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

by Kyle Sloka-Frey 7 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: Spaces and Culling

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

Hello, this is the fourth part of our series on 3D graphics engines. This time, we will be covering *rasterization*: the process of taking a shape described by mathematical formulas and converting it to an image on your screen.

Tip: All of the concepts in this article are built off of classes that we've established in the first three posts, so be sure to check them out first.

Recap

Here's a review of the classes that we've made so far:

```
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
}
```

You can check out the sample program from the third part of the series to see how they work together.

Now, let's take a look at some new stuff!

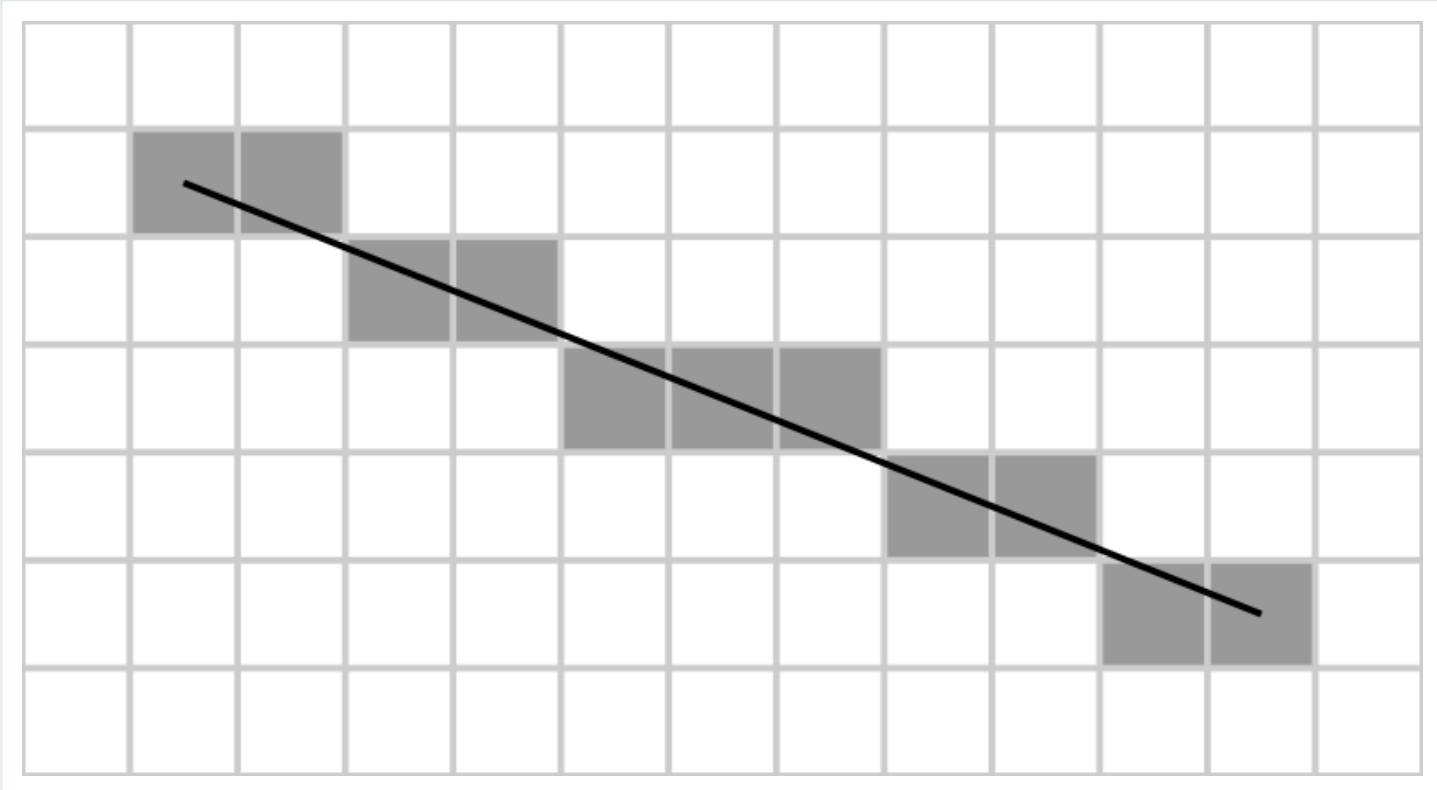
Rasterization

Rasterization (or *rasterisation*, if you like) is the process of taking a shape described in a vector graphics format (or in our case, mathematically) and converting it into a raster image (where the shape is fit onto a pixel structure).

