Chapter 6

Transformations

KEY TOPICS

- Translation
- Scaling
- Rotation
- Concatenation
- Visualization Experience

After all the discussion of how to work with matrices, you're finally ready to look at one of the most common uses of matrices in game programming: affine transformations. **Transformations** is really just a fancy word for moving objects around in your world. It encompasses movement such as forward and backward or up and down motion, scaling objects larger or smaller, and even rotating objects. The term **affine** indicates that the essential shape of the object being moved is preserved. This chapter first looks at how to apply the transformation in two dimensions, and then it extends the process to 3D. In the end, you'll be able to set up combos that control any type of motion you can think of, all with a single matrix. If at any point you find yourself struggling with the mathematical operations, flip back to Chapter 5, "Matrix Operations," for a quick review.

Translation

Let's start with the simple motion of moving objects left, right, up, and down on a flat 2D screen. The fancy name for moving objects in these directions is **translation**. Objects can be translated using both matrix addition and matrix multiplication. If all you plan to do is translate an object, you should definitely use matrix addition, because it is by far faster and easier. However, if you plan to scale and/or rotate your object in the same frame, you have to perform the translation using matrix multiplication instead. Let's tackle the addition method first.

Most people already have an intuitive feel for translation using addition after working with the Cartesian coordinate system. Suppose you had an object at point P(1,2) and you wanted to move it three units to the right and one unit up. What would you do? Well, if you add 3 to the x-coordinate and 1 to the y-coordinate, the object would end up three units to the right and one unit up. That gives you a new location of (1+3,2+1) = (4,3). You can indicate the new position with P', so your object would end up at point P'(4,3) after translating three to the right and one up.

That approach is simple if you're translating only one point or even just a few points. However, most models in games are defined by hundreds, if not thousands, of points, so you need a more systematic way of adding 3 to every *x* and 1 to every *y*. You could set up a loop and perform the following on each point:

$x'=x+3$

$y'=y+1$

This can be rewritten with matrices:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

Remember that when adding matrices you just add corresponding entries, so the matrix format is really the same as the two equations above it. The matrices are more efficient for organizing values in code, so you can set up a loop that inputs the original location and returns the new location of each vertex.

You might not always know the numeric values of the change in *x* and the change in *y* when you set up the code. You might have to wait for user input to determine how far you want the object to move. That's fine. Just set up a generic matrix equation and then plug in the values as soon as they become available.

Earlier in this book, we used the delta symbol (Δ) for "change in," but there's no delta key on the keyboard, so most programmers use *dx* for change in *x* and *dy* for change in *y*. This leads to the general form for translation by addition.

2D Translation by Addition

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} dx \\ dy \end{bmatrix}$$

NOTE

The values for *dx* and *dy* are not restricted to positive numbers. If *dx* is negative, it just indicates to the left instead of to the right, and a negative *dy* indicates down rather than up.

Example 6.1: 2D Translation by Addition

Set up a general matrix equation that will move 2D objects 50 pixels to the right and 100 pixels down on the computer screen, and then use it to move a triangle with vertices at A(20,30), B(0,200), and C(300,400).

Solution

1. Set up the matrix equation. To move 50 pixels to the right and 100 pixels down, *dx* must be 50, and *dy* must be –100 (negative because it's down):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 50 \\ -100 \end{bmatrix}$$

2. Now you have to plug in each old point and add the matrices to see where it moved. First, plug in the old location of vertex A(20,30):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 20 \\ 30 \end{bmatrix} + \begin{bmatrix} 50 \\ -100 \end{bmatrix} = \begin{bmatrix} 70 \\ -70 \end{bmatrix}$$

   So A' is the point (70,–70).

3. If you repeat step 2 with the old locations of B and C, you get the new locations B'(50,100) and C'(350,300). By moving all three vertices, you have moved the whole triangle.

4. Figure 6.1 shows the old location with dashed lines and the new location with solid lines.

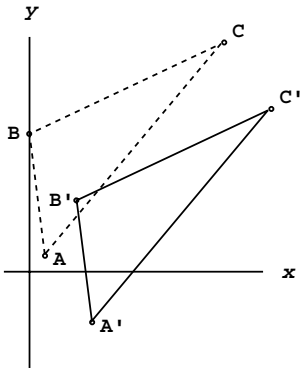

Figure 6.1 Triangle ABC before and after translation.

In Example 6.1 you translated a simple triangle with three vertices. Keep in mind, however, that most models are defined by many more vertices than that. Most polygonal models are covered by hundreds (or thousands) of triangles, but if you can translate one triangle, you can translate many.

You can also translate objects in 3D simply by adding one more entry to each matrix for the z-coordinate.

3D Translation by Addition

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix}$$

Example 6.2: 3D Translation by Addition

Set up a general matrix equation that will move 3D objects 100 pixels to the left, 200 pixels up, and 50 pixels back (behind the screen), and then use it to move a triangle with vertices at A(40,0,100), B(0,350,200), and C(–100,200,–10).

Solution

1. Set up the matrix equation. To move 100 pixels to the left, 200 pixels up, and 50 pixels back, *dx* = –100, *dy* = 200, and *dz* = –50. (If you can't remember which direction is positive or negative, you can always flip back to Chapter 1, "Points and Lines.")

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} -100 \\ 200 \\ -50 \end{bmatrix}$$

2. Now you have to plug in each old point and add the matrices to see where it moved. First, plug in the old location of vertex A(40,0,100):

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 40 \\ 0 \\ 100 \end{bmatrix} + \begin{bmatrix} -100 \\ 200 \\ -50 \end{bmatrix} = \begin{bmatrix} -60 \\ 200 \\ 50 \end{bmatrix}$$

   So A' is the point (–60,200,50).

3. If you repeat step 2 with the old locations of B and C, you get the new locations B'(–100,550,150) and C'(–200,400,–60). By moving all three vertices, you have moved the whole triangle.

Translation by addition is pretty straightforward. Here is an example of how to translate a 3D point using addition:

```
Matrix3X1 translate3DByAddition(Matrix3X1 start, Matrix3X1 trans)
    {
      Matrix3X1 temp;
      temp = addMatrices(start,trans);
      return temp;
    }
```

It is much more common to translate 3D points in games using matrices than it is 2D values. Generally speaking, a basic unit of a game that might represent the player will contain an x and y value. For example, if the player needs to have a position set 10 units to the right, one way to do this is to get away from the matrix notation:

```
Player.setX(Player.getX()+10);
```

If you are feeling comfortable with the matrix notation, feel free to keep it. It will actually be more beneficial when the emphasis comes to working with 3D points as well as rotation in 2D.

Again, if all you need to do is translate an object, use matrix addition. However, if you plan to also scale or rotate the object, you need to use matrix multiplication. You can set up a matrix equation in much the same way. Then, all you need to do is plug in each original location one at a time and multiply the matrices to find the new location.

2D Translation by Multiplication

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where dx = change in x and dy = change in y.

Notice that the old point and the new point have an extra 1 on the end. It's not actually part of the point, but it needs to tag along for the matrix math to work. You might want to research "homogeneous coordinates" for further explanation.

Also, take note of the order in which the matrices are multiplied. If the order were switched (old point * translation matrix), the product would no longer be defined, and the program would crash. Remember from Chapter 5, "Matrix Operations," that the order in which you multiply matrices is critical. The number of columns in the first matrix must equal the number of rows in the second matrix. In this case, you must set it up to be a 3×3 times a 3×1, not the other way around.

Make sure that the matrix equation is always set up in that order—translation matrix * old point.

Let's repeat Example 6.1 using matrix multiplication and see if we get the same results.

Example 6.3: 2D Translation by Multiplication

Set up a general matrix equation (using matrix multiplication) that will move 2D objects 50 pixels to the right and 100 pixels down on the computer screen, and then use it to move a triangle with vertices at A(20,30), B(0,200), and C(300,400).

Solution

1. Set up the matrix equation. To move 50 pixels to the right and 100 pixels down, *dx* must be 50, and *dy* must be –100 (negative because it's down):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 50 \\ 0 & 1 & -100 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. Now you have to plug in each old point and multiply the matrices to see where it moved. First, plug in the old location of vertex A(20,30):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 50 \\ 0 & 1 & -100 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 20 \\ 30 \\ 1 \end{bmatrix} = \begin{bmatrix} 1(20) + 0(30) + 50(1) \\ 0(20) + 1(30) - 100(1) \\ 0(20) + 0(30) + 1(1) \end{bmatrix} = \begin{bmatrix} 70 \\ -70 \\ 1 \end{bmatrix}$$

   So A' is the point (70,–70), which is exactly what you found before.

