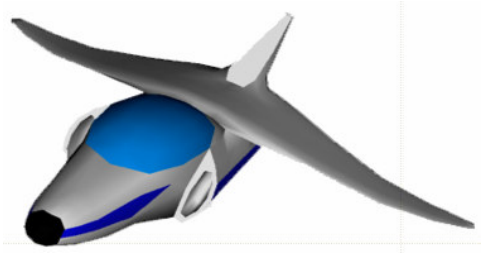


XNA Tutorials

Utah State University
Association for Computing Machinery
XNA Special Interest Group
RB Whitaker
23 December 2007

Line Drawing

Tutorial 17

Overview

We have covered a lot of ground in XNA. But so far, most of our drawing has been triangles—as individual triangles, or as large groups of them (models). In this tutorial, we will go in a slightly different direction and learn the basics of how to draw lines in XNA. For this tutorial specifically, we will use lines to draw the x-, y-, and z-axes on the screen.

Setting Up Vertices

The first thing we need to do is to define some vertices. This step will probably seem very familiar to you. This will be almost identical to the work we did earlier to set up vertices for triangles.

Let's first define an array to store our vertices as a class level variable. Since we want the lines to be colored, let's make it an array of VertexPositionColor elements. Add the following line to your code as a class level variable:

```
protected VertexPositionColor[] vertices;
```

Next let's actually define the values for these vertices. In the `LoadGraphicsContent()` method, inside of the `if (loadAllContent)` statement, add the following code:

```
vertices = new VertexPositionColor[6];
vertices[0] = new VertexPositionColor(Vector3.Zero, Color.Red);
vertices[1] = new VertexPositionColor(Vector3.UnitX, Color.Red);
vertices[2] = new VertexPositionColor(Vector3.Zero, Color.Green);
vertices[3] = new VertexPositionColor(Vector3.UnitY, Color.Green);
vertices[4] = new VertexPositionColor(Vector3.Zero, Color.Blue);
vertices[5] = new VertexPositionColor(Vector3.UnitZ, Color.Blue);
```

This gives us three lines (each has two endpoints). One line goes from (0, 0, 0) to (1, 0, 0) and will be colored red, another goes from (0, 0, 0) to (0, 1, 0) and will be colored green, and the last goes from (0, 0, 0) to (0, 0, 1) and will be colored blue.

Drawing the Lines

We now want to do the actual drawing of the lines. This will again be similar to drawing triangles with only a few changes. We need to add a few lines of code to draw our coordinate axes. The code we will need is shown below:

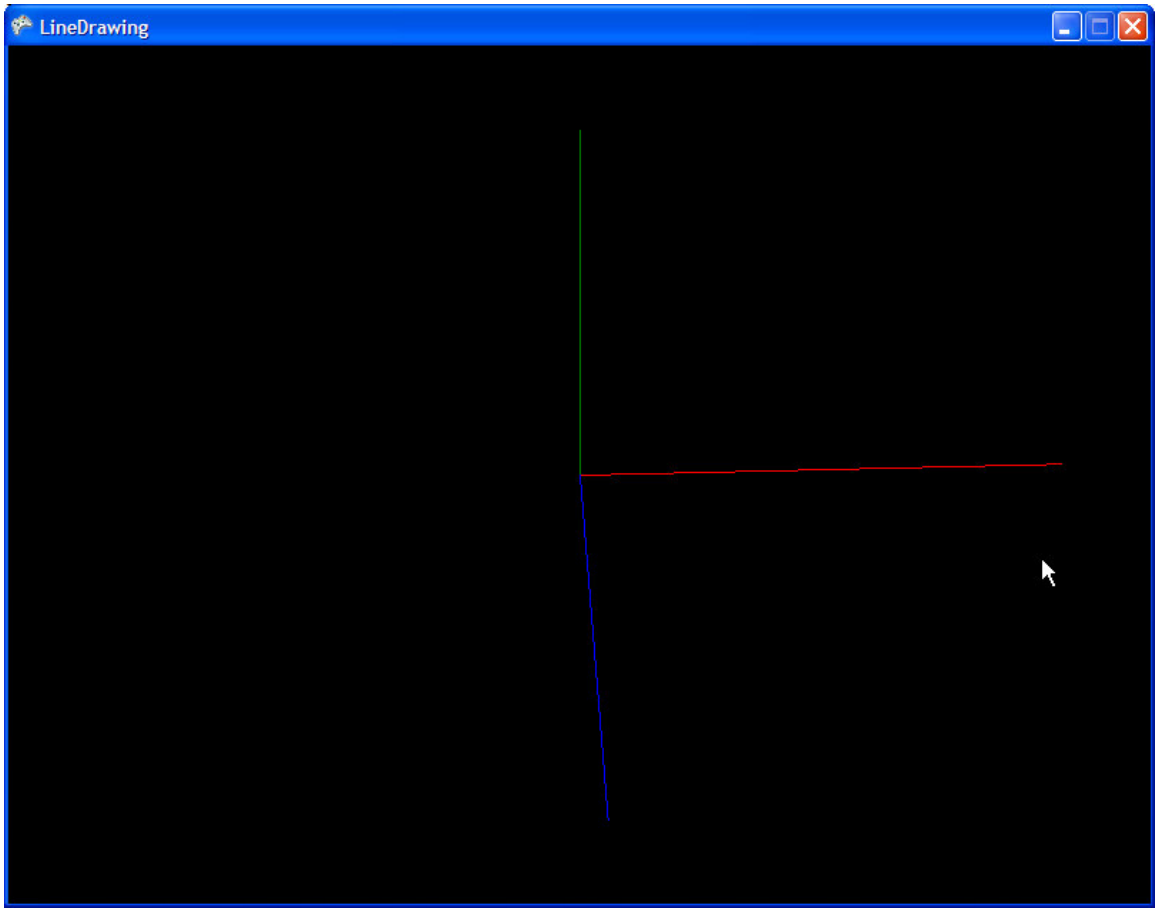
```
effect.Begin();

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

      graphics.GraphicsDevice.VertexDeclaration = new
                  VertexDeclaration(graphics.GraphicsDevice,
                  VertexPositionColor.VertexElements);

      graphics.GraphicsDevice.DrawUserPrimitives(
                  PrimitiveType.LineList, vertices, 0, 3);

      pass.End();
}

effect.End();
```

We start with a call to the current effect's Begin() method. Then, like before, we need to go through all of the passes in the technique that is being used. We start the current pass with a call to the effect pass's begin method.

The next two lines (the middle-most) are where the new information lies. The first line assigns a new vertex declaration to the graphics card. We saw this in another tutorial, but a vertex declaration is simply a structure that says "this is what's in the data that I'm about to send you." We create a new VertexDeclaration object, which requires a reference to the graphics device, and also a list of what data is stored in each vertex, which is accessed by the `VertexPositionColor.VertexElements`. If you were using another type of vertex, like VertexPositionTexture, you would use the VertexPositionTexture.VertexElements reference instead. (By the way, lines look kind of weird/cool when you try to texture them.)

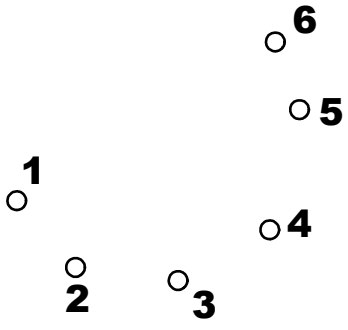
The second line says to draw primitives, like we did before, be the critical feature here is that we are using `PrimitiveType.LineList`, instead of `PrimitiveType.TriangleList`. Just like when we were drawing triangles, we also need to specify the offset into the array, which is 0, and then the total number of primitives to draw. In this case we are drawing all three axes, so we are drawing three lines.

