CHAPTER 5

Coordinate Transformations and OpenGL Matrices

Now it's time to take a short break from learning how to *create* objects in the world, and focus on learning how to *move* the objects around the world. This is a vital ingredient to generating realistic 3D gaming worlds; without it, the 3D scenes you create would be static, boring, and totally non-interactive. OpenGL makes it easy for the programmer to move objects around through the use of various *coordinate transformations*, discussed in this chapter. You will also take a look at how to use your own matrices with OpenGL, which is a feature that is often used to create special-effect transformations on objects.

In this chapter, you'll learn the following:

- The basics of coordinate transformations
- The camera and viewing transformations
- OpenGL matrices and matrix stacks
- Projections
- Using your own matrices with OpenGL

Understanding Coordinate Transformations

Transformations allow us to move, rotate, and manipulate entities in a 3D world. One use of transformations is the capability to project 3D coordinates on a 2D screen. Another use was discussed in Chapter 3, "An Overview of 3D Graphics Theory," which covered the theory side of translate, rotate, and scale. Although it may seem that these transformations modify the objects directly, in reality, they modify the coordinate systems of the objects being transformed. For example, when you rotate a model's coordinate system, the model will appear to be rotated when it is drawn. Similarly, when you translate a model from the origin to a point 100 units away, the model will appear to be 100 units away from the camera when it is drawn.

When rendering 3D scenes, vertices pass through three types of transformations before they are finally rendered on the screen:

- **Viewing transformation.** Specifies the location of the camera.
- **Modeling transformation.** Moves objects around the scene.
- **Projection transformation.** Defines the viewing volume and clipping planes.

There is an additional transformation called the viewport transformation, which maps the two-dimensional projection of the scene into the window on your screen. You don't count the viewport transformation as a transformation that the vertices pass through because it relates strictly to the rendering window. Additionally, there is one other transformation that we will discuss: the modelview transformation. It can be considered a combination of the viewing and modeling transformation. Table 5.1 shows a summary of all these transformations.

Table 5.1 OpenGL Transformations

Transformation	Description
Viewing	Specifies the location of the camera
Modeling	Handles moving objects around the scene
Projection	Defines the viewing volume and clipping planes
Viewport	Maps the 2D projection of the scene into the rendering window
Modelview	Combination of the viewing and modeling transformations

When you are actually implementing these transformations, they must be executed in a specific order. The viewing transformations must execute before the modeling transformations; however, the projection and viewport transformations can be executed at any point before rendering. Figure 5.1 shows the general order that these vertex transformations are executed.

The Camera and Eye Coordinates

One of the most critical concepts to transformations and viewing in OpenGL is the concept of the *camera*, or *eye*, *coordinates*. Eye coordinates come strictly from the Cartesian coordinate system applied to the camera. In OpenGL, the default camera always looks down the negative z axis, as shown in Figure 5.2.

Eye coordinates remain the same no matter what transformations have been applied to them. For example, when rotating an object, you are in effect rotating the coordinate system of the object with respect to the eye's coordinate system. If you were to rotate a triangle 45 degrees counterclockwise, you would be transforming the triangle's coordinate system by 45 degrees counterclockwise. Figure 5.3 shows this transformation.

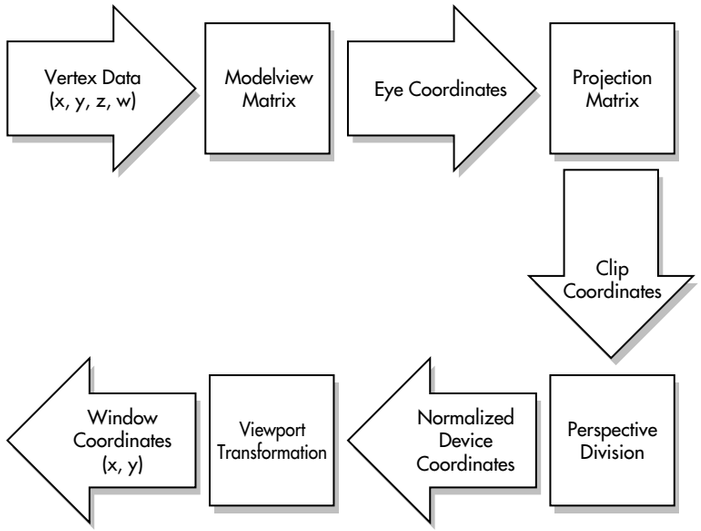

Figure 5.1

The vertex-transformation pipeline.

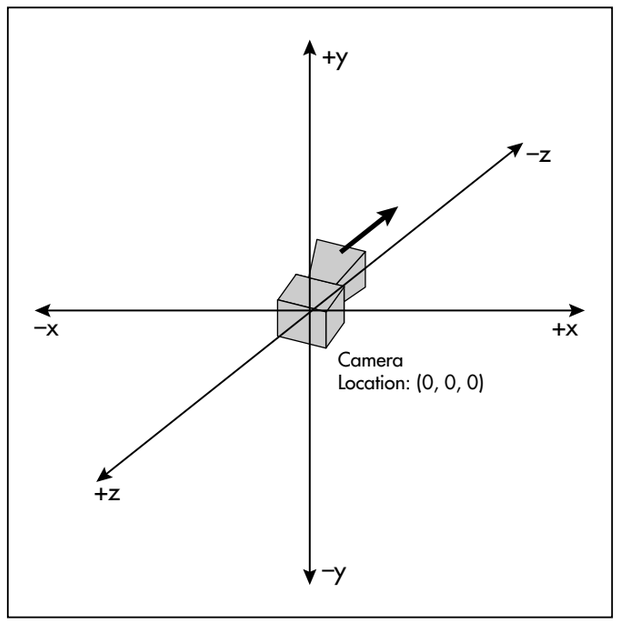

Figure 5.2

The viewer in OpenGL uses the Cartesian coordinate system and looks down the negative z axis.

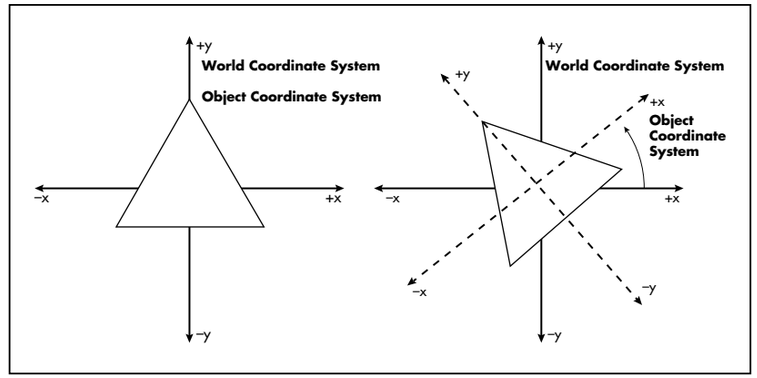

Figure 5.3

Rotating a triangle actually rotates its coordinate system with respect to eye coordinates.

Understanding eye coordinates is essential to understanding OpenGL transformations. We'll be taking a look at how you can modify the current coordinates to transform objects all over your 3D world.

Viewing Transformations

The viewing transformation is the first transformation applied to the scene and is used to position and aim the camera. As already stated, the camera's default orientation is to point down the negative z axis while positioned at the origin (0,0,0). You can move and change the camera's orientation through translation and rotation commands, which in effect manipulate the viewing transformation.

Remember that the viewing transformation must be completed before any other transformations. This is because it moves the current coordinate system with respect to the eye-coordinate system. Any other transformations that you do are based on the modified current coordinate system.

So how do you create the viewing transformation? Well, first you need to clear the *current matrix*. You accomplish this through the `glLoadIdentity()` command, specified as

```
void glLoadIdentity(void);
```

This sets the current matrix equal to the identity matrix and is necessary because most transformation commands manipulate the current matrix and set it to their own values. This can cause unexpected results, so you need to remember to clear the matrix.

After initializing the current matrix, you can create the viewing matrix in several different ways. One way is to just set the viewing matrix equal to the identity matrix. This will result in the default location and orientation of the camera, which would be at the origin and looking down the negative z axis. Other ways include the following:

- Using the `gluLookAt()` function to specify a line of sight that extends from the camera. This is a function that encapsulates a set of translation and rotation commands.
- Using the translation and rotation modeling commands `glTranslate*()` and `glRotate*()`. These commands are discussed in more detail later in this chapter; for now, suffice it to say that this method moves the objects in the world relative to a stationary camera.
- Creating your own routines that use the translation and rotation routines for your own coordinate system (for example, polar coordinates for a camera orbiting around an object).

