5

MOVING AROUND IN SPACE: COORDINATE TRANSFORMATIONS

by Richard S. Wright, Jr.

What you'll learn in this chapter:

How to...	Functions You'll Use
Establish your position in the scene	`gluLookAt/glTranslate/glRotate`
Position objects within the scene	`glTranslate/glRotate`
Scale objects	`glScale`
Establish a perspective transformation	`gluPerspective`
Perform your own matrix transformations	`glLoadMatrix/glMultMatrix`

In Chapter 4, "Drawing in Space: Lines, Points, and Polygons," you learned how to draw points, lines, and various primitives in 3D. To turn a collection of shapes into a coherent scene, you must arrange them in relation to one another and to the viewer. In this chapter, you'll start moving shapes and objects around in your coordinate system. (Actually, you don't move the objects, but rather shift the coordinate system to create the view you want.) The ability to place and orient your objects in a scene is a crucial tool for any 3D graphics programmer. As you will see, it is actually convenient to describe your objects' dimensions around the origin and then translate and rotate the objects into the desired position.

Is This the Dreaded Math Chapter?

In most books on 3D graphics programming, yes, this would be the dreaded math chapter. However, you can relax; we take a more moderate approach to these principles than some texts.

The keys to object and coordinate transformations are two modeling matrices maintained by OpenGL. To familiarize you with these matrices, this chapter strikes a compromise between two extremes in computer graphics philosophy. On one hand, we could warn you, "Please review a textbook on linear algebra before reading this chapter." On the other hand, we could perpetuate the deceptive reassurance that you can "learn to do 3D graphics without all those complex mathematical formulas." But we don't agree with either camp.

In reality, you can get along just fine without understanding the finer mathematics of 3D graphics, just as you can drive your car every day without having to know anything at all about automotive mechanics and the internal combustion engine. But you'd better know enough about your car to realize that you need an oil change every so often, that you have to fill the tank with gas regularly, and that you must change the tires when they get bald. This knowledge makes you a responsible (and safe!) automobile owner. If you want to be a responsible and capable OpenGL programmer, the same standards apply. You want to understand at least the basics so you know what can be done and what tools best suit the job.

Even if you don't have the ability to multiply two matrices in your head, you need to know what matrices are and that they are the means to OpenGL's 3D magic. But before you go dusting off that old linear algebra textbook (doesn't everyone have one?), have no fear: OpenGL does all the math for you. Think of it as using a calculator to do long division when you don't know how to do it on paper. Although you don't have to do it yourself, you still know what it is and how to apply it. See—you can have your cake and eat it too!

That said, a basic understanding of 3D math and transformations can be a powerful and essential tool for real-time programming. If you think we are going to treat this subject superficially, fear not! In Chapter 19, "Real-Time Programming with OpenGL," we go a little deeper into understanding the mechanics of geometry transformation. This understanding is a requirement for high-performance 3D graphics, but not necessarily a requirement to start doing useful work. In most of this book, we cover walking; in Chapter 19, we take all this knowledge and show you how to run with it.

Understanding Transformations

Transformations make possible the projection of 3D coordinates onto a 2D screen. Transformations also allow you to rotate objects around, move them about, and even stretch, shrink, and warp them. Rather than modify your object directly, a transformation modifies the coordinate system. Once a transformation rotates the coordinate system, the object appears rotated when it is drawn. Three types of transformations

occur between the time you specify your vertices and the time they appear on the screen: viewing, modeling, and projection. In this section, we examine the principles of each type of transformation, which you will find summarized in Table 5.1.

Table 5.1 Summary of the OpenGL Transformations

Transformation	Use
Viewing	Specifies the location of the viewer or camera
Modeling	Moves objects around the scene
Modelview	Describes the duality of viewing and modeling transformations
Projection	Clips and sizes the viewing volume
Viewport	Scales the final output to the window

Eye Coordinates

An important concept throughout this chapter is that of eye coordinates. *Eye coordinates* are from the viewpoint of the observer, regardless of any transformations that may occur; think of them as "absolute" screen coordinates. Thus, eye coordinates are not real coordinates, but rather represent a virtual fixed coordinate system that is used as a common frame of reference. All of the transformations discussed in this chapter are described in terms of their effects relative to the eye coordinate system.

Figure 5.1 shows the eye coordinate system from two viewpoints. On the left (a), the eye coordinates are represented as seen by the observer of the scene (that is, perpendicular to the monitor). On the right (b), the eye coordinate system is rotated slightly so you can better see the relation of the z-axis. Positive x and y are pointed right and up, respectively, from the viewer's perspective. Positive z travels away from the origin toward the user, and negative z values travel farther away from the viewpoint into the screen.

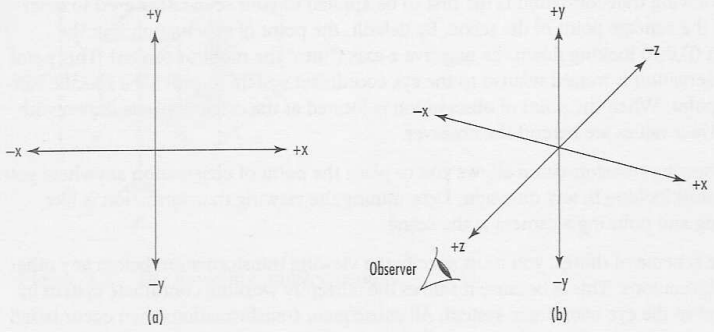

Figure 5.1 Two perspectives of eye coordinates.

When you draw in 3D with OpenGL, you use the Cartesian coordinate system. In the absence of any transformations, the system in use is identical to the eye coordinate system. All of the various transformations change the current coordinate system with respect to the eye coordinates. This, in essence, is how you move and rotate objects in your scene—by moving and rotating the coordinate system with respect to eye coordinates. Figure 5.2 gives a two-dimensional example of the coordinate system rotated 45° clockwise by eye coordinates. A square plotted on this rotated coordinate system would also appear rotated.

