Let's Build a 3D Graphics Engine: Rasterizing Triangles and Quads

by Kyle Sloka-Frey 13 Jun 2013

Difficulty: Intermediate Length: Medium Languages: English

Programming Platform Agnostic HTML5

This post is part of a series called Let's Build a 3D Graphics Software Engine.

Let's Build a 3D Graphics Engine: Rasterizing Line Segments and Circles

Let's Build a 3D Graphics Engine: Colors

Welcome to the fifth part of our Let's Build a 3D Graphics Engine series! This time, we will be building two new classes for rasterizing: one for triangles and one for basic quadrilaterals. Then, we are going to take pieces from those two classes and put together a final, almighty polygon class.

Tip: This is a part of a series, so if you want to get the most out of it make sure that you read the other tutorials leading up to this one.

Recap

We've built quite a bit into our engine so far! Here's what we have:

- Point and Vector classes (the building blocks of our engine).
- Transformation functions for our points.
- A Camera class (sets our viewport, and culls points outside of the screen).
- Two classes for rasterizing (line segments and circles).

Here is a quick reference for all of the classes we've built:

```
Point Class
{
    Variables:
        num tuple[3]; //(x,y,z)
    Operators:
        Point AddVectorToPoint(Vector);
        Point SubtractVectorFromPoint(Vector);
        Vector SubtractPointFromPoint(Point);
        Null SetPointToPoint(Point);
    Functions:
        drawPoint; //draw a point at its position tuple
}

Vector Class
{
    Variables:
        num tuple[3]; //(x,y,z)
    Operators:
        Vector AddVectorToVector(Vector);
        Vector SubtractVectorFromVector(Vector);
        Vector RotateXY(degrees);
        Vector RotateYZ(degrees);
        Vector RotateXZ(degrees);
        Vector Scale(s0,s1,s2); //receives a scaling 3-tuple, returns the scaled vector
}

Camera Class
{
    Vars:
        int minX, maxX;
        int minY, maxY;
        int minZ, maxZ;
        array objectsInWorld; //an array of all existent objects
    Functions:
        null drawScene(); //draws all needed objects to the screen
}

LineSegment Class
{
    Variables:
        int startX, startY; //the starting point of our line segment
        int endX, endY; //the ending point of our line segment
    Function:
        array returnPointsInSegment; //all points lying on this line segment
}
```

We are going to rely heavily on the `LineSegment` class to create our `Triangle` and `Quad` classes, so make sure to refamiliarize yourself with it before moving on.

Rasterizing Triangles

Putting together a `Triangle` class for the engine is fairly simple, especially since the `LineSegment` class is where all of our rasterization is actually going to take place. This class will allow three points to be set, and will draw a line segment between them to make the completed triangle.

A basic outline of the class could look like this:

```
Triangle Class
{
    Variables:
        //co-ordinates for the three points of our triangles
        int Point1X, Point1Y;
        int Point2X, Point2Y;
        int Point3X, Point3Y;
    Function:
        array returnPointsInTriangle; //all points within the triangle's perimeter
}
```

For the sake of standards, we are going to assume that the three points declared within our triangle are in a clockwise pattern.

Using our `LineSegment` class, then, we can set up our `returnPointsInTriangle()` function like this:

```
function returnPointsInTriangle()
{
    array PointsToReturn; //create a temporary array to hold the triangle's points

    //Create three line segments and store their points in the array
    PointsToReturn.push(new LineSegment(this.Point1X, this.Point1Y, this.Point2X, this.
    PointsToReturn.push(new LineSegment(this.Point2X, this.Point2Y, this.Point3X, this.
    PointsToReturn.push(new LineSegment(this.Point3X, this.Point3Y, this.Point1X, this.

    return(PointsToReturn);
}
```

Not too bad, right? Since we already have a lot of the work being done within our `LineSegment` class, we only have to continue stringing them together to create more complex shapes. This makes it easy to create ever more complicated polygons on the screen, simply by adding on more `LineSegments` (and storing more points within the class itself).

Next, let's take a look at how we can add more points to this system by creating a square class.

Advertisement

Getting Squared Away

Putting together a class to handle quadrilaterals only involves adding a few extra things to our `Triangle` class. With another set of points, our quadrilateral class would look like this:

```
Quad Class
{
    Variables:
        int Point1X, Point1Y; //co-ordinates for the four points of our quadrilateral
        int Point2X, Point2Y;
        int Point3X, Point3Y;
        int Point4X, Point4Y;

    Function:
        array returnPointsInQuad; //return all points within the quadrilateral
}
```

Then, we just add in the additional line segment to the `returnPointsInQuad` function, like so:

```
function returnPointsInQuad()
{
    array PointsToReturn; //create a temporary array to hold the quad's points

    //Create four line segments and store their points in the array
    PointsToReturn.push(new LineSegment(this.Point1X, this.Point1Y, this.Point2X, this.
    PointsToReturn.push(new LineSegment(this.Point2X, this.Point2Y, this.Point3X, this.
    PointsToReturn.push(new LineSegment(this.Point3X, this.Point3Y, this.Point4X, this.
    PointsToReturn.push(new LineSegment(this.Point4X, this.Point4Y, this.Point1X, this.

    return(PointsToReturn);
}
```

While building new classes like this is pretty straight-forward, there is a much easier way to encapsulate all of our polygons into one class. By using the magic of loops and arrays, we can put together a polygon class that could make almost any size shape that you could want!

Where Have All the Polys-Gon?

To create an ever expanding polygon class, we need to do two things. The first is to move all of our points into an array, which would give us a class outline similar to something like this:

```
Polygon Class
{
    Variables:
        array Points; //holds all of the polygon's points in an array

    Function:
        array returnPointsInPolygon; //an array holding all of the polygon's points
}
```

The second is to use a loop to allow an unnamed number of line segments to be traversed in our `returnPointsInPolygon()` function, which could look something like this:

```
function returnPointsInPolygon
{
    array PointsToReturn; //a temporary array to hold the polygon's points

    //loop through all points in the polygon, moving one co-ordinate pair at a time (by
    for(int x = 0; x < this.Points.length; x+=2)
    {
        if(this is not the last point)
        {
            //create a line segment between this point and the next one in the array
            PointsToReturn.push(new LineSegment(this.Points[x], this.Points[x+1], this.
        }
        else if(this is the last point)
        {
            //create a line segment between this point and the first point in the array
            PointsToReturn.push(new LineSegment(this.Points[x-2], this.Points[x-1], thi
        }
    }

    //return the array of points
    return PointsToReturn;
}
```

With this class added to our engine, we can now create anything from a triangle to some 39-sided abomination with the same line of code.

Polygon Creator

To play with our new polygon class, let's make a program that shows the extent of its reach. Our program is going to allow the user to add to or remove sides from the displayed polygon using key presses. Of course, we will have to set limits for the number of sides that our polygon can have, since having less than three sides will no longer make it a polygon. We don't really have to keep an eye on the upper limits of our polygon because they should scale nicely. However, we are going to limit polygons to having ten sides at maximum since we will be setting its new points from within the code.

Our program specifications can be broken down into these smaller parts:

- Draw a polygon to the screen initially.
- When the 'a' key is pressed, lower the number of sides on the polygon by 1.
- When the 's' key is pressed, increase the number of sides on the polygon by 1.
- Prevent the polygon's number of sides from falling below 3.
- Prevent the polygon's number of sides from increasing above 10.

Let's look at what our code could look like:

```
main{
    //setup for your favorite Graphics API here
    //setup for keyboard input (may not be required) here

    var camera = new Camera(); //create an instance of our camera class
    camera.objectsInWorld[]; //initialize the camera's object array

    //set up the camera's view space
    camera.minX = 0;
    camera.maxX = screenWidth;
    camera.minY = 0;
    camera.maxY = screenHeight;
    camera.minZ = 0;
    camera.maxZ = 100;

    //create an array of points for each polygon size
    var threeSides = new Array(100,100,100,50,50,50);
    var fourSides = new Array(points in here);
    var fiveSides = new Array(points in here);
    var sixSides = new Array(points in here);
    var sevenSides = new Array(points in here);
    var eightSides = new Array(points in here);
    var nineSides = new Array(points in here);
    var tenSides = new Array(points in here);

    //store all of the arrays in another array for easier access
    var sidesArray = new Array(threeSides, fourSides, fiveSides, sixSides, sevenSides,

    //keep track of how many points the polygon currently has
    var polygonPoints = 3;

    //create the initial polygon to be displayed
    var polygon = new Polygon(sidesArray[0][0], sidesArray[0][1], sidesArray[0][2], sid
    //draw the initial polygon to the screen
    camera.drawScene();

    //while the user has not pressed the escape key
    while(key != esc) {
        if(key pressed == 'a')
        {
            //if the polygon is not at risk of falling below 3
            if(polygonPoints != 3)
            {
                //reduce the number of points
                polygonPoints--;
                //change the polygon to have the correct number of points
            }
            //redraw the scene
            camera.drawScene();
        }
        else if(key pressed == 's')
        {
            //if the polygon is not at risk of going above 10
            if(polygonPoints != 10)
            {
                //increase the number of points
                polygonPoints++;
                //change the polygon to have the correct number of points
            }
            //redraw the scene
            camera.drawScene();
        }
    }
}
```

Our little program should let you adjust a polygon on-screen now! Check out the demo. If you would like to beef up this program a bit, you might want to try putting the polygon alteration section into some form of an algorithm to make the scaling easier on yourself. I'm unsure if one already exists, but if it does, you could easily have an infinitely scaling polygon on your hands!

Conclusion

We've quite an extensive amount of rasterizing built into our engine now, letting us create almost any shape that we might need (though some only through combination). Next time we will be moving away from drawing shapes and talking more about their properties. If you're interested in bringing some color to your screen, then be sure to check out the next part!

Kyle Sloka-Frey

Kyle Sloka-Frey

Data Scientist, Father, Game Dev, Developer, Futurist

Kyle is a data scientist, father, game dev, and all around analytics and organization lover. He enjoys space and all things Futurism.

kyleslokafrey