Using the gluLookAt() Function

Because we have not yet talked about the modeling transformations, let's take a look at the `gluLookAt()` function, defined as

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble
centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);
```

You can use this function to define the camera's location and orientation. The first set of three parameters (*eyex*, *eyey*, *eyez*) specifies the location of the camera. The value (0,0,0) would naturally specify the origin. The next set of parameters (*centerx*, *centery*, *centerz*) specifies where the camera is pointing, also called the *line of sight*. This typically specifies a point somewhere in the middle of the scene that is currently being examined. The last set of parameters (*upx*, *upy*, *upz*) is a vector that tells which direction is up. Figure 5.4 shows how all of these parameters work on the camera with the `gluLookAt()` function.

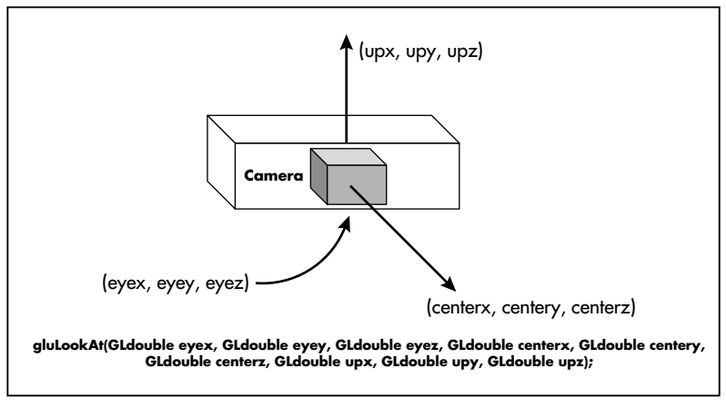

Figure 5.4

The `gluLookAt()` parameters specify the location and orientation of the camera.

Here is a short code snippet that uses the `gluLookAt()` function. Don't worry about any code you don't understand yet. You will get to it at some point. In any case, here is the code:

```
void DisplayScene()
{
     glClear(GL_COLOR_BUFFER_BIT);        // clear the color buffer
     glColor3f(1.0f, 0.0f, 0.0f);         // set color to red
     glLoadIdentity();                    // clear the current matrix

     // Now we set the viewing transformation with the gluLookAt() function.
     // This sets the camera at the position (0,0,10) and looking down the
     // negative z axis (0.0, 0.0, -100.0).
     // (eyex, eyey, eyez) = (0.0, 0.0, 10.0)
     // (centerx, centery, centerz) = (0.0, 0.0, -100.0)
     // (upx, upy, upz) = (0.0, 1.0, 0.0)
     gluLookAt(0.0f, 0.0f, 10.0f, 0.0f, 0.0f, -100.0f, 0.0f, 1.0f, 0.0f);

     // draw a triangle at the origin
     glBegin(GL_TRIANGLE);
          glVertexf(10.0f, 0.0f, 0.0f);
          glVertexf(0.0f, 10.0f, 0.0f);
          glVertexf(-10.0f, 0.0f, 0.0f);
     glEnd();

     // flush the buffer
     glFlush();
}
```

As you can see, the `gluLookAt()` function is rather easy to use. By manipulating the parameters, you can move the camera to any position and orientation that you want.

Using the glRotate*() and glTranslate*() Functions

A drawback to the `gluLookAt()` function, however, is that you must link the GLU library with your application. What if you don't want to use the GLU library? Well, one solution is to simply use the `glRotate*()` and `glTranslate*()` modeling-transformation functions. These functions modify the location of the objects in the world relative to a stationary camera. So rather than move the actual camera coordinates, you move the entire world around the camera. If you do not already understand the modeling-transformation functions, you might want to skip ahead to that section before looking at the following code. This code uses the modeling functions to produce the same effect on the camera as the `gluLookAt()` code.

```
void DisplayScene()
{
     glClear(GL_COLOR_BUFFER_BIT);        // clear the color buffer
     glColor3f(1.0f, 0.0f, 0.0f);         // set color to red
     glLoadIdentity();                    // clear the current matrix

     // Now we set the viewing transformation with the glTranslatef() function.
     // We move the modeling transformation to (0.0, 0.0, -10.0), which effectively
     // moves the camera to the position (0.0, 0.0, 10.0).
     glTranslatef(0.0f, 0.0f, -10.0f);

     // draw a triangle at the origin
     glBegin(GL_TRIANGLE);
          glVertexf(10.0f, 0.0f, 0.0f);
          glVertexf(0.0f, 10.0f, 0.0f);
            glVertexf(-10.0f, 0.0f, 0.0f);
     glEnd();

     // flush the buffer
     glFlush();
}
```

In this case, there isn't a serious difference in code from the `gluLookAt()` function because all you are doing is moving the camera along the z axis. But if you were orienting the camera at an odd angle, you would need to use the `glRotate()` function as well, which leads to the next way of manipulating the camera: your own custom routines.

Creating Your Own Custom Routines

Suppose you want to create your own flight simulator. In a typical flight simulator, the camera is positioned in the pilot's seat, so it moves and is oriented in the same manner as the plane. Plane orientation is defined by pitch, yaw, and roll, which are rotation angles relative to the center of gravity of the plane (in your case, the pilot/camera position). Using the modeling-transformation functions, you could create the following function to create the viewing transformation:

```
void PlaneView(GLfloat planeX, GLfloat planeY, glFloat planeZ, // the plane's position
               GLfloat roll, GLfloat pitch, GLfloat yaw)       // orientation
{
     // roll is rotation about the z axis
     glRotatef(roll, 0.0f, 0.0f, 1.0f);

     // yaw, or heading, is rotation about the y axis
     glRotatef(yaw, 0.0f, 1.0f, 0.0f);

     // pitch is rotation about the x axis
     glRotatef(pitch, 1.0f, 0.0f, 0.0f);