3. If you repeat step 2 with the old locations of B and C, you get the new locations B'(50,100) and C'(350,300), just like last time. By moving all three vertices, you have moved the whole triangle. Notice that it ends up moving to the same place, as it did in Example 6.1.

If you look closely at the matrix multiplication, you can see the important role that the extra 1 plays. Notice the dot product that you calculated for *x*'. You kept the old *x* (multiply by 1), ignored the old *y* (multiply by 0), and added *dx* (multiply by 1). Then, to calculate *y*', you ignored the old *x,* kept the old *y,* and added *dy*. You needed that extra 1 to add the *dx* and *dy*.

The same thing happens when you translate 3D objects using matrix multiplication.

Let's take a quick look at one way to handle this in code. Here is a function that will multiply a 2D point using matrix multiplication:

```
Matrix3X1 translate2DByMultiplication(Matrix3X1 start,float dx,
➥float dy)
    {
        Matrix3X3 temp;
        Matrix3X1 result;

        //Zero out the matrix.
        temp = createFixed3X3Matrix(0);

        //setup the 3x3 for multiplication;
        temp.index[0][0] = 1;
        temp.index[1][1] = 1;
        temp.index[2][2] = 1;

        //put in the translation amount
        temp.index[0][2] = dx;
        temp.index[1][2] = dy;

        result = multiplyMatrixNxM(temp,start);
        return result;
    }
```

There are some important things to note here. First of all, we go through and set up the positions [0][0];[1][1];[2][2] to the value of 1. The reason for this is so that the dot product will work out. Then we set the additional positions by the amount we want to move by in each direction. Once that is settled, we multiply the 3×3 matrix against our matrix that was holding our original position. The step to setting up the matrix for proper evaluation will become much more important as we move toward 3D multiplication.

3D Translation by Multiplication

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where *dx* = change in *x*, *dy* = change in *y*, and *dz* = change in *z*.

The process for 3D is the same as that for 2D; the only difference is that you're now using a 4×4 translation matrix. Notice that the extra 1 is still along for the ride so that the matrix math works out.

Example 6.4: 3D Translation by Multiplication

Set up a general matrix equation (using matrix multiplication) that will move 3D objects 100 pixels to the left, 200 pixels up, and 50 pixels back (behind the screen), and then use it to move a triangle with vertices at A(40,0,100), B(0,350,200), and C(–100,200,–10).

Solution

1. Set up the matrix equation. To move 100 pixels to the left, 200 pixels up, and 50 pixels back, *dx* = –100, *dy* = 200, and *dz* = –50. (If you can't remember which direction is positive or negative, you can always flip back to Chapter 1.)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -100 \\ 0 & 1 & 0 & 200 \\ 0 & 0 & 1 & -50 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2. Now you have to plug in each old point and add the matrices to see where it moved. First, plug in the old location of vertex A(40,0,100):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -100 \\ 0 & 1 & 0 & 200 \\ 0 & 0 & 1 & -50 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 40 \\ 0 \\ 100 \\ 1 \end{bmatrix} = \begin{bmatrix} 1(40) + 0(0) + 0(100) - 100(1) \\ 0(40) + 1(0) + 0(100) + 200(1) \\ 0(40) + 0(0) + 1(100) - 50(1) \\ 0(40) + 0(0) + 0(100) + 1(1) \end{bmatrix} = \begin{bmatrix} -60 \\ 200 \\ 50 \\ 1 \end{bmatrix}$$

So A' is the point (–60,200,50), just like last time.

3. If you repeat step 2 with the old locations of B and C, you get the new locations B'(–100,550,150) and C'(–200,400,–60). By moving all three vertices, you have moved the whole triangle. Notice that again you have found the same new location as you did using matrix addition.

Translation through multiplication is very similar for 3D. The major difference is that you are working with 4×4 and 4×1 matrices. The rest is fairly straightforward. Here is the function that translates 3D by multiplication:

```
Matrix4X1 translate3DByMultiply(Matrix4X1 start,float dx,
➥float dy,float dz)
     {
          Matrix4X4 temp;
          Matrix4X1 result;

          //Zero out the matrix.
          temp = createFixed4X4Matrix(0);

          //setup the 4X4 for multiplication;
          temp.index[0][0] = 1;
          temp.index[1][1] = 1;
          temp.index[2][2] = 1;
          temp.index[3][3] = 1;

          //put in the translation amount
          temp.index[0][3] = dx;
          temp.index[1][3] = dy;
          temp.index[2][3] = dz;

          result = multiplyMatrixNxM(temp,start);
          return result;
     }
```

Again, we go through and initialize the important components of the matrix, then use the starting coordinates to set the rightmost column. The result is a 4×1 matrix that holds the new position of the point.

NOTE

Notice that each time you calculate the new location, the last entry is always a 1. In code, you're wasting precious time if you calculate the extra 1 for every vertex. All you really need to find is x', y', and z', so don't bother calculating the last entry each time. Can you implement this as a small optimization into the translation code?

The beginning of this section said that matrix addition is by far easier and faster. After practicing both methods, I think you'll agree. Unfortunately, the only way to combine translation with scaling and rotating is to use matrix multiplication. The good news is that if you feel comfortable with translation, scaling and rotating should fall right into place for you.

Self-Assessment

Using the matrix equation provided, find the new locations of the following vertices:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 40 \\ -20 \end{bmatrix}$$

1. D(30,80)
2. E(–50,200)
3. F(100,0)
4. Set up a general matrix equation (using matrix addition) that will move 3D objects 50 pixels to the right, 300 pixels down, and 0 pixels back (behind the screen), and then use it to move a triangle with vertices at G(200,–30,–50), H(90,0,–40), and J(–400,50,–100).
5. Set up a general matrix equation (using matrix multiplication) that will move 2D objects 200 pixels to the left and 20 pixels up on the computer screen. Then use it to move a triangle with vertices at L(–100,30), M(50,–80), and N(70,0).
6. Repeat question 4 using matrix multiplication. Do you get the same three new locations?

Scaling

Matrix multiplication can also be used to scale objects in your game. Just like translation, if you scale each individual vertex, you end up scaling the whole object. Let's set up another matrix equation for scaling. Again, we'll start with 2D and then extend to 3D.

2D Scaling

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where Sx = scale factor in the x direction and Sy = scale factor in the y direction.

As soon as you plug in the scale factors (Sx and Sy), the process is the same as translation: Plug in each vertex one at a time, and multiply the matrices to find its new location. If you want to perform a **uniform scale** to keep the proportions the same, just make sure that $Sx = Sy$. You don't have to perform a uniform scale, however. If you plug in two different values for Sx and Sy, you'll end up with a **differential scale**.

When choosing values for Sx and Sy, keep in mind that any number between 0 and 1 makes objects smaller, and any number greater than 1 scales objects larger. (Negative values flip the object into a different quadrant.) Let's look at a couple examples.

Example 6.5: 2D Uniform Scale

Set up a general matrix equation that will uniformly scale 2D objects 3 times larger, and then use it to scale the rectangle pictured in Figure 6.2.

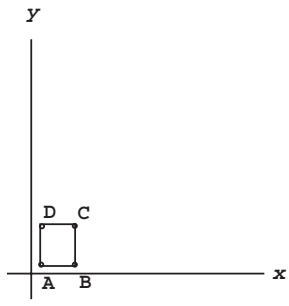

Figure 6.2 A rectangle to be uniformly scaled.

Solution

1. Set up the matrix equation. To uniformly scale objects 3 times larger, both scale factors must be equal to 3:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. Now you have to plug in each old point and multiply the matrices to find the new location. First, plug in the old location of vertex A(10,10):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 3(10) + 0(10) + 0(1) \\ 0(10) + 3(10) + 0(1) \\ 0(10) + 0(10) + 1(1) \end{bmatrix} = \begin{bmatrix} 30 \\ 30 \\ 1 \end{bmatrix}$$

   So A' is the point (30,30).

3. If you repeat step 2 with the old locations of B, C, and D, you get the new locations B'(150,30), C'(150,120), and D'(30,120). By scaling all four vertices, you have scaled the whole rectangle 3 times larger.

4. The old location and the new location are graphed in Figure 6.3.

Turning this into code isn't very difficult, but extends slightly as we try to make something more of it. Here is a function that will take a matrix and multiply the value by a scale passed to it:

```
Matrix3X1 scale2DByMultiplication(Matrix3X1 start, float dx, float dy)
    {
        Matrix3X3 temp;
        Matrix3X1 result;

        //Zero out the matrix.
        temp = createFixed3X3Matrix(0);

        //setup the 3x3 for multiplication;
        temp.index[0][0] = dx;
        temp.index[1][1] = dy;
        temp.index[2][2] = 1;

        result = multiplyMatrixNxM(temp,start);
        return result;
    }
```

Notice that this function will scale uniformly or nonuniformly based on the values passed through to `dx` and `dy`. If you want to make it scale uniformly, just ask for a single scale factor and assign them to be equal.

This is great for scaling up single points, but most objects that are scaled are of some geometric shape. Let's look at part of the driver function from the sample code that indicates how to scale a rectangle uniformly:

```
void scale2D()
    {
      Matrix3X1 start,temp;
      float dx,dy, height,width;
 cout<<"Please enter the coordinates and dimensions  of a
 ➥rectangle.\n";
      cout<<"X coordinates\n";
      cin>>start.index[0];
      cout<<"Now the Y coordinate.\n";
      cin>>start.index[1];
      cout<<"Enter the rectangle's height.\n";
      cin>>height;
      cout<<"Enter the rectangle's width.\n";
      cin>>width;
```

```
    //make sure the last part of the matrix is a 1.
    start.index[2] = 1;
    cout<<endl;
    cout<<"Now enter the amount to scale by.\n";
    cin>>dx;
    dy = dx;

    temp = scale2DByMultiplication(start,dx,dy);
    width = temp.index[0]+width;
    height = temp.index[1]+height;

    cout<<"The new position is
    ➥"<<temp.index[0]<<","<<temp.index[1]<<"\n";
    cout<<"The right coord is
    ➥"<<width<<","<<temp.index[1]<<"\n";
    cout<<"The bottom coord is
    ➥"<<height<<","<<temp.index[0]<<"\n";
}
```

This function will give the scaled position of all the points in the rectangle and is much more useful than just scaling a single point. We will look at a similar example when we focus on rotation a little later in this chapter.

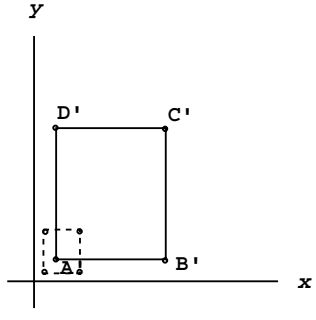

Figure 6.3 Old and new locations of a uniformly scaled rectangle.

Look at Figure 6.3 for a moment. Notice that the dimensions of the rectangle are indeed 3 times larger. However, the rectangle appears to have also moved away from the origin. Unfortunately, the scaling matrix is set up to scale objects with respect to the origin, which is why the rectangle moved away from the origin when it was scaled larger. If you had scaled it down, the rectangle would have moved closer to the origin. Unfortunately, if you want to keep the object in the same place and scale it (with respect to its own center), that is a combo. We'll examine that particular combo later in this chapter. In the meantime, let's look at an example of differential scaling with respect to the origin.

NOTE

One way to deal with this issue is to set up a local coordinate system with its own origin. For example, if I'm scaling a human figure who's standing on the ground, I can place the local origin on the soles of the figure's shoes and center it on the other two axes. Then a single scaling matrix would allow the figure to grow from the ground without moving laterally. Quite often, assets that look perfect in the preview applications get sent back to the artist after they start running in the game because their origins were misplaced.

Example 6.6: 2D Differential Scale

Suppose you have a rectangular-shaped object (see Figure 6.4) with vertices at A(20,0), B(50,0), C(50,100), and D(20,100), and a huge boulder falls on it. You want the object to look like it got squished, so set up a matrix equation that will scale objects 1.5 times in the *x* direction and 0.1 in the *y* direction, and then use it to scale the rectangle.

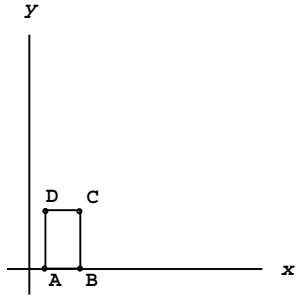

Figure 6.4 A rectangle to be differentially scaled.

Solution

1. Set up the matrix equation. In this case, the scale factor in the *x* direction is 1.5 and the scale factor in the *y* direction is 0.1:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. Now you have to plug in each old point and multiply the matrices to find the new location. First, plug in the old location of vertex A(20,0):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 20 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5(20) + 0(0) + 0(1) \\ 0(20) + 0.1(0) + 0(1) \\ 0(20) + 0(0) + 1(1) \end{bmatrix} = \begin{bmatrix} 30 \\ 0 \\ 1 \end{bmatrix}$$

   So A' is the point (30,0).

3. If you repeat step 2 with the old locations of B, C, and D, you get the new locations B'(75,0), C'(75,10), and D'(30,10). By scaling all four vertices, you have scaled the whole rectangle.

4. The old location and the new location are graphed in Figure 6.5. Notice that the rectangle has become wider and flatter.

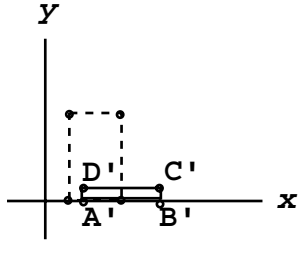

Figure 6.5 Old and new locations of a differentially scaled rectangle.

The scaling process works the exact same way in 3D. You just need to add an extra dimension to each matrix in the equation.

3D Scaling

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where *Sx* = scale factor in the *x* direction, *Sy* = scale factor in the *y* direction, and *Sz* = scale factor in the *z* direction.

Example 6.7: 3D Uniform Scale

Set up a general matrix equation that will uniformly scale 3D objects 5 times larger, and then use it to scale a triangle with vertices at A(50,0,–10), B(0,20,–100), and C(200,150,–50).

Solution

1. Set up the matrix equation. To uniformly scale objects 5 times larger, all three scale factors must be equal to 5:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2. Plug in each old point, and multiply the matrices to find the new location. First, plug in the old location of vertex A(50,0,–10):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 50 \\ 0 \\ -10 \\ 1 \end{bmatrix} = \begin{bmatrix} 5(50) + 0(0) + 0(-10) + 0(1) \\ 0(50) + 5(0) + 0(-10) + 0(1) \\ 0(50) + 0(0) + 5(-10) + 0(1) \\ 0(50) + 0(0) + 0(-10) + 1(1) \end{bmatrix} = \begin{bmatrix} 250 \\ 0 \\ -50 \\ 1 \end{bmatrix}$$

   So A' is the point (250,0,–50).

3. If you repeat step 2 with the old locations of B and C, you get the new locations B'(0,100,–500) and C'(1000,750,–250). By scaling all three vertices, you have scaled the whole triangle 5 times larger.

You can also perform differential scaling in 3D, so let's look at an example of that.

Example 6.8: 3D Differential Scale

Set up a general matrix equation that will scale 3D objects two times taller and half as deep (z direction), and then use it to scale a triangle with vertices at A(50,0,–10), B(0,20,–100), and C(200,150,–50).

Solution

1. Set up the matrix equation. To scale objects two times taller and half as deep, the scale factor in the x direction must be 1 (no change), the scale factor in the y direction must be 2, and the scale factor in the z direction must be 0.5:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2. Now you have to plug in each old point and multiply the matrices to find the new location. First, plug in the old location of vertex A(50,0,–10):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 50 \\ 0 \\ -10 \\ 1 \end{bmatrix} = \begin{bmatrix} 1(50) + 0(0) + 0(-10) + 0(1) \\ 0(50) + 2(0) + 0(-10) + 0(1) \\ 0(50) + 0(0) + 0.5(-10) + 0(1) \\ 0(50) + 0(0) + 0(-10) + 1(1) \end{bmatrix} = \begin{bmatrix} 50 \\ 0 \\ -5 \\ 1 \end{bmatrix}$$

   So A' is the point (50,0,–5).

3. If you repeat step 2 with the old locations of B and C, you get the new locations B'(0,40,–50) and C'(200,300,–25). By scaling all three vertices, you have scaled the whole triangle.