If we run our program now, we should get something like the image below:

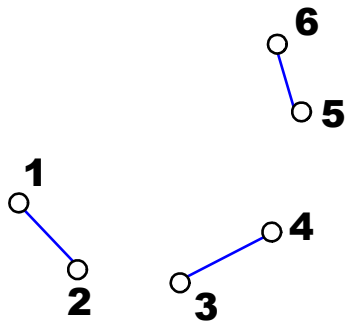


Line Strips

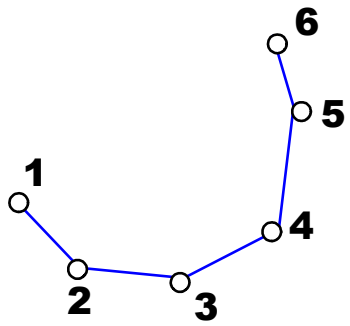
While we are talking about drawing lines, let's take a look at one other possible way to draw lines. This way is using line strips. The difference between a line list and a line strip is that a line list assumes that every pair of two lines in the array are a new line segment. A line strip assumes that every new element in the array is supposed to be a new segment from the previous point to the current point. For example, look at the image below. Here are six points that are in an array, marked in the order that they appear in the array.



With a line list, this would be drawn like the image below:



In the above image, each pair of points (1 and 2, 3 and 4, 5 and 6) make up a new line segment because the graphics card is told to treat the array as a line list. The image below shows how these points would be drawn in a line strip:



This image shows that as the program reads a new vertex from the array, it assumes that there is supposed to be a line from the last vertex to this new vertex.

Let's try an example, just to see how line strips work. Let's draw a circle. (Or more accurately, let's draw a bunch of lines that are roughly the shape of a circle, and the user won't even be able to tell that it isn't a circle!)

Let's go back up to where we defined our list of vertices and put in the following code:

```
protected int divisions = 64;
protected VertexPositionColor[] vertices2;
```

`divisions` is the total number of line segments we are going to have, all spread out evenly along a circle. This way it will be easy to change. It might be worth your time later to come back and change this to lower numbers and higher numbers to see the difference. At 64 it should look a lot like a circle though. `vertices2` will be another array of vertices that we will need to define.

Go down to the `LoadGraphicsContent()` method and just below where we defined our information for `vertices`, let's define the vertices for our circle. Rather than defining each of them by hand, let's make a loop do the work for us. I used the following code to do mine:

```
vertices2 = new VertexPositionColor[divisions + 1];
float angleIncrement = (float)(Math.PI * 2 / divisions);
```

```
for (int i = 0; i < vertices2.Length; i++)
{
      float angle = angleIncrement * i;
      Vector3 position = new Vector3((float)Math.Sin(angle),
                  (float)Math.Cos(angle), 0);
      vertices2[i] = new VertexPositionColor(position, new
                  Color(position));
}
```

