CHAPTER 4

Creating 3D Objects

IN THIS CHAPTER

In this chapter, we examine 3D concepts and how the XNA Framework exposes different types and objects that allow us to easily create 3D worlds. We will create a couple 3D demos that explain the basics. We will also create 3D objects directly inside of our code. Finally, we will move these objects on the screen.

Vertices

Everything in a 3D game is represented by 3D points. We can use one of two ways to get 3D objects on the screen: We can plot the points ourselves, or we can load them from a 3D file (which has all the points stored already). Later, in Chapter 6, "Loading and Texturing 3D Objects," you will learn how to load 3D files to use in our games. For now, we are going to create the vertices ourselves.

We defined these vertices with an x, y, and z coordinate (x, y, z). In XNA we represent a vertex with a vector, which leads us to the next section.

Vectors

XNA provides three different vector structs for us—`Vector2`, `Vector3`, and `Vector4`. `Vector2` only has an x and y component. We typically use this 2D vector in 2D games and when working with a texture. `Vector3` adds in the z component. Not only do we store vertices as a vector, but we also store velocity as a vector. We discuss velocity in Chapter 16, "Physics Basics." The last vector struct that XNA provides for us is a 4D struct appropriately called `Vector4`. Later examples in this book will use this struct to pass color information around because it has four components.

We can perform different math operations on vectors, which prove to be very helpful. We do not discuss 3D math in detail in this book because there are many texts available that cover it. Fortunately, XNA allows us to use the built-in helper functions without needing a deep understanding of the inner workings of the code. With that said, it is extremely beneficial to understand the math behind the different functions.

Matrices

In XNA a matrix is a 4×4 table of data. It is a two-dimensional array. An identity matrix, also referred to as a *unit matrix*, is similar to the number 1 in that if we multiply any other number by 1 we always end up with the number we started out with (5 * 1 = 5). Multiplying a matrix by an identity matrix will produce a matrix with the same value as the original matrix. Although the identity matrix can be useful in and of itself, the key is actually that the individual fields in the 4×4 array are structured so that we can combine many transformations into a single matrix. The XNA Framework provides a struct to hold matrix data—not surprisingly, it is called `Matrix`.

Transformations

The data a matrix contains is called a transformation. The three common types of transformations are translation, scaling, and rotation. These transformations do just that: They transform our 3D objects.

Translation

Translating an object simply means we are moving the object. We translate an object from one point to another point by moving each point inside of the object correctly.

Scaling

Scaling an object will make the object larger or smaller. This is done by actually moving the points in the object closer together or further apart, depending on whether we are scaling down or scaling up.

Rotation

Rotating an object will turn the object on one or more axes. By moving the points in 3D space, we can make our object spin.

Transformation versus Translation

A translation is a type of a transformation. Transformations include translations (movement), scaling (size), and rotation. A translation is one type of transformation, but they are not the same thing.

Transformations Reloaded

An object can have one transformation applied to it, or it can have many transformations applied to it. We might only want to translate (move) an object, so we can update the object's world matrix to move it around in the world. We might just want the object to spin around, so we apply a rotation transformation to the object over and over so it will rotate. We might need an object we created from a 3D editor to be smaller to fit better in our world. In that case we can apply a scaling transformation to the object. Of course, we might need to take this object we loaded in from the 3D editor and scale it down and rotate it 30 degrees to the left so it will face some object, and we might need to move it closer to the object it is facing. In this case, we would actually do all three types of transformations to get the desired results. We might even need to rotate it downward 5 degrees as well, and that is perfectly acceptable.

We can have many different transformations applied to an object. This is done by multiplying different matrices together. However, there is a catch—there is always a catch, right? The catch is that because we are doing these transformations using matrix math, we need to be aware of something very important. We are multiplying our transformation matrices together to get the results we want. Unlike multiplying normal integers, multiplying matrices is not commutative. This means that Matrix A * Matrix B != Matrix B * Matrix A. So in our earlier example where we want to scale our object and rotate it (two different times) and then move it, we need to be careful in which order we perform those operations. You will see how to do this a little later in the chapter.

Creating a Camera

That is enough theory for a bit. We are going to create a camera so we can view our world. Now we can create a new Windows game project to get started with this section. We'll this project XNADemo. To begin, we need to create the following private member fields:

```
private Matrix projection;
private Matrix view;
```

We then need to add a call to `InitializeCamera` in the beginning of our `LoadGraphicsContent` method. The `InitializeCamera` method will have no parameters and no return value. We will begin to populate the method, which can be marked as private, in the next three sections.

Projection

The `Matrix` struct has a lot of helper functions built in that we can utilize. The `Matrix.CreatePerspectiveFieldOfView` is the method we want to look at now:

```
float aspectRatio = (float)graphics.GraphicsDevice.Viewport.Width /
    (float)graphics.GraphicsDevice.Viewport.Height;
Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4, aspectRatio,
    0.0001f, 1000.0f, out projection);
```

First, we set up a local variable called `aspectRatio`. This is to store, you guessed it, the aspect ratio of our screen. For the Xbox 360 the aspect ratio of the back buffer will determine how the game is displayed on the gamer's TV. If we develop with a widescreen aspect ratio and the user has a standard TV, the game will have a letterbox look to it. Conversely, if we develop with a standard aspect ratio and the user has a widescreen TV, the Xbox 360 will stretch the display. To avoid this we should account for both situations and then adjust the value of our aspect ratio variable to the default values of the viewport of the graphics device, like in the preceding code. If we needed to find the default value to which the gamer has his or her Xbox 360 set, we can gather that information by querying the `DisplayMode` property of the graphics device during or after the `Initialization` method is called by the framework.

However, if we want to force a widescreen aspect ratio on the Xbox 360, we could set the `PreferredBackBufferWidth` and `PreferredBackBufferHeight` properties on the graphics object right after creating it. Many gamers do not care for the black bars, so we should use this with caution. Forcing a widescreen aspect ratio on Windows is a little more complicated, but the XNA Game Studio documentation has a great "How To" page explaining how to do it. Once in the documentation, look for "How to: Restrict Graphics Devices to Widescreen Aspect Ratios in Full Screen" under the Application Model in the Programming Guide.

Second, we create our field of view. The first parameter we pass in is 45 degrees. We could have used `MathHelper.ToRadians(45.0f)`, but there is no need to do the math because the `MathHelper` class already has the value as a constant. The second parameter is the aspect ratio, which we already calculated. The third and fourth parameters are our near and far clipping planes, respectively. The plane values represent how far the plane is from our camera. It means anything past the far clipping plane will not be drawn onto the screen. It also means anything closer to us than the near clipping plane will not be drawn either. Only the points that fall in between those two planes and are within a 45-degree angle of where we are looking will be drawn on the screen. The last parameter is where we populate our projection matrix. This is an overloaded method. (One version actually returns the projection, but we will utilize the overload that has reference and out parameters, which is faster because it doesn't have to copy the value of the data.)

View

Now that we have our projection matrix set, we can set up our view matrix. Although our projection can be thought of as the camera's internals (like choosing a lens for the camera), our view can be thought of as what our camera sees. The view matrix contains which way is up for the camera, which way the camera is facing, and the actual position of the camera. To set up our view matrix, we are going to use another XNA matrix helper method. The `Matrix.CreateLookAt` method takes three parameters. Let's create and initialize these private member fields now.

```
private Vector3 cameraPosition = new Vector3(0.0f, 0.0f, 3.0f);
private Vector3 cameraTarget = Vector3.Zero;
private Vector3 cameraUpVector = Vector3.Up;
```

Now we can actually call the `CreateLookAt` method inside of our `InitializeCamera` method. We should add the following code at the end of the method:

```
Matrix.CreateLookAt(ref cameraPosition, ref cameraTarget,
    ref cameraUpVector, out view);
```

The first parameter we pass in is our camera position. We are passing in the coordinates (0,0,3) for our camera position to start with, so our camera position will remain at the origin of the x and y axis, but it will move backward from the origin 3 units. The second parameter of the `CreateLookAt` method is the target of where we are aiming the camera. In this example, we are aiming the camera at the origin of the world `Vector3.Zero` (0,0,0). Finally, we pass in the camera's up vector. For this we use the `Up` property on `Vector3`, which means (0,1,0). Notice we actually created a variable for this so we can pass it in by reference. This is also an overloaded method, and because we want this to be fast we will pass the variables in by reference instead of by value. Fortunately, we do not lose much readability with this performance gain.

World

At this point if we compiled and ran the demo we would still see the lovely blank cornflower blue screen because we have not set up our world matrix or put anything in the world to actually look at. Let's fix that now.

As you saw, the templates provide a lot of methods stubbed out for us. One of these very important methods is the `Draw` method. Find this method and add the following line of code right below the `TODO: Add your drawing code here` comment:

```
Matrix world = Matrix.Identity;
```

This simply sets our world matrix to an identity matrix, which means that there is no scaling, no rotating, and no translating (movement). The identity matrix has a translation of (0,0,0), so this will effectively set our world matrix to the origin of the world.

At this point we have our camera successfully set up, but we have not actually drawn anything. We are going to correct that starting with the next section.

Vertex Buffers

3D objects are made up of triangles. Every object is one triangle or more. For example, a sphere is just made up of triangles; the more triangles, the more rounded the sphere is. Take a look at Figure 4.1 to see how this works. Now that you know that every 3D object we render is made up of triangles and that a triangle is simply three vertices in 3D space, we can use vertex buffers to store a list of 3D points. As the name implies, a vertex buffer is simply memory (a buffer) that holds a list of vertices.

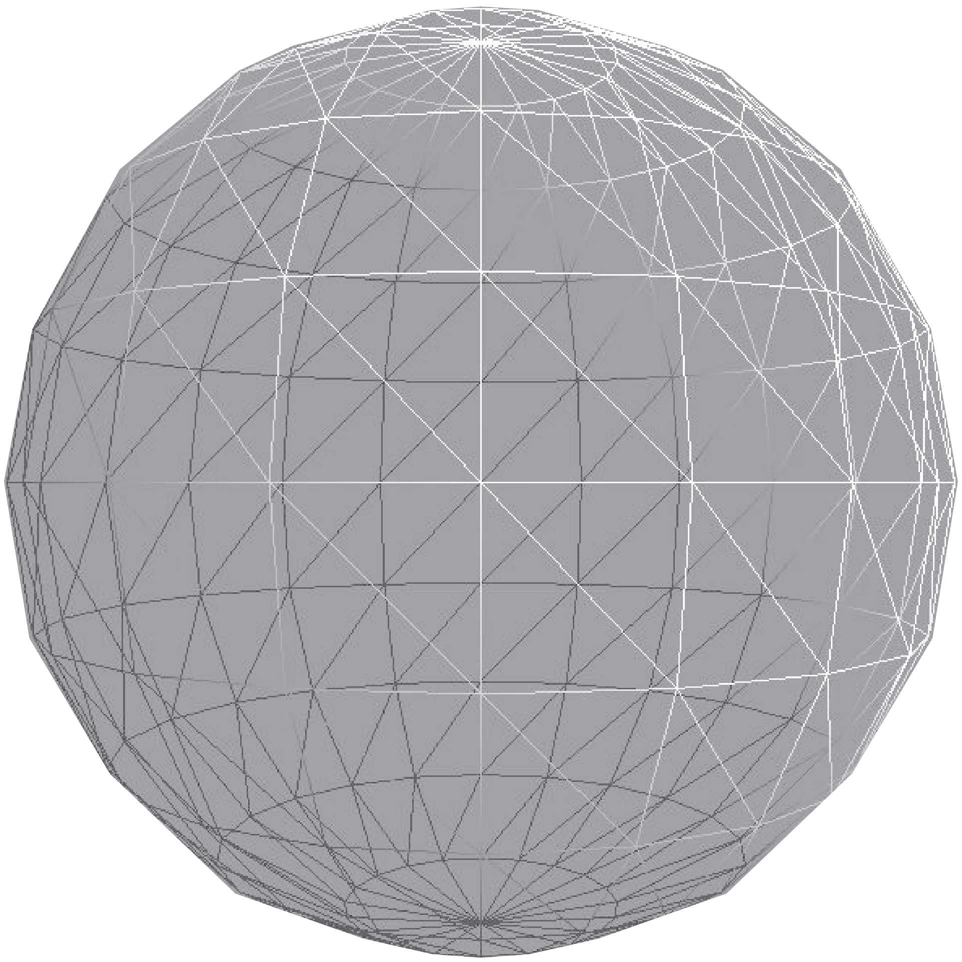

FIGURE 4.1 All 3D objects are made up of triangles.

XNA uses a right-handed coordinate system. This means that the x axis goes from left to right (left being negative, and right being positive), the y axis goes up and down (down being negative, and up being positive), and z goes forward and backward (forward being negative, and backward being positive). You can visualize this by extending your right arm out to your right and positioning your hand like you are holding a gun. Now rotate your wrist so your palm is facing the sky. At this point your pointer finger should be pointing to the right (this is our x axis going in a positive direction to the right). Your thumb should be pointing behind you (this is our z axis going in a positive direction backward). Now, uncurl your three fingers so they are pointing to the sky (this represents the y axis with a positive direction upward). Take a look at Figure 4.2 to help solidify how the right-handed coordinate system works.

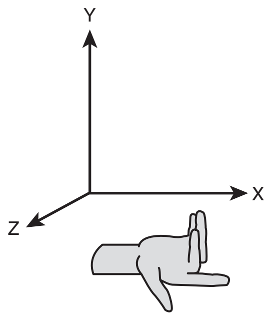

FIGURE 4.2 This demonstrates a right-handed coordinate system.

Now that you know what the positive direction is for each axis, we are ready to start plotting our points. XNA uses counterclockwise culling. *Culling* is a performance measure graphic cards take to keep from rendering objects that are not facing the camera. XNA has three options for culling: `CullClockwiseFace`, `CullCounterClockwiseFace`, and `None`. The default culling mode for XNA is `CullCounterClockwiseFace`, so to see our objects we have to set up our points in the opposite order—clockwise.

