Michael McQuade

Zhigang Deng

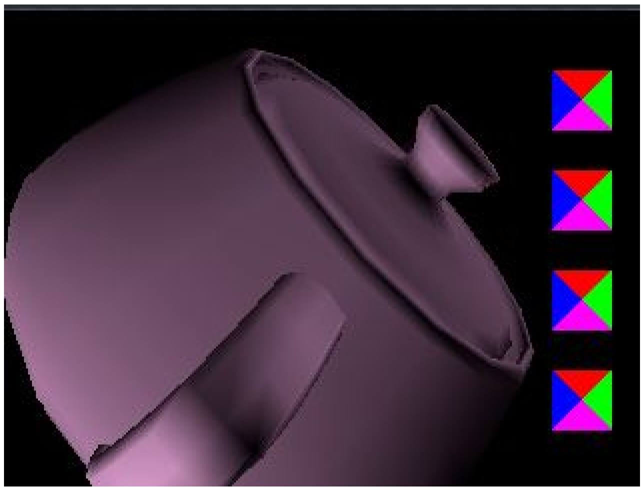
Computer Graphics

9 March 2021

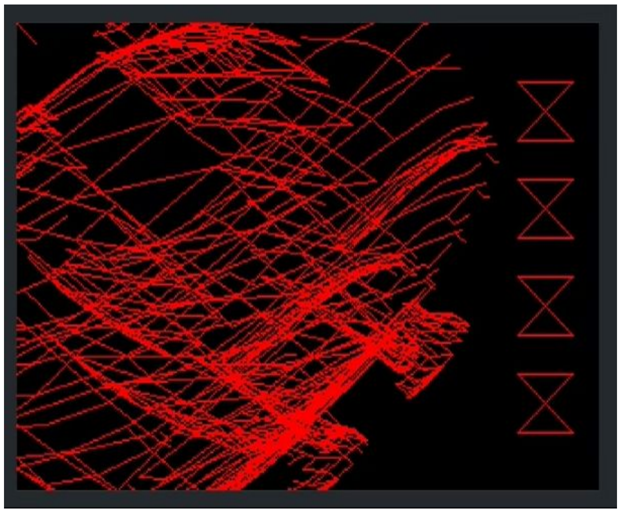
Rasterization

The topic of this report is rasterization, or converting vectors into their graphical representation. It is common to describe polygons like triangles using the vectors that make up the triangle. In the case of a triangle, if you know three points, anyone can connect the lines and complete the triangle. This isn't any different for a computer algorithm. This assignment explores that topic, and the steps needed to convert from a series of vectors representing a triangle into the triangle drawn on the screen, and eventually, the series of triangles that make up a three-dimensional mesh.

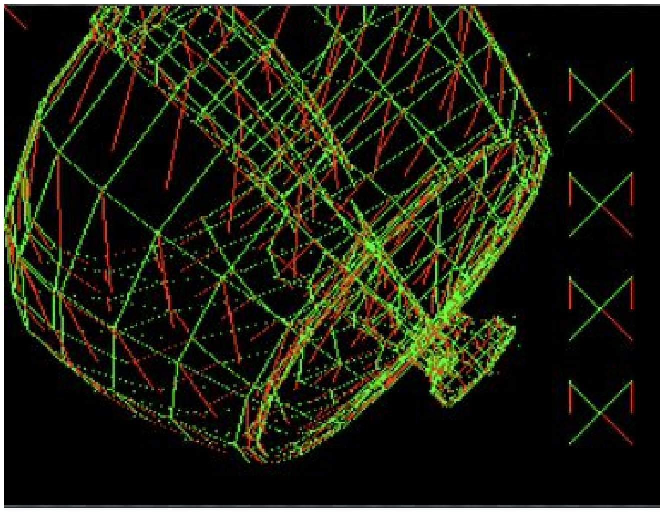
In the previous assignment, the task was to draw a series of rectangles given two opposing corners. In this assignment, the input file consists of a series of vertices, where each group of three vertices make up a triangle. The output of the program is an image of a cup that those triangles represented. The final result should look like the image below:


Example output

Before trying to draw a triangle, or a mesh, the first, maybe obvious approach is to draw a line. If a triangle is made of a point P, Q, and S, then if you draw lines from P to Q, P to S, and S to Q, you will have the outline of the triangle. This is a great first step. Now, how does one draw a line? The first pass at drawing a line didn't work out too well, because of problems when lines overlap or they are too steep, it's necessary to cut the line in half and render from each side, see image below for first failed attempt. Fortunately, Bressenham's line algorithm has already solved this problem, and implementing it within C++ is relatively simple. Implementing the line drawing algorithm alone will give an output as such:



First attempt at drawing lines

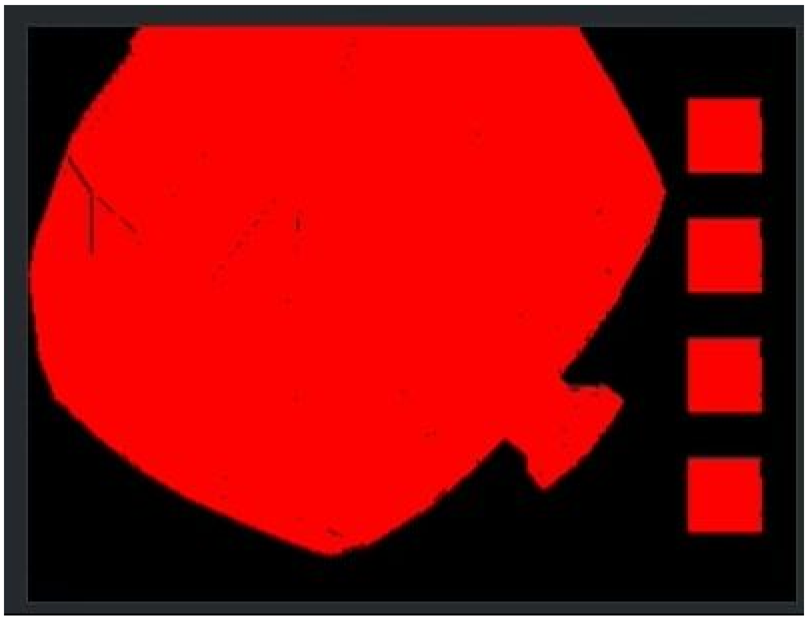


Drawing the cup with Bressenham's Algorithm

However, after implementing this algorithm, a new problem will present itself, filling the triangles. It turns out, drawing a rectangle is much easier than drawing a triangle. This is because by knowing the corners of a rectangle, you automatically know all of the points that lie within the rectangle. This same logic can't be used with triangles. However, there exists a mathematical way to detect if a point lies within a triangle. Enter, barycentric coordinates. In barycentric coordinates, the 3 vertices that make up a triangle have the coordinates of (1,0,0), (0,1,0), and

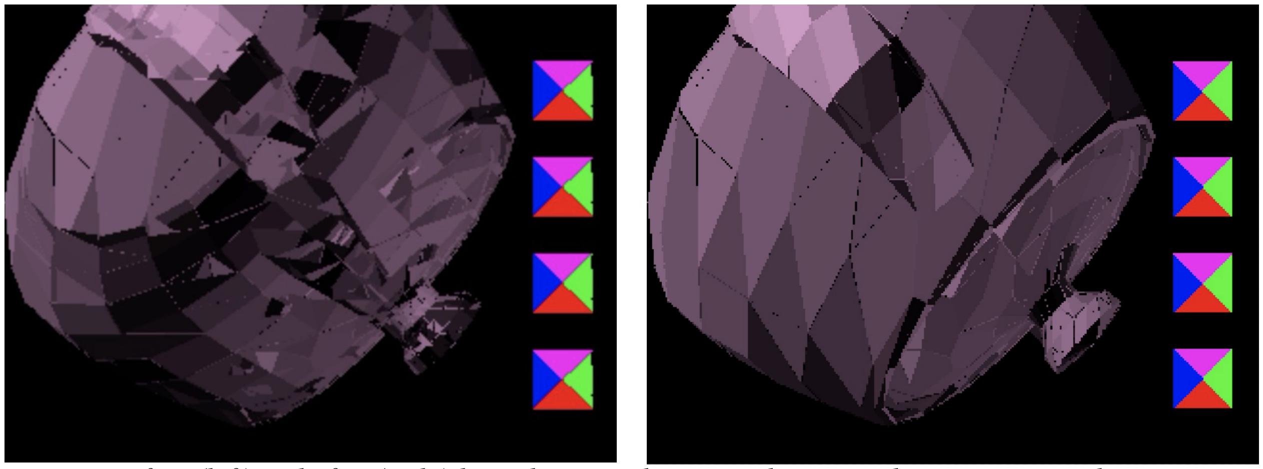
(0,0,1). In this system, it becomes trivial to know if a point is within the triangle or not. Any point that has a negative coordinate is not a part of the triangle.