Earlier, we saw a function that could process uniform and non-uniform 2D scaling. Here is a function that works the exact same way for 3D scaling:

```
Matrix4X1 scale3DByMultiply(Matrix4X1 start, float dx, float dy,
➥float dz)
    {
        Matrix4X4 temp;
        Matrix4X1 result;
```

```
        //Zero out the matrix to make sure nothing is left
        ➥uninitialized.
        temp = createFixed4X4Matrix(0);

        //setup the 3x3 for multiplication;
        temp.index[0][0] = dx;
        temp.index[1][1] = dy;
        temp.index[2][2] = dz;
        temp.index[3][3] = 1;

        result = multiplyMatrixNxM(temp,start);
        return result;
    }
```

This operation is performed the same way as the 2D scale function with the addition of a new variable in the form of `dz`. If you want this function to perform a uniform scale on a point or group of points, ask the user for one scale factor and assign the same value to `dx`, `dy`, and `dz`. The provided sample code for this chapter demonstrates this technique to rotate a set of vertices. Be sure to check it out!

Notice that the process for scaling objects is very similar to that of translating by matrix multiplication. This will make it easy to combine translation and scaling later in this chapter. In the meantime, practice scaling 2D and 3D objects with respect to the origin. This will make the combo of scaling with respect to any other point much easier when you get there.

Self-Assessment

1. Set up a general matrix equation that will uniformly scale 2D objects in half, and then use it to scale a triangle with vertices at A(–50,20), B(–10,20), and C(–30,40).

2. Set up a general matrix equation that will make objects tall and skinny by scaling 1/4 in the *x* direction and 3 times in the *y* direction, and then use it to scale a triangle with vertices at A(–50,20), B(–10,20), and C(–30,40).

3. Set up a general matrix equation that will uniformly scale 3D objects 10 times larger, and then use it to scale a triangle with vertices at D(0,30,–100), E(–50,100,–20), and F(–20,0,–300).

4. Set up a general matrix equation that will scale objects twice as big in the *x* direction, no change in the *y* direction, and half as big in the *z* direction, and then use it to scale a triangle with vertices at D(0,30,–100), E(–50,100,–20), and F(–20,0,–300).

Rotation

There are actually two different approaches to rotating objects. One method uses quaternions, but the math is complex. The other method is very similar to that of scaling and translating, because it uses a similar matrix equation. This second method is called **Euler rotation** (pronounced "oiler rotation," not "yewler"). You'll set up a matrix equation first in 2D and then in 3D. As soon as the matrix equation is set up, all you have to do is plug in the vertices one at a time and multiply the matrices to find the new position.

2D Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where θ is the angle of rotation.

As soon as you know the angle of rotation, you can take the sine and cosine and turn them into decimal numbers so that you can multiply the matrices. You might need to revisit Chapter 3, "Trigonometry Snippets," for a discussion of angles. Remember that positive angles rotate counterclockwise, and negative angles rotate clockwise. Also, if you know the angle ahead of time and you're calculating the sine and cosine, you can leave the angle in degrees. However, if the computer program is fed the angle and it must take the sine and cosine, remember that the angle must be in radian measure.

Programming Transformations Using Matrices

When using the widely accepted `math.h` header, key functions such as sine and cosine have arguments that accept their parameters in radians. It's important to properly format the values passed to these functions before asking them to do any calculations. Here are a couple of quick macro functions that you can use when needed:

```
#define RadsToDegrees( radian ) ((radian) * (180.0f / M_PI))
```

As you probably recognize, this first method takes a radian argument and returns a degree conversion. More on the definition of `M_PI` in just a moment.

```
#define DegreesToRads( degrees ) ((degrees) * (M_PI/ 180.0f))
```

This function does the reverse and gives us radian arguments for our degree values.

The usage of `M_PI` comes from the `math.h` header. Make sure that you not only include the math header in your programs, but also set the directive to use the math header definitions for things like `PI`. The listed term `M_PI` comes from the math header and is used to represent pi throughout the chapter. If the math header defines are supported on your platform, you should be able to access them like this:

```
#define _USE_MATH_DEFINES
#include <math.h>
```

It's important to set the *`USE`*`_MATH_DEFINES` definition before you include the library so that it knows in advance to work with those parameters.

Example 6.9: 2D Rotation

Set up a general matrix equation that will rotate 2D objects 90°, and then use it to rotate the triangle pictured in Figure 6.6 with vertices at A(50,40), B(100,40), and C(75,200).

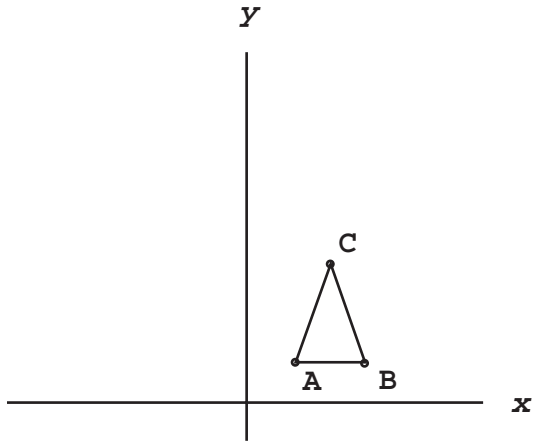

Figure 6.6 A 2D triangle to be rotated.

Solution

1. Set up the matrix equation. The angle of rotation is 90°:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. Now you have to plug in each old point and multiply the matrices to find the new location. First, plug in the old location of vertex A(50,40):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 50 \\ 40 \\ 1 \end{bmatrix} = \begin{bmatrix} 0(50) - 1(40) + 0(1) \\ 1(50) + 0(40) + 0(1) \\ 0(50) + 0(40) + 1(1) \end{bmatrix} = \begin{bmatrix} -40 \\ 50 \\ 1 \end{bmatrix}$$

   So A' is the point (–40,50).

3. If you repeat step 2 with the old locations of B and C, you get the new locations B'(–40,100) and C'(–200,75). By rotating all three vertices, you have rotated the whole triangle. The original location and the new location are shown in Figure 6.7.

Rotating objects in 2D is an important thing to be able to process. The calculations are pretty straightforward. Let's take a look at a function that rotates a 2D point:

```
Matrix3X1 rotate2D(Matrix3X1 start, float theta)
     {
          Matrix3X3 temp;
          Matrix3X1 result;
          //Zero out the matrix.
          temp = createFixed3X3Matrix(0);
          //place the needed rotational values into the matrix.
          temp.index[0][0] = cos(DegreesToRads(theta));
          temp.index[1][1] = cos(DegreesToRads(theta));
          temp.index[2][2] = 1;
          temp.index[0][1] = -1*(sin(DegreesToRads(theta)));
          temp.index[1][0] = sin(DegreesToRads(theta));
          temp.index[2][2] = 1;
          result = multiplyMatrixNxM(temp,start);
          return result;
     }
```

You can see we are using the `DegreesToRads()` macro that was referenced at the beginning of the chapter. Stepping through, we initialize the values of the rotation matrix based on the amount of rotation specified by the user. Once the matrix is set up, we can then just multiply the current matrix by the rotation matrix and we will have the newly translated points.

Here is some sample code that will allow us to translate a whole rectangle around a center point:

```
void rotate2D()
     {
          Matrix3X1 start,temp;
          float height,width,theta;
          cout<<"Let's rotate a 2D!\n";
          cout<<"Please enter the coordinates and dimensions.\n";
          cout<<"X coordinates\n";
          cin>>start.index[0];
          cout<<"Now the Y coordinate.\n";
```

```
        cin>>start.index[1];
        cout<<"Enter the rectangle's height.\n";
        cin>>height;
        cout<<"Enter the rectangle's width.\n";
        cin>>width;
        //make sure the last part of the matrix is a 1.
        start.index[2] = 1;
        cout<<endl;
        cout<<"Now enter the amount to rotate by in degrees.\n";
        cin>>theta;
        //Now that we have our info, lets rotate!
        temp = rotate2D(start,theta);

    //This gives the new locations calced from the temp matrix.
        width = temp.index[0]+width;
        height = temp.index[1]+height;

        cout<<"The right coordinate of the rectangle
        ➥<<width<<","<<temp.index[1]<<"\n";
        cout<<"The bottom coordiante of the rectangle
        ➥"<<height<<","<<temp.index[0]<<"\n";
}
```

The main reason that rotation is so important with respect to this square is that most 2D images are represented in a square format, even if the alpha channel blocks out some of the colors from the user. It is common to use the current position to handle the display of graphics. This rotation algorithm lets the user specify what facing he wants a particular graphic to have. Some games will directly tie the input from the user to the facing and process the updates using a similar rotation algorithm.

