent chapters. Modeling an object about its local origin involves defining it in terms of model coordinates, a coordinate system that is used specifically to define a particular graphical object. The model coordinates are defined by specifying the coordinates of each point by either defining constant coordinates or by computing them from some known geometric object. This is done with code that will look something like

```c
vertex(x1, y1, z1);
vertex(x2, y2, z2);
...
vertex(xN, yN, zN);
```

Because the coordinate system is part of an object’s design, it may be different for every part of a scene. In order to integrate each object, built with its own coordinates, into a single overall 3D world space, the object must be placed in the world space by using an appropriate modeling transformation. Modeling transformations, like all the transformations we will describe throughout the book, are functions that move objects while preserving their geometric properties. The transformations that are available to us in a graphics system are rotations, translations, and scaling. Rotations hold a line through the origin of a coordinate system fixed and rotate all the points in a scene by a fixed angle around the line, translations add a fixed value to each of the coordinates of each point in a scene, and scaling multiplies each coordinate of a point by a fixed value. These will be discussed in much more detail in the chapter on modeling below. All transformations may be represented as matrices, so sometimes in a graphics API you will see a mention of a matrix; this almost always means that a transformation is involved.

In practice, graphics programmers use a relatively small set of simple, built-in transformations and build up the model transformations through a sequence of these simple transformations. Because each transformation works on the geometry it sees, we see the effect of the associative law for functions; in a piece of code represented by metacode such as

```c
transformOne(...);
transformTwo(...);
transformThree(...);
geometry(...);
```

we see that `transformThree` is applied to the original geometry, `transformTwo` to the results of that transformation, and `transformOne` to the results of the second transformation. Letting \( t_1, t_2, \) and \( t_3 \) be the three transformations, respectively, we see by the application of the associative law for function composition that

\[
t_1(t_2(t_3(geometry))) = (t_1*t_2*t_3)(geometry)
\]

This shows us that in a product of transformations, applied by multiplying on the left, the transformation nearest the geometry is applied first, and that this principle extends across multiple
transformations. This will be very important in the overall understanding of the overall order in which we operate on scenes, as we describe at the end of this section.

The modeling transformation for an object in a scene can change over time to create motion in a scene. For example, in a rigid-body animation, an object can be moved through the scene just by changing its model transformation between frames. This change can be made through standard built-in facilities in most graphics APIs, including OpenGL; we will discuss how this is done later.

**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.

![Figure 2.1: a point, a line segment, a polygon, and a polyhedron](image)

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 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 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.

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 \textit{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 \( W=1 \) 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 \(w\neq0\). The four-dimensional representation of points with a non-zero \(w\) component is called \textit{homogeneous coordinates}, and calculating the three-dimensional equivalent for a homogeneous representation by dividing by \(w\) is called \textit{homogenizing} the point. When we discuss transformations, we will sometimes think of them as \(4\times4\) matrices because we will need them to operate on points in homogeneous coordinates.

Not all points in 4-dimensional space can be identified with points in 3-space, however. The point \((x, y, z, 0)\) is not identified with a point in 3-space because it cannot be homogenized, but it is identified with the direction defined by the vector \(\langle x, y, z \rangle\). This can be thought of as a “point at infinity” in a certain direction. 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.

\textit{Some examples}

In this section we will describe the kinds of simple objects that are directly supported by most graphics APIs. We begin with very simple objects and proceed to more complex ones, but you will find that both simple and complex objects will be needed in your work. 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.

The simple way that a graphics API handles lines hides an important concept, however. A line is a continuous object with real-valued coordinates, and it is displayed on a discrete object with integer screen coordinates. This is, of course, the difference between model space and eye space on one hand and screen space on the other. While we focus on geometric thinking in terms that overlook the details of conversions from eye space to screen space, you need to realize that algorithms for such conversions lie at the foundation of computer graphics and that your ability to think in higher-level terms is a tribute to the work that has built these foundations.

Connected lines

Connected lines—collections of 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 line loops, 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. The difference between a line strip and a line loop is that the former does not connect the last vertex defined to the first vertex, leaving the figure open; the latter includes this extra segment and creates a closed figure. Thus the number of line segments drawn by the a line strip will be one fewer than the number of vertices in the vertex list, while a line loop will draw the same number of segments as vertices. 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.

![Figure 2.2: a line strip and a line loop](image)

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. Note that this figure and similar figures that show simple geometric primitives are presented as if they were drawn in 2D space. In fact they are not, but in order to make them look three-dimensional we would need to use some kind of shading, which is a separate concept discussed in a later chapter (and which is used to present the triangle fan of Figure 2.18). We thus ask you to think of these as three-dimensional, even though they look flat.

