CS 184: Computer Graphics and Imaging, Spring 2023

Project 1: Rasterizer

Matthew Sun

Overview

In this project, I was tasked to build a rasterizer, which is a program that renders images from .svg files for which the fundamental polygon is the triangle. Three functionalities were utilized to complete the given tasks, namely supersampling, barycentric coordinate interpolation (BCI), and texture mapping. Supersampling is essential to the process of anti-aliasing, which removes the appearance of jagged edges from rendering. BCI allows for gradient coloring. Texture mapping utilizes mipmaps and bilinear interpolation to apply an anti-aliased texture to a scene. Within this write-up, I will summarize how I implemented this rasterizer and show the results demonstrating each of these three techniques. The project was interesting and gave me much insight into how my computer draws the shapes I see when I watch videos or play games, as well as the amount of work that goes into creating such a pipeline.

Section I: Rasterization

Part 1: Rasterizing single-color triangles

This function seeks to build a triangle and render it correctly. The following is a detailed, step-by-step explanation of my implementation.

1. Create a unit vector z, which is a vector that points directly in the z-direction.
2. Build a triangle using the given coordinates and Vector3D structures.
    a. The reason 3-dimensional vectors were used is because the property of vector cross-products and the right-hand rule allows for the verification of the orientation of the triangle.
3. Set the enumeration order of the vertices, which ensures that they will always be enumerated in the clockwise order.
    a. This was done by swapping two points if they didn't have the desired orientation as determined in 2.a.
4. Build the edges by taking the difference between each pair of vertices.
5. Create a set of normals that point into the triangle at each vertex (used for the line test).
6. Define the bounds of each triangle by checking the minimum and maximum x- and y-coordinates for each triangle.
7. Using a nested for-loop, iterate over each point. For each point:
    a. Perform a line test to check if the point is within the triangle (or is on an edge).
        i. Using the line test, a technique described in lecture, is as if not more efficient than checking each pixel in the bound box using arithmetic.
    b. If it is, fill in the point with the specified color. Otherwise, do nothing.

The following image is the result of drawing the provided .svg file using this rasterization function. As is clearly visible in the image, there are jagged edges, but the triangles have been filled in, as well as the edges.

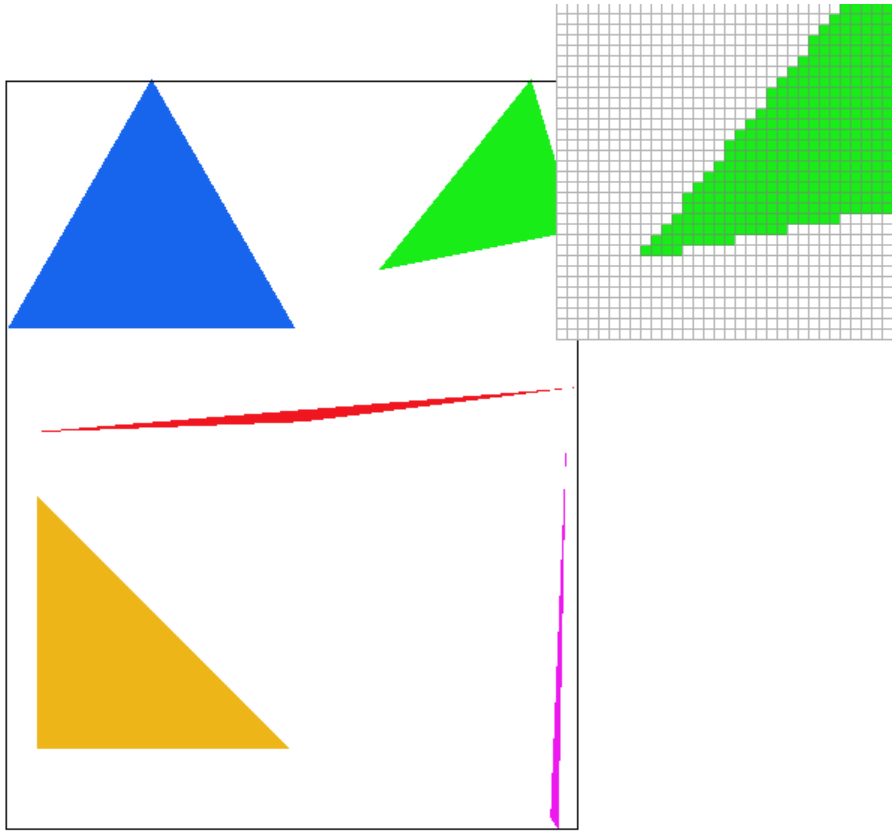

Figure 1. A render of 5 differently colored and differently shaped filled triangles, with the pixel inspector honed on a vertex. The edges can be seen jagged and rough, which will be resolved in Part 2.

Part 2: Antialiasing triangles

Supersampling allows the mapping of a higher-quality image to a lower-quality image by averaging pixel color values. This is the baseline of anti-aliasing. Such a process is described as such. Subdivide the coordinate increment between each pixel by the specified sample rate. For each subpixel, use the line test to check if the coordinate is within the triangle. The final color of the actual pixel is the average of the colors of the subpixels, which are determined using the line test.

The project spec suggested that a data structure for storing the higher-resolution image be created and downsampled when resolving to the framebuffer. While there is a disadvantage in terms of memory usage and write operation costs with the inclusion of this supersampling buffer as opposed to simply performing on-the-fly computations, having the buffer proved useful in the later parts of the project. However, using the buffer required the modification of several functions that resized the buffer and allocated memory based on the sample rate parameter. Additionally, a modification to the resolve_to_framebuffer() method was required to perform the averaging for the supersampling.

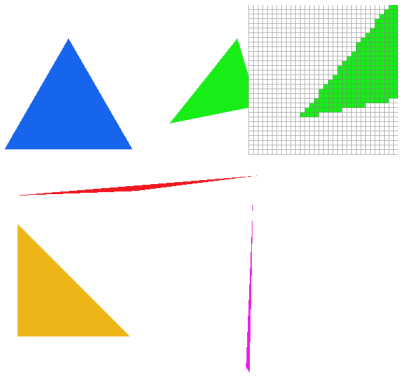

Figure 2.1. Sample Rate 1. Same as if anti-aliasing had not been used.

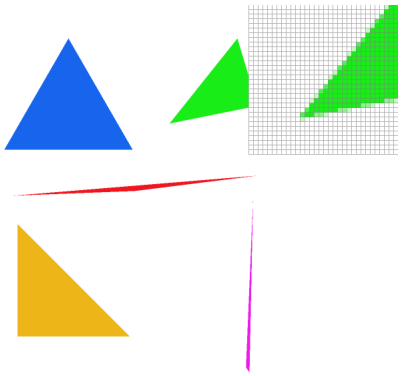

Figure 2.2. Sample Rate 4. The jagged edges are noticably softer in the larger image, and the close-up displays.

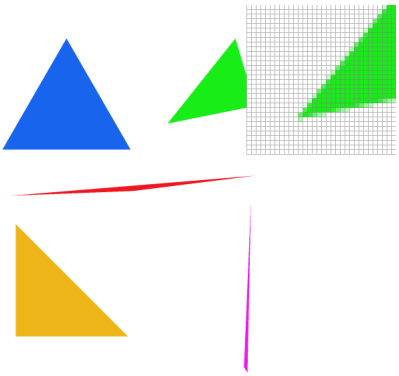

Figure 2.3. Sample Rate 9. While the change is less noticable, because the sample rate is higher, aliasing is further reduced.

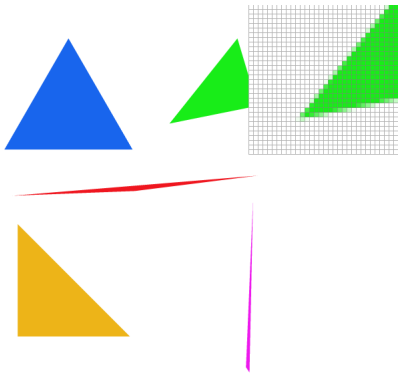

Figure 2.4. Sample Rate 16. At this point, the change is even less noticable. However, on the top side of the green triangle, some difference can still be seen.

Figure 2. The table shows how supersampling reduces the jagged edges, which allows for a smoother rendering. The pixelation decreases with each increase in sampling rate

Part 3: Transforms

I redecorated the cube man in Berkeley colors and made him do a classic flexing pose. I think he looks pretty cool.

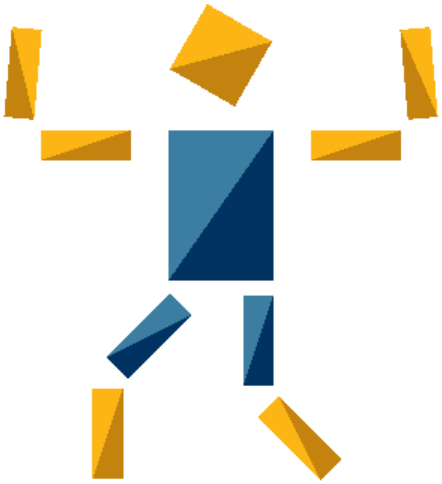

Figure 3. Cube man doing his cube things.

Section II: Sampling

Part 4: Barycentric coordinates

Barycentric coordinates are a system defined through three reference values (at least, in this use case). These coordinates allow for the definition of a weighted sum of the reference values. Specifically, moving along the alpha, beta, and gamma axes exclusively of such a coordinate system has the direct effect of assigning greater weight values to the reference values of A, B, and C, respectively. Since the sum of the barycentric coordinates are constrained to be 1, there remain two degrees of freedom. Thus, this system is especially useful for interpolating the values of the fill of a triangle in a particular feature space (RGB color space, in this case).

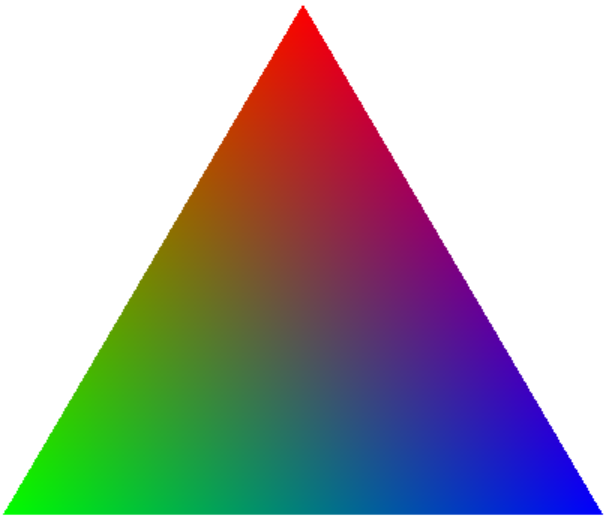

Figure 4.1. A triangle with each vertex colored differently. The colors that fill the triangle were calculated using barycentric coordinates based on the colors on each vertex, namely red (1, 0, 0), green (0, 1, 0), and blue (0, 0, 1).

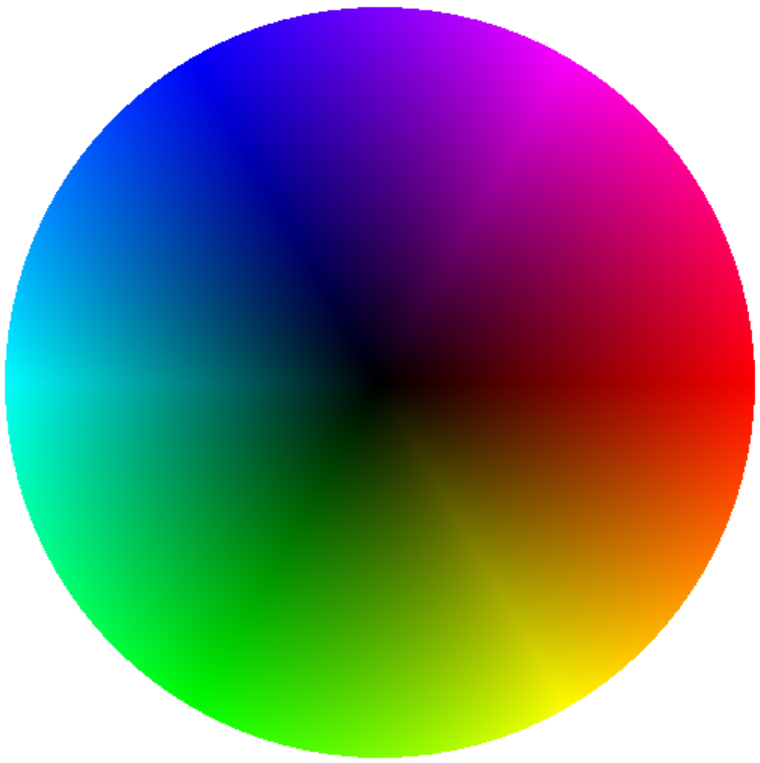

Figure 4.2. The color wheel is created by fanning several acute isosceles triangles radially about the center. Barycentric coordinates are used to smoothly interpolate the colors between the vertices of each triangle, resulting in the blended

colors seen in the image.

Part 5: "Pixel sampling" for texture mapping

For texture mapping, the challenge lies primarily in the mapping of texture to screenspace. This is due to the fact that the map between the texturespace and screenspace can be highly non-linear. Pixel sampling methods like the two used in the project (nearest-neighbor and bilinear) are attempts at tackling this challenge.

Nearest-neighbor pixel sampling for a given normalized UV coordinate can be implemented by simply rescaling the normalized coordinate by the texture map dimensions and identifying the closest texel. Bilinear sampling can be implemented by locating the four nearest texels and in the texture map and performing a series of LERP operations over the horizontal and vertical axes.

The figure below showcases the differences between these two techniques. The zoomed view shows the advantages of each rather clearly. Supersampling allows for more fine selection of colors, allowing for smoother coloration. The largest differences between the two methods occurs when a lower-resolution texture is mapped onto a higher-resolution screenspace due to nearest-neighbor mapping causes a texel to be mapped to multiple pixels.

Figure 5.1. Nearest-neighbor, sample rate 1.

Figure 5.2. Bilinear, sample rate 1.

Figure 5.3. Nearest-neighbor, sample rate 16.

Figure 5.4. Bilinear, sample rate 16.

Figure 5. This chart clearly illustrates the differences between nearest-neighbor and bilinear sampling. While nearest-neighbor most faithfully replicates the texture map, it tends to result in an oversharp image. Bilinear tends to produce a more natural/smooth image.

Part 6: "Level sampling" with mipmaps for texture mapping

Level sampling becomes rather important when dealing with mipmaps and attempts to resolve issues that stem from minification. This refers to the single pixel in screen space spans several texels in texture space. This in turn creates an issue where translating and mapping the texture into a coherent visual onto screenspace. If the texturespace has a sufficiently high frequency of content in a minified region, moving by a single pixel of screenspace could result in the selection of a very different texturespace pixel with a very different color.

A mipmap gathers a collection of downsampled, texel-averaged textures from which coherent texels can be selected and returned to screenspace pixels. This process is called level sampling, and it can be implemented by using the columns of the mipmap's Jacobian matrix, in which the magnitudes of the column vectors provide an approximate metric representing the texel footprint. Using these metrics, the mipmap level best suited for the coloration of the screenspace based on map-induced regional minification can be calculated.

Figure 6.1. Zero, nearest-neighbor.

Figure 6.2. Zero, bilinear.

Figure 6.3. Nearest

Figure 6.4. Nearest, bilinear.

Figure 6. This chart clearly illustrates the differences between level zero mapping and nearest level mapping. Zero is a raw mapping of the texture. Linear causes a strong blending of the texture and results in colors that may or may not be too smooth, but result in better visual coherence.

Website

Follow this link to reach the gitpage hosting this project write-up.