Because math isn't always as precise as we need it to be for computer graphics, we must use algorithms to fit the shapes it describes onto our integer-based screen. For example, a point can fall onto the co-ordinate $(3.2, 4.6)$ in mathematics, but when we render it, we must nudge it to $(3, 5)$ so it can fit into the pixel structure of our screen.

Each type of shape that we rasterize will have its own algorithm for doing so. Let's start with one of the more simple shapes to rasterize: the *line segment*.

Line Segments



Source: https://en.wikipedia.org/wiki/File:Bresenham.svg

Line segments are one of the simplest shapes that can be drawn, and so are often one of the first things covered in any geometry class. They are represented by two distinct points (one beginning point and one ending point), and the line that connects the two. The most commonly used algorithm in rasterizing a line segment is called *Bresenham's Algorithm*.

Step by step, Bresenham's Algorithm works like this:

1. Receive a line segment's beginning and ending points as input.
2. Identify a line segment's direction by determining its \(dx\) and \(dy\) properties (\(dx = x_{1} - x_{0}\), \(dy = y_{1} - y_{0}\)).
3. Determine `sx`, `sy`, and error catching properties (I'll show the mathematical definition for these below).
4. Round each point in the line segment to either the pixel above or below.

Before we implement Bresenham's Algorithm, lets put together a basic line segment class to be used in our engine:

```
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
}
```

If you want to perform a transformation on our new `LineSegment` class, all you have to do is apply your chosen transformation to the beginning and ending points of the `LineSegment` and then place them back into the class. All of the points that fall between will be processed when the `LineSegment` itself is drawn, as Bresenham's Algorithm only requires the beginning and ending points to find each subsequent point.

In order for the `LineSegment` class to fit with our current engine, we can't actually have a `draw()` function built into the class, which is why I've opted for using a `returnPointsInSegment` function instead. This function will return an array of every point that exists within the line segment, allowing us to easily draw and cull the line segment as appropriate.

Our function `returnPointsInSegment()` looks a bit like this (in JavaScript):

```
function returnPointsInSegment()
{
    //create a list to store all of the line segment's points in
    var pointArray = new Array();
    //set this function's variables based on the class's starting and ending points
    var x0 = this.startX;
    var y0 = this.startY;
    var x1 = this.endX;
    var y1 = this.endY;
    //define vector differences and other variables required for Bresenham's Algorithm
    var dx = Math.abs(x1-x0);
    var dy = Math.abs(y1-y0);
    var sx = (x0 & x1) ? 1 : -1; //step x
    var sy = (y0 & y1) ? 1 : -1; //step y
    var err = dx-dy; //get the initial error value
    //set the first point in the array
    pointArray.push(new Point(x0,y0));

    //Main processing loop
    while(!((x0 == x1) && (y0 == y1)))
    {
        var e2 = err * 2; //hold the error value
        //use the error value to determine if the point should be rounded up or down
        if(e2 => -dy)
        {
            err -= dy;
            x0 += sx;
        }
        if(e2 < dx)
        {
            err += dx;
            y0 += sy;
        }
        //add the new point to the array
        pointArray.push(new Point(x0, y0));
    }
    return pointArray;
}
```

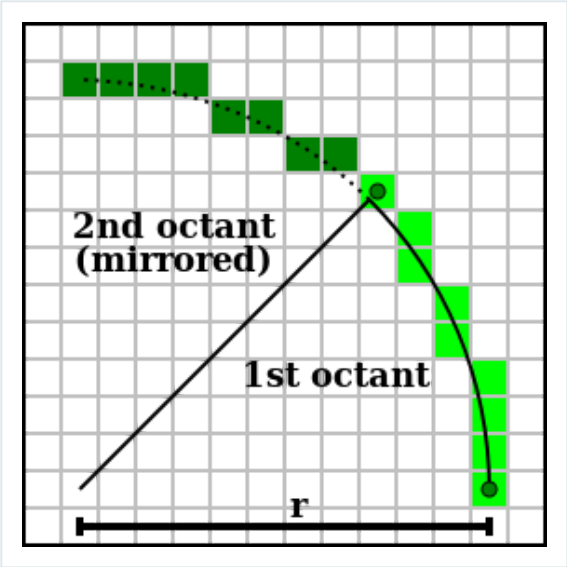
The easiest way to add the rendering of our line segments into our camera class is to add in a simple `if` structure, similar to the following:

```
//loop through array of objects
if (class type == Point)
{
    //do our current rendering code
}
else if (class type == LineSegment)
{
```

```
08      var segmentArray = LineSegment.returnPointsInSegment();
09      //loop through points in the array, drawing and culling them as we have previously
10  }
```

That's all you need to get your first shape class up and running! If you want to learn more about the more technical aspects of Bresenham's Algorithm (particularly the error sections), you can check out the Wikipedia article on it.