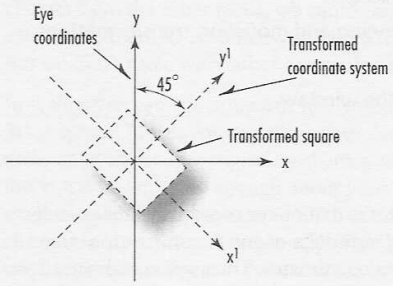

Figure 5.2 A coordinate system rotated with respect to eye coordinates.

In this chapter, you'll study the methods by which you modify the current coordinate system before drawing your objects. You can even save the state of the current system, do some transformations and drawing, and then restore the state and start over again. By chaining these events together, you will be able to place objects all about the scene and in various orientations.

Viewing Transformations

The viewing transformation is the first to be applied to your scene. It is used to determine the vantage point of the scene. By default, the point of observation is at the origin (0,0,0) looking down the negative z-axis ("into" the monitor screen). This point of observation is moved relative to the eye coordinate system to provide a specific vantage point. When the point of observation is located at the origin, objects drawn with positive z values are behind the observer.

The viewing transformation allows you to place the point of observation anywhere you want and looking in any direction. Determining the viewing transformation is like placing and pointing a camera at the scene.

In the scheme of things, you must specify the viewing transformation before any other transformations. This is because it moves the currently working coordinate system in respect to the eye coordinate system. All subsequent transformations then occur based

on the newly modified coordinate system. Later, you'll see more easily how this works, when we actually start looking at how to make these transformations.

Modeling Transformations

Modeling transformations are used to manipulate your model and the particular objects within it. These transformations move objects into place, rotate them, and scale them. Figure 5.3 illustrates three modeling transformations that you will apply to your objects. Figure 5.3a shows translation, where an object is moved along a given axis. Figure 5.3b shows a rotation, where an object is rotated about one of the axes. Finally, Figure 5.3c shows the effects of scaling, where the dimensions of the object are increased or decreased by a specified amount. Scaling can occur nonuniformly (the various dimensions can be scaled by different amounts), so you can use scaling to stretch and shrink objects.

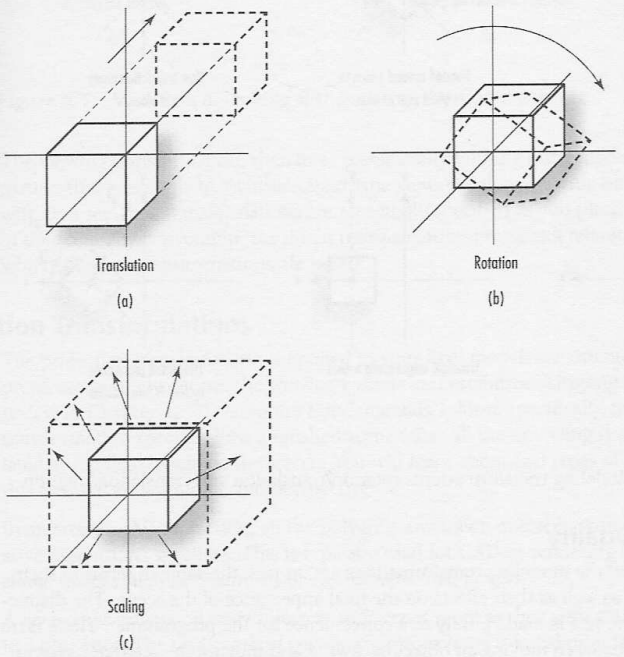

Figure 5.3 The modeling transformations.

The mathematics behind these transformations are greatly simplified by the mathematical notation of the matrix. You can achieve each of the transformations we have discussed by multiplying a matrix that contains the vertices by a matrix that describes the transformation. Thus, all the transformations achievable with OpenGL can be described as a multiplication of two or more matrices.

What Is a Matrix?

A matrix is nothing more than a set of numbers arranged in uniform rows and columns—in programming terms, a two-dimensional array. A matrix doesn't have to be square, but each row or column must have the same number of elements as every other row or column in the matrix. Figure 5.7 presents some examples of matrices. (These don't represent anything in particular but only serve to demonstrate matrix structure.) Note that it is valid for a matrix to have a single column.

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \quad \begin{bmatrix} 0 & 42 \\ 1.5 & 0.877 \\ 2 & 14 \end{bmatrix} \quad \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Figure 5.7 Three examples of matrices.

Matrix transformations are actually not difficult to understand but can be intimidating. Because an understanding of matrix transformations is fundamental to many 3D tasks, you should still make an attempt to become familiar with them later. However, because you don't need to understand this topic to use OpenGL, we won't devote the space required here. Appendix B, "Further Reading," lists some good texts on this subject.

The Transformation Pipeline

To effect the types of transformations described in this chapter, you will modify two matrices in particular: the modelview matrix and the projection matrix. Don't worry, OpenGL gives you some high-level functions that you can call for these transformations. After you've mastered the basics of the OpenGL API, you will undoubtedly start trying some of the more advanced 3D rendering techniques. Only then will you need to call the lower-level functions that actually set the values contained in the matrices.

The road from raw vertex data to screen coordinates is a long one. Figure 5.8 is a flowchart of this process. First, your vertex is converted to a 1-by-4 matrix in which the first three values are the x, y, and z coordinates. The fourth number is a scaling factor that you can apply manually by using the vertex functions that take four values. This is the w coordinate, usually 1.0 by default. You will seldom modify this value directly.