In Figure 6.7, notice that the triangle rotated and moved. You might have expected it to stay in the same place and rotate. Unfortunately, that is also a combo.

NOTE

Again, this issue can be resolved by setting up a local origin for the model (also called its pivot).

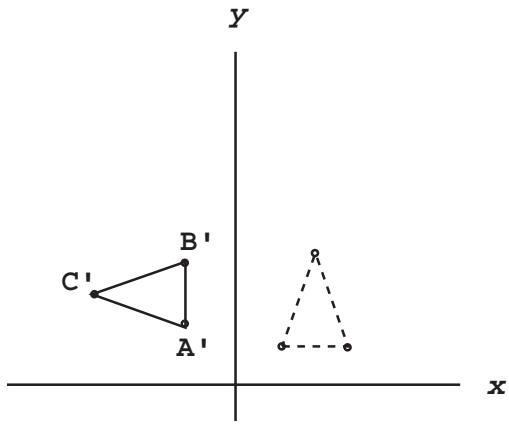

Figure 6.7 A 2D triangle before and after rotation.

By default, the rotation matrix rotates objects with respect to the origin, just like the scaling matrix does. The resulting effect is that the object appears to be orbiting about the origin. If you want the object to rotate about its own center or a particular vertex so that it appears to be tipping over, that is a combo, and we'll discuss that process in the next section.

In the meantime, let's take a look at 3D rotation. 2D has only one plane to rotate in, much like a flat screen. However, 3D has three different planes to rotate in—the *xy* plane, the *yz* plane, and the *xz* plane. These three planes are shown in Figure 6.8.

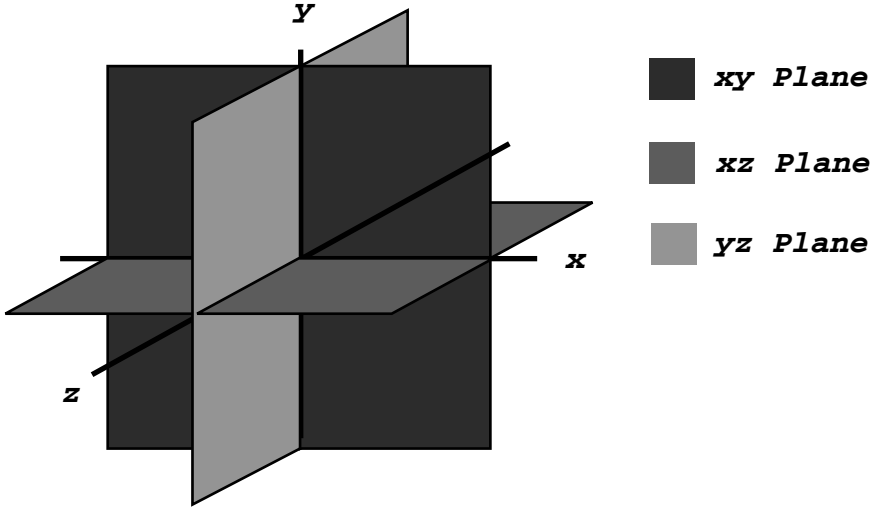

Figure 6.8 The three planes in 3D.

Notice that the *xy* plane is the same as the flat 2D plane you're used to working with for the computer screen. Because there are three separate planes to rotate within, there are three separate rotation matrices for 3D. Let's examine them one at a time.

3D Rotation About the Z-Axis (Roll)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where θ is the angle of rotation.

Remember that the z-axis comes in and out of the screen, so if you rotate around the z-axis it's the same as rotating inside the flat screen (*xy* plane). That's why this rotation matrix looks almost the same as the 2D rotation matrix; it just has an extra row and column of identity to make it a 4×4 matrix. Notice also that this particular rotation is labeled *roll*. This name comes from flight simulator terminology. If you stand up with your arms extended to the side and bend from side to side, it's the same motion as a plane rolling.

To rotate around the z-axis, use the following function:

```
Matrix4X1 rotate3DZ(Matrix4X1 start, float theta)
    {
        Matrix4X4 temp;
        Matrix4X1 result;

        //Zero out the matrix.
        temp = createFixed4X4Matrix(0);

        //put the needed rotation values into the matrix from
        ➥theta.
        temp.index[0][0] = cos(DegreesToRads(theta));
        temp.index[1][1] = cos(DegreesToRads(theta));
        temp.index[2][2] = 1;
        temp.index[3][3] = 1;
```

```
        temp.index[0][1] = -1*(sin(DegreesToRads(theta)));
        temp.index[1][0] = sin(DegreesToRads(theta));


        result = multiplyMatrixNxM(temp,start);
        return result;
    }
```

Make sure that when you are setting up your own rotation algorithms that you watch the radian calculations. Also make sure you put the values in the proper positions.

3D Rotation About the X-Axis (Pitch)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where θ is the angle of rotation.

The x-axis runs left and right, so if you rotate around the x-axis, it's the same motion as standing up and bending over to take a bow.

In flight simulator terms, that's the same motion as pitching, so many programmers call rotation about the x-axis *pitch*. Notice the shift in the location of the sines and cosines. The first row and column typically represent *x* values, and in this case the first row and column have identity values. If you're rotating about the x-axis, the x-coordinates of the vertices shouldn't change, so that makes sense. This little trick might help you memorize the 3D rotation matrices. It works for rotating about the y- and z-axes as well.

Here's one way to rotate about the X-axis:

```
Matrix4X1 rotate3DX(Matrix4X1 start, float theta)
    {
        Matrix4X4 temp;
        Matrix4X1 result;
```

```
        //Zero out the matrix.
        temp = createFixed4X4Matrix(0);

        //place needed rotation values into the matrix based on
        ➥theta.
        temp.index[0][0] = 1;
        temp.index[1][1] = cos(DegreesToRads(theta));
        temp.index[2][2] = cos(DegreesToRads(theta));
        temp.index[3][3] = 1;

        temp.index[1][2] = -1*(sin(DegreesToRads(theta)));
        temp.index[2][1] = sin(DegreesToRads(theta));


        result = multiplyMatrixNxM(temp,start);
        return result;
    }
```

The main changes here are the locations of the sine and cosine values of theta. This example is used in the sample source code provided for this chapter.

3D Rotation About the Y-Axis (Yaw)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where θ is the angle of rotation.

Notice in this case that the second row and column (which represent *y* values) have identity values. Rotation about the y-axis in simulation terminology is the same as *yaw*. If you stand in place and spin around like a ballerina, that's the motion described as yaw. You can also visualize yaw by picturing someone spinning around a vertical pole. As soon as you have each matrix equation set up, all you have to do is plug in the original location of each vertex and multiply it to find the new location.

Unfortunately, these three types of rotation must be planned separately, but you'll see in the next section that they can eventually be combined.

Here's a function to rotate about the Y-axis:

```
Matrix4X1 rotate3DY(Matrix4X1 start, float theta)
     {
          Matrix4X4 temp;
          Matrix4X1 result;

          //Zero out the matrix.
          temp = createFixed4X4Matrix(0);

          //place the rotational values into the matrix based on
          ➥theta.
          temp.index[0][0] = cos(DegreesToRads(theta));
          temp.index[1][1] = 1;
          temp.index[2][2] = cos(DegreesToRads(theta));
          temp.index[3][3] = 1;

          temp.index[2][0] = -1*(sin(DegreesToRads(theta)));
          temp.index[0][2] = sin(DegreesToRads(theta));


          result = multiplyMatrixNxM(temp,start);
          return result;
     }
```

Remember to double-check the locations of the sine and cosine values.

NOTE

These three rotation matrices are also designed to rotate with respect to the origin, so they all have the same orbiting effect as the 2D rotation matrix. Again, if you want to rotate with respect to another point, that has to be set up as a combo.

Example 6.10: 3D Rotation About the Y-Axis

Set up a general matrix equation that will rotate 3D objects π^R about the y-axis, and then use it to rotate a triangle with vertices at A(100,0,–50), B(40,–30,0), and C(–20,100,50).