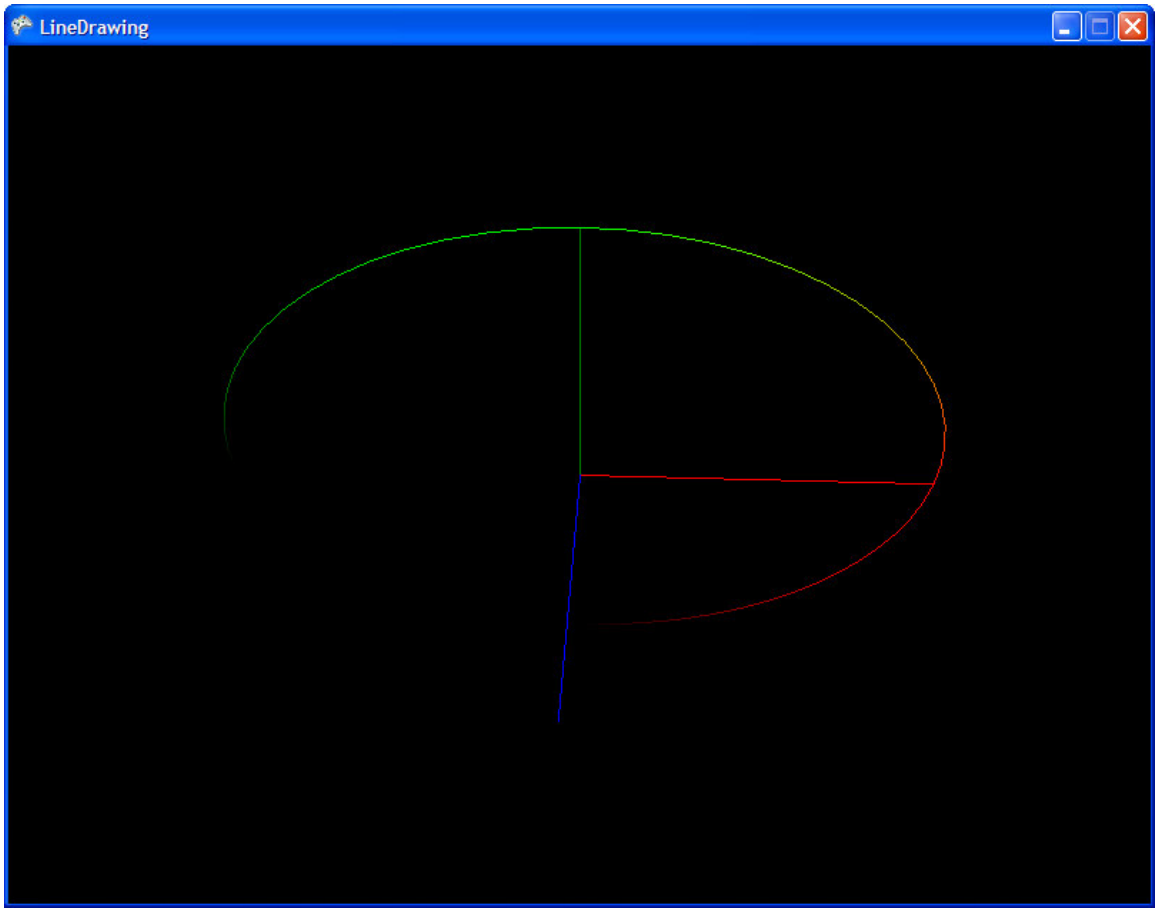
In the first line, we declare our array to have divisions + 1 elements, because we will need one extra vertex to complete the loop. The next line creates a variable called angleIncrement, which stores the angle between any two vertices. For instance, if we had four divisions, each of them would be 90° apart from each other. We will keep track of this angle for use later in the loop. We then start our loop and look at each of the vertices in our list. The first thing I did was calculate the current angle. We already knew the angle increment, and so the current angle is just the angle increment multiplied by the number of this vertex. We next calculate the position of the point, by using trigonometry. Our circle will be in the xy-plane. Finally, we create a new VertexPositionColor object with our desired position. I used a little trick to color the circle. The Color class has a constructor that takes a Vector3 and uses the values in it as the values for red, green, and blue. You'll see that this makes for a nice (but easy) color gradient around the circle, but since part of the circle has negative positions, the color will be black and won't show up. You could also just put in a color (say `Color.White`) and just made the whole circle one particular color.

The only thing left for us to do is to draw the circle. So go to the `Draw()` method and put in the following code. Put it between the `pass.Begin()` and `pass.End()` calls, just after the calls that drew our axes.

```
graphics.GraphicsDevice.DrawUserPrimitives(PrimitiveType.LineStrip,
            vertices2, 0, divisions);
```

Here, we are doing like we did before with line lists, only we are using `PrimitiveType.LineStrip` instead. We are also drawing the data in `vertices2`, rather than `vertices`, and we want `divisions` total line segments, so we've made those changes too.

We can now run our program again, and we should see something like the image below:



As a side note, if you want to draw a complete model as a wire frame, Supplementary Tutorial 5 covers that.