TIP

It is helpful to use some graph paper (or regular notebook paper for that matter) to plot out points. Simply put points where you want them, and make sure when you put them into the code that you do it in a clockwise order.

Let's plot some points. Ultimately, we want to make a square. We know that all 3D objects can be made with triangles, and we can see that a square is made up of two triangles. We will position the first triangle at (-1,1,0); (1,-1,0); (-1,-1,0). That means the first point (-1,1,0) will be positioned on the x axis one unit to the left, and it will be one unit up the y axis and will stay at the origin on the z axis. The code needed to set up these points is as follows:

```
private void InitializeVertices(){
    Vector3 position;
    Vector2 textureCoordinates;

    vertices = new VertexPositionNormalTexture[3];

    //top left
    position = new Vector3(-1, 1, 0);
    textureCoordinates = new Vector2(0, 0);
    vertices[0] = new VertexPositionNormalTexture(position, Vector3.Forward,
        textureCoordinates);
    //bottom right
    position = new Vector3(1, -1, 0);
    textureCoordinates = new Vector2(1, 1);
    vertices[1] = new VertexPositionNormalTexture(position, Vector3.Forward,
        textureCoordinates);

    //bottom left
    position = new Vector3(-1, -1, 0);
    textureCoordinates = new Vector2(0, 1);
    vertices[2] = new VertexPositionNormalTexture(position, Vector3.Forward,
        textureCoordinates);
}
```

As you look at this function, notice that two variables have been created: `position` and `textureCoordinates`. XNA has different structs that describe the type of data a vertex will hold. In most cases, for 3D games we will need to store the position, normal, and texture coordinates. We discuss normals later, but for now it is sufficient to understand that they let the graphics device know how to reflect light off the face (triangle). The most important part of the vertex variable is the position of the point in 3D space. You saw earlier that XNA allows us to store that information in a `Vector3` struct. We can either set the data in the constructor as we did in this code, or we can explicitly set its `X`, `Y`, and `Z` properties.

I'll skip over explaining the texture coordinates momentarily, but notice they use the `Vector2` struct XNA provides for us. We need to add the following private member field to our class we have been using to store our vertices:

```
private VertexPositionNormalTexture[] vertices;
```

We need to call this method in our application. The appropriate place to call the `InitializeVertices` method is inside of the `LoadContent` method.

If we compile and run our application now we still do not see anything on the screen. This is because we have not actually told the program to *draw* our triangle! We will want to find our `Draw` method, and before the last call to the base class `base.Draw(gameTime)` we need to add the following code:

```
graphics.GraphicsDevice.VertexDeclaration = new
    VertexDeclaration(graphics.GraphicsDevice,
    VertexPositionNormalTexture.VertexElements);

BasicEffect effect = new BasicEffect(graphics.GraphicsDevice, null);

effect.Projection = projection;
effect.View = view;

effect.EnableDefaultLighting();
```

```
world = Matrix.Identity;
effect.World = world;
effect.Begin();

foreach (EffectPass pass in effect.CurrentTechnique.Passes)
{
    pass.Begin();
    graphics.GraphicsDevice.DrawUserPrimitives(
        PrimitiveType.TriangleList, vertices, 0,
        vertices.Length / 3);

    pass.End();
}

effect.End();
```

You might think there is a lot of code here just to draw the points we have created on the screen. Well, there is, but it is all very straightforward and we can plow on through. Before we do, though, let's take a minute and talk about effects.

Effects

Effects are used to get anything in our XNA 3D game to actually show up on the screen. They handle things such as lights, textures, and even the position of the points. We will talk about effects extensively in Part VI, "High Level Shader Language (HLSL)." For now, we can utilize the `BasicEffect` class that XNA provides. This keeps us from having to actually create an effect file, so we can get started quickly.

The first thing to notice is that we create a new variable to hold our effect. We do this by passing in the graphics device as our first parameter, and we are passing in null as the effect pool because we are only using one effect and don't need a pool to share among multiple effects. After creating our effect, we want to set some of the properties so we can use it. Notice we set the world, view, and projection matrices for the effect as well as tell the effect to turn on the default lighting. We discuss lighting in detail in the HLSL part of the book, but for now, this will light up the 3D scene so we can see our objects.

TIP

When working with 3D, it is a good idea to leave the background color set to `Color.CornflowerBlue` or some other nonblack color. The reason for this is if the lights are not set up correctly, the object will render in black (no light is shining on it). So if the background color is black, you might think that the object didn't render at all.

Now back to our code. Notice that we call the `Begin` method on our effect and we also call the `End` method. Anything we draw on the screen in between these two calls will have that effect applied to them. The next section of code is our `foreach` loop. This loop iterates through all the passes of our effect. Effects will have one or more techniques. A technique will have one or more passes. For this basic effect, we have only one technique and one pass. You will learn about techniques and passes in more detail in Part VI. At this point we have another begin and end pair, but this time it is for the pass of the current (only) technique in our effect. Inside of this pass is where we finally get to draw our triangle onto the screen. This is done using the `DrawUserPrimitives` method in the graphics device object:

```
graphics.GraphicsDevice.DrawUserPrimitives(
    PrimitiveType.TriangleList, vertices, 0, vertices.Length / 3);
```

We are passing in the type of primitive we will be rendering. The primitives we are drawing are triangles, so we are going to pass in a triangle list. This is the most common primitive type used in modern games. Refer to Table 4.1 for a list of different primitive types and how they can be used. The second parameter we pass in is the actual vertex data we created in our `InitializeVertices` method. The third parameter is the offset of the point data where we want to start drawing—in our case, we want to start with the first point, so that is 0. Finally, we need to pass in the number of triangles we are drawing on the screen. We can calculate this by taking the number of points we have stored and dividing it by 3 (because there are three points in a triangle). For our example, this will return one triangle. If we compile and run the code at this point we should see a triangle drawn on our screen. It is not very pretty, because it is a dull shade of gray, but it is a triangle nonetheless (see Figure 4.3).

TABLE 4.1 PrimitiveType Enumeration from the XNA Documentation

Member Name	Description
`LineList`	Renders the vertices as a list of isolated straight-line segments.
`LineStrip`	Renders the vertices as a single polyline.
`PointList`	Renders the vertices as a collection of isolated points. This value is unsupported for indexed primitives.
`TriangleFan`	Renders the vertices as a triangle fan.
`TriangleList`	Renders the specified vertices as a sequence of isolated triangles. Each group of three vertices defines a separate triangle. Back-face culling is affected by the current winding-order render state.
`TriangleStrip`	Renders the vertices as a triangle strip. The back-face culling flag is flipped automatically on even-numbered triangles.