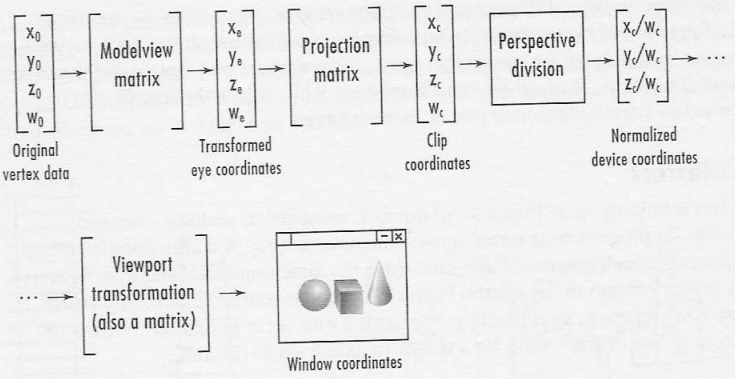

Figure 5.8 The vertex transformation pipeline.

The vertex is then multiplied by the modelview matrix, which yields the transformed eye coordinates. The eye coordinates are then multiplied by the projection matrix to yield clip coordinates. This effectively eliminates all data outside the viewing volume. The clip coordinates are then divided by the w coordinate to yield normalized device coordinates. The w value may have been modified by the projection matrix or the modelview matrix, depending on the transformations that may have occurred. Again, OpenGL and the high-level matrix functions hide all this from you.

Finally, your coordinate triplet is mapped to a 2D plane by the viewport transformation. This is also represented by a matrix, but not one that you will specify or modify directly. OpenGL sets it up internally depending on the values you specified to `glViewport`.

The Modelview Matrix

The modelview matrix is a 4×4 matrix that represents the transformed coordinate system you are using to place and orient your objects. The vertices you provide for your primitives are used as a single-column matrix and multiplied by the modelview matrix to yield new transformed coordinates in relation to the eye coordinate system.

In Figure 5.9, a matrix containing data for a single vertex is multiplied by the modelview matrix to yield new eye coordinates. The vertex data is actually four elements with an extra value w that represents a scaling factor. This value is set by default to 1.0, and rarely will you change this yourself.

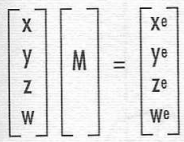

Figure 5.9 Matrix equation that applies the modelview transformation to a single vertex.

Translation

Let's take an example that modifies the modelview matrix. Say you want to draw a cube using the GLUT library's `glutWireCube` function. You simply call

```
glutWireCube(10.0f);
```

and you have a cube centered at the origin that measures 10 units on a side. To move the cube up the y-axis by 10 units before drawing it, you multiply the modelview matrix by a matrix that describes a translation of 10 units up the y-axis and then do your drawing. In skeleton form, the code looks like this:

```
// Construct a translation matrix for positive 10 Y
...

// Multiply it by the modelview matrix
...

// Draw the cube
glutWireCube(10.0f);
```

Actually, such a matrix is fairly easy to construct, but it requires quite a few lines of code. Fortunately, a high-level function is provided that does this for you:

```
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
```

This function takes as parameters the amount to translate along the x, y, and z directions. It then constructs an appropriate matrix and does the multiplication. Now, the pseudo code looks like the following, and the effect is illustrated in Figure 5.10.

```
// Translate up the y-axis 10 units
glTranslatef(0.0f, 10.0f, 0.0f);

// Draw the cube
glutWireCube(10.0f);
```

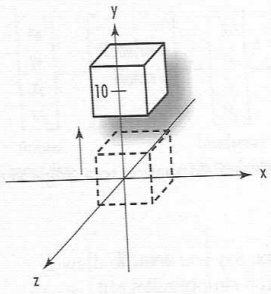

Figure 5.10 A cube translated 10 units in the positive y direction.

Rotation

To rotate an object about one of the three axes, you have to devise a rotation matrix to be multiplied by the modelview matrix. Again, a high-level function comes to the rescue:

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

Here, we perform a rotation around the vector specified by the x, y, and z arguments. The angle of rotation is in the counterclockwise direction measured in degrees and specified by the argument angle. In the simplest of cases, the rotation is around one of the axes, so you need to specify only that value.

You can also perform a rotation around an arbitrary axis by specifying x, y, and z values for that vector. To see the axis of rotation, you can just draw a line from the origin to the point represented by (x,y,z). The following code rotates the cube by 45° around an arbitrary axis specified by (1,1,1), as illustrated in Figure 5.11:

```
// Perform the transformation
glRotatef(90.0f, 1.0f, 1.0f, 1.0f);

// Draw the cube
glutWireCube(10.0f);
```

Scaling

A scaling transformation increases the size of your object by expanding all the vertices along the three axes by the factors specified. The function

```
glScalef(GLfloat x, GLfloat y, GLfloat z);
```

multiplies the x, y, and z values by the scaling factors specified.

The final appearance of your scene or object can depend greatly on the order in which the modeling transformations are applied. This is particularly true of translation and rotation. Figure 5.4a shows the progression of a square rotated first about the z-axis and then translated down the newly transformed x-axis. In Figure 5.4b, the same square is first translated down the x-axis and then rotated around the z-axis. The difference in the final dispositions of the square occurs because each transformation is performed with respect to the last transformation performed. In Figure 5.4a, the square is rotated with respect to the origin first. In 5.4b, after the square is translated, the rotation is performed around the newly translated origin.

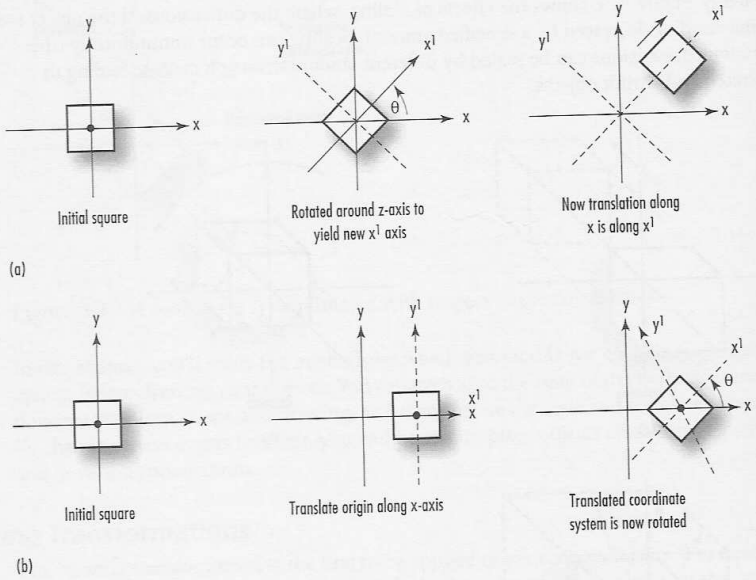

Figure 5.4 Modeling transformations: rotation/translation and translation/rotation.