     // move the plane to the plane's world coordinates
     glTranslatef(-planeX, -planeY, -planeZ);
}
```

Using this function would place the camera in the pilot's seat of your airplane regardless of the orientation or location of the plane. This is just one of the uses of your own customized routines. Other uses include applications of polar coordinates, such as rotation about a fixed point and use of the modeling-transformation functions to create what is called "*Quake*-like movement," where the mouse and keyboard can be used to control the camera.

Modeling Transformations

The modeling transformations allow you to manipulate the position and set the orientation of a model by moving, rotating, and scaling it. You can perform these operations one at a time or as a combination of events. Figure 5.5 illustrates the three operations that you can use on objects:

- **Translation.** This operation is the act of moving an object along a specified axis.
- **Rotation.** This is where an object is rotated about one of the axes.
- **Scaling.** This is when you increase or decrease the size of an object. With scaling, you can specify different values for different axes. This gives you the ability to stretch and shrink objects non-uniformly.

The order that you specify modeling transformations is very important to the final rendition of your scene. For example, as shown in Figure 5.6, rotating and then translating an object has a completely different effect than translating and then rotating the object. Let's say you have an arrow located at the origin, and the first transformation you apply is a rotation of 30 degrees around the z axis. You then apply a translation transformation of 5 units along the x axis. The final position of the triangle would be (5, 4.33) with the arrow pointing at a 30-degree angle from the positive x axis. Now, let's say you translate the arrow by 5 units along the x axis instead of rotating it first. After the translation, the arrow would be located at (5,0). When you apply the rotation transformation, the arrow would still be located at (5,0), but it would be pointing at a 30-degree angle from the x axis.

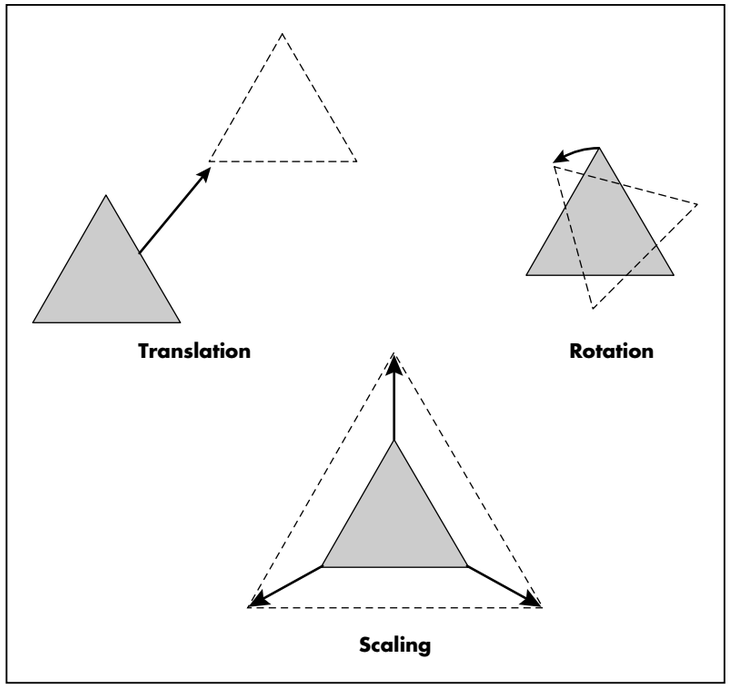

Figure 5.5

The three modeling transformations.

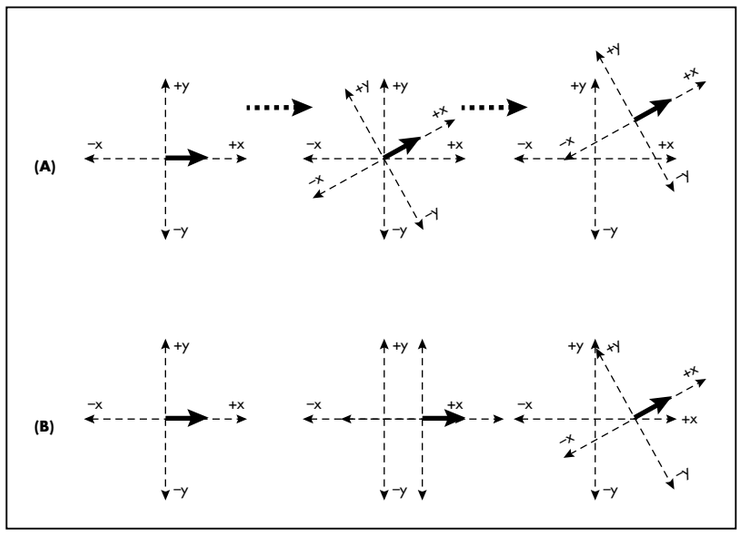

Figure 5.6

(a) Performing rotation before translation; (b) Performing translation before rotation.

Projection Transformations

The projection transformation defines the viewing volume and clipping planes. It is performed after the modelview transformation, which we have not yet covered in detail. You can think of the projection transformation as determining which objects belong in the viewing volume and how they should look. It is very much like choosing a camera lens that is used to look into the world. The field of view you choose when creating the projection transformation determines what type of lens you have. For instance, a wider field of view would be like having a wide-angle lens, where you could see a huge area of the scene without much detail. With a smaller field of view, which would be similar to a telephoto lens, you would be able to look at objects as though they were closer to you than they actually are.

OpenGL offers two types of projections:

- **Perspective projection.** This type of projection shows 3D worlds exactly how you see things in real life. With perspective projection, objects that are farther away appear smaller than objects that are closer to the camera.
- **Orthographic projection.** This type of projection shows objects on the screen in their true size, regardless of their distance from the camera. This projection is useful for CAD software, where objects are drawn with specific views to show the dimensions of an object.

The Viewport Transformation

The last transformation is the *viewport transformation*. This transformation maps the two-dimensional scene created by the perspective transformation onto your window's rendering surface. You can think of the viewport transformation as determining whether the final image should be enlarged or shrunk, depending on the size of the rendering surface.

OpenGL and Matrices

Now that you've learned about the various transformations involved in OpenGL, let's take a look at how you actually use them. Transformations in OpenGL rely on the *matrix* for all mathematical computations. As you will soon see, OpenGL has what is called the *matrix stack*, which is useful for constructing complicated models composed of many simple objects. You will be taking a look at each of the transformations and look more into the matrix stack in this section.

The Modelview Matrix

The modelview matrix defines the coordinate system that is being used to place and orient objects. It is a 4×4 matrix that is multiplied by vertices and transformations to create a new matrix that reflects the result of any transformations that have been applied to the vertices.

You can specify that you want to modify the modelview matrix through the OpenGL command `glMatrixMode()`, which is defined as

```
void glMatrixMode(GLenum mode);
```

Before calling any transformation commands, you must specify whether you want to modify the modelview matrix or the projection matrix. In order to modify the modelview matrix, you use the argument `GL_MODELVIEW`. This will set the modelview matrix to the current matrix, which means that it can be modified with subsequent transformation commands. Doing this would look like

```
void glMatrixMode(GL_MODELVIEW);
```

Other arguments for `glMatrixMode` include `GL_PROJECTION` and `GL_TEXTURE`. `GL_PROJECTION` is used to specify the projection matrix, and `GL_TEXTURE` is used to indicate the texture matrix, which we will discuss in Chapter 8, "Texture Mapping."

In most cases, you will want to reset the modelview matrix after you set it to the current matrix. To do this, you call the `glLoadIdentity()` function, discussed earlier. Calling this function will set the modelview matrix equal to the identity matrix and reset the current coordinate system to the origin. Here's a snippet of how you would reset the modelview matrix:

```
// ...
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();            // reset the modelview matrix

// ... do transformations

glBegin(GL_POINTS);
     glVertex3f(0.0f, 0.0f, 0.0f);
glEnd();

// ... continue with program
```

Translation

Translation allows you to move an object from one place to another in the 3D world. You can accomplish this with OpenGL using the functions `glTranslatef()` and `glTranslated()`, which are defined as follows:

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


The only difference between these two functions is their parameter types. You pass `float` parameters to `glTranslatef()` and `double` parameters to `glTranslated()`. Which one you decide to use depends on the level of precision you desire.

The parameters *x*, *y*, and *z* specify the amount to translate along the x, y, and z axes. For example, if you execute the command

```
glTranslatef(3.0f, 1.0f, 8.0f);
```

your object will move three units along the positive x axis, one unit along the positive y axis, and eight units along the positive z axis.

Suppose you want to move a cube from the origin to the position (5, 5, 5). You first load the modelview matrix and reset it to the identity matrix. Then you translate the current matrix to the position (5,5,5) before calling your `DrawCube()` function. In code, this looks like

```
glMatrixMode(GL_MODELVIEW);          // set current matrix to modelview
glLoadIdentity();                    // reset modelview to identity matrix
glTranslatef(5.0f, 5.0f, 5.0f);      // move to (5,5,5)
DrawCube();                          // draw the cube
```

Figure 5.7 illustrates how this code executes.

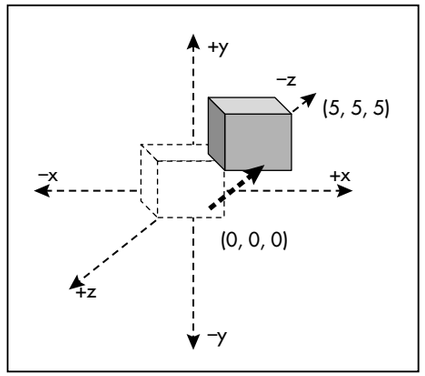

Figure 5.7
Translating a cube from the origin to (5,5,5).

Rotation

Rotation in OpenGL is accomplished through the `glRotate*()` function, which is defined as

```
void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
void glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z);
```

Again, you can use either `doubles` or `floats` for your parameters. With this function, you are performing a rotation around the vector specified by the *x*, *y*, and *z* parameters. The angle of rotation is specified by *`angle`* and is measured in degrees in the counterclockwise direction.

For example, if you wanted to rotate around the y axis 135 degrees in the counterclockwise direction, you would use the following code:

```
glRotatef(135.0f, 0.0f, 1.0f, 0.0f);
```

The value of `1.0f` for the *y* argument specifies a unit vector pointing in the direction of the y axis. When doing the rotation, you only need to specify unit vectors to rotate about the axis you desire. Figure 5.8 illustrates how the `glRotate*()` function works.

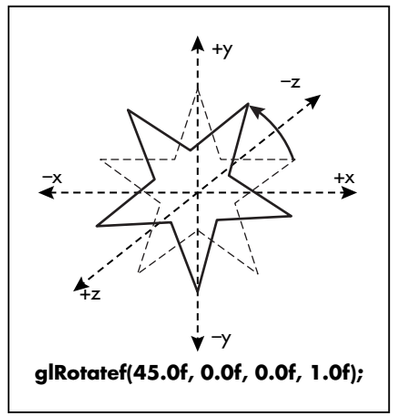

Figure 5.8

The `glRotate*()` function takes the angle of rotation and a unit vector for the axis of rotation as parameters.

If you wanted to rotate clockwise, you would set the angle of rotation as a negative number. To rotate around the y axis 135 degrees in the clockwise direction, you use the following code:

```
glRotatef(-135.0f, 0.0f, 1.0f, 0.0f);
```

What if you wanted to rotate around an arbitrary axis? You can accomplish this by specifying the arbitrary axis vector in the *x*, *y*, and *z* parameters. By drawing a line from the origin to the point represented by `(x,y,z)`, you can see the arbitrary axis around which you will rotate. For instance, if you rotate 90 degrees about the axis specified by the vector (1,1,0), you rotate about the axis that goes from the origin to the point (1,1,0). In code, this looks like the following:

```
glRotatef(90.0f, 1.0f, 1.0f, 0.0f);
```

Figure 5.9 illustrates how it works.

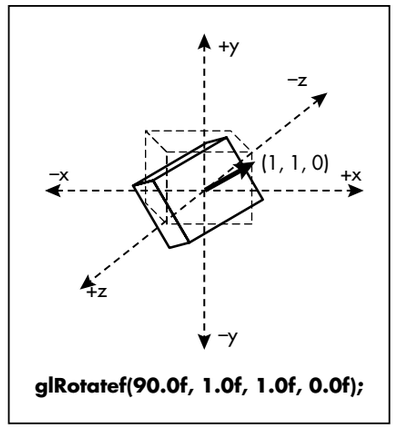

Figure 5.9

Rotation about an arbitrary axis.

And now a quick snippet of code that rotates a cube 60 degrees along the x axis and 45 degrees along the y axis:

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

glRotatef(60.0f, 1.0f, 0.0f, 0.0f);     // rotate 60 degrees around x axis
glRotatef(45.0f, 0.0f, 1.0f, 0.0f);     // rotate 45 degrees around y axis
DrawCube();                             // draw the cube
```

Scaling

Scaling is when you increase or decrease the size of an object. Vertices of an object are expanded or shrunk along the three axes depending on the scaling factor for each axis. You perform scaling through the OpenGL function `glScale*()`, which is defined as

```
void glScalef(GLfloat x, GLfloat y, GLfloat z);
void glScaled(GLdouble x, GLdouble y, GLdouble z);
```

The values passed to the *x*, *y*, and *z* parameters specify the scale factor along each axis. For example, the following line doubles the current size of an object:

```
glScalef(2.0f, 2.0f, 2.0f);
```

Now, let's say you had a cube, and you wanted to double its width (the x axis) without changing its height (the y axis) and depth (the z axis). You would use the following:

```
glScalef(2.0f, 1.0f, 1.0f);
```

What if you wanted to shrink an object? Well, because the scaling factors are each multiplied by the vertices, you simply choose a value less than one, like this:

```
glScalef(0.5f, 0.5f, 0.5f);
```

This line will shrink an object by half its original size. A value of 0.2 would shrink it by one-fifth, 0.1 by one-tenth, and so on. Now, if you set a scaling factor to 1.0, then the axis it belongs to will not be scaled. This is equivalent to multiplying a number by 1.0. Otherwise, values less than 1.0 will shrink the object, and values greater than 1.0 will enlarge the object. Figure 5.10 illustrates the `glScale*()` function.

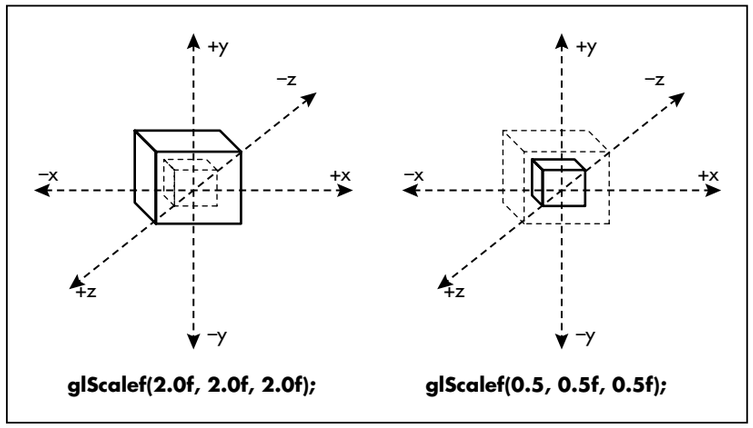

Figure 5.10

The `glScale*()` function.

Here is some code that will double the size of a cube:

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

glScalef(2.0f, 2.0f, 2.0f);        // double the size
DrawCube();                        // draw the cube
```

Matrix Stacks

The modelview matrix we've been playing with so far is actually only the top of a stack of matrices, which is naturally called the OpenGL matrix stack. There are three types of matrix stacks in OpenGL:

- The modelview matrix stack
- The projection matrix stack
- The texture matrix stack

The modelview matrix is actually the top of the modelview matrix stack, and as you will see, the projection matrix is the top of the projection matrix stack. Figure 5.11 gives some more information about these matrix stacks. The texture matrix stack is used for the transformation of coordinates.

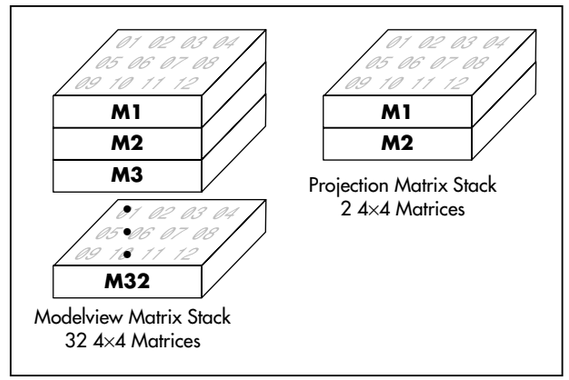

Figure 5.11

The modelview and projection matrix stacks are made up of 32 4×4 matrices and two 4×4 matrices, respectively, for the Microsoft OpenGL implementation.

The modelview matrix stack is used to construct complicated models out of more-simple ones. For example, consider how a robot might be built out of boxes. If you divide the robot into individual components, you have the torso, two arms, one head, and two legs. So in our program, we'd have a function to draw the torso, a function for one arm, a function for the head, and one for the legs. Each of these functions draws its respective component centered around the origin and at a normal orientation.

When you draw the robot, you would first draw the torso. Then, to draw the left arm, you would call the arm-drawing routine after translating to the position of the left arm relative to the torso. To draw the right arm, you would translate to the position of the right arm, again relative to the torso. Likewise, the legs and head would be drawn in their respective positions relative to the torso.

Matrix stacks provide this type of functionality in OpenGL. You can move object *A* relative to object *B*'s origin, draw object *A* around its own origin, and then throw away the whole transformation so you are again relative to object *B*'s origin. Two stack operations make this possible: `glPushMatrix()` and `glPopMatrix()`.

The `glPushMatrix()` function copies the current matrix and places it as the second matrix in the stack after pushing all the other matrices in the current stack down one level. Using this function

is like telling OpenGL to remember the current position in the world for a few moments while you visit another portion of the world. `glPushMatrix()` is defined as

```
void glPushMatrix(void);
```

If you push too many matrices onto the stack, then OpenGL gives a `GL_STACK_OVERFLOW` error.

The `glPopMatrix()` function discards the top matrix on the stack, destroying its contents, and places the second matrix at the top of the stack. All other matrices in the stack are moved up one. Using this function is like telling OpenGL to take you back to your original position after you've been visiting another portion of the world. `glPopMatrix()` is defined as