![Figure 2.3: triangle strip and triangle fan](image)

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](image)

As an example of the use of quad strips and triangle fans, let’s create your a model of a sphere. As we will see in the next chapter, both the GLU and GLUT toolkits include pre-built sphere models, 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.”

In the chapter on mathematical fundamentals, we will describe the use of spherical coordinates in modeling. We can use spherical coordinates to model the sphere at first, and then we can later convert to Cartesian coordinates as we describe in that chapter 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:

```cpp
// 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()

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 as we might want to do for other kinds of objects. Note the order in which we set the points in the triangle fans and in the quad strips, as we described when we introduced these concepts; this is not immediately an obvious order and you may want to think about it a bit. When you do, you will find that the point sequence for a quad strip is exactly the same as the point sequence for a triangle strip.

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 will note in the 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 later discussion for more details, but we include Figure 2.5 below to illustrate the difference.

![Figure 2.5: convex and non-convex polygons](image)

An interesting property of convex polygons is that if you take two adjacent vertices and then write the remaining vertices in order as they occur around the polygon, you have exactly the same vertex sequence as if you were defining a triangle fan. Thus just as was the case for quad strips and triangle strips, you can see a way to implement a convex polygon as a triangle fan.

Polyhedron

In Figure 2.1 we saw that a polyhedron is one of the basic objects we use in our modeling, especially when we will focus almost exclusively on 3D computer graphics. We specify a polyhedron by specifying all the polygons that make up its boundary. In general, most graphics APIs leave the specification of polyhedrons up to the user, which can make them fairly difficult objects to define as you are learning the subject. With experience, however, you will develop a set of polyhedra that you’re familiar with and can use them with comfort.
While a graphics API may not have a general set of polyhedra, however, some provide a set of basic polyhedra that can be very useful to you. These depend on the API so we cannot be more specific here, but the next chapter includes a description of the polyhedra provided by OpenGL.

**Aliasing and antialiasing**

When you create a point, line, or polygon in your image, the system will define the pixels on the screen that represent the geometry within the discrete integer-coordinate 2D screen space. The standard way of selecting pixels is all-or-none: a pixel is computed to be either in the geometry, in which case it is colored as the geometry specifies, or not in the geometry, in which case it is left in whatever color it already was. Because of the relatively coarse nature of screen space, this all-or-nothing approach can leave a great deal to be desired because it created jagged edges along the space between geometry and background. This appearance is called *aliasing*, and it is shown in the left-hand image of Figure 2.6.

There are a number of techniques to reduce the effects of aliasing, and collectively the techniques are called *antialiasing*. They all work by recognizing that the boundary of a true geometry can go through individual pixels in a way that only partially covers a pixel. Each technique finds a way to account for this varying coverage and then lights the pixel according to the amount of coverage of the pixel with the geometry. Because the background may vary, this variable lighting is often managed by controlling the blending value for the pixel’s color, using the color \((R, G, B, A)\) where \((R, G, B)\) is the geometry color and \(A\) is the proportion of the pixel covered by the object’s geometry. An image that uses antialiasing is shown in the right-hand image of Figure 2.6. For more detail on color blending, see the later chapter on color.

![Figure 2.6: aliased lines (left) and antialiased lines (right)](image)

As we said, there are many ways to determine the coverage of a pixel by the geometry. One way that is often used for very high-quality images is to *supersample* the pixel, that is, to assume a much higher image resolution than is really present and to see how many of these “subpixels” lie in the geometry. The proportion of subpixels that would be covered will serve as the proportion for the antialiasing value. However, supersampling is not an ordinary function of a graphics API, so we would expect a simpler approach to be used. Because APIs use linear geometries—all the basic geometry is polygon-based—it is possible to calculate exactly how the 2D world space line intersects each pixel and then how much of the pixel is covered. This is a more standard kind of API computation, though the details will certainly vary between APIs and even between different implementations of an API. You may want to look at your API’s manuals for more details.
Normals

When you define the geometry of an object, you may also want or need to define the direction the object faces as well as the coordinate values for the point. This is done by defining a normal for the object. Normals are often fairly easy to obtain. In the appendix to this chapter you will see ways to calculate normals for plane polygons fairly easily; for many of the kinds of objects that are available with a graphics API, normals are built into the object definition; and if an object is defined by mathematical formulas, you can often get normals by doing some straightforward calculations.

The sphere described above is a good example of getting normals by calculation. For a sphere, the normal to the sphere at a given point is the radius vector at that point. For a unit sphere with center at the origin, the radius vector to a point has the same components as the coordinates of the point. So if you know the coordinates of the point, you know the normal at that point.

To add the normal information to the modeling definition, then, you can simply use functions that set the normal for a geometric primitive, as you would expect to have from your graphics API, and you would get code that looks something like the following excerpt from the example above:

```plaintext
for j = -90+180/M to 90-180/M // latitude without sphere caps
  doQuadStrip()
    // one quad strip per band around the sphere at any latitude
    for i = 0 to 360 // longitude
      set normal to (1, i, j)
      set vertex at (1, i, j)
      set vertex at (1, i, j+180/M)
      set vertex at (1, i+360/N, j)
      set vertex at (1, i+360/N, j+180/M)
  endQuadStrip()
```

![Figure 2.7: a normal to a surface quad on a sphere](image)

Data structures to hold objects

When you define a polyhedron for your graphics work, as we discussed above, there are many ways you can hold the information that describes a polyhedral graphics object. One of the simplest is the triangle list—an array of triples, with each set of three triples representing 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 and whose formal specifications are in the Appendix.

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. The normals are vectors, one per face, but these are also given as arrays of three points. In C, these would be given as follows:

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

point3 vertices[8] = {{-1.0, -1.0, -1.0},
                      {-1.0, -1.0,  1.0 },
                      {-1.0,  1.0, -1.0 },
                      {-1.0,  1.0,  1.0 },
                      { 1.0, -1.0, -1.0 },
                      { 1.0, -1.0,  1.0 },
                      { 1.0,  1.0, -1.0 },
                      { 1.0,  1.0,  1.0 }};

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

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

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

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) `setVertex(...)` and `setNormal(...)` functions. 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 needed by your API, then you may omit the first setVertex call in the pseudocode for the function cube() below.

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

We added a simple structure for a list of normals, with one normal per face, which echoes the structure of the faces. This supports what is often called flat shading, or shading where each face has a single color. In many applications, though, you might want to have smooth shading, where colors blend smoothly across each face of your polygon. For this, each vertex needs to have its individual normal representing the perpendicular to the object at that vertex. In this case, you often need to specify the normal each time you specify a vertex, and a normal list that follows the vertex list would allow you to do that easily. For the code above, for example, we would not have a per-face normal but instead each setVertex operation could be replaced by the pair of operations

```c
    setNormal(normals[edges[cube[face][0]][0]]);
    setVertex(vertices[edges[cube[face][0]]][0]);
```

Neither the simple triangle list nor the more complex structure of vertex, normal, 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 only talked about line strips, triangle strips, triangle fans, and quad strips above because these are more often supported by a graphics API. Geometry compression approaches not only save space, but are also 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.

### 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 \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 perfectly reasonable 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 should be able to 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.

Transformations

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, a light, a direction, 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.

![Figure 2.8](image)

Figure 2.8: 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)

*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 applying the scaling transformation changes the point to \((x \times S_x, y \times S_y, z \times S_z)\). 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 2.8 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.
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 2.9. 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

\[
\begin{align*}
\text{translate}( \text{Tx, Ty, Tz} ) \\
\text{rotate}( \text{angle, x-axis} ) \\
\text{scale}( 1., 2., 1. ) \\
\text{drawBall}()
\end{align*}
\]
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 travels from left to right, while the position of the ball describes a parabola as it moves, modeling the effect of gravity on the ball’s flight. This kind of composite transformation construction is described in the next section, and as we point out there, the order of these transformation calls is critical in order to achieve the effect we need.

Transformations are mathematical operations that map 3D space to 3D space, and so mathematics has standard ways to represent them. This is discussed in the next chapter, and processes such as composite transformations are linked to the standard operations on these objects.

**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 \( \alpha \) relative to the \( Z \)-axis, you could start with a cube one unit on a side 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 \( A \), \( B \), and \( C \),
2. second, rotate the cube by the angle \( \alpha \) 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. 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, \( f(g(x)) \neq g(f(x)) \)). Unless you have some experience with noncommutative operations from a course such as linear algebra, 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 \(90^\circ\) 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 \((3, 1, 0)\). 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) 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 in two different orders