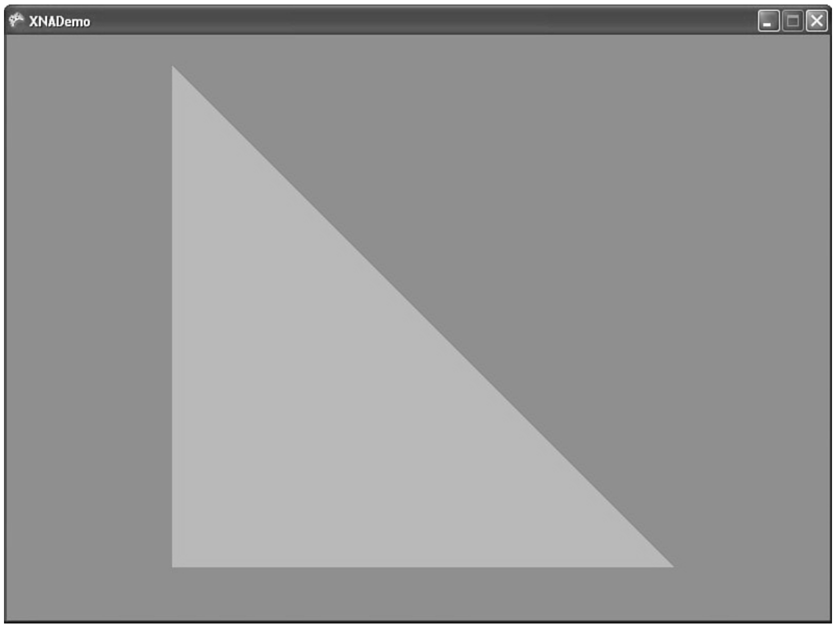

FIGURE 4.3 Drawing a triangle as our first demo.

Textures

We have a triangle finally drawn on the screen, but it does not look particularly good. We can fix that by adding a texture. Copy the texture from the Chapter4\XNADemo\XNADemo folder on the CD (texture.jpg) and paste that into the Content project. This invokes the XNA Content Pipeline, which we discuss in Part III, “Content Pipeline”. For now, you just need to know that the content pipeline makes the texture available as loadable content complete with a name by which we can access it. The asset will get the name “texture” (because that is the name of the file). We need to declare a private member field to store our texture:

```
private Texture2D texture;
```

We define our texture as a `Texture2D` object. This is another class that XNA provides for us. `Texture2D` inherits from the `Texture` class, which allows us to manipulate a texture resource. Now we need to actually load our texture into that variable. We do this in the `LoadContent` method by adding this line of code:

```
texture = Content.Load<Texture2D>("texture");
```

Now we have our texture added to our project and loaded into a variable (with very little code), but we have yet to associate that texture to the effect that we used to draw the triangle. We will do that now by adding the following two lines of code right before our call to `effect.Begin` inside of our `Draw` method:

```
effect.TextureEnabled = true;
effect.Texture = texture;
```

This simply tells the effect we are using that we want to use textures, and then we actually assign the texture to our effect. It is really that simple. Go ahead and compile and run the code to see our nicely textured triangle!

Index Buffers

We have covered a lot of ground so far, but we aren't done yet. We want to create a rectangle on the screen, and to do this we need another triangle. So that means we need three more vertices, or do we? Actually, we only need one more vertex to create our square because the second triangle we need to complete the square shares two of our existing points already. Feel free to review the sections earlier in this chapter where we talked about vertex buffers. We set up three points to make the triangle. To use the code as is, we would need to create another three points, but two of those points are redundant and the amount of data it takes to represent the `VertexPositionNormalTexture` struct is not minimal, so we do not want to duplicate all that data if we do not need to. Fortunately, we do not. XNA provides us with index buffers.

Index buffers simply store indices that correspond to our vertex buffer. So to resolve our current dilemma of not wanting to duplicate our heavy vertex data, we will instead duplicate our index data, which is much smaller. Our vertex buffer will only store four points (instead of six), and our index buffer will store six indices that correspond to our vertices in the order we want them to be drawn. We need to increase our vertex array to hold four values instead of three. Make the following change in the `InitializeVertices` method:

```
vertices = new VertexPositionNormalTexture[4];
```

An index buffer simply describes the order in which we want the vertices in our vertex buffer to be drawn in our scene.

Find the `InitializeVertices` method in our code and add the last point we need for our rectangle. Try to do this before looking at the following code.

```
//top right
position = new Vector3(1, 1, 0);
textureCoordinates = new Vector2(1, 0);
vertices[3] = new VertexPositionNormalTexture(position, Vector3.Forward,
    textureCoordinates);
```

As you were writing the code, I imagine you were wondering about the texture coordinates for the points. We finally talked about textures, but not really how we mapped the texture to the vertices we created. We will take a moment and do that now before we continue our discussion of index buffers.

Texture coordinates start at the top left at (0,0) and end at the bottom right at (1,1). The bottom-left texture coordinate is (0,1), and the top right is (1,0). Take a look at Figure 4.4 to see an example.

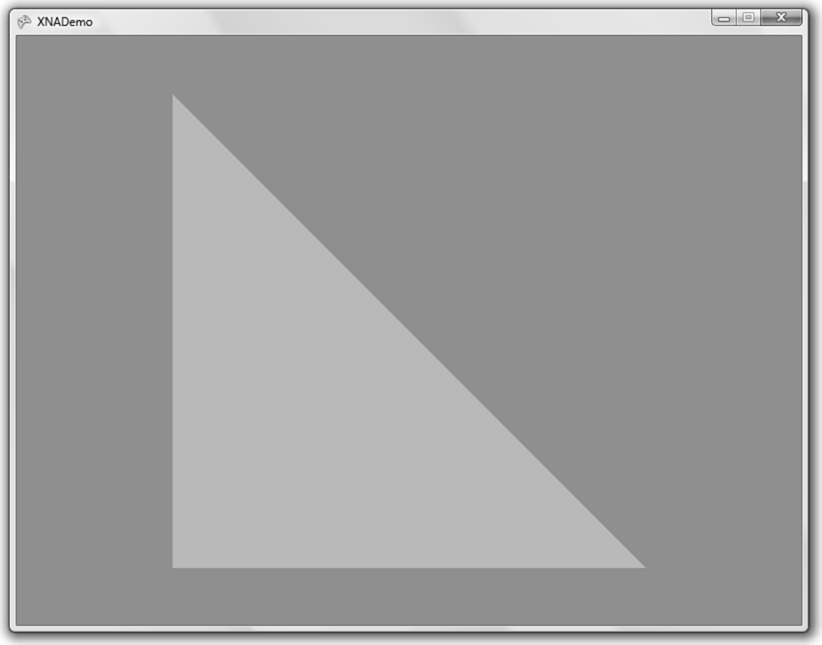

FIGURE 4.4 Texture coordinates start at the top left at (0,0) and end at the bottom right at (1,1).

If we wanted to map a vertex to the bottom-center pixel of a texture, what should the values be? The horizontal axis is our x axis, and the vertical axis is our y axis. We know we need a 1 in our y coordinate to get to the very bottom of the texture. To get to the middle of that bottom row, we would need to take the value in between 0 and 1, which is 0.5. So if we wanted to map a vertex to the bottom-center pixel of a texture, we would map it at (0.5, 1). Back to our demo: Because the vertex we just added was the top-right point of the rectangle, the texture coordinate we assigned to it was (1,0).

