Miles Rollins-Waterman

Intro to Computer Graphics

Professor Hanocka

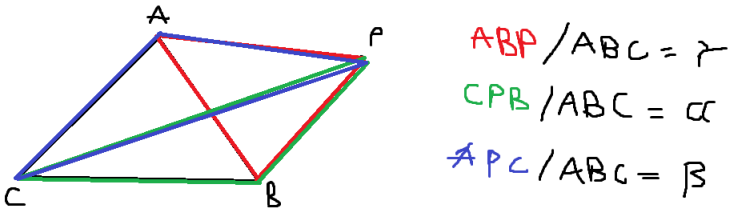
Friday, January 15, 2026

Documentation

TRIANGLE:

For this part of the project, I began by figuring out how to establish a bounding box since that seemed like an easy place to start. In order to find said box, I used a simple brute force solution of iterating through all the x and y coordinates of the triangle, checking each against the current max/min to find the global max/min.

After finding the bounding box, the next step was to create a function which would check if an individual point in the svg viewbox was in a shape read by the svg reader. To do this for triangles, I calculated the barycentric coordinates of the given point using the area of the triangle in question, and the areas of the triangles formed by connecting every pair of vertices on the
original triangle with the given point.



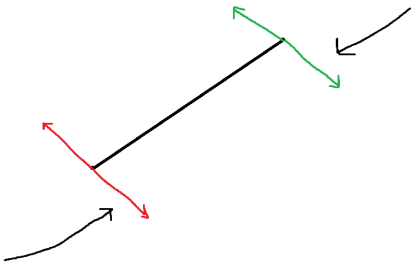
If any of these coordinates are less than 0 (or less than given a certain tolerance!), then the point is not in the triangle. Finally, in order to draw each shape, I iterated through all the svg

viewbox points within the previously established bounding box and checked whether each of them was in the given triangle. This method proved mildly effective, but I was still missing some pixels on the edge. In order to improve my model, I began iterating over image space coordinates (i.e real pixels) instead of svg space points. Then, for each image space coordinate, I would find it's center, and translate that to it's svg viewbox equivalent. This viewbox point is checked for placement in the triangle, and the image space pixel is colored if its center point's viewbox equivalent is contained within the shape.

Method for calculating Barycentric coordinates taken from the course textbook and Wikipedia

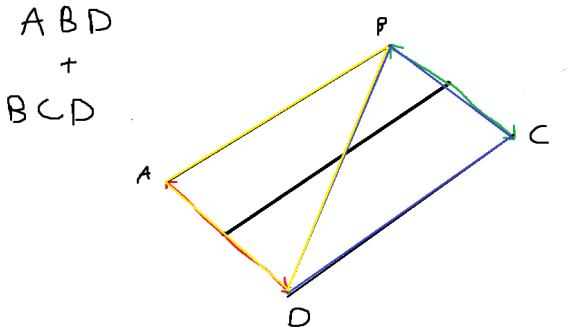
LINE:

I began by finding the bounding box for a line. This itself required first finding the 4 corners of the rectangle the line would become after taking its width into account. To do this, I found the *dx* and *dy* of the line as well as the line length (using the distance formula). From there, I made *dx* and *dy* into unit vectors by dividing them each by line length. Finally, I multiplied each by half the width of the line to find delta *x* and *y* that would result in a perpendicular line exactly half the width of the line, with its origin at the line's end point.After



After finding this delta *x* and *y*, I applied them to the line's original end point coordinates to find the end points of the perpendicular lines, which in turn would be the vertices of the rectangle the line represents (once its width is applied). Finding the bounding box was then as simple as finding the largest and smallest x and y values among the rectangle's vertices, similar to what I did with the triangle.

To check if a point was in/one the line, I decided to represent the line as two adjacent triangles.



After selecting two specific grouping of vertices from the rectangle, such that each group of vertices formed a triangle as shown above, I fed these vertex groups to the function I previously constructed to check if a point was in a triangle. If that function returned true for either constructed triangle, the point was on/in the line.

Method for constructing a rectangle from a line taken from Stack Overflow

CIRCLE:

For this part of the project I began again with the bounding box, which was simple. Given the center point of the circle, the max and min x values can be found by adding and subtracting (respectively) the radius of the circle from the x-value of the circle's center. The max and min y values are found the same way.

Checking if a point was in the circle was similarly trivial. For any point, if the distance between that point and the center of the circle is greater than the radius of the circle, the point cannot lie in the circle.

ANTI-ALIASING:

This part of the project was not difficult since I had already made the switch to iterating through image space coordinates. For each image space pixel I created a program that found the svg space points of a 3x3 grid beginning with the pixel's image space coordinate, the top left hand corner of the pixel itself. After finding this sample, I checked each svg space point in the 3x3 grid for coverage by the given shape. I then divided this number by 9 to find the percentage or ratio of coverage for that pixel. That ratio determined how much of the shape's color would

be applied to that pixel, and then that color was blended with anything already at that pixel in the image.