Solution

1. Set up the matrix equation. The angle of rotation is π^R or 180° (see Chapter 3 for the conversion):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos 180° & 0 & \sin 180° & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 180° & 0 & \cos 180° & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2. Now you have to plug in each old point and multiply the matrices to find the new location. First, plug in the old location of vertex A(100,0,–50):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 100 \\ 0 \\ -50 \\ 1 \end{bmatrix} = \begin{bmatrix} -1(100) + 0(0) + 0(-50) + 0(1) \\ 0(100) + 1(0) + 0(-50) + 0(1) \\ 0(100) + 0(0) - 1(-50) + 0(1) \\ 0(100) + 0(0) + 0(-50) + 1(1) \end{bmatrix} = \begin{bmatrix} -100 \\ 0 \\ 50 \\ 1 \end{bmatrix}$$

   So A' is the point (–100,0,50).

3. If you repeat step 2 with the old locations of B and C, you get the new locations B'(–40,–30,0) and C'(20,100,–50). By rotating all three vertices, you have rotated the whole triangle.

If you can rotate about the y-axis, you can rotate about the other two axes as well. They all work the same way. Notice that the examples use multiples of 90°, so the sines and cosines are all 0s and 1s. However, be aware that for other angles, the sines and cosines have decimal numbers, so the new locations aren't always nice whole numbers. Also, remember that all rotation matrices are set up for rotation about the origin. The next section discusses how to rotate with respect to any other point.

The functions from last chapter that were used to create random matrices are a good way to test the matrix multiplications. You should also give the user a way to enter manual values as well. Check the sample source code for this chapter to see 3D rotation around each axis in action.

Self-Assessment

1. Set up a general matrix equation that will rotate 2D objects 30°, and then use it to rotate a triangle with vertices at A(–10,50), B(80,0), and C(–50,100).

2. Set up a general matrix equation that will make 3D objects stand in place and rotate 1 degree at a time (about the y-axis), and then use it to rotate a triangle with vertices at D(0,20,–50), E(–10,0,20), and F(30,20,0).

3. Set up a general matrix equation that will make 3D objects pitch 45°, and then use it to rotate a triangle with vertices at G(0,30,–100), H(–50,100,–20), and J(–20,0,–300).

4. Set up a general matrix equation that will rotate 3D objects $(\pi/2)^R$ about the z-axis, and then use it to rotate a triangle with vertices at L(0,30,–100), M(–50,100,–20), and N(–20,0,–300).

Concatenation

This whole chapter has been building up to this particular section. **Concatenation** is just a fancy name for combining transformation matrices into a single combo matrix. Earlier in this chapter I mentioned several common combinations: scaling with respect to an object's own center point, rotating with respect to the center point (or a vertex), and combining the 3D rotations. We'll use them as examples to demonstrate the concatenation process.

This process can be used for any combination of translation, scaling, and rotation, not just the examples mentioned earlier. Any time you plan to perform more than one transformation within the same frame, you can save *a lot* of processor time by combining them. It might take you more time up front when you're coding, but it will greatly enhance performance when the game is running.

Let's step through the process using the example of rotating an object about its own center point. Let's revisit the 2D triangle you rotated in Example 6.9. You rotated that triangle 90° with respect to the origin. This time, you'll rotate it 90° with respect to its own center point, which is (75,93). This particular combo is a three-step process (see Figure 6.9):

1. Translate so that the center is at the origin (left 75 and down 93).
2. Rotate 90°.
3. Translate so that the center is back in its original position (right 75 and up 93).

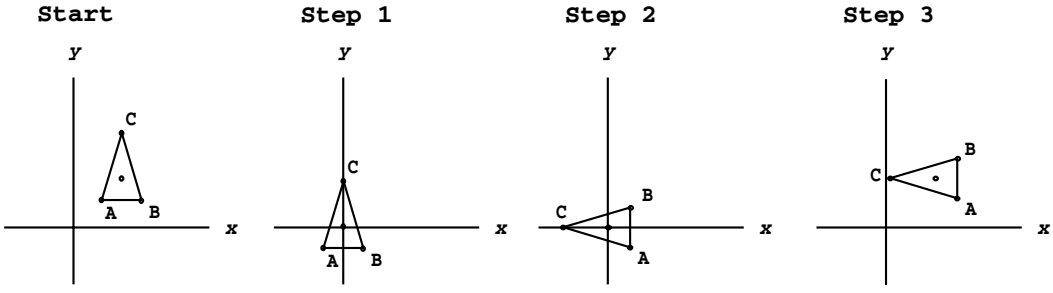

Figure 6.9 The three-step process.

By moving the center point to the origin, you almost trick the computer into rotating with respect to that point instead. You can use this same three-step process to scale with respect to any point other than the origin. As soon as you know the steps of your combo and the order in which you want to perform them, the next step is setting up a stack of transformation matrices so that they can be combined. Start with the old location on the far right. Then stack to the left (in order) the steps of your combo. In this case, the new location is equal to the following:

$$\begin{bmatrix} 1 & 0 & 75 \\ 0 & 1 & 93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -75 \\ 0 & 1 & -93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

NOTE

This particular example has only three steps, but your combo can have as many steps as you like. As long as you want them all to happen in the same frame, go ahead and combine them.

Be very careful when setting up the stack. Remember from Chapter 5 that matrix multiplication is not commutative. This means that the order in which you multiply matrices is very important. If you accidentally flip two matrices in the stack, you end up at a completely different location than you expected, so always keep them in chronological order.

As soon as you have all the steps set up, the last step is to combine them into one matrix. The only way to do this is to multiply them all together. In this case, the new location matrix is equal to the following:

$$\begin{bmatrix} 1 & 0 & 75 \\ 0 & 1 & 93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -75 \\ 0 & 1 & -93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 75 \\ 0 & 1 & 93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -75 \\ 0 & 1 & -93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -1 & 75 \\ 1 & 0 & 93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -75 \\ 0 & 1 & -93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -1 & 168 \\ 1 & 0 & 18 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Just keep multiplying to the right until you get down to one matrix. This final matrix performs all three steps at once. In this particular case, all the player sees is the initial and final positions shown in Figure 6.9. The in-between steps are never seen, because they all happen within the same frame. Now, instead of placing all three matrices in the code, all you need is this combo matrix equation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 168 \\ 1 & 0 & 18 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This might seem like a lot of work up front, and it is. However, it's better for you to do all that matrix multiplication once in the beginning than to have the computer repeat it for *every* vertex. A complex model has thousands of vertices, so this will have a significant effect.

At this point, the process is the same as all the others: Just plug in each vertex one at a time, and multiply the matrices to see where it moved to. By moving all the individual vertices, you end up moving the whole object. In this case, when you plug in the three original vertices A(50,40), B(100,40), and C(75,200), you get the new locations A'(128,68), B'(128,118), and C'(–32,93). Figure 6.10 shows both the original and final locations.

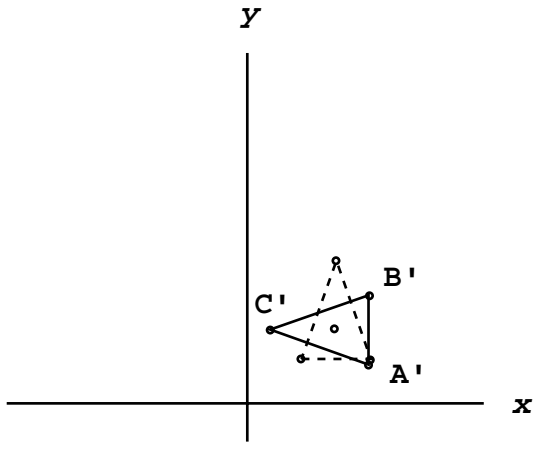

Figure 6.10 2D rotation with respect to the center point.

This same process also works in three dimensions. Let's look at an example of a 3D combo.

Example 6.11: 3D Scaling with Respect to the Center Point

Suppose you want a 3D model in your game to stay in the same place but scale down to half the size. Set up a general matrix equation that will scale any 3D object in half with respect to its own center point (x_c,y_c,z_c) using a single combo matrix.

Solution

1. Organize the three steps of the combo:
   - ➤ Translate so that the center is at the origin.
   - ➤ Uniformly scale in half.
   - ➤ Translate so that the center is back in its original position.