The Modelview Duality

The viewing and the modeling transformations are, in fact, the same in terms of their internal effects as well as their effects on the final appearance of the scene. The distinction between the two is made purely as a convenience for the programmer. There is no real difference between moving an object backward and moving the reference system forward; as shown in Figure 5.5, the net effect is the same. (You experience this first-hand when you're sitting in your car at an intersection and you see the car next to you roll forward; it might seem to you that your own car is rolling backwards.) The term

modelview indicates that you can think of this transformation either as the modeling transformation or the viewing transformation, but in fact there is no distinction; thus, it is the modelview transformation.

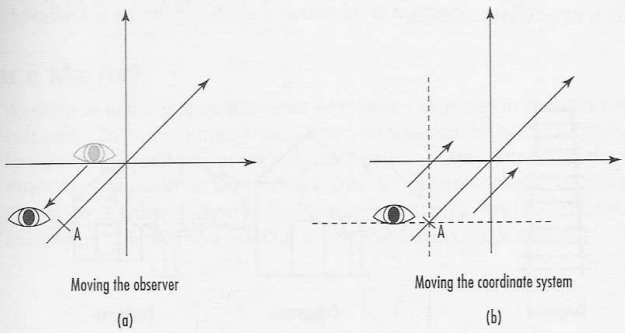

Figure 5.5 Two ways of looking at the viewing transformation.

The viewing transformation, therefore, is essentially nothing but a modeling transformation that you apply to a virtual object (the viewer) before drawing objects. As you will soon see, new transformations are repeatedly specified as you place more objects in the scene. By convention, the initial transformation provides a reference from which all other transformations are based.

Projection Transformations

The projection transformation is applied to your final modelview orientation. This projection actually defines the viewing volume and establishes clipping planes (see the review in Chapter 1, "3D Graphics Fundamentals"). More specifically, the projection transformation specifies how a finished scene (after all the modeling is done) is translated to the final image on the screen. You will learn about two types of projections in this chapter: orthographic and perspective.

In an *orthographic* projection, all the polygons are drawn onscreen with exactly the relative dimensions specified. This is typically used for CAD or rendering two-dimensional images such as blueprints or two-dimensional graphs.

A *perspective* projection shows objects and scenes more as they appear in real life than in a blueprint. The trademark of perspective projections is foreshortening, which makes distant objects appear smaller than nearby objects of the same size. Lines in 3D space that might be parallel do not always appear parallel to the viewer. In a railroad track, for instance, the rails are parallel, but with perspective projection, they appear to converge at some distant point. We call this point the vanishing point.

The benefit of perspective projection is that you don't have to figure out where lines converge or how much smaller distant objects are. All you need to do is specify the scene using the modelview transformations and then apply the perspective projection. OpenGL works all the magic for you. Figure 5.6 compares orthographic and perspective projections on two different scenes.

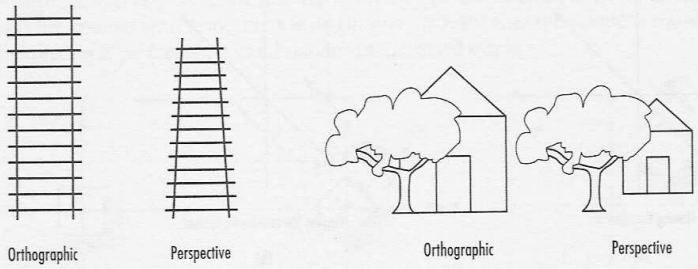

Figure 5.6 Two examples of orthographic versus perspective projections.

Orthographic projections are used most often for 2D drawing purposes where you want an exact correspondence between pixels and drawing units. This might be for a schematic layout, or perhaps a 2D graphing application. You also can use an orthographic projection for 3D renderings when the depth of the rendering has a very small depth in comparison to the distance from the viewpoint. Perspective projections are used for rendering scenes that contain wide open spaces, or objects that need to have the foreshortening effect applied. For the most part, perspective projections are typical.

Viewport Transformations

When all is said and done, you end up with a two-dimensional projection of your scene that will be mapped to a window somewhere on your screen. This mapping to physical window coordinates is the last transformation that is done, and it is called the *viewport* transformation. The viewport was discussed briefly in Chapter 3, "Using OpenGL," where you used it to stretch an image or keep a scene squarely placed in a rectangular window.

Matrix Munching

Now that you're armed with some basic vocabulary and definitions of transformations, you're ready for some simple matrix mathematics. Let's examine how OpenGL performs these transformations and get to know the functions you call to achieve your desired effects.

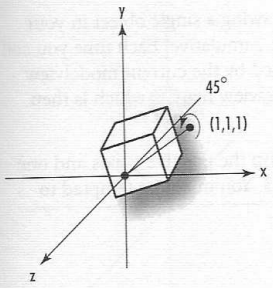

Figure 5.11 A cube rotated about an arbitrary axis.

Scaling does not have to be uniform. You can use it to stretch or squeeze objects as well. For example, the following code produces a cube that is twice as large along the x- and z-axes as the cubes discussed in the previous examples, but still the same along the y-axis. The result is shown in Figure 5.12.

```
// Perform the scaling transformation
glScalef(2.0f, 1.0f, 2.0f);

// Draw the cube
glutWireCube(10.0f);
```

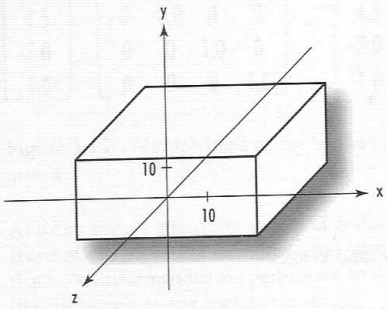

Figure 5.12 A nonuniform scaling of a cube.

The Identity Matrix

You might be wondering about now why we had to bother with all this matrix stuff in the first place. Can't we just call these transformation functions to move our objects around and be done with it? Do we really need to know that it is the modelview matrix that is modified?

The answer is yes and no, (but it's no only if you are drawing a single object in your scene). This is because the effects of these functions are cumulative. Each time you call one, the appropriate matrix is constructed and multiplied by the current modelview matrix. The new matrix then becomes the current modelview matrix, which is then multiplied by the next transformation, and so on.

Suppose you want to draw two spheres—one 10 units up the positive y-axis and one 10 units out the positive x-axis, as shown in Figure 5.13. You might be tempted to write code that looks something like this:

```
// Go 10 units up the y-axis
glTranslatef(0.0f, 10.0f, 0.0f);

// Draw the first sphere
glutSolidSphere(1.0f,15,15);

// Go 10 units out the x-axis
glTranslatef(10.0f, 0.0f, 0.0f);

// Draw the second sphere
glutSolidSphere(1.0f);
```

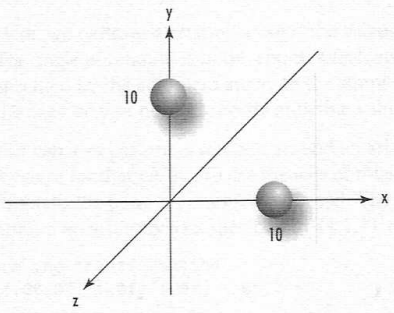

Figure 5.13 Two spheres drawn on the y- and x- axes.

Consider, however, that each call to `glTranslate` is cumulative on the modelview matrix, so the second call translates 10 units in the positive x direction from the previous translation in the y direction. This yields the results shown in Figure 5.14.

You can make an extra call to `glTranslate` to back down the y-axis 10 units in the negative direction, but this makes some complex scenes difficult to code and debug not to mention you throw extra transformation math at the CPU. A simpler method is to reset the modelview matrix to a known state—in this case, centered at the origin of our eye coordinate system.

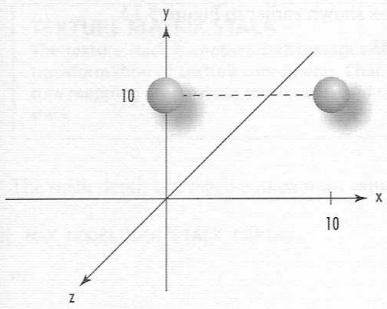

Figure 5.14 The result of two consecutive translations.

You reset the origin by loading the modelview matrix with the identity matrix. The *identity matrix* specifies that no transformation is to occur, in effect saying that all the coordinates you specify when drawing are in eye coordinates. An identity matrix contains all 0s with the exception of a diagonal row of 1s. When this matrix is multiplied by any vertex matrix, the result is that the vertex matrix is unchanged. Figure 5.15 shows this equation.

$$\begin{bmatrix} 8.0 \\ 4.5 \\ -2.0 \\ 1.0 \end{bmatrix} \begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix} = \begin{bmatrix} 8.0 \\ 4.5 \\ -2.0 \\ 1.0 \end{bmatrix}$$

Figure 5.15 Multiplying a vertex by the identity matrix yields the same vertex matrix.

As we've already stated, the details of performing matrix multiplication are outside the scope of this book. For now, just remember this: Loading the identity matrix means that no transformations are performed on the vertices. In essence, you are resetting the modelview matrix back to the origin.

The following two lines load the identity matrix into the modelview matrix:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

The first line specifies that the current operating matrix is the modelview matrix. Once you set the current operating matrix (the matrix that your matrix functions are affecting), it remains the active matrix until you change it. The second line loads the current matrix (in this case, the modelview matrix) with the identity matrix.

Now, the following code produces results as shown earlier in Figure 5.13:

```
// Set current matrix to modelview and reset
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// Go 10 units up the y-axis
glTranslatef(0.0f, 10.0f, 0.0f);

// Draw the first sphere
glutSolidSphere(1.0f, 15, 15);

// Reset modelview matrix again
glLoadIdentity();

// Go 10 units out the x-axis
glTranslatef(10.0f, 0.0f, 0.0f);

// Draw the second sphere
glutSolidSphere(1.0f, 15, 15);
```

The Matrix Stacks

It is not always desirable to reset the modelview matrix to identity before placing every object. Often, you want to save the current transformation state and then restore it after some objects have been placed. This approach is most convenient when you have initially transformed the modelview matrix as your viewing transformation (and thus are no longer located at the origin).

To facilitate this, OpenGL maintains a *matrix stack* for both the modelview and projection matrices. A matrix stack works just like an ordinary program stack. You can push the current matrix onto the stack to save it and then make your changes to the current matrix. Popping the matrix off the stack restores it. Figure 5.16 shows the stack principle in action.

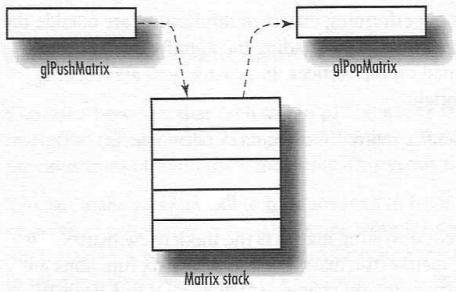

Figure 5.16 The matrix stack in action.

TEXTURE MATRIX STACK
The texture stack is another matrix stack available to the programmer. It is used for the transformation of texture coordinates. Chapter 8, "Texture Mapping," examines texture mapping and texture coordinates and contains a discussion of the texture matrix stack.

The stack depth can reach a maximum value that can be retrieved with a call to either

```
glGet(GL_MAX_MODELVIEW_STACK_DEPTH);
```

or

```
glGet(GL_MAX_PROJECTION_STACK_DEPTH);
```

If you exceed the stack depth, you get a `GL_STACK_OVERFLOW` error; if you try to pop a matrix value off the stack when there is none, you generate a `GL_STACK_UNDERFLOW` error. The stack depth is implementation dependent. For the Microsoft software implementation, the values are 32 for the modelview and 2 for the projection stack.

A Nuclear Example

Let's put to use what we have learned. In the next example, we build a crude, animated model of an atom. This atom has a single sphere at the center to represent the nucleus and three electrons in orbit about the atom. We use an orthographic projection, as we have previously in this book. (Some other interesting projections are covered in the upcoming section, "Using Projections.")

Our ATOM program uses the GLUT timer callback mechanism (discussed in Chapter 3) to redraw the scene about 10 times per second. Each time the `Render` function is called, the angle of revolution about the nucleus is incremented. Also, each electron lies in a different plane. Listing 5.1 shows the `Render` function for this example, and the output from the ATOM program is shown in Figure 5.17.

Listing 5.1 *Render Function from ATOM Sample Program*

```
// Called to draw scene
void RenderScene(void)
    {
    // Angle of revolution around the nucleus
    static float fElect1 = 0.0f;

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

continued on next page

continued from previous page

```
    // Reset the modelview matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();


    // Translate the whole scene out and into view
    // This is the initial viewing transformation
    glTranslatef(0.0f, 0.0f, -100.0f);

    // Red Nucleus

    glColor3ub(255, 0, 0);
    glutSolidSphere(10.0f, 15, 15);

    // Yellow Electrons

    glColor3ub(255,255,0);

    // First Electron Orbit
    // Save viewing transformation
    glPushMatrix();

    // Rotate by angle of revolution
    glRotatef(fElect1, 0.0f, 1.0f, 0.0f);

    // Translate out from origin to orbit distance
    glTranslatef(90.0f, 0.0f, 0.0f);

    // Draw the electron
    glutSolidSphere(6.0f, 15, 15);

    // Restore the viewing transformation
    glPopMatrix();

    // Second Electron Orbit
    glPushMatrix();
    glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
    glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
    glTranslatef(-70.0f, 0.0f, 0.0f);
    glutSolidSphere(6.0f, 15, 15);
    glPopMatrix();


    // Third Electron Orbit
```

```
    glPushMatrix();
    glRotatef(360.0f, -45.0f, 0.0f, 0.0f, 1.0f);
    glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
    glTranslatef(0.0f, 0.0f, 60.0f);
    glutSolidSphere(6.0f, 15, 15);
    glPopMatrix();


    // Increment the angle of revolution
    fElect1 += 10.0f;
    if(fElect1 > 360.0f)
        fElect1 = 0.0f;

    // Show the image
    glutSwapBuffers();
    }
```

Let's examine the code for placing one of the electrons, a couple of lines at a time. The first line saves the current modelview matrix by pushing the current transformation on the stack:

```
// First Electron Orbit
// Save viewing transformation
glPushMatrix();
```

Now, the coordinate system is rotated around the y-axis by an angle `fElect1`:

```
// Rotate by angle of revolution
glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
```

The electron is drawn by translating down the newly rotated coordinate system:

```
// Translate out from origin to orbit distance
    glTranslatef(90.0f, 0.0f, 0.0f);
```

Then, the electron is drawn (as a solid sphere), and we restore the modelview matrix by popping it off the matrix stack:

```
// Draw the electron
    glutSolidSphere(6.0f, 15, 15);

    // Restore the viewing transformation
    glPopMatrix();
```

The other electrons are placed similarly.

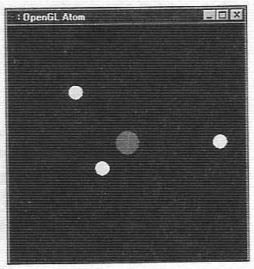

Figure 5.17 Output from the ATOM sample program.

Using Projections

In our examples so far, we have used the modelview matrix to position our vantage point of the viewing volume and to place our objects therein. The projection matrix actually specifies the size and shape of our viewing volume.

Thus far in this book, we have created a simple parallel viewing volume using the function `glOrtho`, setting the near and far, left and right, and top and bottom clipping coordinates. When the projection matrix is loaded with the identity matrix, the diagonal line of 1s specifies that the clipping planes extend from the origin to positive or negative 1 in all directions. The projection matrix does no scaling or perspective adjustments. To see these effects, you must use a perspective projection. The next two sample programs, ORTHO and PERSPECT, are not covered in detail from the standpoint of their source code. These examples use lighting and shading that you haven't covered yet to help highlight the differences between an orthographic and a perspective projection. These interactive samples make it much easier for you to see firsthand how the projection can distort the appearance of an object. If possible, you should run these examples while reading the next two sections.

Orthographic Projections

An orthographic projection, used for most of this book so far, is square on all sides. The logical width is equal at the front, back, top, bottom, left, and right sides. This produces a parallel projection, which is useful for drawings of specific objects that do not have any foreshortening when viewed from a distance. This is good for CAD, 2D graphs, or architectural drawings, for which you want to represent the exact dimensions and measurements onscreen.

Figure 5.18 shows the output from the sample program ORTHO on the CD in this chapter's subdirectory. To produce this hollow, tube-like box, we used an orthographic projection just as we did for all our previous examples. Figure 5.19 shows the same box rotated more to the side so you can see how long it actually is.

Figure 5.18 A hollow square tube shown with an orthographic projection.

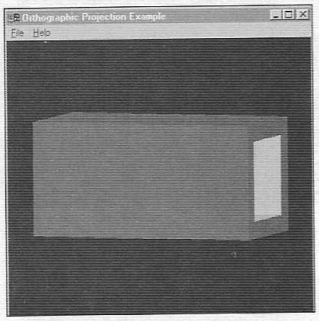

Figure 5.19 A side view showing the length of the square tube.

In Figure 5.20, you're looking directly down the barrel of the tube. Because the tube does not converge in the distance, this is not an entirely accurate view of how such a tube appears in real life. To add some perspective, we must use a perspective projection.

Perspective Projections

A perspective projection performs perspective division to shorten and shrink objects that are farther away from the viewer. The width of the back of the viewing volume does not have the same measurements as the front of the viewing volume. Thus, an object of the same logical dimensions appears larger at the front of the viewing volume than if it were drawn at the back of the viewing volume.

The picture in our next example is of a geometric shape called a frustum. A frustum is a section of a pyramid viewed from the narrow end to the broad end. Figure 5.21 shows the frustum, with the observer in place.

Figure 5.20 Looking down the barrel of the tube.

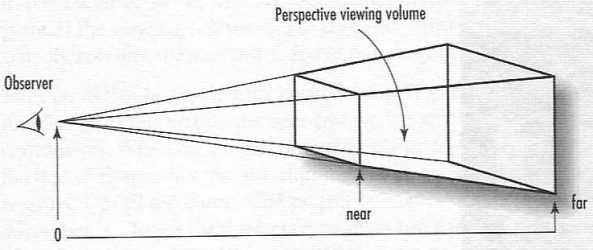

Figure 5.21 A perspective projection defined by a frustum.

You can define a frustum with the function `glFrustum`. Its parameters are the coordinates and distances between the front and back clipping planes. However, `glFrustum` is not intuitive about setting up your projection to get the desired effects. The utility function `gluPerspective` is easier to use and somewhat more intuitive for most purposes:

```
void gluPerspective(GLdouble fovy, GLdouble aspect,
                    GLdouble zNear, GLdouble zFar);
```

Parameters for the `gluPerspective` function are a field-of-view angle in the vertical direction; the aspect ratio of the height to width; and the distances to the near and far clipping planes. (See Figure 5.22.) You find the aspect ratio by dividing the width (w) by the height (h) of the front clipping plane.

Listing 5.2 shows how we change our orthographic projection from the previous examples to use a perspective projection. Foreshortening adds realism to our earlier orthographic projections of the square tube (Figures 5.23, 5.24, and 5.25). The only substantial change we made for our typical projection code in Listing 5.2 is the added call to `gluPerspective`.

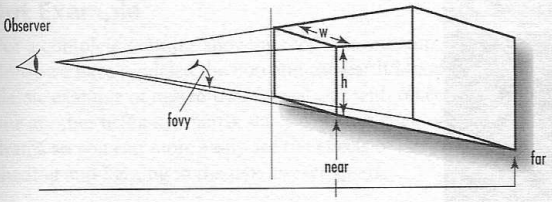

Figure 5.22 The frustum as defined by `gluPerspective`.

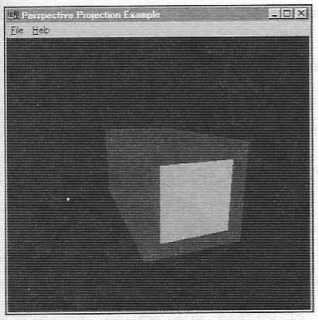

Figure 5.23 The square tube with a perspective projection.

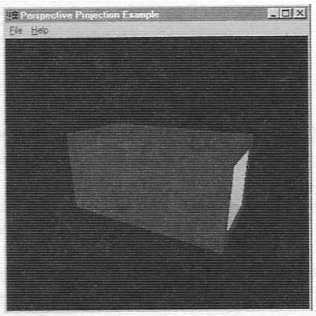

Figure 5.24 Side view with foreshortening.

Figure 5.25 Looking down the barrel of the tube with perspective added.

Listing 5.2 *Setting Up the Perspective Projection for the PERSPECT Sample Program*

```
// Change viewing volume and viewport.  Called when window is resized
void ChangeSize(GLsizei w, GLsizei h)
    {
    GLfloat fAspect;

    // Prevent a divide by zero
    if(h == 0)
        h = 1;

    // Set viewport to window dimensions
glViewport(0, 0, w, h);

    fAspect = (GLfloat)w/(GLfloat)h;

    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Produce the perspective projection
    gluPerspective(60.0f, fAspect, 1.0, 400.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    }
```

We made the same changes to the ATOM example in ATOM2 to add perspective. Run the two side-by-side and you see how the electrons appear to be smaller as they swing far away behind the nucleus.

A Far-Out Example

For a complete example showing modelview manipulation and perspective projections, we have modeled the Sun and the Earth/Moon system in revolution. This is a classic example of nested transformations with objects being transformed relative to one another using the matrix stack. We have enabled some lighting and shading for drama so you can more easily see the effects of our operations. You'll be learning about shading and lighting in the next two chapters.

In our model, we have the Earth moving around the Sun and the Moon revolving around the Earth. A light source is placed behind the observer to illuminate the Sun sphere (which also produces the illusion that the Sun is the light source). The light is then moved to the center of the Sun in order to light the Earth and Moon from the direction of the Sun, thus producing phases. This is a dramatic example of how easy it is to produce realistic effects with OpenGL.

Listing 5.3 shows the code that sets up our projection and the rendering code that keeps the system in motion. A timer elsewhere in the program invalidates the window 10 times a second to keep the `Render` function in action. Notice in Figures 5.26 and 5.27 that when the Earth appears larger, it's on the near side of the Sun; on the far side, it appears smaller.

Listing 5.3 *Code That Produces the Sun/Earth/Moon System*

```
// Change viewing volume and viewport.  Called when window is resized
void ChangeSize(GLsizei w, GLsizei h)
    {
    GLfloat fAspect;

    // Prevent a divide by zero
    if(h == 0)
        h = 1;

    // Set viewport to window dimensions
    glViewport(0, 0, w, h);

    // Calculate aspect ratio of the window
    fAspect = (GLfloat)w/(GLfloat)h;

