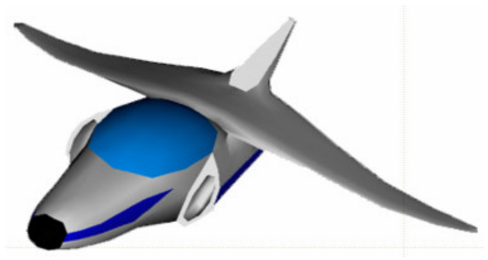


XNA Tutorials

Utah State University
Association for Computing Machinery
XNA Special Interest Group
RB Whitaker
14 October 2007

World Coordinates

Tutorial 6

Overview

In the last couple of tutorials, been using an effect called "Pretransformed." It was called this because it did a bunch of stuff, labeled collectively as *transformations*, for us. This was great because it did a lot of work for us. We didn't have to do any transformations ourselves.

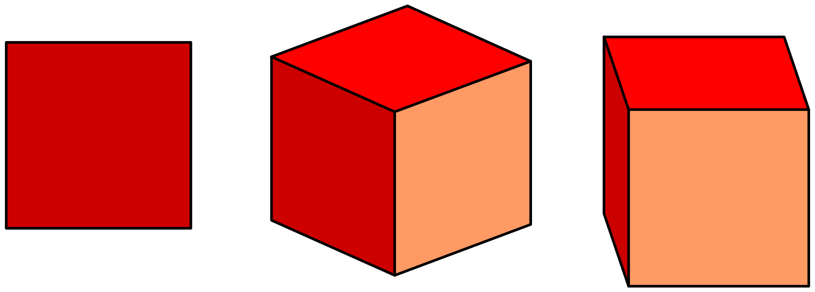
But for what we want to do next, we need to be able to do transformations. This tutorial will cover what transformations are, why we want to use them, how to use an effect that lets us perform transformations, and finally, how to actually perform transformations.

World Coordinates

In our program, we have objects in a three dimensional world. These coordinates are called **World Coordinates**. On our screen, we have two dimensional coordinate. These are called **Screen Coordinates**. You can imagine that at some point, we will have to find a way to convert three dimensional world coordinates to two dimensional screen coordinates.

The way this is done in graphics is by using matrices. If you remember your linear algebra, then this probably makes sense to you. If you don't remember linear algebra, then don't worry. The computer really takes care of almost all of this for us.

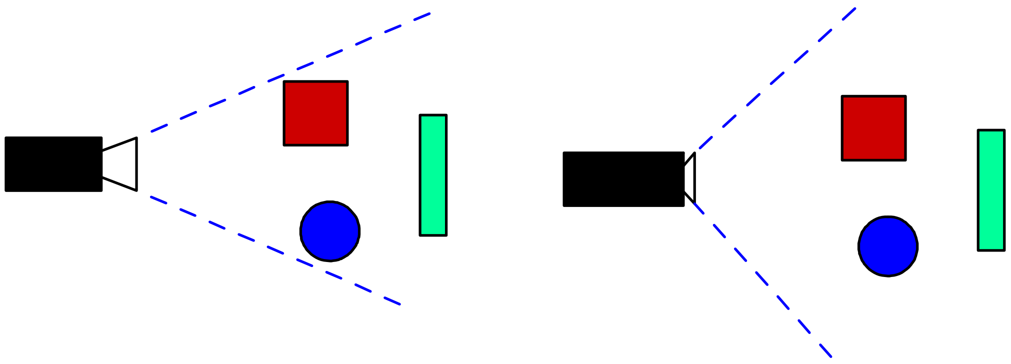
There can be lots of different ways to do this conversion. You can imagine that if we want to draw a box from two different points of view that the resulting screen coordinates of our box will be very different from each other, as shown in the image below.



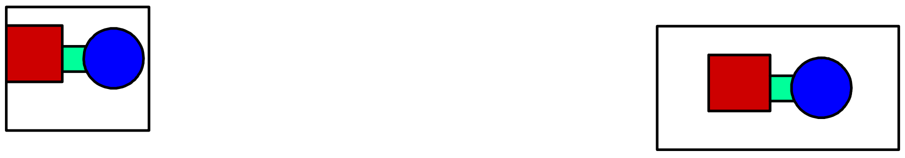
Although this may sound a little confusing, it actually is not too bad to perform. All we really need to do is tell the program where we want to be located at, and what kind of camera we want to use. The graphics card can take care of the rest of the math needed to convert our world coordinates to screen coordinates, as long as it has that information.

Let's get back into our code and add a function to our class that sets up the camera. I called mine `protected void SetupCamera()`.

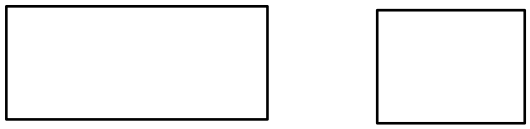
The first task we need to do is to determine what kind of camera we want to use. There are four attributes of our camera. First, is the field of view angle. The diagram below shows two cameras with different field of view angles. The one on the left has a field of view angle of about 45°, while the one on the right has a field of view angle of about 90°.



Below shows what the screen might look like if you are looking at the scene from these two different cameras. You can see that the camera on the right produces a wider image. We can see well beyond the sides of the objects in the scene.



The second feature of our camera is called the aspect ratio. The aspect ratio is a number that says how long the window is compared to its height. Even in real life, we run into aspect ratios. With movies, there are two kinds of formats, determined by their aspect ratio. The two common aspect ratios are widescreen, which is 2.35 times longer than it is tall, and full screen, which is 1.33 times longer than it is tall. These two sizes are shown below.



When we are doing graphics, our window could be just about any size, so we will have to figure that in when we calculate our aspect ratio.

The last two features of our camera are closely related, so we'll discuss them together. In order for our scene to be drawn right, we need to specify a near and far clipping plane. Everything between our near and far clipping plane will be drawn, but if an object is closer to us than the near clipping plane, or further from us than the far clipping plane, it will be ignored.

Now that we understand how our camera works, let's put in the code to setup our camera. Let's add the following lines of code to our `SetupCamera()` method:

```
float fovAngle = MathHelper.PiOver4;  // 45 degrees in radians
float aspectRatio = this.Window.ClientBounds.Width /
                                this.Window.ClientBounds.Height;
float nearClippingPlane = 0.1f;
float farClippingPlane = 100f;

Matrix projectionMatrix = Matrix.CreatePerspectiveFieldOfView(
                  fovAngle, aspectRatio, nearClippingPlane,
                  farClippingPlane);
```

The next thing we need to do is place our camera in the scene somewhere. So first let's add the line below to our `SetupCamera()` method:

```
Vector3 cameraLocation = new Vector3(0, 0, 5);
```

Now we need to specify what we are looking at, so let's add the next line to our code as well:

```
Vector3 lookAt = new Vector3(0, 0, 0);
```

Now our camera knows what to look at, but it could rotate around the axis that it is looking in. The last thing we need to do is tell it what direction we want to be up. The line below says we want the y-axis to be up. (By the way, the Vector3 class defines an Up vector, which we could use instead.)

```
Vector3 up = new Vector3(0, 1, 0);
```

Now we need to combine all three of these together into one matrix, with the following line of code:

```
Matrix viewMatrix = Matrix.CreateLookAt(cameraLocation, target, up);
```

The *Colored* Effect

We are about ready start using our new camera. There is one major thing we need to do first. The effect we were using before, the "Pretransformed" effect, is not good enough for what we want to do next. So let's switch, and use an effect that *can*. This effect is the "Colored" effect, and we get it from the effect file just like we did with the "Pretransformed" effect. Find the line in the line in the Draw() method that looks like `effect.CurrentTechnique = effect.Techniques["Pretransformed"];` and change it to `effect.CurrentTechnique = effect.Techniques["Colored"];`

Sending Data to the Effect

Now go back to the `SetupCamera()` method. At the bottom of that method, we need to put in some code to send our camera information over to the effect on the graphics card. Add the following three lines here.

```
effect.Parameters["xWorld"].SetValue(Matrix.Identity);
effect.Parameters["xProjection"].SetValue(projectionMatrix);
effect.Parameters["xView"].SetValue(viewMatrix);
```

These lines send information over to the effect on the graphics card, and assign them to a particular value. The strings in the brackets are defined variables in the effect. If you write your own effect, you can define these as you want. Since "xWorld", "xProjection", and "xView" are the ones defined in our effect, those are the ones we need to use.

From the discussion above, you should understand what the "xProjection" and "xView" parameters do. We will discuss the "xWorld" parameter more later.