2. Now you can set up a matrix equation with the individual transformation matrices stacked in order (right to left):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_c \\ 0 & 1 & 0 & y_c \\ 0 & 0 & 1 & z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3. The last (but hardest) step is to multiply all the transformation matrices together. Remember that the order is important, so be sure to multiply left to right.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_c \\ 0 & 1 & 0 & y_c \\ 0 & 0 & 1 & z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 0 & x_c \\ 0 & 0.5 & 0 & y_c \\ 0 & 0 & 0.5 & z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 0 & (0.5)x_c \\ 0 & 0.5 & 0 & (0.5)y_c \\ 0 & 0 & 0.5 & (0.5)z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4. This last line is the final matrix equation that will scale any 3D object in half with respect to its own center point in just one step.

The preceding section talked about 3D rotation and showed that it must be broken into three separate parts: roll, pitch, and yaw. You might need to flip back to that section before tackling the next example, where you'll combine the three steps into one step.

Example 6.12: 3D Rotation Combined

Set up a general matrix equation that will make any 3D object roll 30°, pitch 180°, and yaw 90° using a single combo matrix. Then apply it to a triangle with vertices at A(200,0,–30), B(0,50,–150), and C(40,20,–100).

Solution

1. Organize the three steps of the combo:
   - ➤ Rotate 30° about the z-axis.
   - ➤ Rotate 180° about the x-axis.
   - ➤ Rotate 90° about the y-axis.
2. Now you can set up a matrix equation with the individual transformation matrices stacked in order (right to left):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos 90° & 0 & \sin 90° & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 90° & 0 & \cos 90° & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 180° & -\sin 180° & 0 \\ 0 & \sin 180° & \cos 180° & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 30° & -\sin 30° & 0 & 0 \\ \sin 30° & \cos 30° & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3. The last (but hardest) step is to multiply all the transformation matrices together. Order is important, so be sure to multiply left to right.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.8660 & -0.5 & 0 & 0 \\ 0.5 & 0.8660 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.8660 & -0.5 & 0 & 0 \\ 0.5 & 0.8660 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -0.5 & -0.8660 & 0 & 0 \\ -0.8660 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4. Now that you have a matrix equation that performs all three rotations at once, all you have to do is plug in each vertex and multiply the matrices to find the new location, just like before. For example, if you plug in the old location of vertex A(200,0,–30), you get the following:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -0.5 & -0.8660 & 0 & 0 \\ -0.8660 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 200 \\ 0 \\ -30 \\ 1 \end{bmatrix} = \begin{bmatrix} 0(200) + 0(0) - 1(-30) + 0(1) \\ -0.5(200) - 0.8660(0) + 0(-30) + 0(1) \\ -0.8660(200) + 0.5(0) + 0(-30) + 0(1) \\ 0(200) + 0(0) + 0(-30) + 1(1) \end{bmatrix} = \begin{bmatrix} 30 \\ -100 \\ 173.2 \\ 1 \end{bmatrix}$$

So A' is the point (30,–100,173.2).

5. If you repeat step 4 with the old locations of B and C, you get the new locations B'(150,–43.3,25) and C'(100,–37.32,–24.64). By rotating all three vertices, you have rotated the whole triangle.

Programming combo matrices represent the last big step in transformations and require complete understanding of everything already presented in this chapter.

The first step is to actually create the combo matrix. Fortunately, we already have all the tools we need to build this matrix. Here is a function that will create a 3D combo matrix in preparation for the actual combo transformation:

```
Matrix4X4 createRotationCombo(float thetax,float thetay,float thetaz)
    {
        Matrix4X4 X,Y,Z,temp,result;

        X = createFixed4X4Matrix(0.0f);
        Y = createFixed4X4Matrix(0.0f);
        Z = createFixed4X4Matrix(0.0f);
        temp = createFixed4X4Matrix(0.0f);
        result = createFixed4X4Matrix(0.0f);

        //place the needed X rotational values into the matrix.
        X.index[0][0] = 1;
        X.index[1][1] = cos(DegreesToRads(thetax));
        X.index[2][2] = cos(DegreesToRads(thetax));
        X.index[3][3] = 1;

        X.index[2][1] = -1*(sin(DegreesToRads(thetax)));
        X.index[1][2] = sin(DegreesToRads(thetax));
```

```
        //place the needed Y-Axis rotational values into the
        ➥matrix.
        Y.index[0][0] = cos(DegreesToRads(thetay));
        Y.index[1][1] = 1;
        Y.index[2][2] = cos(DegreesToRads(thetay));
        Y.index[3][3] = 1;

        Y.index[2][0] = -1*(sin(DegreesToRads(thetay)));
        Y.index[0][2] = 1;sin(DegreesToRads(thetay));

        //place the needed Z-axis rotational values into the
        ➥matrix.
        Z.index[0][0]  = cos(DegreesToRads(thetaz));
        Z.index[1][1]  = cos(DegreesToRads(thetaz));
        Z.index[2][2]  = 1;
        Z.index[3][3]  = 1;

        Z.index[0][1]  = -1*(sin(DegreesToRads(thetaz)));
        Z.index[1][0]  = sin(DegreesToRads(thetaz));

        //Create the single Combo Matrix.
        temp = multiply4X4Matrices(Y,X);
        result = multiply4X4Matrices(temp,Z);
        return result;
    }
```

Most of this should look familiar. We took the guts out of each of the rotation functions we saw earlier in the chapter and set them up directly within this function. Once we have prepared the various matrices, we multiply them right to left. In this case, we are using temporary placeholders to bridge the limitation of our functions. You can process these in slightly faster ways without assigning the values to an intermediary matrix—just be careful and make sure that you are ordering them properly.

The final output of this function is just the finalized combo matrix; we will still need to process the translation. Now that the hard work is done, let's take a look at the transformation:

```
Matrix4X1 rotate3DWithCombo(Matrix4X4 combo, Matrix4X1 vertex)
     {
       Matrix4X1 temp;
       temp = multiplyMatrixNxM(combo,vertex);
       return temp;
     }
```

We continue through and multiply each vertex by the combo matrix. Remember that the rotation values for the different axis are contained within the single combo matrix. We need to make this rotation call against each vertex to discover the new orientation of the object. At this point, this process probably is starting to seem almost trivial.

By nature, programmers are always looking for math shortcuts. Unfortunately with combo matrices, the numbers become intertwined, so there's no shortcut here. The only way to find the correct values for each entry in the combo matrix is to stack them and multiply. There is some good news, though. When you go back to debug, an interesting pattern emerges. Let's look back at the example used to introduce this section: rotating 90° with respect to a triangle's own center point. The combo matrix equation you found is as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 168 \\ 1 & 0 & 18 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

If you look closely at this combo matrix, you can see that the rotation information is stored in the upper-left 2×2 section, and the overall translation information is stored in the last column.

Try setting up a few different combos; you should find the following pattern.

2D Combo Matrix

For every 2D combo matrix:

$$\begin{bmatrix} r_{00} & r_{01} & t_x \\ r_{10} & r_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

entries with an *r* store scaling and rotation information, and entries with a *t* store overall translation information.

Notice that the bottom row of the combo is always (0,0,1). This means that when you go back to debug, you might want to first check the bottom row, because that's an easy fix. If that's not the problem, look at the top two rows. If something is off with the translation, check the last column. If something is not right with scaling or rotation, check the first two columns. This process of elimination might save you some time when you go back to debug.

The same patterns also apply to 3D combo matrices.

3D Combo Matrix

For every 3D combo matrix:

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

entries with an *r* store scaling and rotation information, and entries with a *t* store overall translation information.

Again, scaling and rotation information is stored in the upper-left 3×3 section. Examples 6.11 and 6.12 verify this. Again, the last column holds the overall translation information. You also might have noticed that the first row corresponds to *x,* the second row corresponds to *y,* the third row corresponds to *z,* and the bottom row is always the same: (0,0,0,1). All these tricks should help you debug faster.

The last part of this section revisits the idea of the transpose, so you might need to flip back to Chapter 5 for a quick refresher. When you're ready, we'll look at 2D matrix equations. A typical combo matrix equation looks like this:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & t_x \\ r_{10} & r_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

You might have noticed that this chapter uses single columns to represent the vertices. The good news is that OpenGL and most traditional mathematicians use the single-column format as well. The bad news is that DirectX uses single rows instead of single

columns, so you have to make a few adjustments to convert from OpenGL format to DirectX.