    // Set the perspective coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Field of view of 45 degrees, near and far planes 1.0 and 425
    gluPerspective(45.0f, fAspect, 1.0, 425.0);
```

continued on next page

continued from previous page

```
    // Modelview matrix reset
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    }
// Called to draw scene
void RenderScene(void)
    {
    // Earth and Moon angle of revolution
    static float fMoonRot = 0.0f;
    static float fEarthRot = 0.0f;

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Save the matrix state and do the rotations
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    // Set light position before viewing transformation
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);

    // Translate the whole scene out and into view
    glTranslatef(0.0f, 0.0f, -300.0f);

    // Set material color, Red
    // Sun
    glColor3ub(255, 255, 0);
    glutSolidSphere(15.0f, 15, 15);

    // Move the light after we draw the sun!
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);

    // Rotate coordinate system
    glRotatef(fEarthRot, 0.0f, 1.0f, 0.0f);

    // Draw the Earth
    glColor3ub(0,0,255);
    glTranslatef(105.0f,0.0f,0.0f);
    glutSolidSphere(15.0f, 15, 15);


    // Rotate from Earth-based coordinates and draw Moon
    glRGB(200,200,200);
    glRotatef(fMoonRot,0.0f, 1.0f, 0.0f);
    glTranslatef(30.0f, 0.0f, 0.0f);
    fMoonRot+= 15.0f;
    if(fMoonRot > 360.0f)
        fMoonRot = 0.0f;
```

```
glutSolidSphere(6.0f, 15, 15);


// Restore the matrix state
glPopMatrix();    // Modelview matrix


// Step earth orbit 5 degrees
fEarthRot += 5.0f;
if(fEarthRot > 360.0f)
    fEarthRot = 0.0f;

// Show the image    glutSwapBuffers();
}
```

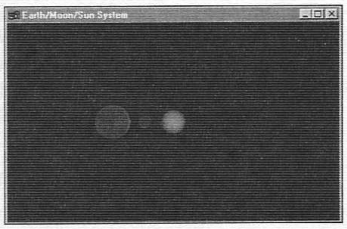

Figure 5.26 The Sun/Earth/Moon system with the Earth on the near side.

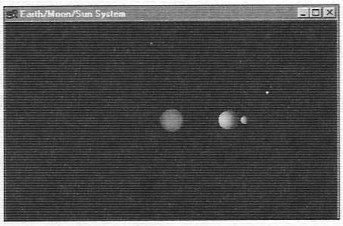

Figure 5.27 The Sun/Earth/Moon system with the Earth on the far side.

Advanced Matrix Manipulation

You don't have to use the high-level functions to produce your transformations. We recommend that you do, however, because those functions often are highly optimized for their particular purpose, whereas the low-level functions are designed for general use. For example, you can actually translate a vertex much faster than using a

matrix multiply, and many drivers employ such optimizations. Two of these low-level functions make it possible for you to load your own matrix and multiply it into either the modelview or projection matrix stacks.

Loading a Matrix

You can load an arbitrary matrix into the projection, modelview, or texture matrix stacks. First, declare an array to hold the 16 values of a 4×4 matrix. Make the desired matrix stack the current one, and call `glLoadMatrix`.

The matrix is stored in column-major order, which simply means that each column is traversed first from top to bottom. Figure 5.28 shows the matrix elements in numbered order. The following code shows an array being loaded with the identity matrix and then being loaded into the modelview matrix stack. This is equivalent to calling `glLoadIdentity` using the higher-level functions.

```
// Equivalent, but more flexible
glFloat m[] = { 1.0f, 0.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f, 0.0f,
              0.0f, 0.0f, 1.0f, 0.0f,
              0.0f, 0.0f, 0.0f, 1.0f };

glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(m);
```

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix}$$

Figure 5.28 Column-major matrix ordering.

Performing Your Own Transformations

You can load an array with an arbitrary matrix if you want and multiply it, too, into one of the three matrix stacks. The following code shows a transformation matrix that translates 10 units along the x-axis. This matrix is then multiplied into the modelview matrix. You can also achieve this affect (and probably more efficiently) by calling `glTranslatef`:

```
// Define the Translation matrix
glFloat m[] = { 1.0f, 0.0f, 0.0f, 10.0f,
0.0f, 1.0f, 0.0f, 0.0f,
              0.0f, 0.0f, 1.0f, 0.0f,
```

```
            0.0f, 0.0f, 0.0f, 1.0f };

    // Multiply the translation matrix by the current modelview
    // matrix. The new matrix becomes the modelview matrix
    glMatrixMode(GL_MODELVIEW);
    glMultMatrixf(m);
```

Other Transformations

There's no particular advantage in duplicating the functionality of `gLoadIdentity` or `glTranslatef` by specifying a matrix. The real reason for allowing manipulation of arbitrary matrices is to allow for complex matrix transformations. One such use is for drawing shadows, and you'll see that in action toward the end of the next chapter. You can also use these matrix functions to do your own transformations. Why would you want to do this after I told you to let OpenGL do it for you? You'll see in Chapter 19 how we can employ a new type of culling called *frustum culling* to eliminate geometry before sending primitives to the OpenGL driver. As you'll learn then, this approach can have a dramatic performance impact. You'll also learn about another real-time technique for angular transformations using *quaternions*, and the problem they solve: *gimbal lock*.

Summary

In this chapter, you've learned concepts crucial to using OpenGL for creation of 3D scenes. Even if you can't juggle matrices in your head, you now know what matrices are and how they are used to perform the various transformations. You've also learned how to manipulate the modelview and projection matrix stacks to place your objects in the scene and to determine how they are viewed onscreen.

Finally, we also showed you the functions needed to perform your own matrix magic if you are so inclined. These functions allow you to create your own matrices and load them into the matrix stack or multiply them by the current matrix first.

Reference Section

glFrustum

Purpose	Multiplies the current matrix by a perspective matrix.
Include File	<gl.h>
Syntax	`void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);`