```
void glPopMatrix(void);
```

If you try to use this function when there is only one matrix in the stack, OpenGL will give a `GL_STACK_UNDERFLOW` error.

Figure 5.12 shows how the `glPushMatrix()` and `glPopMatrix()` functions operate on the matrix stack.

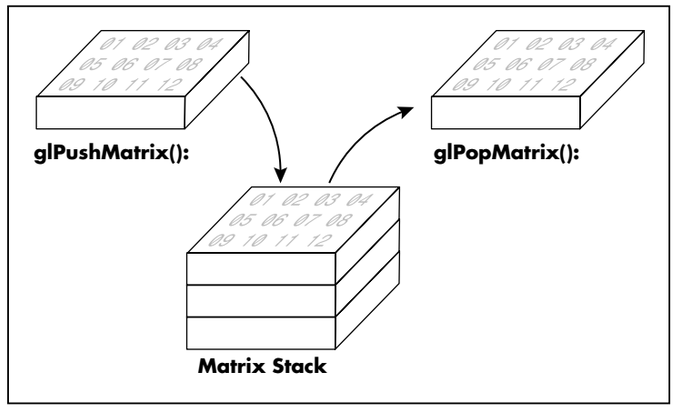

Figure 5.12

Pushing and popping on the matrix stack.

The Robot Example

Let's take a break and look at an example that uses everything we've talked about so far in this chapter. The source code on the following pages is for a small OpenGL demo that shows a walking robot around which the camera rotates. The robot is constructed of cubes that you scale to different shapes and sizes to give it the arms, legs, torso, and head. Take special note of how you use the `glPushMatrix()` and `glPopMatrix()` functions to place and move the robot.

Without further ado, here is the code:


```
#define WIN32_LEAN_AND_MEAN             // trim the excess fat from Windows

////// Includes
#include <windows.h>                    // standard Windows app include
#include <gl/gl.h>                      // standard OpenGL include
#include <gl/glu.h>                     // OpenGL utilities
#include <gl/glaux.h>                   // OpenGL auxiliary functions

////// Global Variables
float angle = 0.0f;                     // current angle of the camera
HDC g_HDC;                              // global device context
bool fullScreen = false;

////// Robot Variables
float legAngle[2] = { 0.0f, 0.0f };     // each leg's current angle
float armAngle[2] = { 0.0f, 0.0f };     // each arm's current angle

// DrawCube
// desc: since each component of the robot is made up of
//       cubes, we will use a single function that will
//       draw a cube at a specified location.
void DrawCube(float xPos, float yPos, float zPos)
{
     glPushMatrix();
          glTranslatef(xPos, yPos, zPos);
          glBegin(GL_POLYGON);
               glVertex3f(0.0f, 0.0f, 0.0f);  // top face
               glVertex3f(0.0f, 0.0f, -1.0f);
               glVertex3f(-1.0f, 0.0f, -1.0f);
               glVertex3f(-1.0f, 0.0f, 0.0f);
               glVertex3f(0.0f, 0.0f, 0.0f);  // front face
               glVertex3f(-1.0f, 0.0f, 0.0f);
               glVertex3f(-1.0f, -1.0f, 0.0f);
               glVertex3f(0.0f, -1.0f, 0.0f);
               glVertex3f(0.0f, 0.0f, 0.0f);   // right face
               glVertex3f(0.0f, -1.0f, 0.0f);
               glVertex3f(0.0f, -1.0f, -1.0f);
               glVertex3f(0.0f, 0.0f, -1.0f);
               glVertex3f(-1.0f, 0.0f, 0.0f);  // left face
               glVertex3f(-1.0f, 0.0f, -1.0f);
               glVertex3f(-1.0f, -1.0f, -1.0f);
```

```
                glVertex3f(-1.0f, -1.0f, 0.0f);
                glVertex3f(0.0f, 0.0f, 0.0f);  // bottom face
                glVertex3f(0.0f, -1.0f, -1.0f);
                glVertex3f(-1.0f, -1.0f, -1.0f);
                glVertex3f(-1.0f, -1.0f, 0.0f);
                glVertex3f(0.0f, 0.0f, 0.0f);  // back face
                glVertex3f(-1.0f, 0.0f, -1.0f);
                glVertex3f(-1.0f, -1.0f, -1.0f);
                glVertex3f(0.0f, -1.0f, -1.0f);
           glEnd();
     glPopMatrix();
}

// DrawArm
// desc: draws one arm
void DrawArm(float xPos, float yPos, float zPos)
{
     glPushMatrix();
          glColor3f(1.0f, 0.0f, 0.0f);       // red
          glTranslatef(xPos, yPos, zPos);
          glScalef(1.0f, 4.0f, 1.0f);        // arm is a 1x4x1 cube
          DrawCube(0.0f, 0.0f, 0.0f);
     glPopMatrix();
}

// DrawHead
// desc: draws the robot head
void DrawHead(float xPos, float yPos, float zPos)
{
     glPushMatrix();
          glColor3f(1.0f, 1.0f, 1.0f);       // white
          glTranslatef(xPos, yPos, zPos);
          glScalef(2.0f, 2.0f, 2.0f);        // head is a 2x2x2 cube
          DrawCube(0.0f, 0.0f, 0.0f);
     glPopMatrix();
}

// DrawTorso
// desc: draws the robot torso
void DrawTorso(float xPos, float yPos, float zPos)
```



```
{
     glPushMatrix();
          glColor3f(0.0f, 0.0f, 1.0f);     // blue
          glTranslatef(xPos, yPos, zPos);
          glScalef(3.0f, 5.0f, 2.0f);      // torso is a 3x5x2 cube
          DrawCube(0.0f, 0.0f, 0.0f);
     glPopMatrix();
}

// DrawLeg
// desc: draws a single leg
void DrawLeg(float xPos, float yPos, float zPos)
{
     glPushMatrix();
          glColor3f(1.0f, 1.0f, 0.0f);     // yellow
          glTranslatef(xPos, yPos, zPos);
          glScalef(1.0f, 5.0f, 1.0f);      // leg is a 1x5x1 cube
          DrawCube(0.0f, 0.0f, 0.0f);
     glPopMatrix();
}

// DrawRobot
// desc: draws the robot located at (xPos,yPos,zPos)
void DrawRobot(float xPos, float yPos, float zPos)
{
     static bool leg1 = true;        // robot's leg states
     static bool leg2 = false;        // true = forward, false = back

     static bool arm1 = true;
     static bool arm2 = false;

     glPushMatrix();

          glTranslatef(xPos, yPos, zPos);    // draw robot at desired coordinates

          // draw components
          DrawHead(1.0f, 2.0f, 0.0f);
          DrawTorso(1.5f, 0.0f, 0.0f);
          glPushMatrix();
```

```
        // if leg is moving forward, increase angle, else decrease angle
        if (arm1)
             armAngle[0] = armAngle[0] + 1.0f;
        else
             armAngle[0] = armAngle[0] - 1.0f;

        // once leg has reached its maximum angle in a direction,
        // reverse it
        if (armAngle[0] >= 15.0f)
             arm1 = false;
        if (armAngle[0] <= -15.0f)
             arm1 = true;

        // move the leg away from the torso and rotate it to give
        //"walking" effect
        glTranslatef(0.0f, -0.5f, 0.0f);
        glRotatef(armAngle[0], 1.0f, 0.0f, 0.0f);
        DrawArm(2.5f, 0.0f, -0.5f);
    glPopMatrix();

    glPushMatrix();
        // if leg is moving forward, increase angle, else decrease angle
        if (arm2)
             armAngle[1] = armAngle[1] + 1.0f;
        else
             armAngle[1] = armAngle[1] - 1.0f;

        // once leg has reached its maximum angle in a direction,
        // reverse it
        if (armAngle[1] >= 15.0f)
             arm2 = false;
        if (armAngle[1] <= -15.0f)
             arm2 = true;

        // move the leg away from the torso and rotate it to give
        //"walking" effect
        glTranslatef(0.0f, -0.5f, 0.0f);
        glRotatef(armAngle[1], 1.0f, 0.0f, 0.0f);
        DrawArm(-1.5f, 0.0f, -0.5f);
    glPopMatrix();
```



```
// we want to rotate the legs relative to the robot's position in the
//world. this is leg 1, the robot's right leg
glPushMatrix();

     // if leg is moving forward, increase angle, else decrease angle
     if (leg1)
          legAngle[0] = legAngle[0] + 1.0f;
     else
          legAngle[0] = legAngle[0] - 1.0f;

     // once leg has reached its maximum angle in a direction,
     // reverse it
     if (legAngle[0] >= 15.0f)
          leg1 = false;
     if (legAngle[0] <= -15.0f)
          leg1 = true;

     // move the leg away from the torso and rotate it to give
     //"walking" effect
     glTranslatef(0.0f, -0.5f, 0.0f);
     glRotatef(legAngle[0], 1.0f, 0.0f, 0.0f);

     // draw the leg
     DrawLeg(-0.5f, -5.0f, -0.5f);

     glPopMatrix();