```plaintext
rotate(60, 0, 0, 1) rotate(90, 0, 0, 1) scale(3, 1, .5) cube()
rotate(90, 0, 1, 0) rotate(60, 0, 0, 1) scale(3, 1, .5) cube()
```

3/15/03
then the results are quite different, as is shown in Figure 2.10.

Figure 2.10: the results from two different orderings of the same rotations

Transformations are implemented as matrices for computational purposes. Recall that we are able to represent points as 4-tuples of real numbers; transformations are implemented as 4x4 matrices that map the space of 4-tuples into itself. Although we will not explicitly use this representation in our work, it is used by graphics APIs and helps explain how transformations work; for example, you can understand why transformations are not commutative by understanding that matrix multiplication is not commutative. (Try it out for yourself!) And if we realize that a 4x4 matrix is equivalent to an array of 16 real numbers, we can think of transformation stacks as stacks of such matrices. While this book does not require matrix operations for transformations, there may be times when you’ll need to manipulate transformations in ways that go beyond your API, so be aware of this.

When it comes time to apply transformations to your models, we need to think about how we represent the problem for computational purposes. Mathematical notation can be applied in many ways, so your previous mathematical experience may or may not help you very much in deciding how you can think about this problem. In order to have a good model for thinking about complex transformation sequences, we will 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. We can also think about this is in terms of building composite functions by multiplying the individual functions, and with the convention above 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:

```
translate * rotate * scale * geometry
```

or, thinking of multiplication as function composition, as

```
translate(rotate(scale(geometry)))
```

This might be implemented by a sequence of function calls like that below that is not intended to represent any particular API:

```
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
```

At first glance, this sequence looks to be exactly the opposite of the sequence noted above. In fact, however, 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 2.11 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:

$$P \ V \ T_0 \ T_1 \ T_2 \ ... \ T_n \ T_{n+1} \ ... \ T_{\text{last}} \ ... \ \text{geometry}$$

Here, $P$ is the projection transformation, $V$ is the viewing transformation, and $T_0, T_1, ... T_{\text{last}}$ 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 \texttt{reshape} function; the viewing transformation is defined in the \texttt{init} function or at the beginning of the \texttt{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 $T_0, T_1, ... T_{\text{last}}$ 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

Creating a model and transforming it into world space can take a good deal of work. You may need to compute vertex coordinates in model space and you will need to apply modeling transformations to these coordinates to get the final vertex coordinates in world space that are finally sent to the projection and rendering processes. If the model is used frequently, and if it must be re-calculated each time it is drawn, it can make a scene quite slow to display. As we will see later, applying a transformation involves a matrix multiplication that could involve as many as 16 operations for each transformation and each vertex, although in practice many transformations can be done with many fewer.

As a way to save time in displaying the image, many graphics APIs allow you to “compile” the geometry in a model in a way that will allow it to be displayed much more quickly. This compiled geometry is basically what is sent to the rendering pipeline as the display list, as described in Chapter 8 below. When the compiled model is displayed, no re-calculation of vertices and no computation of transformations are needed, and only the saved results of these computations is sent to the graphics system. Geometry that is to be compiled should be carefully chosen so that it is not changed between displays; if changes are needed, you will need to re-compile the object. Once you have seen what parts you can compile, you can compile them and use the compiled versions to make the display faster. We will discuss how OpenGL compiles geometry in the next chapter. If you use another graphics API, look for details in its documentation.

An example

To help us see how you can make useful graphical objects from simple modeling, let’s consider a 3D arrow that we could use to point out things in a 3D scene. Our goal is to make an arrow such as the one in Figure 2.12, with the arrow oriented downward and aligned with the Y-axis. An

Figure 2.12: the 3D arrow in standard position
A vector like this could be used easily by scaling it as needed, rotating it to orient it as desired, and then translating it to have its point wherever it's needed.

In order to make this vector, we start with two simpler shapes that are themselves useful. (These are sufficiently useful that they are provided as built-in functions in the GLU and GLUT toolkits we describe in the next chapter.) These simpler shapes are designed to be in standard positions and have standard sizes.

The first of these simple shapes is a cylinder. We will design this as a template that can be made to have any size and any orientation by using simple transformations. Our template orientation will be to have the centerline of the cylinder parallel as X-axis and our template size will be to have the cylinder have radius 1 and length 1. We will design the cylinder to have the cross-section of a regular polygon with $NSIDES$ sides. This will look strange, but will be easy to scale. The template is shown in the left-hand side of Figure 2.13, and a sketch of the code for the cylinder template function cylinder() will look like:

```c
angle = 0.;
anglestep = 360. / (float)NSIDES;
for (i = 0; i < NSIDES; i++) {
    nextangle = angle + anglestep;
    beginQuad();
    vertex(0., cos(angle), sin(angle));
    vertex(1., cos(angle), sin(angle));
    vertex(1., cos(nextangle), sin(nextangle));
    vertex(0., cos(nextangle), sin(nextangle));
    endQuad();
    angle = nextangle;
}
```

![Figure 2.13: the templates of the parts of the arrow: the cylinder (left) and the cone (right)](image)

The second simple shape is a cone whose centerline is the Y-axis with a base of radius 1 and a height of 1. As with the cylinder, this template will be easy to scale and orient as needed for various uses. We will again use a regular polygon of $NSIDES$ sides for the base of the cone. The template shape is shown in the right-hand side of Figure 2.13, and a sketch of the code for the cone template function cone() will look like:

```c
angle = 0.;
anglestep = 360. / (float)NSIDES;
beginTriangleFan();
vertex(0., 1., 0.);
```

![Figure 2.13: the templates of the parts of the arrow: the cylinder (left) and the cone (right)](image)
for (i = 0; i < NSIDES; i++)
    vertex(cos(angle), 0., sin(angle));
endTriangleFan();
angle = 0.;
beginPolygon();
    for (i = 0; i < NSIDES; i++)
        vertex(cos(angle), 0., sin(angle));
endPolygon();

With both template shapes implemented, we can then build a template arrow function arrow3D() as sketched below. We will use a cylinder twice as long and with half the radius of the original cylinder template, oriented along the Z-axis, and the cone in its original form, to form the shape of an arrow. We will then move that shape so its point is at the origin, and will rotate the shape to lie along the Z-axis as defined.

    // place and orient arrow as a whole
    pushTransformStack();
    rotate(180., 1., 0., 0.);  // 180 degrees around x-axis
    translate(-1., 0., 0.);     // move arrow so point is at origin
    // scale and orient cylinder part of arrow
    pushTransformStack();
    rotate(90., 0., 1., 0.);   // 90 degrees around z-axis
    scale( 2., 0.5, 0.5);
    cylinder();
    popTransformStack();
    // now use the cone as defined without any transforms
    cone();
    popTransformStack();

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.
Scene Graphs and Modeling Graphs

Introduction

In this chapter, we have defined 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 programmers may find considerable difficulty with modeling 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. This is even more difficult when the eye point is one of the moving or hierarchically-organized parts. Hierarchical modeling has long been done by using trees or tree-like structures to organize the components of the model, and we will find this kind of approach to be very useful.

Recent graphics systems, such as Java3D and VRML 2, have formalized the concept of a scene graph as a powerful tool both for modeling scenes and for organizing the drawing 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 we will introduce the scene graph structure and will adapt it to a slightly simplified modeling graph that you can use to design scenes. We will also 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 with 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 2.14 will give us a basis to discuss the general approach to graph-structured modeling as it can be applied to beginning computer graphics. Remember that we will be simplifying some aspects of this graph before applying it to our modeling.

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, as seen in the left-hand branch of Figure 2.14. 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.

Figure 2.14: the structure of the scene graph as defined in Java3D

The view branch of the scene graph includes the specification of the display device, and thus the projection appropriate for that device, as shown in the right-hand branch of Figure 2.14. 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 2.14, 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.) 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 in 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.

Figure 2.15: a scene that we will describe with a scene graph

The initial analysis of the scene in Figure 2.15, organized along the lines of view and content branches, leads to an initial (and partial) graph structure shown in Figure 2.16. The content branch

![Scene Graph Diagram]

Figure 2.16: a scene graph that organizes the modeling of our simple scene
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 the scene graph, we are not nearly finished. The initial scene graph of Figure 2.16 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 2.15 and shown below in Figure 2.17) or relative to other components of the model (as described later in the chapter), 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 2.17.

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 2.16 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. It is, of course, quite possible that in some models a light or the eye might be attached to a group instead of being positioned independently, and this can lead to some interesting examples that we describe later. 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 several other kinds of operations. Eventually each vertex of the geometry will have not only geometry, in terms of its coordinates, but also normal components, texture coordinates, and several other properties. Here, however, we are primarily concerned with the geometry content of the shape node; much of the rest of these notes is devoted to building the appearance properties of the shape node, because the appearance content is perhaps the most important part of graphics for building high-quality images.

![Scene Graph Diagram](image)

Figure 2.18: the scene graph after integrating the viewing transformation into the content branch

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 2.17 as shown in Figure 2.18 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.

The scene graph for a particular image is not unique because there are many ways to organize a scene and many ways to organize the way you carry out the graphic operations that the scene graph specifies. Once you have written a first scene graph for a scene, you may want to think some more about the scene to see whether there is another way to organize the scene graph to create a more efficient program from the scene graph or to make the scene graph present a more clear description of the scene. Remember that the scene graph is a design tool, and there are always many ways to create a design for any problem.