The first step is to take the transpose of everything. That sends the single columns to single rows, and it adjusts the combo matrix to match. But you're not done yet. Remember that the order in which you multiply matrices is important. When you used single columns, you multiplied a 3×3 matrix by a 3×1. However, when you transposed everything, you ended up with a 3×3 times a 1×3, which is undefined. This leads to the second step, which is to reverse the order of multiplication. These two steps give you a matrix equation that is ready for DirectX:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} r_{00} & r_{10} & 0 \\ r_{01} & r_{11} & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

Conversion Between the OpenGL and DirectX Formats

AB = $\mathbf{B^T A^T}$ for matrices **A** and **B**:

$$\underset{A}{\begin{bmatrix} r_{00} & r_{01} & t_x \\ r_{10} & r_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix}} \underset{B}{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}} = \underset{B^T}{\begin{bmatrix} x & y & 1 \end{bmatrix}} \underset{A^T}{\begin{bmatrix} r_{00} & r_{10} & 0 \\ r_{01} & r_{11} & 0 \\ t_x & t_y & 1 \end{bmatrix}}$$

This same conversion process also works in 3D. Just remember to take the transpose of each matrix *and* reverse the order so that the matrix multiplication is defined.

NOTE

Even if you don't plan to use OpenGL, you might find it easier to set up the combos using single-column format and then switch to single rows at the last minute. Matrix multiplication is always easier going left to right.

This entire chapter built up to the point where you could finally construct combo matrices. You started by performing translation using matrix addition. Again, if all you want to do is translate, use matrix addition. However, if you know you need to set up a combo with scaling and/or rotation, you must use matrix multiplication. Remember to always start with the original vertex on the far right, stack to the left, and multiply to the right. In the end you can always switch to single-row format if you need to work

with DirectX. I think you'll find that whether they use OpenGL or DirectX, transformation matrices are still at the heart of most 3D games.

Self-Assessment

1. Set up a combo matrix that will rotate a 2D object –90° with respect to its own center point, which is (10,50), all in one step.
2. Use the matrix in question 1 to rotate a triangle with vertices at A(–10,50), B(90,0), and C(–50,100).
3. Find a general matrix equation that will rotate any 2D object -90° with respect to its own center point, which is (x_c,y_c), all in one step.
4. Convert your answer to question 3 to a format that can be used by DirectX.
5. Set up a general matrix equation that will make any 3D object roll 45°, pitch 90°, and yaw –90° using a single combo matrix. Then apply it to a triangle with vertices at D(200,0,–30), E(0,50,–150), and F(40,20,–100).
6. Convert your matrix equation from question 3 to a format that can be used by DirectX.

Visualization Experience

On the CD-ROM, you will find a demo named Transformations. Here's a brief description from the programmers:

This program was designed to help you understand the principles of 2D transformations applied to an object. Matrices are used to rotate or translate a quad around the grid. Users can use this application to get a visual representation while learning the basics of matrix concatenation.

The user can add, select, move, and trace quads on the grid as well as undo and redo actions applied to a quad. During the experience, the user can view information about the current location of each quad.

—Michael Wigand and Michael Fawcett

This demo is also interactive. You can plug values into a transformation matrix and then see how the object moves as a result. You might want to try setting up a few examples by hand first and then run the demo to see if it has the effect you expected.

You can run the demo by double-clicking the Transformations.exe file. As soon as the demo is loaded, you should see a window with a red grid and a colored box in the middle. You can move this box around the screen using transformation matrices. The toolbar is labeled for you in Figure 6.11 so that you can follow the instructions listed next.

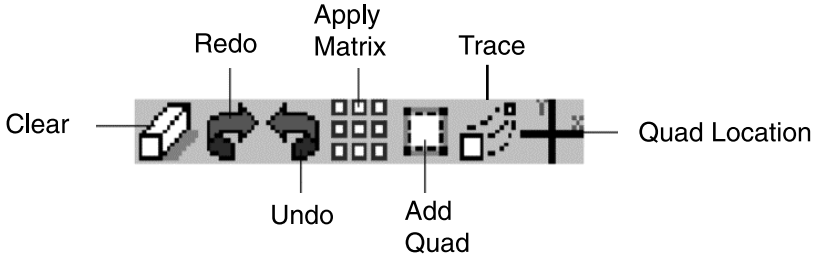

Figure 6.11 The toolbar for the Transformations demo.

To get started, click the Apply matrix button. A window pops up with a 3×3 matrix. Enter the following values:

$$\begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

The box should move five units to the right and eight up.

Click the Trace button on the toolbar. This leaves an imprint of the current location. Click the Add a quad button on the toolbar. This adds a new box with the bottom-left vertex at the origin.

This time, try to move the box five units to the left and three up. After you have done so, leave the box selected (outlined in white), trace its location, and apply the following matrix to it:

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

What happened this time? That's right—the object scaled 3 times larger with respect to the origin. This means that it moved away from the origin as it got larger. Click the Undo button to go back to the previous location. See if you can scale the box 3 times larger without moving away from the origin. Remember that this is a three-step combo:

$$\begin{bmatrix} 1 & 0 & -5 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix}$$

If you multiply these three matrices in the correct order, you should get the following combo matrix:

$$\begin{bmatrix} 3 & 0 & 10 \\ 0 & 3 & -6 \\ 0 & 0 & 1 \end{bmatrix}$$

This time you scaled 3 times larger with respect to the box's center instead of the origin. If your screen is starting to get too messy, you can click the Clear button at this point to wipe the slate clean. If not, just create a new quad for the next set of transforms.

This time, move the box nine units to the right and four units up using this matrix:

$$\begin{bmatrix} 1 & 0 & 9 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

Trace that location and then rotate 90° with this matrix:

$$\begin{bmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

What happened? It looks like the box just translated up and to the left, but that's not entirely what happened. Look at the blue corner of the box. Press Undo and Redo a couple times. You might notice that the box did in fact rotate 90° in addition to translating. Otherwise, you can also view the coordinates of all four vertices by clicking the Quad position button on the toolbar. Write down the coordinates in the new position,

press Undo, and view the coordinates of the old position. You can also verify that the box rotated that way. So what really happened is that the box rotated *with respect to* the origin.

Try to rotate the box with respect to the red corner instead. That way, it stays in the same place and looks like it's just tipping over. If you're not currently at the old location, click the Undo button to go back. Now you can view the quad position to find the coordinates of the red vertex. You should find that the red corner is currently at (9,4). Now you can set up a combo to rotate 90° with respect to the red vertex. You can either perform each of these steps individually or multiply them together for one combo matrix:

$$\begin{bmatrix} 1 & 0 & 9 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & -9 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 13 \\ 1 & 0 & -5 \\ 0 & 0 & 1 \end{bmatrix}$$

Did you get the outcome you expected? I suggest that you try a few combos of your own. Experiment with the demo a little bit. You might want to consider performing your own combos by hand first and then use the demo to check your work. You can write down the four original vertices and calculate where they should end up and then view the quad position to see if that's where the computer places them in the demo. Hopefully they come out to be the same!

Self-Assessment Solutions

Translation

1. D'(70,60)
2. E'(–10,180)
3. F'(140,–20)
4. G'(250,–330,–50), H'(140,–300,–40), J'(–350,–250,–100)
5. L'(–300,50), M'(–150,–60), N'(–130,20)
6. Yes. G'(250,–330,–50), H'(140,–300,–40), J'(–350,–250,–100)

Scaling

1. A'(–25,10), B'(–5,10), C'(–15,20)
2. A'(–12.5,60), B'(–2.5,60), C'(–7.5,120)
3. D'(0,300,–1000), E'(–500,1000,–200), F'(–200,0,–3000)
4. D'(0,30,–50), E'(–100,100,–10), F'(–40,0,–150)

Rotation

1. A'(–33.66,38.30), B'(69.28,40), C'(–93.3,61.6)
2. D'(–0.8725,20,–49.9925), E'(–9.6495,0,20.1715), F'(29.9955,20,–0.5235)
3. G'(0,91.91,–49.49), H'(–50,84.84,56.56), J'(–20,212.13,–212.13)
4. L'(–30,0,–100), M'(–100,–50,–20), N'(0,–20,–300)

Concatenation

1. $$\begin{bmatrix} 0 & 1 & -40 \\ -1 & 0 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$
2. A'(10,70), B'(–40,30), C'(60,110)
3. $$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & (-y_c + x_c) \\ -1 & 0 & (x_c + y_c) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$
4. $$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ (-y_c + x_c) & (x_c + y_c) & 1 \end{bmatrix}$$
5. D'(–141.4,30,141.4), E'(–35.4,150,–35.4), F'(–42.4,100,14.1)

6.

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} -.707 & 0 & .707 & 0 \\ -.707 & 0 & -.707 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$