Now that you have a better understanding of why our texture mapped to our triangle correctly, we can get back to our index buffer. We have added a vertex to our code and now we need to create an index buffer to reference these four points. We need to create another private member field called `indices`:

```
private short[] indices;
```

Notice that we declared this as an array of `short`. We could have used `int`, but `short` takes up less room and we aren't going to have more than 65,535 indices in this demo. The next thing we need to do is actually create our method that will initialize our indices. We will name this `InitializeIndices`, and we will call this method from inside our `LoadContent` method right after we make the call to `InitializeVertices`. Make sure that the vertex was added right before we initialized our vertex buffer and after we created all the other vertices. This way, the code for `InitializeIndices` shown next will work for us. It assumes the latest addition to our list of vertices is at the bottom of the list.

```
private void InitializeIndices()
{
    //6 vertices make up 2 triangles which make up our rectangle
    indices = new short[6];

    //triangle 1 (bottom portion)
    indices[0] = 0; // top left
    indices[1] = 1; // bottom right
    indices[2] = 2; // bottom left

    //triangle 2 (top portion)
    indices[3] = 0; // top left
    indices[4] = 3; // top right
    indices[5] = 1; // bottom right
}
```

In this method, we know we are going to create two triangles (with three points each), so we create enough space to hold all six indices. We then populate our indices. We took care to add our vertices in a clockwise order when adding them to the vertex list, so we can simply set our first three indices to 0, 1, and 2. The second triangle, however, needs a little more thought. We know we have to add these in clockwise order, so we can start with any vertex and work our way around. Let's start with the top-left vertex (the first vertex we added to our list—index of 0). That means we need to set our next index to be the top-right vertex, which is the one we just added to the end of the list. We set that index to 3. Finally, we set the last point to the bottom-right vertex, which was added to the vertex buffer second and has the index of 1.

Now we have our vertices created, complete with textured coordinates and position and even normals. We have our indices set up to use the vertex buffer in a way that doesn't duplicate any of the complex vertex data. It may appear the data is duplicated, but only the indices are duplicated, not the actual vertex data. We further saved memory by using `short` instead of `int` because we will only have a few indices we need to store to represent our 3D object (our rectangle). Also, some older graphic cards do not support 32-bit (`int`) index buffers. The only thing left for us to do is to actually modify our code that draws the primitive to tell it we are now using an index buffer. To do that, find the `Draw` method and locate the call to `DrawUserPrimitives`. We will want to replace that line with the following line:

```
graphics.GraphicsDevice.DrawUserIndexedPrimitives(
    PrimitiveType.TriangleList, vertices, 0, vertices.Length,
    indices, 0, indices.Length / 3);
```

Notice that we changed the method we are calling on the graphics device. We are now passing in both vertex and index data. Let's break down the parameters we are passing in. We still pass in a triangle array as the first parameter and our array of vertices as the second parameter, and we are leaving our vertex offset at 0. The next parameter is new and simply needs the number of vertices that is in our vertex array. The fifth parameter is our array of indices (this method has an override that accepts an array of `int` as well). The sixth parameter is the offset we want for our index buffer. We want to use all the indices, so we passed in 0. The final parameter, `primitive count`, is the same as the final parameter in the method we just replaced. Because we only have four vertices, we needed to change that to our index array. Our indices array has six references to vertices in it, and we take that value and divide it by 3 to get the number of triangles in our triangle list. When we compile and run the code, we should see a rectangle that was created with our modified vertex buffer and our new index buffer!

As an exercise, modify the points in our vertex to have the rectangle slanted into the screen. This will require modifying a couple of our z values from 0 to something else. Give it a try!

XNA Game Components

Now that we have created this very exciting rectangle, let's take a look at what it did to our performance. In this section we are going to create an XNA `GameComponent`. A game component allows us to separate pieces of logic into their own file that will be called automatically by the XNA Framework. We will take the frame rate code we added in our PerformanceBenchmark project from the last chapter and create a game component out of it. To do this, we need to add another file to our project, which we can call FPS.cs. We need to pick GameComponent as the file type from inside the Add New File dialog box of XNA Game Studio.

With a blank fps.cs in front of us, we should see the class is inheriting from `Microsoft.Xna.Framework.GameComponent`. This is useful for components where we are only updating the internal data, typically through the `Update` method. For our frame rate calculation, however, we need to have our game component expose the `Draw` method because we want to know how many times a second we can draw our world on the screen. So we first need to change from which class we are inheriting. Instead of inheriting from `GameComponent`, we need to inherit from `DrawableGameComponent` so we can access the `Draw` method. We need to override the `Draw` method and use the same code we used in the PerformanceBenchmark project. To see the definition of the `DrawableGameComponent` to determine what is available for us to override, press F12 while the cursor is inside the `DrawableGameComponent` text.

Listing 4.1 contains the same code we used in the PerformanceBenchmark project. The difference is that it is inside of a drawable game component now. The biggest difference is our constructor, so let's take a minute to dissect that now. As you learned in the last chapter, to measure a true frame rate we need to get the screen to draw as many times as it can and not wait on the monitor to do a vertical refresh before updating the screen. We could put this code inside of our main game class, but for the projects in this book we typically let the game run at a fixed pace and only change it when we are trying to measure our true frame rate. Because of this assumption, the code is set up to take these values in via the constructor of the FPS game component. Typically a game component only requires passing in a game instance to the constructor, but we can require other parameters if we need to. For this game component, we are passing the values we initially set at the game level. We have a default constructor that will have the game render as fast as possible. These settings are per game, not per component.

LISTING 4.1 A Drawable Game Component That Calculates Our FPS

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace XNADemo
{
    public sealed partial class FPS
        : Microsoft.Xna.Framework.DrawableGameComponent
    {
        private float fps;
        private float updateInterval = 1.0f;
        private float timeSinceLastUpdate = 0.0f;
        private float framecount = 0;

        public FPS(Game game)
           : this(game, false, false, game.TargetElapsedTime) { }

        public FPS(Game game, bool synchWithVerticalRetrace,
                   bool isFixedTimeStep, TimeSpan targetElapsedTime)
            : base(game)
        {
            GraphicsDeviceManager graphics =
                (GraphicsDeviceManager)Game.Services.GetService(
                typeof(IGraphicsDeviceManager));
            graphics.SynchronizeWithVerticalRetrace = synchWithVerticalRetrace;
            Game.IsFixedTimeStep = isFixedTimeStep;
            Game.TargetElapsedTime = targetElapsedTime;
        }

        public sealed override void Initialize()
        {
            // TODO: Add your initialization code here
            base.Initialize();
        }

        public sealed override void Update(GameTime gameTime)
        {
            // TODO: Add your update code here

            base.Update(gameTime);
        }

        public sealed override void Draw(GameTime gameTime)
        {
            float elapsed = (float)gameTime.ElapsedRealTime.TotalSeconds;
            framecount++;
            timeSinceLastUpdate += elapsed;
            if (timeSinceLastUpdate > updateInterval)
            {
                fps = framecount / timeSinceLastUpdate;

#if XBOX360
                System.Diagnostics.Debug.WriteLine("FPS: " + fps.ToString());
#else
                Game.Window.Title = "FPS: " + fps.ToString();
#endif
                framecount = 0;
                timeSinceLastUpdate -= updateInterval;
            }
            base.Draw(gameTime);
        }
    }
}
```

Now that we have the game component created and added to our project, we need to actually use it inside of the demo. To do this we need to create a private member field in our game class, as follows:

```
private FPS fps;
```

Then we can add the following code inside of our game constructor after we initialize our graphics variable:

```
#if DEBUG
    fps = new FPS(this);
#else
    fps = new FPS(this, true, true, this.TargetElapsedTime);
#endif
    Components.Add(fps);
```

We wrapped this with the DEBUG compiler directive, but we might want to run in debug mode without rendering the code as fast as possible. Either we can change how we are initializing our fps variable by passing in explicit values to the constructor or we can create another configuration (that is, PROFILE). If we compiled in release or debug mode, the game is going to run at the normal pace but will still display the frame rate. This would allow us to see if the frame rate is dropping and we are falling behind, but in order to see how much room we have, we would want to run it under the new configuration (PROFILE). After initializing the fps object we then add the component to our game's component collection. The XNA Framework will then call the component's Update and Draw methods (and other virtual methods) at the same time it calls the game's methods.

It can be very beneficial to separate logic and items we need to draw to the screen. It provides a nice clean way to separate our code but it does have some overhead. It is definitely not wise to handle all the objects we want to draw as game components. Instead, if we want to separate our enemies from our player, it might be beneficial to have our player in its own game component and then have an "enemy manager" as its own game component. The enemy manager could then handle itself which enemies it needs to draw, move, and so on. This way, as enemies come and go, the manager is handling all that logic and not the core game class adding and removing a bunch of enemy components. Game components can really help, but we cannot go overboard with them or our performance will suffer.

Checking Performance

Now that we have our fps functionality inside of a game component, we can check out whether or not the code we wrote for the demo to display the rectangle is performing well. Fortunately, we recorded the frame rate we were getting in the last chapter, so we have a baseline from which to work.

We will need to set up an Xbox 360 game project for this solution as we discussed in Chapter 2, "XNA and the Xbox 360." Once we have it set up, we can run our application on our machine and on the Xbox 360 to measure performance.

Machine A ran the benchmark code at about 280 fps. The Xbox 360 ran the same benchmark code at about 5,220 fps. With our new code, Machine A is running at 206 fps. The Xbox 360 is running at only 114 fps—ouch! What did we do wrong? Well, because we tested for performance right away, we know that it has to be an issue with our Draw method, so we should take a look at it again to see what is going on. We can also run the XNA Framework Remote Performance Monitor for the Xbox 360 and see what the garbage collector is doing. If a refresher is needed, you can find information on running this application in Chapter 2.

By launching our demo through the performance monitor tool, we can see that the "Objects Moved by Compactor" value is between 75,000 and 85,000 *every second*. We can see the "Objects not moved by Compactor" value is constantly growing with about 5,000 or more per second. This is obviously not good, and it is why we are thrashing our Xbox 360. Looking in the code we can see that we are creating a new instance of the `BasicEffect` class on every frame. That has got to be hurting us, so we can make that a member field of the game class because we are never changing it. We can break it out and actually initialize the effect inside our `LoadContent` method as follows:

```
effect = new BasicEffect(graphics.GraphicsDevice, null);
```

Now we can run our application again and look at the frame rate it is spitting out in our debug window. It is much better now—about 2,850 fps. However, that is still a far cry from our 5,220 fps. Checking the frame rate on Machine A reveals that we are running at 207 fps. So although the change really made a difference on the Xbox 360, it did not do much for us on the Windows side of things. Of course, this is expected because the issue we were having was with the garbage collector. Remember from the last chapter that each time we were creating a new `BasicEffect` object in Windows, the code was creating the effect object and destroying it all in the same frame, so when the garbage collector ran, it simply removed the dead objects. On the Xbox 360, however, the garbage collector has to go through the entire heap to determine what is dead as it starts to get full. So we have helped our situation on the console, but we are still only running at 53% of what we were at the baseline. Let's dig around some more. Notice the following lines of code at the top of our `Draw` method:

```
graphics.GraphicsDevice.VertexDeclaration = new
    VertexDeclaration(graphics.GraphicsDevice,
    VertexPositionNormalTexture.VertexElements);
```

This cannot be doing the garbage collector any favors. Although we need to set our vertex declaration on every frame, we do not need to create it every frame. We can create a private member field to hold our vertex declaration as follows:

```
private VertexDeclaration vertexDeclaration;
```

Now we can actually initialize that variable inside of the `LoadContent` method right after our `BasicEffect` initialization. The code for this is as follows:

```
vertexDeclaration = new VertexDeclaration(graphics.GraphicsDevice,
    VertexPositionNormalTexture.VertexElements);
```

Finally, we can change the original statement inside of `Draw` to set graphics device vertex declaration to the variable we just initialized:

```
graphics.GraphicsDevice.VertexDeclaration = vertexDeclaration;
```

Now we are only creating the vertex declaration once and setting it once instead of every frame. This is encouraging, because we are now at about 3,230 fps on the Xbox 360 and the performance is still the same on Machine A at about 207 fps. So just with a little bit of effort we optimized our code from running at a mere 114 fps on the Xbox 360 to a more reasonable 3,230 fps.

What else can we do? Surely just displaying two triangles on the screen should not decrease our frame rate by 39%. As we go back to the performance monitoring tool, we can see that our "Objects Moved by Compactor" and "Objects Not Moved by Compactor" values are at a much better number—zero!

If we comment out the `DrawUserIndexedPrimitives` method, we see that our frame rate jumps back up to 5,220 fps. So it means the "problem" exists in this method. Is there anything that can be done with the code we have? Because our baseline code did not do anything and this code is actually drawing something (even if it is only two triangles), it might be that this is as good as it gets—after all, more than 3,000 fps is not that shabby! However, there has been some debate in regard to whether the `DrawUser*` methods are as fast as their `Draw*` counterparts. We are going to find out if that is the case.

DrawUserIndexedPrimitives versus DrawIndexedPrimitives

Make a copy of the solution folder we are working on and rename the new folder XNADemo–DIP. We can leave the solution file and everything else the same name. After opening this new project, we need to give it another title in the AssemblyInfo.cs file. We also need to change the GUID value so that when we deploy this, it will show up as a new entry in our XNA Game Launcher list. We can just replace any one digit with another digit for this example. This way, we can easily compare this demo with the one we just finished inside of the remote performance monitor for the Xbox 360. Once this has been done, we need to modify our game code by adding the following code to the end of the `InitializeIndices` method:

```
IndexBuffer ib = new IndexBuffer(graphics.GraphicsDevice,
    sizeof(short) * indices.Length, BufferUsage.WriteOnly,
    IndexElementSize.SixteenBits);
ib.SetData(indices);

graphics.GraphicsDevice.Indices = ib;
```

We are initializing the index buffer on our graphics device by telling it the size of our indices array. We use `BufferUsage.WriteOnly` because we will not be reading from the list (reading would fail with this setting) and it allows the graphics driver to determine the best location in memory to efficiently perform the rendering and write operations. Finally, we tell it that we have an array of `short` by setting the `IndexElementSize` to 16 bits, the size of the short type (`System.Int16`). The second statement actually sets the array of indices inside of the graphic devices index buffer.

We needed to define our index buffer because the method we are replacing in the `Draw` method needs to have the data explicitly set on the graphics device. The following is the code we are going to replace our `DrawUserIndexedPrimitives` inside the `Draw` method with:

```
graphics.GraphicsDevice.DrawIndexedPrimitives(
    PrimitiveType.TriangleList, 0, 0, vertices.Length, 0, indices.Length / 3);
```

Finally, we need to set the source of our graphic devices vertex buffer. We do that with the following code, which should be placed at the end of the `InitializeVertices` method:

```
vertexBuffer = new VertexBuffer(graphics.GraphicsDevice,
    VertexPositionNormalTexture.SizeInBytes * vertices.Length,
    ResourceUsage.WriteOnly, ResourceManagementMode.Automatic);
vertexBuffer.SetData(vertices);

graphics.GraphicsDevice.Vertices[0].SetSource(vertexBuffer, 0,
    VertexPositionNormalTexture.SizeInBytes);
```

The first statement is used to populate our vertex buffer with the actual vertices we created. We pass in the graphics device followed by the size of the buffer. The size of the buffer is determined by taking the size of the struct we are using to represent our vertex (in this case it is `VertexPositionNormalTexture`) and multiplying that by the number of points we have. In this case it is three, but instead of hard-coding three, we grab the length property of our vertex array. The third parameter describes how we plan to use this vertex buffer. We can find all the different options for this enumeration by looking in the documentation that was installed with XNA Game Studio. We are setting the `BufferUsage` parameter just like we did before. The final parameter of this method tells XNA to handle our memory management automatically. For more information on this enumeration, take look at the documentation. The second statement takes our vertex data we set in our `InitializeVertices` method and sets our vertex buffer with it.

We can set up our `vertexBuffer` private member field next:

```
private VertexBuffer vertexBuffer;
```

After these changes, we can compile and run the code on the Xbox 360 and see that our frame rate has done nothing. The conclusion to draw here is that it does not make a difference if we use `DrawUserPrimitives` or `DrawPrimitives` methods. This was a good exercise, but for our example here it did not make a difference. Whichever version is more convenient for us as we develop our game is the one we should use. However, the results could change if we are drawing more vertices. Another performance test could be to add more vertices and indices to see if one scales better than the other. For example, instead of only rendering four vertices, you could render 10,000 and see how the two methods compare. Have fun finding the most efficient way to use these methods in your particular situation.

Transformations Revolutions

Just in case it is not clear where these transformation section titles come from, when dealing with matrices it is very hard not to think about *The Matrix* movies. What a great trilogy.

Anyway, the reason we are back yet again to discuss transformations is because you need to gain practical knowledge about transformations and not just the theory you learned about earlier. Therefore, we will look at some of the transformation functions that XNA has included in the framework.

In the earlier scenario we had a 3D object that we wanted to scale, rotate (twice), and then translate. We said that we had to do it in a particular order but did not take the discussion any further. Now that we know how to create a 3D object, we can run through the exercise of transforming the object the way we want.

We can make a copy of the first demo we created in this chapter (XNADemo) and call it Transformations. We need to rename the assembly title and the GUID again so we will not overwrite the demo with the same GUID on the Xbox 360. We can actually rename each project as well as the solution and change the namespaces if that is preferred.

In the scenario from earlier, we wanted to move to a position and scale our object down and then rotate to the left and down some. This needs to be done in the correct order because matrix multiplication is not commutative. We are going to modify our code and move our existing rectangle into the distance. Then we will create another rectangle and transform it to get the desired effect.

To start, we need to refactor a section of code from our `Draw` method to create a new method called `DrawRectangle`. Cut all the code after we create the world matrix variable and before we call the `Draw` method on our base class. Now paste the code into the newly created `DrawRectangle` method, as follows:

```
private void DrawRectangle(ref Matrix world)
{
    effect.World = world;
    effect.Begin();

    //As we are doing a basic effect, there is no need to loop
    //basic effect will only have one pass on the technique
    effect.CurrentTechnique.Passes[0].Begin();

    graphics.GraphicsDevice.DrawUserIndexedPrimitives(
        PrimitiveType.TriangleList, vertices, 0, vertices.Length,
        indices, 0, indices.Length / 3);
    effect.CurrentTechnique.Passes[0].End();

    effect.End();
}
```

The method takes a matrix as a parameter. This is going to be the matrix we transform before sending it to the effect. To demonstrate another way of setting the passes on our `BasicEffect`, we can just grab the first pass on the current technique instead of doing a `foreach` loop because we know there is only one pass we can get by with this optimization.

Now we need to call this method inside of our `Draw` method where we just removed the code. So right above the call to `Draw` on our base class we can add the following code:

```
DrawRectangle(ref world);
```

If we run this, we should get the exact same results as before. We are still rendering one rectangle in the exact same position—the origin of the world. We can remove the member variable `world` and where we set it at the top of our `Draw` method because we are not using it in this example. Following is our new `Draw` method with the changes mentioned so far:

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    effect.Projection = projection;
    effect.View = view;

    effect.EnableDefaultLighting();

    effect.TextureEnabled = true;
    effect.Texture = texture;

    Matrix world = Matrix.Identity;
    DrawRectangle(ref world);

    base.Draw(gameTime);
}
```

Let's move our existing triangle into the distance. We can move it backward and to the right some. To do this we will need to use the built-in XNA matrix helper method `Matrix.CreateTranslation`. We know we are at the origin (0,0,0), and we want to move back and to the right. Remember that XNA uses a right-handed coordinate system, so this means that to move the rectangle backward we need to subtract from the z position. To move it right we need to add to the x position. `CreateTranslation` takes in `Vector3` as a parameter, so we can set our values into the vector before passing it into the helper function. Change where we set the world matrix to the following:

```
Matrix world = Matrix.CreateTranslation(new Vector3(3.0f, 0, -10.0f));
```

We moved the rectangle to the right by three units and to the back by 10 units. Now we need to add another rectangle. Let's add the code to do this immediately following the first rectangle. To do this we need to simply pass in `Matrix.Identity` as the following code shows:

```
world = Matrix.Identity;
DrawRectangle(ref world);
```

When we run the code we cannot see the rectangle we originally drew because it is further back, and this new rectangle is obstructing our view. Let's scale it down to about 75% of what it is currently. To do this we need to call the XNA matrix helper method `Matrix.CreateScale` as follows:

```
world = Matrix.CreateScale(0.75f);
```

By replacing the identity matrix with this `CreateScale` matrix, we can see our rectangle is now smaller, so we can partially see the one we moved toward the back. A screenshot of this can be seen in Figure 4.5.