     // do the same as above with leg 2, the robot's left leg
     glPushMatrix();

     if (leg2)
          legAngle[1] = legAngle[1] + 1.0f;
     else
          legAngle[1] = legAngle[1] - 1.0f;

     if (legAngle[1] >= 15.0f)
          leg2 = false;
     if (legAngle[1] <= -15.0f)
          leg2 = true;
```

```
                glTranslatef(0.0f, -0.5f, 0.0f);
                glRotatef(legAngle[1], 1.0f, 0.0f, 0.0f);
                DrawLeg(1.5f, -5.0f, -0.5f);

            glPopMatrix();
        glPopMatrix();
}


// Render
// desc: handles drawing of scene
void Render()
{
        glEnable(GL_DEPTH_TEST);                    // enable depth testing

        // do rendering here
        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);       // clear to black
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  // clear color/depth buffer
        glLoadIdentity();                           // reset modelview matrix

        angle = angle + 1.0f;       // increase our rotation angle counter
        if (angle >= 360.0f)        // if we've gone in a circle, reset counter
            angle = 0.0f;

        glPushMatrix();             // put current matrix on stack
            glLoadIdentity();       // reset matrix
            glTranslatef(0.0f, 0.0f, -30.0f);    // move to (0, 0, -30)
            glRotatef(angle, 0.0f, 1.0f, 0.0f);  // rotate the robot on its y axis
            DrawRobot(0.0f, 0.0f, 0.0f);         // draw the robot
        glPopMatrix();                           // dispose of current matrix

        glFlush();
        SwapBuffers(g_HDC);         // bring back buffer to foreground
}

// function to set the pixel format for the device context
void SetupPixelFormat(HDC hDC)
{
        int nPixelFormat;                   // our pixel format index
```



```
static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR),  // size of structure
        1,                              // default version
        PFD_DRAW_TO_WINDOW |            // window-drawing support
        PFD_SUPPORT_OPENGL |            // OpenGL support
        PFD_DOUBLEBUFFER,               // double-buffering support
        PFD_TYPE_RGBA,                  // RGBA color mode
        32,                             // 32-bit color mode
    0, 0, 0, 0, 0, 0,           // ignore color bits, non-palletized mode
    0,                          // no alpha buffer
    0,                          // ignore shift bit
    0,                          // no accumulation buffer
    0, 0, 0, 0,                 // ignore accumulation bits
    16,                         // 16-bit z-buffer size
    0,                          // no stencil buffer
    0,                          // no auxiliary buffer
    PFD_MAIN_PLANE,             // main drawing plane
    0,                          // reserved
    0, 0, 0 };                  // layer masks ignored

    // choose best-matching pixel format
    nPixelFormat = ChoosePixelFormat(hDC, &pfd);

    // set pixel format to device context
    SetPixelFormat(hDC, nPixelFormat, &pfd);
}

// the Windows Procedure event handler
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HGLRC hRC;           // rendering context
    static HDC hDC;             // device context
    int width, height;          // window width and height

    switch(message)
    {
        case WM_CREATE:         // window is being created

            hDC = GetDC(hwnd);          // get current window's device context
            g_HDC = hDC;
            SetupPixelFormat(hDC);      // call our pixel format setup function
```

```
        // create rendering context and make it current
        hRC = wglCreateContext(hDC);
        wglMakeCurrent(hDC, hRC);

        return 0;
        break;

    case WM_CLOSE:          // Windows is closing

        // deselect rendering context and delete it
        wglMakeCurrent(hDC, NULL);
        wglDeleteContext(hRC);

        // send WM_QUIT to message queue
        PostQuitMessage(0);

        return 0;
        break;

    case WM_SIZE:
        height = HIWORD(lParam);        // retrieve width and height
        width = LOWORD(lParam);

        if (height==0)                  // don't want a divide by zero
        {
            height=1;
        }

        // reset the viewport to new dimensions
        glViewport(0, 0, width, height);
        glMatrixMode(GL_PROJECTION); // set projection matrix current matrix
        glLoadIdentity();            // reset projection matrix

        // calculate aspect ratio of window
        gluPerspective(54.0f,(GLfloat)width/(GLfloat)height,1.0f,1000.0f);

        glMatrixMode(GL_MODELVIEW);  // set modelview matrix
        glLoadIdentity();            // reset modelview matrix

        return 0;
        break;
```



```
        default:
            break;
    }

    return (DefWindowProc(hwnd, message, wParam, lParam));
}

// the main Windows entry point
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
                   int nShowCmd)
{
    WNDCLASSEX windowClass;        // windows class
    HWND       hwnd;               // window handle
    MSG        msg;                // message
    bool       done;               // flag saying when our app is complete
    DWORD      dwExStyle;          // window extended style
    DWORD      dwStyle;            // window style
    RECT       windowRect;

    // screen/display attributes
    int width = 800;
    int height = 600;
    int bits = 32;

    windowRect.left=(long)0;          // set left value to 0
    windowRect.right=(long)width;     // set right value to requested width
    windowRect.top=(long)0;           // set top value to 0
    windowRect.bottom=(long)height;   // set bottom value to requested height

    // fill out the windows class structure
    windowClass.cbSize                = sizeof(WNDCLASSEX);
    windowClass.style                 = CS_HREDRAW | CS_VREDRAW;
    windowClass.lpfnWndProc           = WndProc;
    windowClass.cbClsExtra            = 0;
    windowClass.cbWndExtra            = 0;
    windowClass.hInstance             = hInstance;
    windowClass.hIcon                 = LoadIcon(NULL, IDI_APPLICATION);
    windowClass.hCursor               = LoadCursor(NULL, IDC_ARROW);
    windowClass.hbrBackground         = NULL;
    windowClass.lpszMenuName          = NULL;
```

```
windowClass.lpszClassName        = "MyClass";
windowClass.hIconSm              = LoadIcon(NULL, IDI_WINLOGO);

// register the windows class
if (!RegisterClassEx(&windowClass))
     return 0;

if (fullScreen)        // full screen?
{
     DEVMODE dmScreenSettings;          // device mode
     memset(&dmScreenSettings,0,sizeof(dmScreenSettings));
     dmScreenSettings.dmSize = sizeof(dmScreenSettings);
     dmScreenSettings.dmPelsWidth = width;          // screen width
     dmScreenSettings.dmPelsHeight = height;        // screen height
     dmScreenSettings.dmBitsPerPel = bits;          // bits per pixel
     dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

     if (ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN) !=
                               DISP_CHANGE_SUCCESSFUL)
     {
          // setting display mode failed, switch to windowed
          MessageBox(NULL, "Display mode failed", NULL, MB_OK);
          fullScreen=FALSE;
     }
}

if (fullScreen)                      // are we still in full-screen mode?
{
     dwExStyle=WS_EX_APPWINDOW;      // window extended style
     dwStyle=WS_POPUP;               // Windows style
     ShowCursor(FALSE);              // hide mouse pointer
}
else
{
     dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;  // window extended style
     dwStyle=WS_OVERLAPPEDWINDOW;                   // Windows style
}

AdjustWindowRectEx(&windowRect, dwStyle, FALSE, dwExStyle);
```



```
// class registered, so now create our window
hwnd = CreateWindowEx(NULL,  "MyClass",         // class name
                      "OpenGL Robot",           // app name
                      dwStyle | WS_CLIPCHILDREN |
                      WS_CLIPSIBLINGS,
                      0, 0,                     // x,y coordinate
                      windowRect.right - windowRect.left,
                      windowRect.bottom - windowRect.top, // width, height
                      NULL,                     // handle to parent
                      NULL,                     // handle to menu
                      hInstance,                // application instance
                      NULL);                    // no extra params

// check if window creation failed (hwnd would equal NULL)
if (!hwnd)
     return 0;

ShowWindow(hwnd, SW_SHOW);          // display the window
UpdateWindow(hwnd);                 // update the window

done = false;                       // initialize the loop condition variable

// main message loop
while (!done)
{
     PeekMessage(&msg, hwnd, NULL, NULL, PM_REMOVE);

     if (msg.message == WM_QUIT)    // do we receive a WM_QUIT message?
     {
          done = true;              // if so, time to quit the application
     }
     else
     {
          Render();
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
}
```

