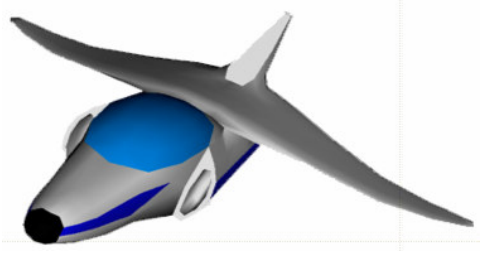


XNA Tutorials

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

Transformations

Tutorial 7

Overview

Now that we have an effect capable of handling transformations, let's do some. We'll first talk about what a transformation is, and then we'll talk about doing each of the basic types in turn.

Transformations

In graphics, a transformation is a way of taking a point in 3D world space, and changing it. We can apply a transformation to all of the points of a triangle, or even all of the points of a high detail model, and apply the transformation to the whole thing.

In graphics, transformations are done with matrices. But fortunately for us, C# has lots of built in matrix transformations, and so we don't really need to know what is going on in order to use it.

Transformations could take any form. In particular, though, there are three types of transformations that will be interesting to us. These are rotations, translations, and scaling transformations. We will talk about what each of these do and how to do each of them in turn.

Rotations

A rotation rotates our points around a line in space. The Matrix class in C# has methods defined for producing rotation matrices. We just need to supply what line we want to rotate around, and how far we want to rotate. For example, if we want a matrix that will cause rotations around the y-axis, we would use the following line:

```
Matrix rotationMatrix = Matrix.CreateRotationY(MathHelper.ToRadians(45));
```

Note that the parameter for the CreateRotationY() method needs to be in radians, so we can use the MathHelper class to convert from degrees if needed. Likewise, there are CreateRotationX and CreateRotationZ methods if you want to rotate around those axes.

If you want to rotate around some arbitrary line (axis) in space, besides those three, you can use the method Matrix.CreateFromAxisAngle() method, which requires a second parameter that defines the axis. The axis is specified as a vector, but the vector

must be normalized (have a length of 1) first. This can be done with the three lines below:

```
Vector3 axis = new Vector3(3, 1, 2);
axis.Normalize();
Matrix rotationMatrix = Matrix.CreateFromAxisAngle(axis,
                        MathHelper.ToRadians(45));
```

We're now going to discuss translations and scaling transformations, but feel free to jump ahead to the "Coding the Transformation" section if you want to just dive in and code your transformations.

Translations

A translation is a transformation that moves our objects from one location to another. We can use the Matrix class again to create a translation matrix. The following line of code will create a translation matrix:

```
Matrix translationMatrix = Matrix.CreateTranslation(new Vector3(1, .5, 0));
```

This line will create a translation matrix that will move our objects 1 unit in the x direction, .5 units in the y direction, and 0 units in the z direction.

Scaling Transformations

A scaling transformation is one that will resize our object. The following line will create a scaling matrix:

```
Matrix scaleMatrix = Matrix.CreateScale(.5);
```

This will create a scaling matrix that will make everything 0.5 times as big as it used to be (half the size). The one trick to scaling matrices is that it scales around the origin (0, 0, 0). That means if you have a bunch of points out around the point (100, 100, 100) and you double their scale, they will all be around the point (200, 200, 200).

Coding the Transformation

Alright, we know enough about transformations that we can put them into our code now. The first step is to create the transformation matrix that we want to use. So let's start with a rotation matrix for now.

Let's create a variable that says how much to rotate our triangle. Add the following line to your code below the other class variables (content, device, vertexArray, etc.):

```
private float angle = MathHelper.ToRadians(45);
```

Now let's create the transformation matrix. Add the following line of code to your Draw method, before the foreach loop:

```
Matrix rotationMatrix = Matrix.CreateRotationY(angle);
```

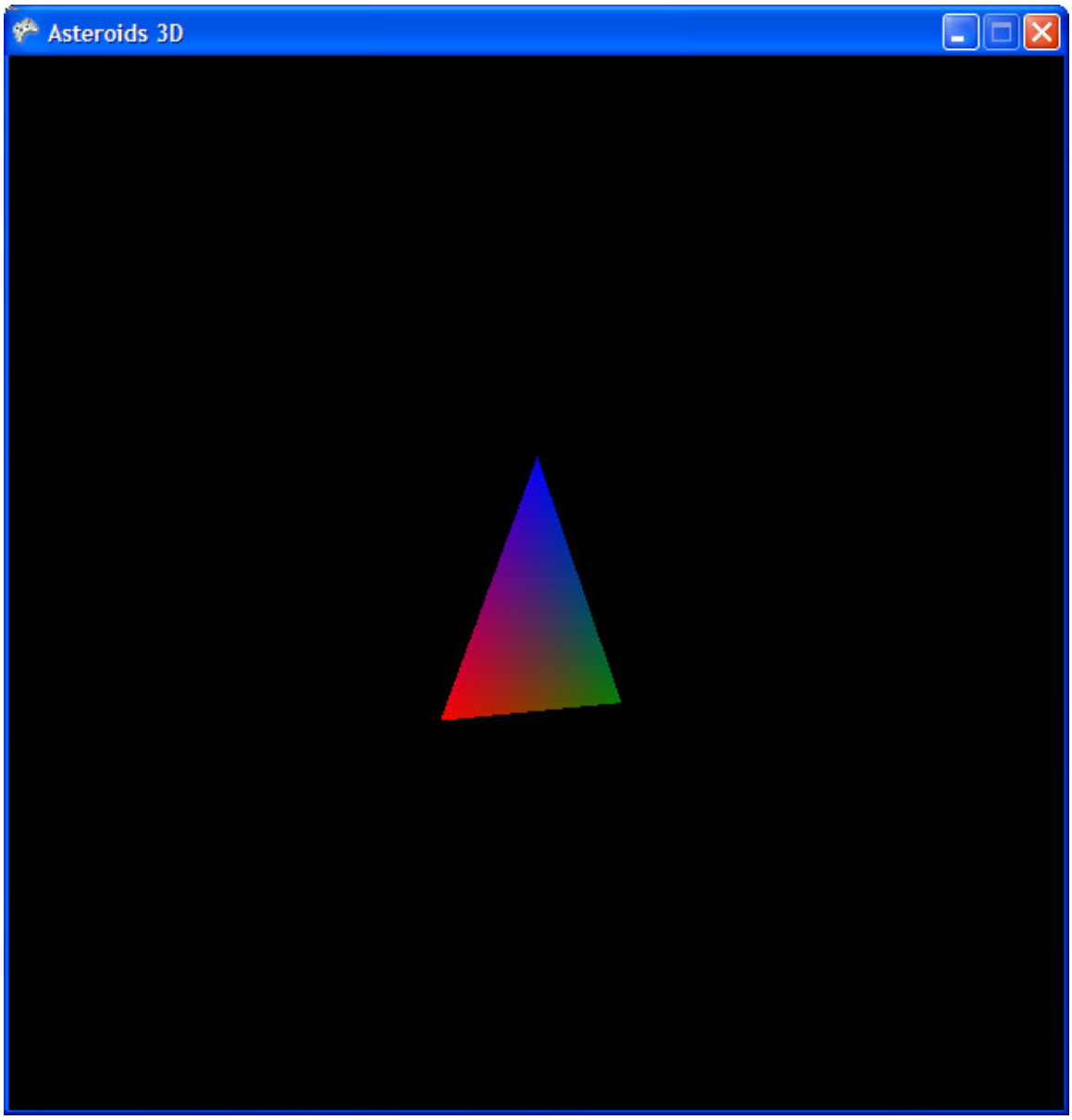
Now we have to tell the effect that this is what we want to do. Add the following line of code just after the last line:

```
effect.Parameters["xWorld"].SetValue(rotationMatrix);
```

The Draw() method should now look like this:

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

	effect.CurrentTechnique = effect.Techniques["Colored"];
	effect.Begin();

	Matrix rotationMatrix = Matrix.CreateRotationY(angle);
	Matrix worldMatrix = rotationMatrix;
	effect.Parameters["xWorld"].SetValue(worldMatrix);

	foreach (EffectPass pass in effect.CurrentTechnique.Passes)
	{
		pass.Begin();

		device.VertexDeclaration = new VertexDeclaration(device,
			VertexPositionColor.VertexElements);
		device.DrawUserPrimitives(PrimitiveType.TriangleList,
			vertexArray, 0, 1);

		pass.End();
	}

	effect.End();

	base.Draw(gameTime);
}
```

We can now run our program again, and see the rotation. The program's display should look like the window below:



Transforming Over Time

Now let's make our triangle spin over time. Actually, this will be very easy. All we need to do is update our value of `angle`. This is the kind of thing we want to put in the Update method. Find the update method and add the following line to it:

```
angle += .1f;
```

We can now run our program again, and get some cool results. Run it to see the triangle spin. If we wanted to be really fancy, we could have it change by a particular amount depending on how much time had passed since the last time we updated. That way, if our game was running a little slow, our updating could keep up with it. We could get this information out of the `gameTime` object that was passed to the `Update()` method.

Now you might be wondering why the triangle disappears half of the time. Triangles in the graphics world have a front and a back side. Usually, the back side of triangles is not drawn, because that means they are probably facing away from you, and the other side of the object will be drawn. Not drawing these backwards triangles saves the graphics card a lot of time. This process of not drawing backwards triangles is called culling.

We can turn off culling, and for right now, that might be a good idea. Let's add the following line of code to the Display() method before the foreach loop to turn off culling:

```
device.RenderState.CullMode = CullMode.None;
```

Now if we run our program, we will see the back face of the triangle as well.

Multiple Transformations at Once

What if we want to do multiple transformations together? For example, what if I want to translate *and* rotate? Well, with matrices, that is easy to do. You can combine two transformations into one simply by multiplying the transformation matrices together.

Let's add a simple translation to what we have already done. In the place where we defined the variable rotationMatrix in the `Display()` method, add the following line:

```
Matrix translationMatrix = Matrix.CreateTranslation(
                  new Vector3(0, 1, 0));
```

Now we need to combine this matrix with the other. We just multiply these two matrices together when we assign the variable worldMatrix a value. If you remember from Linear Algebra, the order in which you multiply matices makes a difference (i.e. A * B is not the same as B * A). For us, the order will determine which transformation occurs first. If we multiply two matrices together, their transformations occur in the reverse order of how they are written. For the multiplication A * B, we will do the B transformation, and then the A transformation after that. This holds true if we are multiplying lots of transformations together, too. If we have A * B * C * D, then the order our transformations will occur is D, C, B, and then A.

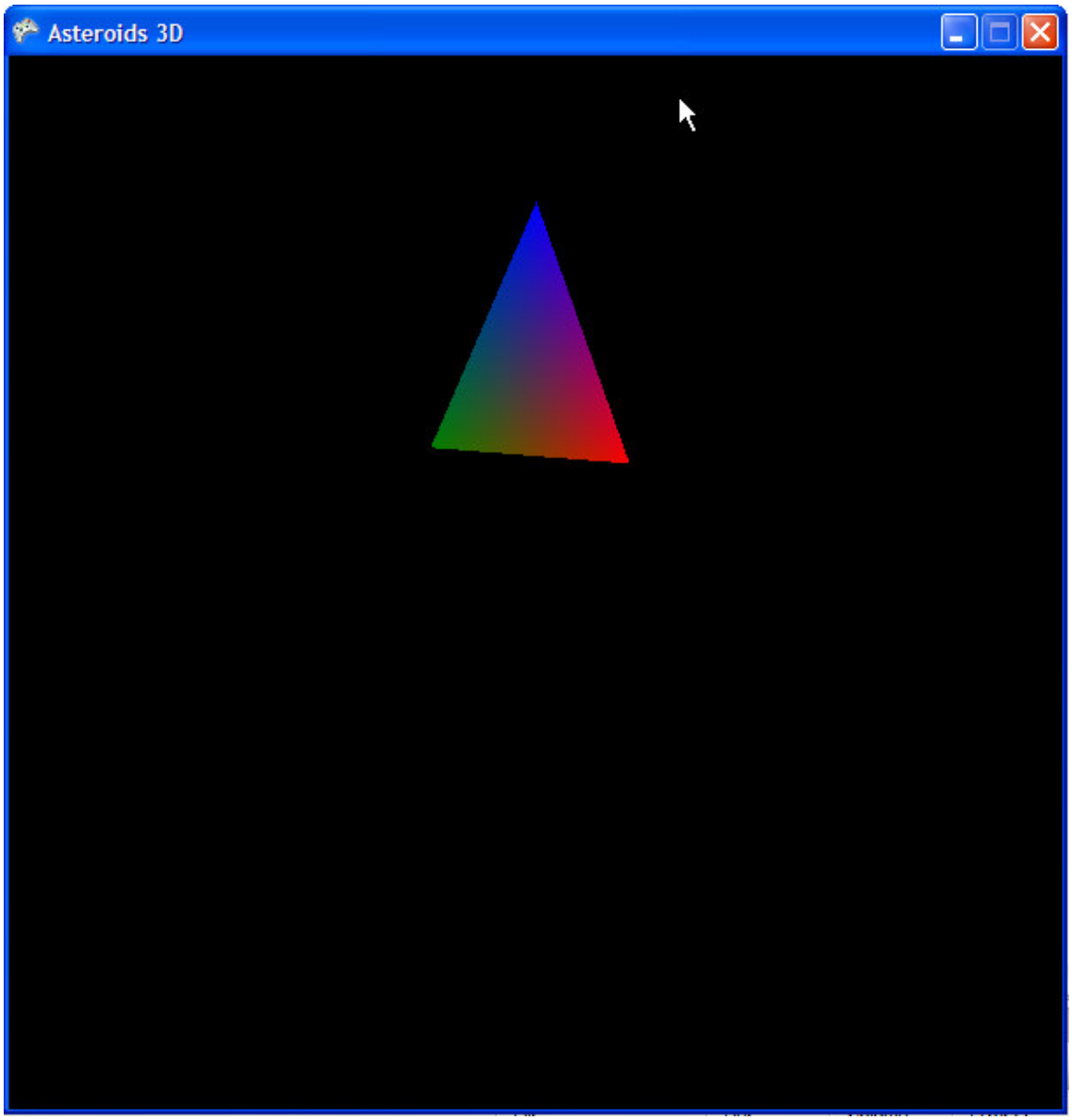
Let's do our rotation first, and then our transformation (although you can switch the order later to see what happens). In the Display method, change the line that says:

```
Matrix worldMatrix = rotationMatrix;
```

To say:

```
Matrix worldMatrix = translationMatrix * rotationMatrix;
```

Now let's run our program again to see the effect. The output should look like the image below, because we are now rotating and translating.


Asteroids 3D