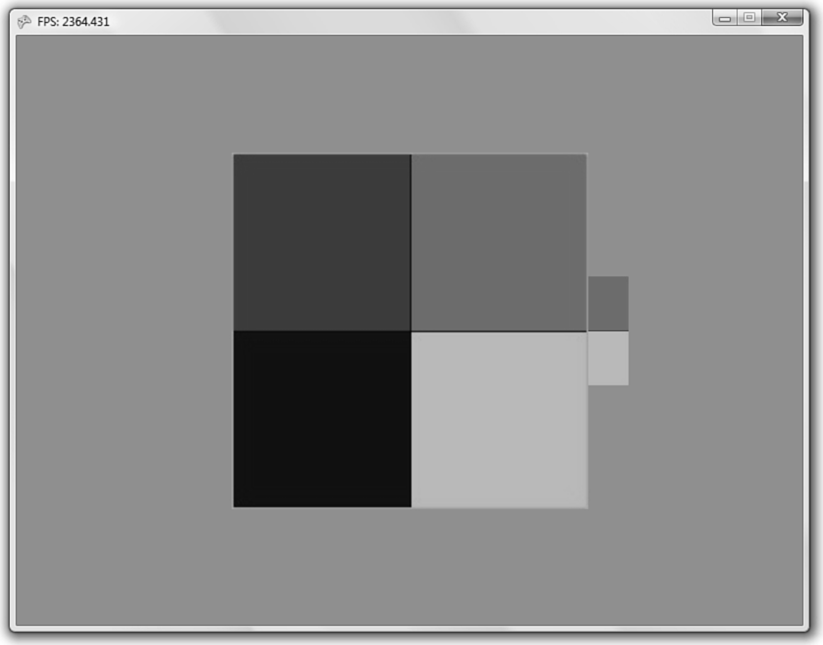

FIGURE 4.5 A smaller rectangle is obstructing the view of a larger rectangle that is further away.

i0672330229_ch04.qxp 12/24/08 11:28 AM Page 25

Let's move this rectangle back about five units, to the left three, and down one unit. We want to keep the scale that we have in place. We need to multiply our matrices together, as discussed earlier, so we change our world matrix again to look like the following code:

```
world = Matrix.CreateScale(0.75f) *
    Matrix.CreateTranslation(new Vector3(-3.0f, -1.0f, -5.0f));
```

We also want to rotate our object to one side and downward. We can do that using a couple other helper methods in the `Matrix` struct. There are three different helper functions, and each will rotate one of the three axes. These methods are called `Matrix.CreateRotationX`, `Matrix.CreateRotationY`, and `Matrix.CreateRotationZ`. To add rotation to make the object turn to the side, we need to rotate around the y axis. To visualize this, take a look at Figure 4.6.

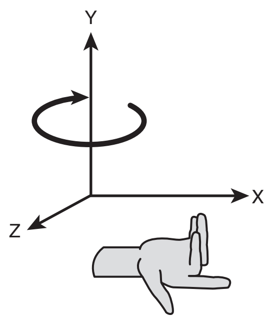

FIGURE 4.6 Rotating around the y axis.

Let's rotate around the y axis by 30 degrees. The rotation helper methods require a value in radians, but the `MathHelper` class allows us to easily convert degrees to radians. Thus, we can rotate 30 degrees around the y axis by using the following code:

```
world = Matrix.CreateScale(0.75f) *
    Matrix.CreateTranslation(new Vector3(-3.0f, -1.0f, -5.0f)) *
    Matrix.CreateRotationY(MathHelper.ToRadians(30.0f));
```

If we run this code, we do not see the rectangle. The reason is that we rotated the matrix after we translated it. By doing this, we effectively told it to rotate around the origin (where it was) instead of at its center. By rotating after translating we are making it orbit around where it was. Instead, we want it to rotate around its own center, and to do this we need to replace the world matrix with the following code:

```
world = Matrix.CreateScale(0.75f) *
    Matrix.CreateRotationY(MathHelper.ToRadians(30.0f)) *
    Matrix.CreateTranslation(new Vector3(-3.0f, -1.0f, -5.0f));
```

Now when we run this, we have rotated around the center of our object to get the desired results. Finally, we want to throw in another rotation for good measure. This time we want to rotate downward, so we need to rotate on the x axis. Assuming we want to rotate it about 15 degrees, we can use the following code:

```
world = Matrix.CreateScale(0.75f) *
    Matrix.CreateRotationX(MathHelper.ToRadians(15.0f)) *
    Matrix.CreateRotationY(MathHelper.ToRadians(30.0f)) *
    Matrix.CreateTranslation(new Vector3(-3.0f, -1.0f, -5.0f));
```

The code scales our rectangle, rotates it around the x axis, rotates it around the y axis, and finally the code moves the rectangle. By multiplying the matrices in the right order, we were able to accomplish the desired effect (see Figure 4.7) .

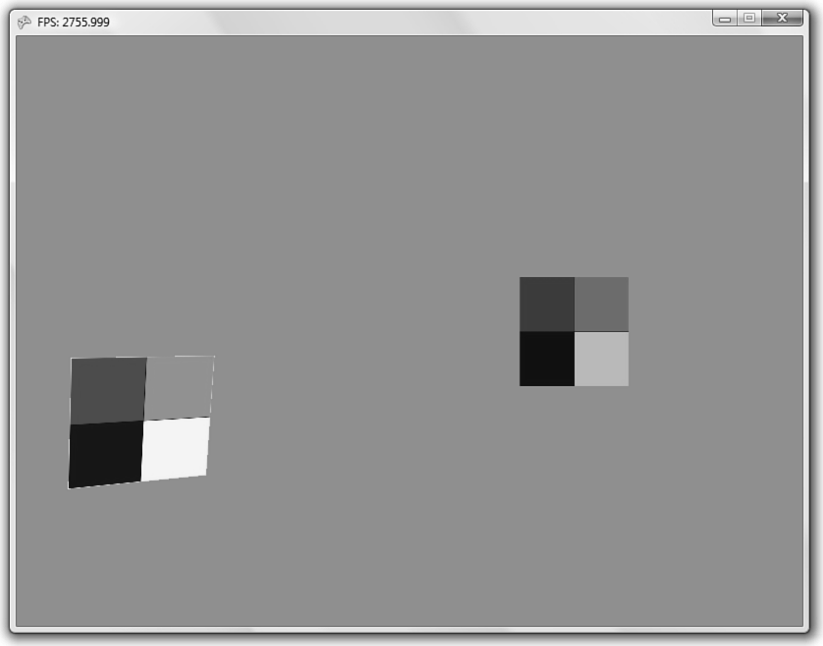

FIGURE 4.7 Applying matrix transformations in the right order will produce the desired effect.

Summary

We covered a lot of ground in this chapter. We discussed the foundation of everything we will do in the 3D worlds we create. We set up a camera to view our 3D world. We discussed basic 3D terminology and how it correlates to XNA. We examined how to use different methods to create 3D objects on the screen with points we manually plotted inside of our code.

We spent quite a bit of time going through the performance-checking process to determine what things we could do to improve our code. We put into practice what you learned in the last chapter.

We ended the chapter by actually performing matrix transformations, a concept you learned about at the beginning of the chapter. We applied multiple transformations to one of our objects and saw how important it was to get the multiplication order right when dealing with transformations.

This chapter is very important, because the rest of the book will build on this foundation. Make sure to take time to let it sink in. Reread it, dwell on it, dream about it—OK, maybe that's a little bit too extreme.