```
    if (fullScreen)
    {
        ChangeDisplaySettings(NULL,0);      // if so switch back to the desktop
        ShowCursor(TRUE);                   // show mouse pointer
    }
    return msg.wParam;
}
```

Wow! That was a lot of code, but you're just now beginning to get into the fun stuff. If you trace through to the `DrawRobot()` function, you will see how you can build and animate a hierarchical model, which is obviously in this case a robot-like figure. Pay careful attention to how you use the `glPushMatrix()` and `glPopMatrix()` functions to place the robot's arms, legs, torso, and head relative to the robot's local coordinate system origin. You could get really fancy and add hands, feet, or other body parts by using the push/pop functions to place the other body parts relative to existing parts. We'll leave that as an exercise for when we get bored. In the meantime, Figure 5.13 shows a screenshot of the robot demo.

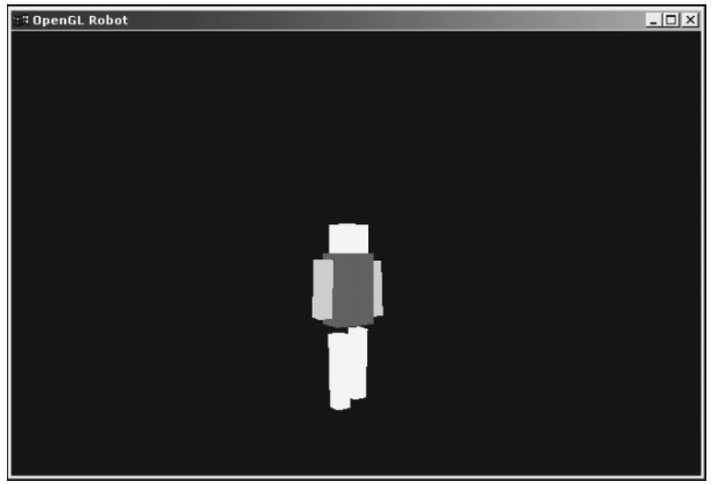

Figure 5.13

A screenshot of the OpenGL robot demo.

Projections

We've mentioned projection transformations several times now, and even used it in code, so it's high time we discussed how they work. As we've pointed out, there are two general classes of projection transformations available in OpenGL: orthographic (or parallel) and perspective. We'll look at both of these in detail.

Setting a projection transformation creates a viewing volume, which serves two purposes. The first is that it specifies a number of clipping planes, which determine which portion of your 3D world is visible at any given time. Objects that are outside this volume are not transformed or rendered. The second purpose of the viewing volume is to determine how objects are drawn. This depends on the shape of the viewing volume, which is the primary difference between orthographic and perspective projections.

Before specifying any kind of projection transformation, though, you need to make sure that the projection matrix stack is currently selected. As with the modelview matrix, this is done with a call to `glMatrixMode()`:

```
glMatrixMode(GL_PROJECTION);
```

In most cases, you'll want to follow this up with a call to `glLoadIdentity()` to clear out anything that may be stored in the matrix stack, so that previous transformations don't get accumulated. Unlike with the modelview matrix, it is very rare to make a lot of changes to the projection matrix.

Once the projection matrix stack is selected, you're ready to specify your projection. We'll look at orthographic projections first, and then at the more commonly used perspective transformations.

Orthographic

As we mentioned before, orthographic, or parallel, projections are those that involve no perspective correction. In other words, no adjustment for distance from the camera is made; objects appear the same size on screen whether they are close or far away. Although this may not look as realistic as perspective projections, it has a number of uses. Traditionally, orthographic projections are included in OpenGL for things like CAD. Orthographic projections can also be used for 2D games or for creating isometric games.

NOTE

Although orthographic projections can be used for isometric games, this is rarely done in practice due to the fact that a higher level of detail can be obtained using conventional 2D methods. This could very well change in the future, however.

OpenGL provides the `glOrtho()` function to set up orthographic projections:

```
glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near,
        GLdouble far);
```

`left` and *`right`* specify the x-coordinate clipping planes, *`bottom`* and *`top`* specify the y-coordinate clipping planes, and *`near`* and *`far`* specify the distance to the z-coordinate clipping planes. Together, these coordinates specify a box-shaped viewing volume. More precisely, opposite planes are parallel to each other, and adjacent planes are perpendicular.

Because orthographic projections are commonly used to create 2D scenes, the Utility Library provides an additional routine to set up orthographic projections for scenes in which you won't really be using the z coordinate:

```
gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
```

left, *right*, *bottom*, and *top* are as with `glOrtho()` above. Using `gluOrtho2D` is equivalent to calling `glOrtho()` with near set to –1.0 and far set to 1.0. When using `gluOrtho2D()`, you'll normally want to use a version of `glVertex()` that takes only two parameters (the x and y coordinates) because the z coordinate will be ignored anyway. It's common in this case to use integer coordinates and to set the view volume to match the x and y coordinates of the viewport.

Perspective

Although orthographic projections can be interesting, perspective projections create more realistic-looking scenes, so that's what you'll likely be using more often. In perspective projections, as an object gets farther from the viewer, it appears smaller on the screen—an effect commonly referred to as *foreshortening*. The viewing volume for a perspective projection is a *frustum*, which looks like a pyramid with the top cut off, with the narrow end toward the viewer. That the far end of the frustum is larger than the near end is what creates the foreshortening effect. The way this works is that OpenGL transforms the frustum so that it becomes a cube. This transformation affects the objects inside the frustum as well, so objects at the wide end of the frustum get compressed more than objects at the narrow end. The greater the ratio between the wide and narrow ends, the more objects will be shrunk. If the ends of the frustum are close in size, there won't be much perspective correction (if they are the same, there will be no correction at all, which is what happens with orthographic projections).

There are a couple ways you can set up the view frustum, and thus the perspective projection. The first we'll look at is the following:

```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
               GLdouble near, GLdouble far);
```

left, *right*, *top*, and *bottom* together specify the x and y coordinates on the near clipping plane, and *near* and *far* specify the distance to the near and far clipping planes. Thus, the top-left corner of the near clipping plane is at (*left*, *top*, *-near*), and the bottom-right corner is at (*right*, *bottom*, *-near*). The corners of the far clipping plane are determined by casting a ray from the viewer through the corners of the near clipping plane and intersecting them with the far clipping plane. So, the closer the viewer is to the near clipping plane, the larger the far clipping plane will be, and the more foreshortening will be apparent.

Using `glFrustum()` enables you to specify an asymmetrical frustum, which may be useful in some instances, but it's not typically what you'll want to do. In addition, thinking about what the viewer can see in terms of a frustum is not particularly intuitive. Instead, it's easier to think about their field of view—that is, how wide of an angle they can see. The OpenGL Utility Library provides a function that allows you to directly specify the field of view, and then calculates the frustum for you. This function is

```
void gluPerspective(GLdouble fov, GLdouble aspect, GLdouble near, GLdouble far);
```

fov specifies, in degrees, the angle in the y direction that is visible to the user. *aspect* is the aspect ratio of the scene, which is the width divided by the height. This determines the field of view in the x direction. *near* and *far* have the same meanings as they've had in the other projection functions in this section.

One thing we haven't mentioned in our discussion of setting up a frustum is how to determine an appropriate ratio between the width of the far and near end (that is, how wide the field of view is). The appropriate field of view is highly application dependent. If you want to create a fish-eye effect, a very wide field of view may be appropriate. For a realistic perspective, something around 90 degrees will probably work best. In general, you'll want to experiment to see what looks right for your particular application.

Setting the Viewport

Some of the projection functions we've just discussed are closely related to the size of the viewport (for example, the aspect ratio in `gluPerspective`). You know that the viewport transformation happens after the projection transformation, so now is as good a time as any to discuss it. Although you can't modify the viewport matrix directly, you can set the size of the viewport, which is all you really need to do.

In essence, the viewport specifies the dimensions and orientation of the 2D window into which you'll be rendering. It is set using `glViewport()`:

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

x and *y* specify the coordinates of the lower-left corner of the viewport, and *width* and *height* specify the size of the window, in pixels.

When a rendering context is first created and attached to your window, the viewport is automatically set to match the dimensions of the window. That may be good enough for some applications, but in most cases, you'll want to update your viewport any time the window is resized. Although the viewport will generally match your window size, there is nothing requiring it to be the same size. There may be times when you want to limit rendering to a sub-region of your window, and setting a smaller viewport is one way to do this.

Projection Example

To get a better idea of the differences between the two major projection types, we've included a simple demo that will you allow to view the same scene in each mode. The demo starts off with a perspective projection; pressing the spacebar will enable you to toggle between perspective (shown in Figure 5.14) and orthographic (shown in Figure 5.15).

The relevant portion of this demo is in the `ResizeScene` and `UpdateProjection` functions, which are listed here for convenience:

```
/*****************************************************************************
ResizeScene()

Updates the viewport and projection based on the screen size.
*****************************************************************************/
GLvoid ResizeScene(GLsizei width, GLsizei height)
{
  // avoid divide by zero
  if (height==0)
  {
    height=1;
  }

  // reset the viewport to the new dimensions
  glViewport(0, 0, width, height);

  // set up the projection, without toggling the projection mode
  UpdateProjection();
} // end ResizeScene()


/*****************************************************************************
UpdateProjection()

Sets the current projection mode. If toggle is set to GL_TRUE, then the
projection will be toggled between perspective and orthograpic. Otherwise,
the previous selection will be used again.
*****************************************************************************/
void UpdateProjection(GLboolean toggle = GL_FALSE)
{
  static GLboolean s_usePerspective = GL_TRUE;
```

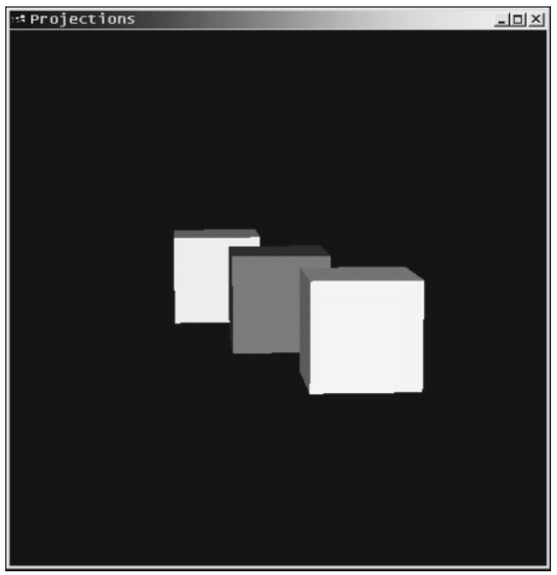

Figure 5.14

Perspective projection.

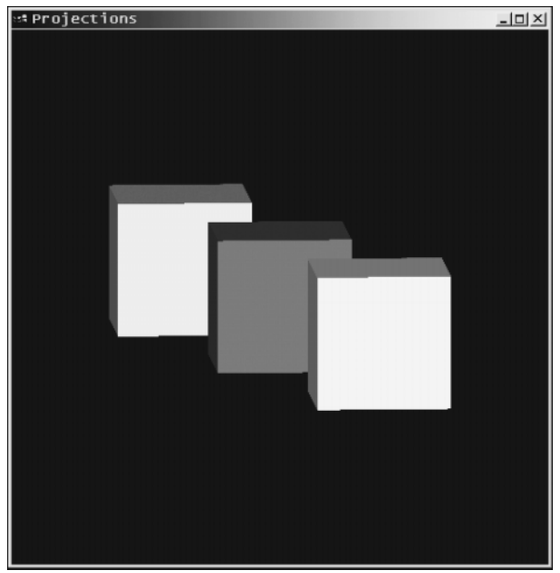

Figure 5.15

Orthographic projection.

```
// toggle the control variable if appropriate
if (toggle)
  s_usePerspective = !s_usePerspective;

// select the projection matrix and clear it out
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

// choose the appropriate projection based on the currently toggled mode
if (s_usePerspective)
{
  // set the perspective with the appropriate aspect ratio
  glFrustum(-1.0, 1.0, -1.0, 1.0, 5, 100);
}
else
{
  // set up an orthographic projection with the same near clip plane
  glOrtho(-1.0, 1.0, -1.0, 1.0, 5, 100);
}

  // select modelview matrix and clear it out
  glMatrixMode(GL_MODELVIEW);
} // end UpdateProjection
```

Using Your Own Matrices

Up until now, we've talked about functions that allow you to modify the matrix stacks without really having to worry about the matrices themselves. This is great, because it allows you to do a lot without having to understand matrix math, and the functions OpenGL provides for you are actually quite powerful and flexible. Eventually, though, you may want to create some advanced effects that are possible only by directly affecting the matrices. This will require that you know your way around matrix math, which we're not going to cover in any more detail than we have already. However, we'll at least show you how to load your own matrix, how to multiply the top of the matrix stack by a custom matrix, and one example of using a custom matrix.

Loading Your Matrix

Before you can load a matrix, you need to specify it. OpenGL matrices are column-major 4×4 matrices of floating point numbers, laid out as in Figure 5.16.

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Figure 5.16

OpenGL's column-major matrix format.

Because the matrices are 4×4, you may be tempted to declare them as two-dimensional arrays, but there is one major problem with this. In C and C++, two-dimensional arrays are row major. For example, to access the bottom-left element of the matrix in Figure 5.16, you might think you'd use `matrix[3][0]`, which is how you'd access the bottom-left corner of a 4×4 C/C++ two-dimensional array. Because OpenGL matrices are column major, however, you'd really be accessing the top-right element of the matrix. To get the bottom-left element, you'd need to use `matrix[0][3]`. This is the opposite of what you're used to in C/C++, making it counterintuitive and error-prone. Rather than using two-dimensional arrays, it's recommended that you use a one-dimensional array of 16 elements. The *n*th element in the array corresponds to element m_n in Figure 5.16.

As an example, if you want to specify the identity matrix (something you'd never need to do in practice due to the `glLoadIdentity()` function), you could use

```
GLfloat identity[16] = { 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
0.0, 0.0, 0.0, 1.0 };
```

That's easy enough. So, now that you've specified a matrix, the next step is to load it. This is done by calling `glLoadMatrix()`, which has two flavors:

```
void glLoadMatrixd(const GLdouble *matrix);

void glLoadMatrixf(const GLfloat *matrix);
```

The only difference between these functions is that one takes an array of `double`s, and the other takes an array of `float`s. When `glLoadMatrix()` is called, whatever is at the top of the currently selected matrix stack is replaced with the values in the *`matrix`* array, which is a 16-element array as specified previously.

Multiplying Matrices

In addition to loading new matrices onto the matrix stack (and thus losing whatever information was previously in it), you can multiply the contents of the active matrix by a new matrix. Again, you'd specify your custom matrix as above, and then call one of the following:

```
void glMultMatrixd(const GLdouble *matrix);
```

```
void glMultMatrixf(const GLfloat *matrix);
```

Again, *matrix* is an array of 16 elements. If the active matrix before the call to `glMultMatrix()` is M_{old}, and the new matrix is M_{new}, then the new matrix will be $M_{old} \times M_{new}$. Note that the ordering is important; because matrix multiplication is not commutative, $M_{old} \times M_{new}$ is not likely to have the same result as $M_{new} \times M_{old}$.

Custom Matrix Example

For an example of using your own matrices, refer to the sample program from Chapter 1, "The Exploration Begins: OpenGL and DirectX." In this program, we used a custom matrix to generate shadows in real time. The code we're interested in for the purposes of the current discussion follows:

```
GLfloat shadowMatrix[16] = { lightPos[1], 0.0,  0.0, 0.0, -lightPos[0], 0.0,
                             -lightPos[2],  -1.0, 0.0, 0.0,  lightPos[1], 0.0, 0.0,
                             0.0,  0.0, lightPos[1] };
...
// project the cube through the shadow matrix
glMultMatrixf(shadowMatrix);
DrawCube();
```

This matrix projects any vertices passed through it onto the y = 0 plane. If you set the current drawing color to black (along with some alpha blending and use of the stencil buffer, which are beyond the scope of this chapter), this has the effect of creating a shadow of the objects being drawn. You store the matrix into the modelview matrix stack by using `glMultMatrix()` rather than `glLoadMatrix()` because you want to preserve other transformations that have been used to orient the scene and position the cube.

One final note needs to be made in regard to using your own matrices: Whenever possible, you should use OpenGL's built-in transformation functions. In many cases, they are able to take advantage of hardware acceleration that you will not have access to.

Summary

In this chapter, you have seen how to manipulate objects in your scene by using transformations. You've also examined how to change the way in which the scene itself is viewed through setting up projections. In the process, you've learned about the projection and modelview matrices and how to manipulate them using both built-in functions and matrices you define yourself. You now have the means to place objects in a 3D world, to move and animate them, and to move around the world. Hmm…sounds like the beginnings of a game!