Circles



Source: http://en.wikipedia.org/wiki/File:Bresenham_circle.svg

Rasterizing a circle is a bit more difficult than rasterizing a line segment, but not much. We will be using the *midpoint circle algorithm* to do all of our heavy lifting, which is an extension of the previously mentioned Bresenham's Algorithm. As such, it follows similar steps to those that were listed above, with some minor differences.

Our new algorithm works like this:

1. Receive a circle's center point and radius.
2. Forcibly set the points in each cardinal direction
3. Cycle through each of our quadrants, drawing their arcs

Our circle class will be very similar to our line segment class, looking something like this:

```
Circle Class
{
    Variables:
        int centerX, centerY; //the center point of our circle
        int radius; //the radius of our circle
    Function:
        array returnPointsInCircle; //all points within this Circle
}
```

Our `returnPointsInCircle()` function is going to behave in the same way that our `LineSegment` class's function does, returning an array of points so that our camera can render and cull them as needed. This lets our engine handle a variety of shapes, with only minor changes needed for each.

Here is what our `returnPointsInCircle()` function is going to look like (in JavaScript):

```
function returnPointsInCircle()
{
    //store all of the circle's points in an array
    var pointArray = new Array();
    //set up the values needed for the algorithm
    var f = 1 - radius; //used to track the progress of the drawn circle (since its sem
    var ddFx = 1; //step x
    var ddFy = -2 * this.radius; //step y
    var x = 0;
    var y = this.radius;

    //this algorithm doesn't account for the farthest points,
    //so we have to set them manually
    pointArray.push(new Point(this.centerX, this.centerY + this.radius));
    pointArray.push(new Point(this.centerX, this.centerY - this.radius));
    pointArray.push(new Point(this.centerX + this.radius, this.centerY));
    pointArray.push(new Point(this.centerX - this.radius, this.centerY));

    while(x < y) {
        if(f >= 0) {
            y--;
            ddFy += 2;
            f += ddFy;
        }
        x++;
        ddFx += 2;
        f += ddFx;

        //build our current arc
        pointArray.push(new Point(x0 + x, y0 + y));
        pointArray.push(new Point(x0 - x, y0 + y));
        pointArray.push(new Point(x0 + x, y0 - y));
```

```
            pointArray.push(new Point(x0 - x, y0 - y));
            pointArray.push(new Point(x0 + y, y0 + x));
            pointArray.push(new Point(x0 - y, y0 + x));
            pointArray.push(new Point(x0 + y, y0 - x));
            pointArray.push(new Point(x0 - y, y0 - x));
        }

        return pointArray;
    }
```

Now, we just add in another `if` statement to our main drawing loop, and these circles are fully integrated!

Here is how the updated drawing loop may look:

```
//loop through array of objects
if(class type == point)
{
    //do our current rendering code
}
else if(class type == LineSegment)
{
    var segmentArray = LineSegment.returnPointsInSegment();
    //loop through points in the array, drawing and culling them as we have previously
}
else if(class type == Circle)
{
    var circleArray = Circle.returnPointsInCircle();
    //loop through points in the array, drawing and culling them as we have previously
}
```

Now that we've got our new classes out of the way, let's make something!

Raster Master

Our program is going to be simple this time. When the user clicks on the screen, we are going to draw a circle whose center point is the point that was clicked, and whose radius is a random number.

Let's take a look at the code:

```

```

```
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 the camera class
    camera.objectsInWorld[]; //create 100 object spaces within the camera's array

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

    while(key != esc) {
        if(mouseClick) {
            //create a new circle
            camera.objectsInWorld.push(new Circle(mouse.x,mouse.y,random(3,10));
            //render everything in the scene
            camera.drawScene();
        }
    }
}
```

With any luck, you should now be able to use your updated engine to draw some awesome circles.

Conclusion

Now that we have some basic rasterization features in our engine, we can finally start drawing some useful things to our screen! Nothing too complicated just yet, but if you wanted to, you could piecing together some stick figures, or something of the kind.

In the next post, we're going to be taking another look at rasterization. - only this time, we're going to be setting up two more classes to be used within our engine: triangles and quadrilaterals. Stay tuned!

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