Our `SetupCamera()` method is now complete, and should look like this:

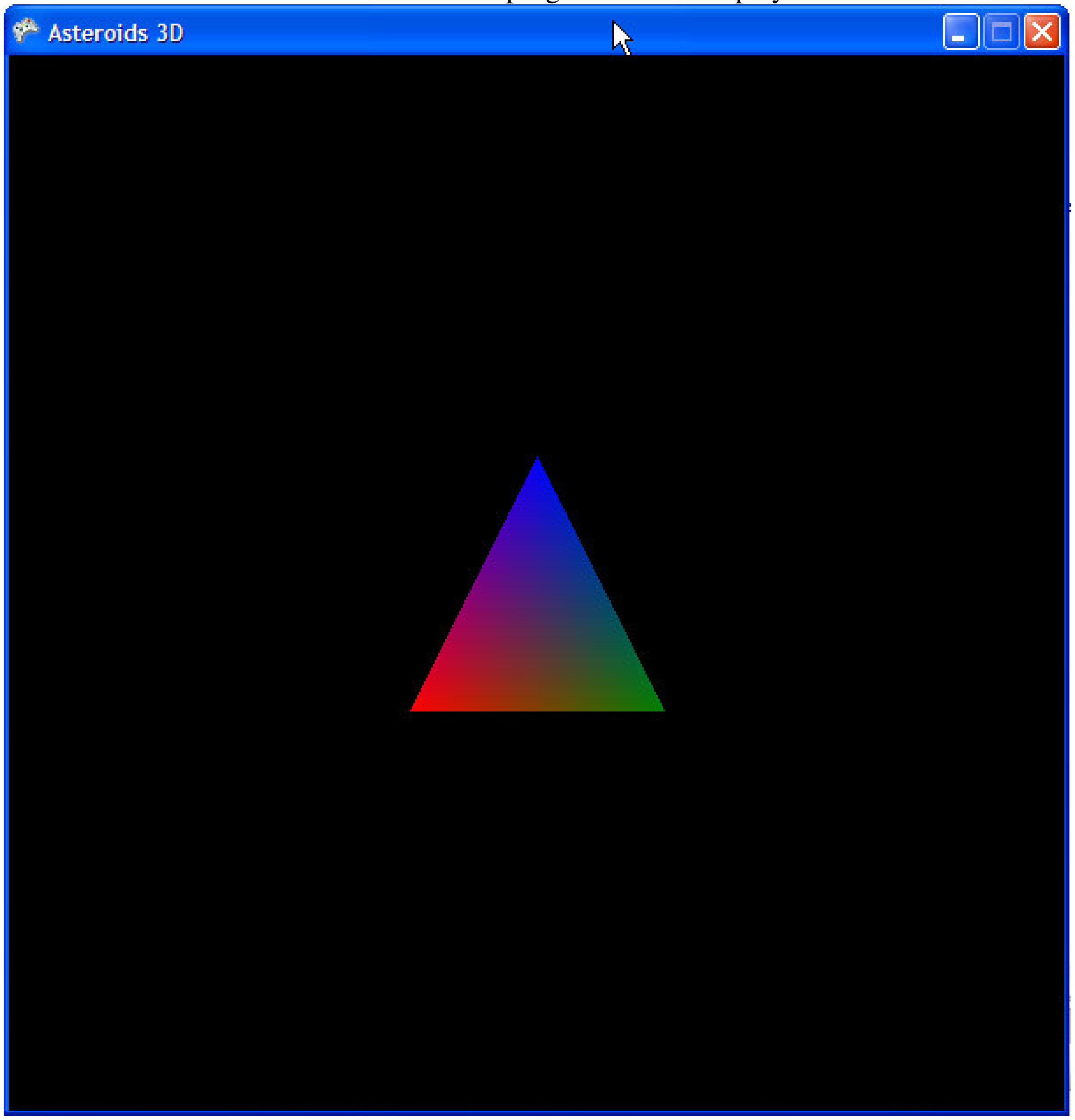
```
protected void SetupCamera()
{
	float fovAngle = MathHelper.PiOver4;  // 45 degrees in radians
	float aspectRatio = this.Window.ClientBounds.Width /
	                        this.Window.ClientBounds.Height;
	float nearClippingPlane = 0.1f;
	float farClippingPlane = 100f;
	Matrix projectionMatrix =
		Matrix.CreatePerspectiveFieldOfView(fovAngle, aspectRatio,
		nearClippingPlane, farClippingPlane);

	Vector3 cameraLocation = new Vector3(0, 0, 3);
	Vector3 target = new Vector3(0, 0, 0);
	Vector3 up = new Vector3(0, 1, 0);
	Matrix viewMatrix = Matrix.CreateLookAt(cameraLocation, target,
		up);

	effect.Parameters["xWorld"].SetValue(Matrix.Identity);
	effect.Parameters["xProjection"].SetValue(projectionMatrix);
	effect.Parameters["xView"].SetValue(viewMatrix);
}
```

The one thing left to do is to call the `SetupCamera()` method from our Initialize() method. Add the line below to the bottom of the Initialize() method:

```
        SetupCamera();
```

We can now run our program again. We should get an image that is almost identical to what it was before. Now, though, we are going to be able to some more interesting things with our triangles in the rest of the tutorials.

Below is a screen shot of what the program should display:


Asteroids 3D