It is very important to note that the scene graph need not describe a static geometry. The transformations in the scene graph may be defined with parameters instead of constant values, and event callbacks can affect the graph by controlling these parameters through user interaction or through computed values. This is discussed 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 information on the three key kinds of 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. Because this is so straightforward, we really do not need to include it in our graph. The viewing transformation is readily created from the transformation information in the view by analyzing the eye placement transformations as we saw above, so it is straightforward to extract this and, more important, to create this transformation from the inversed of the eyepoint transformations. This is discussed in the next section of the chapter. Finally, the modeling transformations for the various components are built by working with the various transformations in the content branch as the components are drawn, and are discussed later in this chapter.

Because all the information we need for both the primitive geometry and all the transformations is held in this simple graph, we will call it the modeling graph for our scene. This modeling graph, basically a scene graph without a view branch but with the viewing information organized at the top as the inverse of the eyepoint placement transformations, will be the basis for the coding of our scenes as we describe in the remainder of the chapter.

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 $T_1T_2T_3\ldots T_K$ is the original transformation sequence, the inverse is $T_K^u\ldots T_3^uT_2^uT_1^u$ 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 \( \text{rotate}(\text{angle}, \text{line}) \) that rotates space by the value \( \text{angle} \) 
around the fixed line \( \text{line} \), we must simply rotate the space by the same angle in the reverse 
direction. Thus the inverse of the rotation transformation is \( \text{rotate}(-\text{angle}, \text{line}) \).

For the translation transformation \( \text{translate}(\text{Tx}, \text{Ty}, \text{Tz}) \) 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 
\( \text{translate}(-\text{Tx}, -\text{Ty}, -\text{Tz}) \).

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(angle, line)
    scale(Sx, Sy, Sz)
\end{verbatim}

is the set of operations

\begin{verbatim}
    scale(1/Sx, 1/Sy, 1/Sz)
    rotate(-angle, line)
    translate(-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.

![Figure 2.19: the same scene as in Figure 2.15 but with the eye point following directly behind the helicopter](image)

In Figure 2.19 we show the change that results in the view of Figure 2.15 when we define the eye 
to be immediately behind the helicopter, and in Figure 2.20 we show the change in the scene graph.
of Figure 2.17 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. You might, for example, have a menu switch that specified that the eye was to be at a fixed point or at a point following the helicopter; then the code for inverting the eye position would be a switch statement that implemented the appropriate transformations depending on the menu choice. This is how the scene graph will help you to organize the viewing process that was described in the earlier chapter on viewing.

![Figure 2.20: the change in the scene graph of Figure 2.11 to implement the view in Figure 2.15](image)

With this scene graph, we can identify the set of transformations \( T_aT_bT_cT_d \ldots T_jT_k \) that are applied to put the helicopter in the scene, and the transformations \( T_uT_vT_w \ldots T_z \) that place the eye point relative to the helicopter. The implementation of the structure of Figure 2.18, then, is to begin the display code with the standard view, followed by \( T_z^{-1}T_v^{-1}T_u^{-1} \) and then \( T_k^{-1}T_j^{-1}T_i^{-1} \ldots T_d^{-1}T_c^{-1}T_b^{-1}T_a^{-1} \), before you begin to write the code for the standard scene as described in Figure 2.21 below.

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 2.18 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.
The scene graph and depth testing

In almost all of the images we expect to create, we would use the hidden-surface abilities provided by our graphics API. As we described in the last chapter, this will probably use some sort of depth buffer or Z-buffer, and the comparisons of depths for hidden surface resolution is done as the parts of the scene are drawn.

However, there may be times when you will want to avoid depth testing and take control of the sequence of drawing your scene components. One such time is described later in the chapter on color and blending, where you need to create a back-to-front drawing sequence in order to simulate transparency with blending operations. In order to do this, you will need to know the depth of each of the pieces of your scene, or the distance of that piece from the eye point. This is easy enough to do if the scene is totally static, but when you allow pieces to move or the eye to move, it becomes much less simple.

The solution to this problem lies in doing a little extra work as you render your scene. Before you actually draw anything, but after you have updated whatever transformations you will use and whatever choices you will make to draw the current version of the scene, apply the same operations but use a tool called a projection that you will find with most graphics APIs. The projection operation allows you to calculate the coordinates of any point in your model space when it is transformed by the viewing and projection transformations into a point in 3D eye space. The depth of that point, then, is simply the Z-coordinate of the projected value. You can draw the entire scene, then, using the projection operation instead of the rendering operation, get the depth values for each piece of the scene, and use the depth values to determine the order in which you will draw the parts. The scene graph will help you make sure you have the right transformations when you project each of the parts, ensuring that you have the right depth values.

Using the modeling graph for coding

Because the modeling graph as we defined it above is intended as a learning tool and not a production 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 rewrite 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 rewrite guidelines are as follows, including the rewrites 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 2.19 above, we would use the tree to write code as shown in skeleton form in Figure 2.21. 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.
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.

```
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 2.21: code sketch to implement the modeling in Figure 2.20

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 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 2.22. 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.

![Figure 2.22: the rabbit’s head](image)

The modeling graph for the rabbit’s head is shown in Figure 2.23. 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 2.23: the modeling graph for 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.

**Summary**

You have seen all the concepts you need for polygon-based modeling, as used in many graphics APIs. You know how to define an object in model space (that is, in a 3D space that is set up just for the object) in terms of graphics primitives such as points, line segments, triangles, quads, and polygons; how to apply the modeling transformations of scaling, translation, and rotation to place objects into a common world space so that the viewing and projection operations can be applied to them; and how to organize a hierarchy of objects in a scene with the scene graph so that the code for the scene can be written easily. You also know how to change transformations so that you can add motion to a scene. You are now ready to look at how the OpenGL graphics API implements these concepts so you can begin doing solid graphics programming, and we will take you there in the next chapter.

**Questions**

1. We know that we can model any polyhedron with triangles, but why can you model a sphere with triangle fans for the polar caps and quad strips for the rest of the object?

2. Put yourself in a familiar environment, but imagine the environment simplified so that it is made up of only boxes, cylinders, and other very basic shapes. Imagine further that your environment has only one door and that everything in the room has to come in that door. Write out the sequence of transformations that would have to be done to place everything in its place in your environment. Now imagine that each of these basic shapes starts out as a standard shape: a unit cube, a cylinder with diameter one and height one, and the like; write out the sequence of transformations that would have to be done to make each object from these basic objects. Finally, if the door would only admit basic objects, put together these two processes to write out the full transformations to create the objects and place them in the space.

3. Now take the environment above and write a scene graph that describes the whole scene, using the basic shapes and transformations you identified in the previous question. Also place your eye in the scene graph starting with a standard view of you standing in the doorway and facing directly into the room. Now imagine that on a table in the space there is a figure of a ballerina spinning around and around, and identify the way the transformations in the scene graph would handle this moving object.

**Exercises**

4. Calculate the coordinates of the vertices of the simpler regular polyhedra: the cube, the tetrahedron, and the octagon. For the octagon and tetrahedron, try using spherical coordinates and converting them to rectangular coordinates; see the chapter on mathematics for modeling.

5. Verify that for any $x, y, z,$ and $w$, the point $(x/w, y/w, z/w, 1)$ is the intersection of the line segment from $(x, y, z, w)$ to $(0, 0, 0, 0)$, and the hyperplane \{(a, b, c, 1) \mid \text{arbitrary } a, b, c \}. Show that this means that an entire line in 4D space is represented by a single point in homogeneous coordinates in 3D space.

6. Show how you can define a cube as six quads. Show how you can refine that definition to write a cube as two quad strips. Can you write a cube as one quad strip?
7. Show how you can write any polygon, convex or not, as a set of triangles. Show further how you can write any convex polygon as a triangle fan. Does it matter which vertex you pick as the first vertex in the triangle fan?

8. Define a polygon in 2D space that is reasonably large and having a side that is not parallel to one of the axes. Find a unit square in the 2D space that intersects that side, and calculate the proportion of the polygon that lies within the unit square. If the square represents a pixel, draw conclusions about what proportion of the pixel’s color should come from the polygon and what proportion from the background.

9. The code for the normals to a quad on a sphere as shown in Figure 2.7 is not accurate because it uses the normal at a vertex instead of the normal in the middle of the quad. How should you calculate the normal so that it is the face normal and not a vertex normal?

10. Make a basic object with no symmetries, and apply simple rotation, simple translation, and simple scaling to it; compare the results with the original object. Then apply second and third transformations after you have applied the first transformation and again see what you get. Show why the order of the transformations matters by applying the same transformations in different order and seeing the results.

11. Scene graphs are basically trees, though different branches may share common shape objects. As trees, they can be traversed in any way that is convenient. Show how you might choose the way you would traverse a scene graph in order to draw back-to-front if you knew the depth of each object in the tree.

12. Add a mouth and tongue to the rabbit’s head, and modify the scene graph for the rabbit’s head to have the rabbit stick out its tongue and wiggle it around.

13. Define a scene graph for a carousel, or merry-go-round. This object has a circular disk as its floor, a cone as its roof, a collection of posts that connect the floor to the roof, and a collection of animals in a circle just inside the outside diameter of the floor, each parallel to the tangent to the floor at the point on the edge nearest the animal. The animals will go up and down in a periodic way as the carousel goes around. You may assume that each animal is a primitive and not try to model it, but you should carefully define all the transformations that build the carousel and place the animals.

Experiments

14. Get some of the models from the avalon.viewpoint.com site and examine the model file to see how you could present the model as a sequence of triangles or other graphics primitives.

15. Write the code for the scene graph of the familiar space from question 3, including the code that manages the inverse transformations for the eye point. Now identify a simple path for the eye, created by parametrizing some of the transformations that place the eye, and create an animation of the scene as it would be seen from the moving eye point.

As we saw in the problems for Chapter 1, the general function

```c
glGetFloatv(GL_MODELVIEW_MATRIX, v)
```

can be used to retrieve the 16 real-number values of the modelview matrix and store them in an array v defined by

```c
GLfloat v[4][4];
```

If we leave the viewing transformation in its default state and apply modeling transformations one at a time, we can get the values of the various modeling matrices using this technique. In each of
the problems below, be sure that your modeling transformation is the only thing on the modelview
stack by setting the modelview matrix to the identity before calling the modeling transform function
you are looking at. It will help greatly if you write a function that will display a 4x4 array nicely so
you can see the elements easily. The matrices you produce in this section should be compared with
the matrices for scaling, rotation, and translation that are described in the chapter below on the
mathematical basis for modeling.

As we did in the similar experiments in Chapter 1, once you have returned the modelview matrix
for the simple transformations, you should change appropriate values in the matrix and re-set the
modelview matrix to this modified matrix. You should then re-draw the figure with the modified
matrix and compare the effects of the original and modified matrix to see the graphic effects, not
just the numerical effects.

16. Start with a simple scaling, set for example with the function glScalef(α, β, γ), and
then get the values of the modelview matrix. You should be able to see the scaling values as
the diagonal values in this matrix. Try using different values of the scale factors and first get
and then print out the matrix in good format.

17. Do as above for a rotation, set for example with the function glRotatef(α, x, y, z)
where x, y, and z are set to be able to isolate the rotation by the angle α around individual axes.
For the x-axis, for example, set x = 1 and y = z = 0. Print out the matrix in good format and
identify the components of the matrix that come from the angle α through trigonometric
functions. Hint: use some simple angles such as 30°, 45°, or 60°.

18. Do as above for a translation, set for example with the function glTranslatef(α, β, γ),
and then get the values for the modelview matrix. Identify the translation values as a column of
values in the matrix. Experiment with different translation values and see how the matrix
changes.

19. Now that you have seen the individual modeling matrices, combine them to see how making
composite transformations compares with the resulting matrix. In particular, take two of the
simple transformations you have examined above and compose them, and see if the matrix of
the composite is the product of the two original matrices. Hint: you may have to think about
the order of multiplication of the matrices.

20. We claimed that composing transformations was not commutative, and justified our statement
by noting that matrix multiplication is not commutative. However, you can verify this much
more directly by composing two transformations and getting the resulting matrix, and then
composing the transformations in reverse order and getting the resulting matrix. The two
matrices should not be equal under most circumstances; check this and see. If you happened to
get the matrices equal, check whether your simple transformations might not have been too
simple and if so, make them a bit more complex and try again.

Projects

21. (A scene graph parser) Define a scene graph data structure as a graph (or tree) with nodes that
have appropriate modeling or transformation statements. For now, these can be pseudocode as
we have used it in this chapter. Write a tree walker that generates the appropriate sequence of
statements to present a scene to a graphics API. Can you see how to make some of the
transformations parametric so you can generate motion in the scene? Can you see how to
generate the statements that invert the eyepoint placement transformations if the eye point is not
in standard position?