Considering this fact, a simple way to fill a triangle might be to scan the image line by line, checking if that point lies within a triangle. If it does, figure out the color it should be, and paint it, do this for every point and every triangle. While this line by line scanning over the entire image is possible, it can quickly turn into millions of operations. Instead of scanning over the entire image, it's better to limit the drawing to the area that probably has a triangle in it. In order to do this, it's effective to create a list of all of the triangles, and sort the list by their y-values. This way, the triangles at the top of the screen are processed first, and it isn't required to iterate over the image more than once.


Filling the triangles to color the teapot

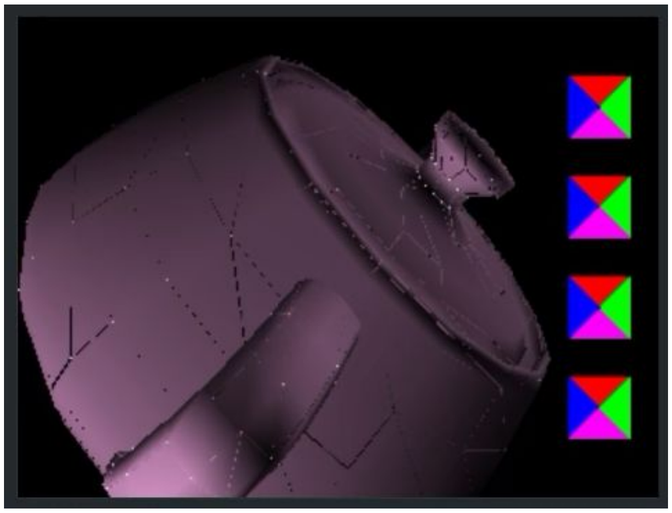
After creating the list of triangles and sorting it, the speed of the application can be further improved by calculating the bounds of the triangle and then only iterating over those bounds. As mentioned earlier, it is harder to iterate over a triangle than a rectangle, so in this case, creating rectangular bounds around the triangle and then checking each point inside of that rectangle to see if the point lies within the triangle is an efficient way to rasterize the triangles.

While doing this, each point is being converted into barycentric coordinates. This is very convenient because these barycentric coordinates can effectively be used as weights for the z-buffer values and color values in addition to identifying which points are in the triangle. In the case of the Z-values, it is sufficient to just multiply the weights against each vertex and divide by the sum of the weights to get a z-value for the point. For the colors, each vertex's color value should be multiplied by the coordinate values in order. That is, red values should be multiplied against the barycentric coordinate X, green with Y, and blue with Z, then the values should be divided by the sum of the coordinates to normalize the values.


Before (left) and after (right) linearly interpolating z-values using barycentric coordinates

Linear interpolation is the idea of blending two values together over the length of a line. In math libraries, there is often a "lerp" function which given a value *t* (a step value), it will step over the line and give the expected value if a line had been drawn between those two points. In colors, this means that interpolating half way between purple and red would give you purple. In the case of these coordinate values, their use as weights effectively linearly interpolates the colors and z values. It's not quite the distances from those points, but it is similar. The benefit of using barycentric coordinates over trying to figure out a weighting based on the distance value is that when a point lies on an edge, the values only come from the two vertices that the point lies between, which is a natural expectation of linear interpolation.

After applying the above methods, the results of implementing rasterization using scanline and barycentric coordinates for blending is the following teapot:



Resources utilized and referenced:

Excellent online tool demonstrating barycentric coordinates
https://www.geogebra.org/m/ZuvmPjmy

Converting to barycentric coordinates
https://en.wikipedia.org/wiki/Barycentric_coordinate_system#Conversion_between_barycentric_and_Cartesian_coordinates

Scanline rendering https://en.wikipedia.org/wiki/Scanline_rendering

Presentation on triangle filling and interpolation
https://www.rose-hulman.edu/class/cs/csse351/m10/triangle_fill.pdf

Bressenham's line algorithm https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm