CS184 Project1 Writeup

Jason Du 3040815573

February 2025

Abstract

This is the overview of the entire project 1. In this project, I implemented a simple rasterizer and added supersampling, pixel sampling, and level sampling techniques to make it more versatile and perform better. Most problems I encountered were coding issues that were solved after understanding the given code better. (P.S.: I use LaTeX to finish this writeup so there may be some misalignment between pictures and texts. I will switch to markdown or HTML for future work.)

1 Drawing Single-Color Triangles

1.1 How do you rasterize triangles?

I first find the triangle's bounding box. Then, to optimize the algorithm, I calculate some factors that are unchanged in the future loop area. All we need to do now is loop through the bounding box area, adjust the detection point to the middle of the pixel by adding 0.5 to its x and y coordinates, and fill with color if it is inside.

1.2 Why is your algorithm no worse?

My algorithm uses the same technique described in the question, which uses a bounding box to cut down computing time.

1.3 Test4 figure result

See Figure 1. Interesting part: The aliasing just caused a triangle to disintegrate.

1.4 Extra optimization

My optimization is:

1. I use barycentric representation to judge each pixel whether it's inside the triangle or not. This drastically improves the performance because every pixel outside the triangle is not filled to save time.

2. I pre-compute things that will be repeatedly used later and it gains performance. Maybe doing all calculations in integers will be faster compared to using float.

See Figure 2 for speed comparison. (P.S.: I don't have a speed comparison table I hope that's OK.)

2 Antialiasing by Supersampling

2.1 Walk through your supersampling

In this modified rasterization pipeline from task 1, I implemented supersampling by increasing the resolution of the sample buffer and averaging the values to generate the final image.

Instead of directly writing pixel colors to the framebuffer, we maintain a sample buffer, which stores `sample_rate` times more color values per pixel. First, we calculate the number of sub-pixel samples per pixel, which is the `sqrt(sample_rate) x sqrt(sample_rate)` mentioned in the task description. Then, when we loop over the pixels in the bounding box, we subdivide each pixel into `sqrt(sample_rate) x sqrt(sample_rate)` smaller regions, compute the coordinates of each subpixel sample and test whether they are inside the triangle. Finally, we store the color value to `sample_buffer`.

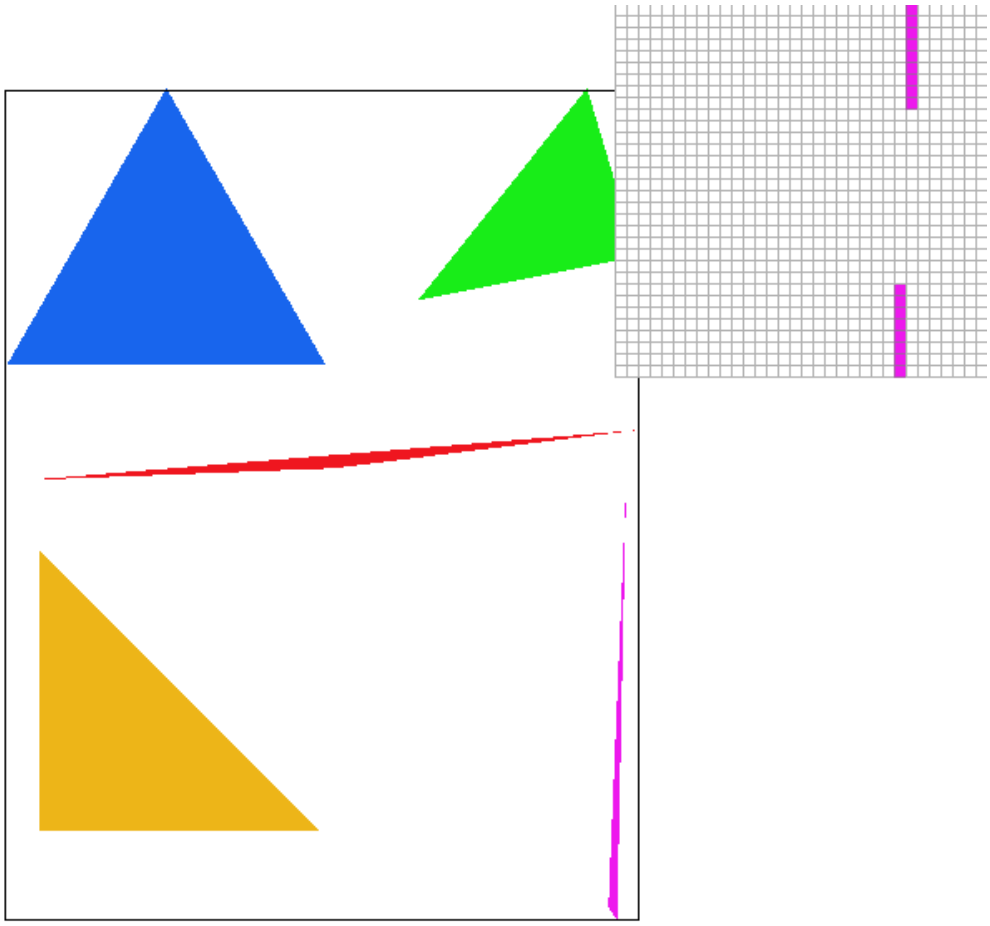
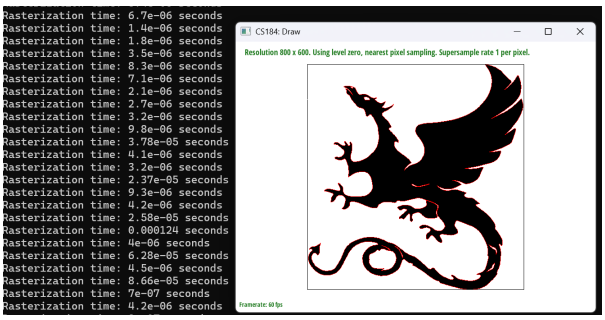
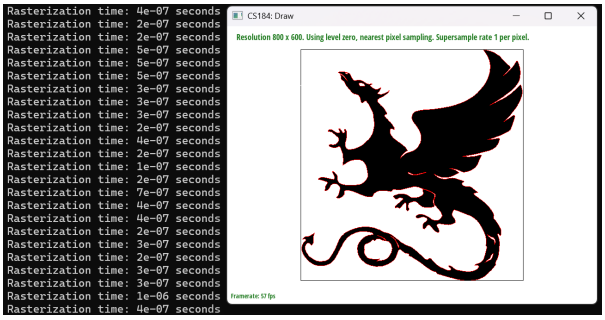


Figure 1: Result of fig4 in Task1



(a) w/ optimization 2



(b) w/o optimization 2

Figure 2: I didn't do the comparison of w/ w/o optimization 1, but w/ optimization 1 it's super fast already.

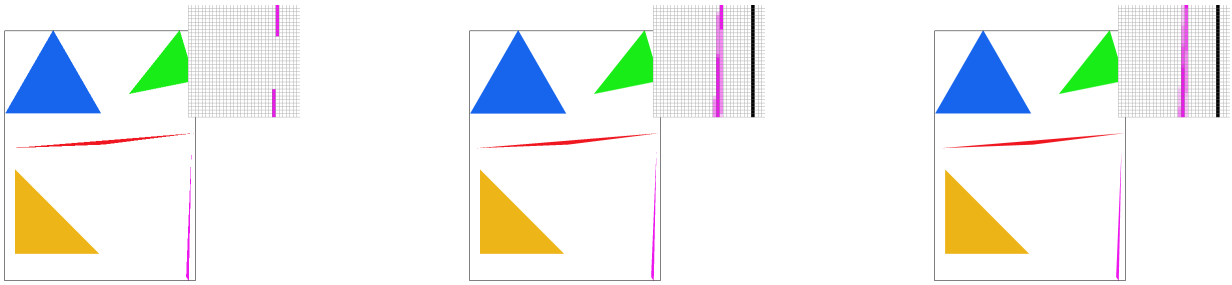
Some changes should also be made to the `resolve_to_framebuffer()` because we rasterized all triangles at supersampled resolution and we have to downsample it to fit framebuffer resolution. So we iterate through each pixel in the framebuffer, average the color values from all subsamples within that pixel, and write the final color to `rgb_framebuffer_target`.

Why is Supersampling Useful? First, it can reduce aliasing by averaging multiple samples, producing smoother edges. Second, it blends colors across edges based on sample coverage so the edge transitions look better. Third, it drastically improves performance with acceptable extra time/computing resources.

Modifications made.

1. Use `sample_buffer` to store multiple color samples per pixel.
2. `rasterize_triangle()` Explained above.
3. `fill_pixel()` Added restrictions to ensure point and line rendering remains correct in the supersampled buffer.
4. `resolve_to_framebuffer()` Explained above.

2.2 Test4 figure result



(a) sample rate 1 (b) sample rate 4 (c) sample rate 16

Figure 3: side-by-side comparison between

For a very skinny corner of a triangle(as demonstrated in the three figures), with a low sampling rate, there's a high chance that the corner of the triangle will not cover any sampling points of a pixel(e.g. go right in between two pixels' sampling points). You get discontinued skinny triangle corners in the first picture(sample rate 1). If you use a higher sampling rate, there's a higher chance for some sampling points to be located inside the triangle, avoiding the problem in low sampling rate scenarios. See Figure 3 for comparison results.

3 Transforms

See Figure 4. I tried to ungroup his skeleton and bend the arms and legs to mimic an infamous actor in China Mainland called Xukun Cai.

4 Barycentric Coordinates

4.1 Explanation

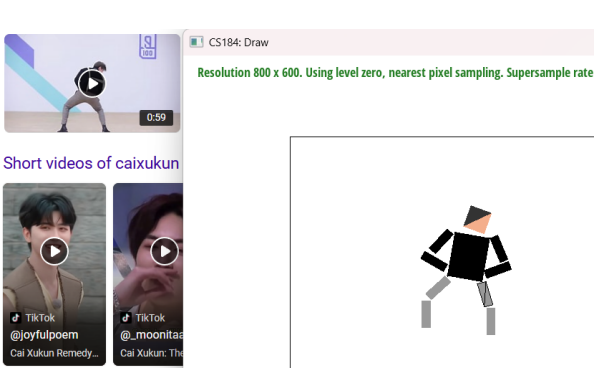
For the triangle result, see Figure 5a. The barycentric coordinate is a way to express the position of a point within a triangle using three values that represent its relative "weight" for the triangle's vertices. It can describe how much "influence" each vertex of the triangle has on the point's position.

$\forall V \in$ triangle $V_AV_BV_C$, point V can be written as:

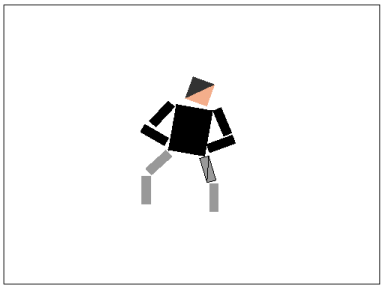
$$V = \alpha V_A + \beta V_B + \gamma V_C \tag{1}$$

where,

$$\alpha + \beta + \gamma = 1 \tag{2}$$



(a) SideBySide picture



(b) My cubeman

Figure 4: Task3 results

To be more specific,

$$\alpha = \frac{S_{VV_BV_C}}{S_{V_AV_BV_C}} \tag{3}$$

$$\beta = \frac{S_{VV_CV_A}}{S_{V_AV_BV_C}} \tag{4}$$

$$\gamma = \frac{S_{VV_AV_B}}{S_{V_AV_BV_C}} \tag{5}$$

4.2 Test7 figure result (Figure 5b)

5 "Pixel sampling" for texture mapping

5.1 Explanation

Pixel sampling is the process of determining the color of a pixel when mapping a texture onto a surface. Since a texture is usually stored as a discrete grid of color values (texels), sampling helps decide how to retrieve and interpolate colors to produce smooth, visually appealing results. It's important because, in texture mapping, we usually meet the problem of texture not aligning perfectly with the texel grid, and we have to use pixel sampling techniques to make sure things are displayed correctly.

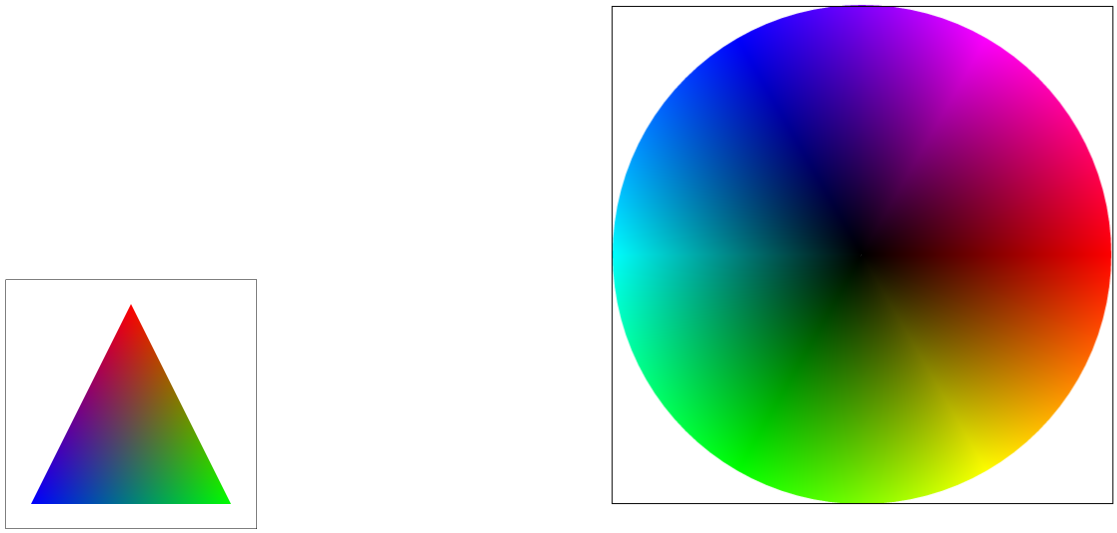
For the code, it's the same with `rasterize_triangle` above(first determines the bounding box of the triangle, next iterates over all pixels within this bounding box, then checks if a pixel is inside the triangle using barycentric coordinates, if the pixel is inside, it interpolates texture coordinates). Then, it uses `SampleParams` struct to store information on different sampling methods and differentials du/dx and du/dy to help with mipmapping.

The nearest-neighbor method simply rounds the (u, v) coordinates to the closest texel. It retrieves the texel at that integer position without any interpolation. Bilinear interpolation takes the four nearest texels and computes a weighted average. Bilinear sampling yields smoother results compared to nearest-neighbor as well as reduced aliasing, but it will be slower.

5.2 Texmap comparison

When at 1 sample per pixel, bilinear sampling clearly beats nearest-neighbor sampling in many places. What I noticed was the breaking in lines due to misalignment in nearest-neighbor sampling, whereas bilinear sampling managed to "connect the lines". See Figure 6 and Figure 7 for details.

Significant differences between the nearest-neighbor and bilinear sampling methods occur in cases where texel-to-pixel misalignment or rapid texture changes are present. For nearest-neighbor, when it produces blocky artifacts



(a) Smoothly blended color triangle

(b) svg/basic/test7.svg result

Figure 5: Task4 barycentric coordinates

because each texel covers multiple pixels without smooth transitions, the bilinear method will create a smooth gradient by interpolating between texels, reducing blockiness. Also, when the texture has sharp patterns, for nearest-neighbor, it just causes abrupt color shifts, making patterns appear jagged, whereas the bilinear method blurs sharp edges slightly and keeps the transition smooth. The reason is that nearest-neighbor chooses one pixel and gets its color, it will, for example, ignore nearby texels of the opposite color, so we see jaggies and blocky artifacts. However, for bilinear sampling, the weighted blending of four texels makes textures appear smoother, which also avoids the sharp transition problem.

6 "Level sampling" with mipmaps for texture mapping

6.1 Explanation

Level sampling is a technique used in texture mapping to determine which mipmap level to sample from when rendering textures. Mipmaps are precomputed, downscaled versions of a texture that help improve rendering performance and reduce aliasing when a large texture is mapped to a small area on the screen. Instead of always sampling from the highest-resolution texture map, level sampling selects an appropriate mipmap level based on how much the texture is shrinking or stretching in screen space.

I followed the hints in the code and implemented/calculated every part of the `SampleParams`. For u, v, I derived them from barycentric coordinates and u0, u1, u2, v0, v1, v2. For du/dx, du/dy and dv/dx, dv/dy, I applied a 1-pixel shift to the original coordinates and recalculated α, β, γ for the new one as well as new barycentric coords. For the mipmap level, I followed this:

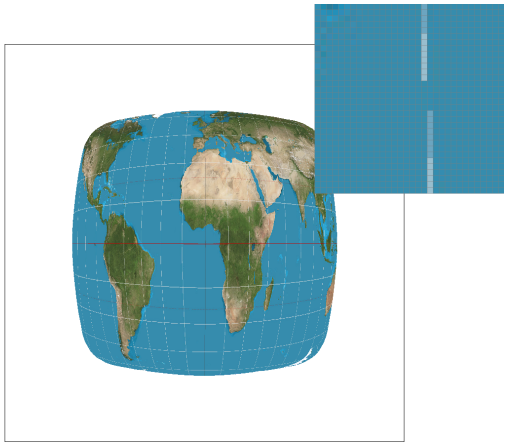
$$D = \log_2 \left(\max \left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2} \right) \right) \tag{6}$$

6.2 Tradeoffs

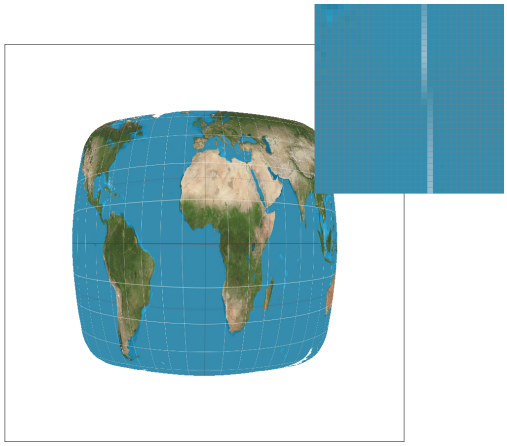
See Table 1.

6.3 My result

The original picture is Figure 8. For results, see Figure 9 and Figure 10.

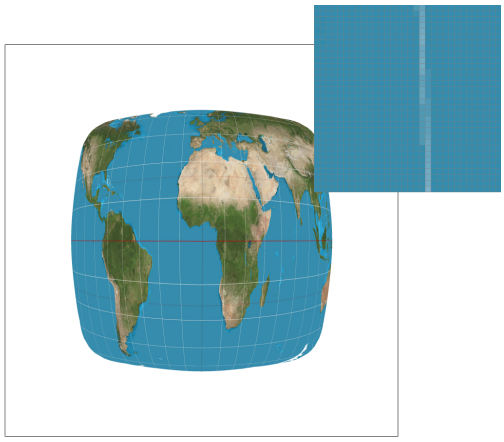


(a) nearest sampling at 1 sample per pixel

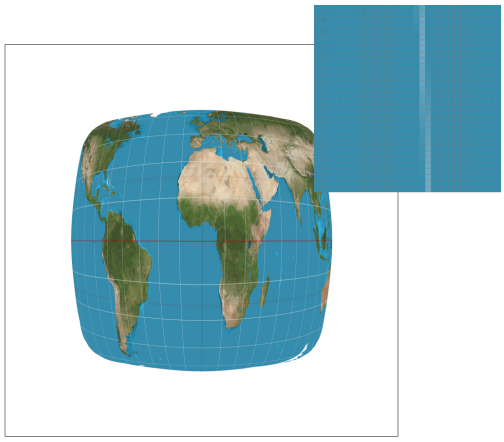


(b) bilinear sampling at 1 sample per pixel

Figure 6: 1 sample per pixel



(a) nearest sampling at 16 samples per pixel



(b) bilinear sampling at 16 samples per pixel

Figure 7: 16 sample per pixel

Technique	Speed	Memory usage	Antialiasing power
Pixel sampling	nearest > bilinear	both low	trilinear > bilinear
Level sampling	$L_{ZERO} > L_{NEAREST} > L_{LINEAR}$	all high	all great
#samples per pixel	SLOW	HIGH	BEST

Table 1: Tradeoffs

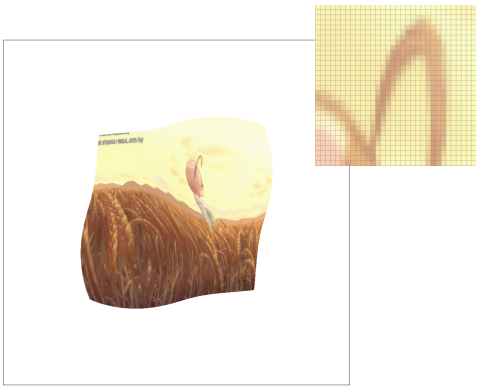
Figure 8: Original picture

(a) L_ZERO and P_NEAREST

(b) L_ZERO and P_LINEAR

Figure 9: Set1

(a) L_NEAREST and P_NEAREST



(b) L_NEAREST and P_LINEAR

Figure 10: Set2