CS 284: Computer Graphics and Imaging, Spring 2023

Project 1: Rasterizer

I-Lun Tsai, Hsuan-Hao Wang, CS284 - where can we find the group number??

Let's goo!

Overview

In this project, we built a rasterizer to draw simple triangles and colored them by interpolating colors from texture images using barycentric coordinates. The final product was a functional vector graphics renderer that could render SVGs (Scalable Vector Graphics) as PNGs. We learned basic techniques such as rasterizing triangles using different sampling rates or different texture mapping methods. Our main takeaway is getting familiar with the techniques used in the rasterization pipeline and understanding the memory and computational tradeoffs between each method.

Section I: Rasterization

Part 1: Rasterizing single-color triangles

To create a rasterized triangle, we implemented the function "rasterize_triangle," which fills pixels within the triangle with the triangle's color. This function iterates through points in the triangle's x and y axes, from the minimum to the maximum values.

We used three of the line tests discussed in class to determine if a point is inside the triangle. For each point, the function checked if the inner product was ≥ 0, or ≤ 0. The point $(x + 0.5, y + 0.5)$ was chosen to calculate the center of the sample. If a point was determined to be inside the triangle, the input color value was filled into the sample buffer's pixel (x, y).

To be more precise, the algorithm performs a single check of every sample within the bounding box because we conducted the point-in-triangle test for every point in the for loops, ensuring that it is no worse than one that checks each sample within the bounding box of the triangle. However, this method may produce alias such as jaggies near the edges of the triangle, as seen in Figure 1 and Figure 2.

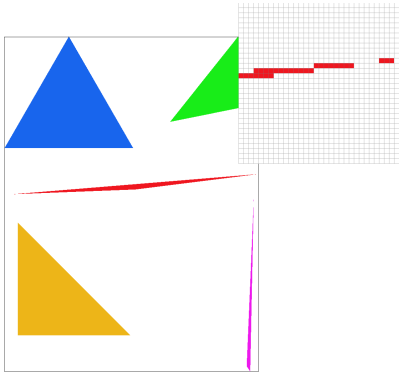

Fig 1. test4.svg

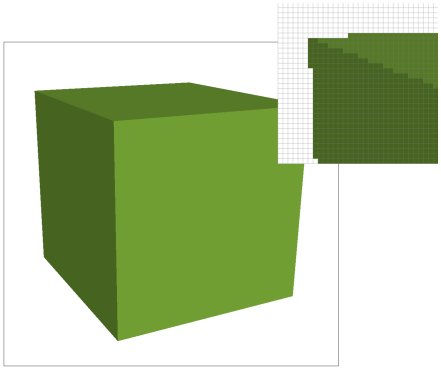

Fig 2. test5.svg

Part 2: Antialiasing triangles

To implement supersampling, I enlarged the sample buffer and updated the rasterization code to account for different sample sizes. When set_sample_rate or set_framebuffer_target is called, the sample buffer is resized by a factor of sample_rate to accommodate the larger supersampling size. In the rasterize_triangle function, I multiplied all coordinates by dilation, where dilation is the square root of sample_rate, making the sample size sample_rate larger than before. The rest of the rasterization process is the same as in task 1.

As the sample buffer is now larger than before, I had to modify the resolve_to_framebuffer function to ensure that the output framebuffer can receive the corresponding pixel value required. Instead of directly filling in the color value in the sample buffer, there is now $dilation \times dilation$ points corresponding to one pixel in the frame buffer. I average the $dilation \times dilation$ color values to obtain the correct color for each pixel.

One of the most important details in supersampling is determining the location of corresponding pixels in the sample buffer. Instead of the original two loops that iterate over width and height, I added two nested for loops from 0 to dilation and calculated all the pixels that contribute to this pixel in the frame buffer.

Supersampling effectively avoids aliasing artifacts like jaggies in Figure 1 and Figure 2. Figures 3 - 6 and Figures 7 - 10 display images with different sample rates, demonstrating how supersampling reduces jaggies. At a sample rate of 1, there are disconnected and unpleasing jaggies in the sharp edges. However, at a sample rate of 16, the edges are smoother and more connected in these high-frequency areas. However, supersampling requires allocating a sample buffer that is sample_rate times larger and performing sample rate times the number of calculations, making the algorithm unaffordable at higher sample_rates.

Fig 3. basic/test4.svg, sample rate = 1

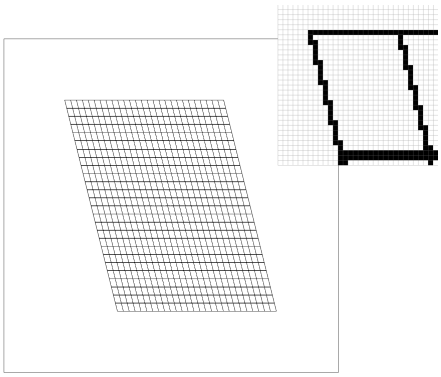

Fig 7. basic/test8.svg, sample rate = 1

Fig 4. basic/test4.svg, sample rate = 4

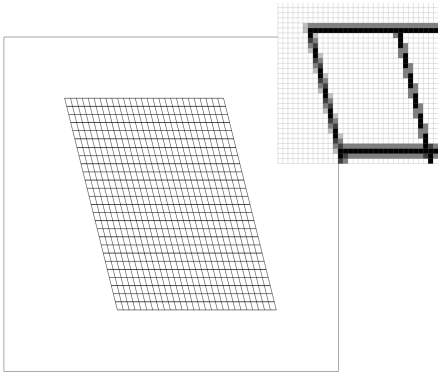

Fig 8. basic/test8.svg, sample rate = 4

Fig 5. basic/test4.svg, sample rate = 9

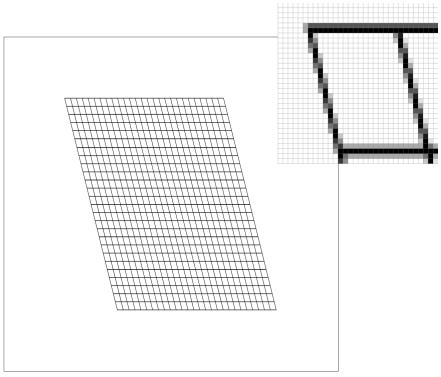

Fig 9. basic/test8.svg, sample rate = 9

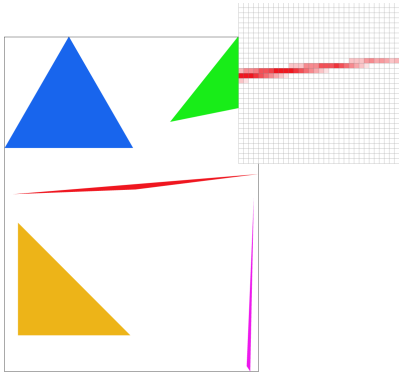

Fig 6. basic/test4.svg, sample rate = 16

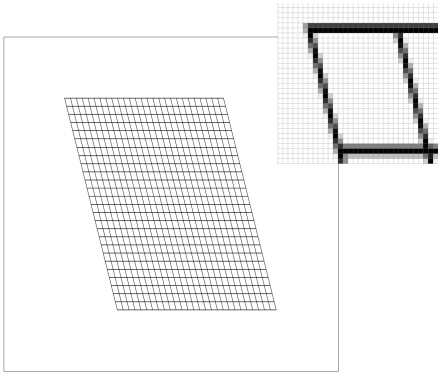

Fig 10. basic/test8.svg, sample rate = 16

Part 3: Transforms

We used the transform, translate, and rotate functions that we implemented to make a YMCABot out of a basic robot. We played around with the (\robot.svg\) file to get familiarized with which code segment corresponded to which body part. The hierarchal structure made it easy for us to move and rotate the robot's entire arm without having to deal with the upper arm and lower arm.

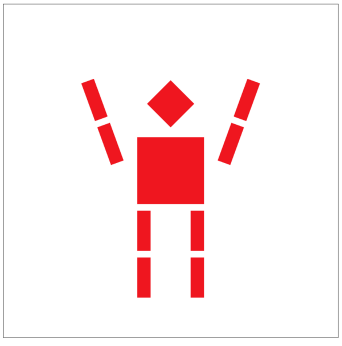

Fig 11. Bot forms Y

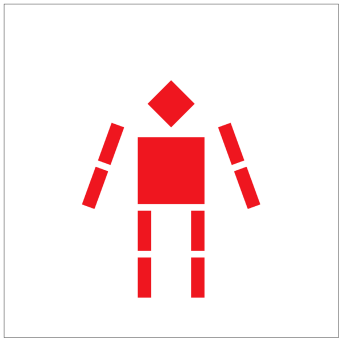

Fig 12. Bot forms M

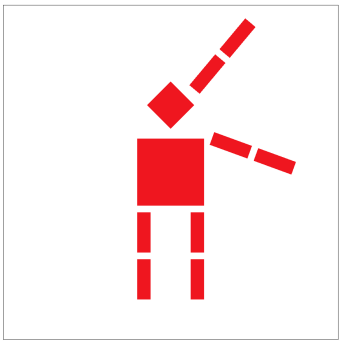

Fig 13. Bot forms C

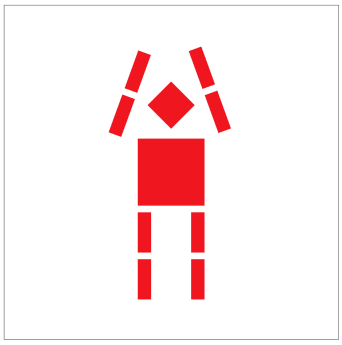

Fig 14. Bot forms A

Section II: Sampling

Part 4: Barycentric coordinates

Barycentric coordinates are a coordinate system for triangles, which specify the position of a point in a triangle with respect to its distances from the triangle's vertices. The barycentric coordinates of a point P in a △ ABC are given by the ratios of the areas of sub-triangles PBC, PCA, and PAB to the area of △ ABC. These ratios are often referred to as (α, β, γ), where

- α = area of △ PBC / area of △ ABC
- β = area of △ PCA / area of △ ABC
- γ = area of △ PAB / area of △ ABC

The barycentric coordinates of P satisfy the condition $\alpha + \beta + \gamma = 1$, and any point within and on the boundaries of the triangle can be represented by a unique set of barycentric coordinates.

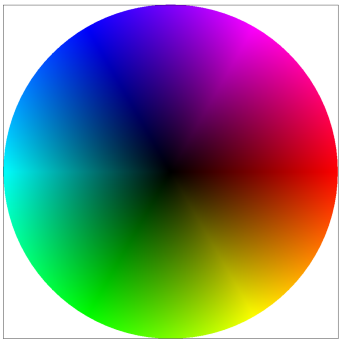

Fig 15. Color spectrum

Barycentric coordinates can also be used to smoothly blend colors in a triangle by performing color interpolation using the barycentric coordinates of each pixel inside the triangle. We describe a step-by-step approach below:

1. Assign a color (with RGB values $0 - 255$) to each vertex of the triangle.
2. For each pixel inside the triangle, compute its barycentric coordinates (α, β, γ) with respect to the triangle.
3. Use the barycentric coordinates to linearly interpolate the colors at each vertex in RGB space to obtain the color at the pixel.

Part 5: "Pixel sampling" for texture mapping

In texture mapping, pixel sampling refers to the process of determining the color or other attributes (e.g. opacity, roughness, geometry) of a surface at a specific point by sampling the texture that is to be applied to that surface.

To determine the texture at a particular point, we first calculated the barycentric coordinates for the pixel of interest, which is used to interpolate its corresponding location in the texture image using the provided corresponding texels of the triangle's vertices. The texel coordinates we obtain from linear interpolation doesn't always correspond with an exact texel in the texture image. Therefore, we use different sampling methods (e.g. nearest neighbor or bilinear interpolation) to determine the texel color. The texel color at that location in the texture image is then sampled to determine the texture color or other attributes to be applied at that point on the object's surface.

Different methods of sampling, such as nearest-pixel or bilinear sampling, can be used to smoothly blend between adjacent texels to create a more similar appearance to the texture image.

- Nearest-pixel sampling takes the color from the nearest texel. In our implementation, we used the round() function to find the nearest integer values of the texel coordinates that were calculated from the pixel's barycentric coordinates.
- Bilinear sampling weight the texel colors according to the distance of the texel of interest from the four nearest texels. The closer a point is to a particular texel, the more weight that texel's color is given in the interpolation. This creates a smooth blend of color between adjacent pixels, resulting in a more similar appearance to the texture image. In our implementation, we used the lerp() function three times for each bilinear sample.

The main difference between pixel sampling and bilinear sampling, as seen in the images above, is how smooth the latitude and longitude lines are. Pixel sampling produces line segments while bilinear sampling blurs the line and produces a continuous line. This is more prominent in the case of lower sampling rate because supersampling also removes visual artifacts to some extent. Bilinear sampling is better in this particular case because it interpolates the colors of neighboring texels, which helps to smooth out any jagged edges or blocky artifacts that would otherwise be visible.

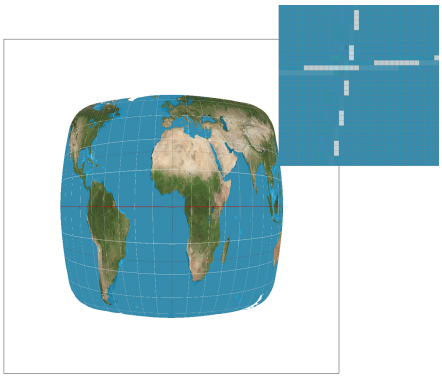

Fig 16. task5_sample1_nearest

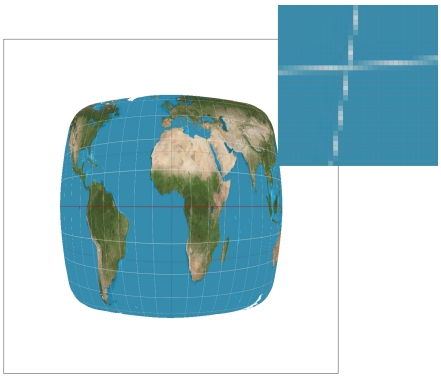

Fig 18. task5_sample1_bilinear

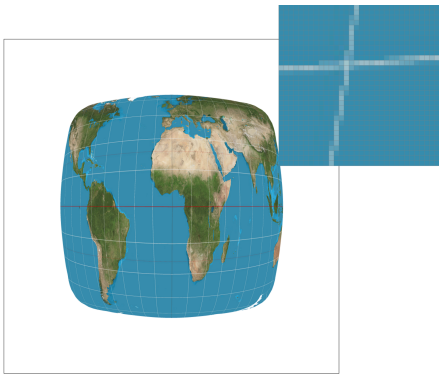

Fig 17. task5_sample16_nearest

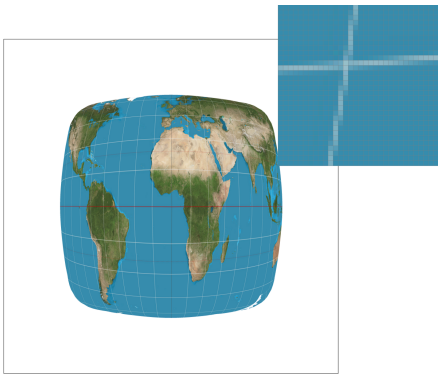

Fig 19. task5_sample16_bilinear

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

Level sampling is a technique for texture mapping that uses lower-resolution versions of a texture (or mipmaps) to speed up rendering, reduce aliasing and blurring. When applying the texture image onto an object, the appropriate mipmap level is selected based on the object's distance to the image frame, and samples the texel colors from that level to compute the color for the pixel. The use of mipmaps allows the us to optimize the texture sampling process by not accessing the full-resolution texture for distant objects.

Our implementation of level sampling can be described in the following step-by-step process:

1. Find the barycentric coordinates of sample ((x, y)) at ($x + 1, y$) and ($x, y + 1$)
2. Use barycentric coordinates to calculate the two vectors

   $$(\frac{du}{dx}, \frac{dv}{dx})$$

   and

   $$(\frac{du}{dy}, \frac{dv}{dy})$$

   and scale them to the dimensions of our texture image
3. Plug values into $D = \log_2 L$ where $L = \max(\sqrt{(\frac{du}{dx})^2 + (\frac{dv}{dx})^2}, \sqrt{(\frac{du}{dy})^2 + (\frac{dv}{dy})^2})$ to find the desired mipmap level

4. If the desired level is in between two mipmap levels, we use linear interpolation to blend the texels from the two closest levels to approximate the color of the desired level.
5. The color is then used to shade the pixel being rendered.

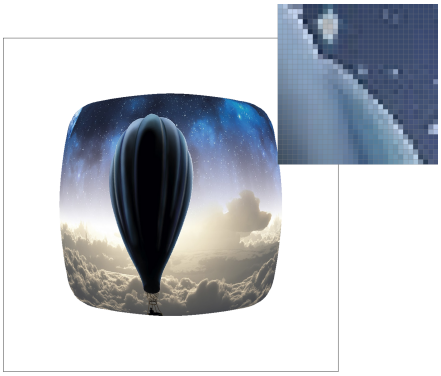

Fig 20. task6_level0_nearest

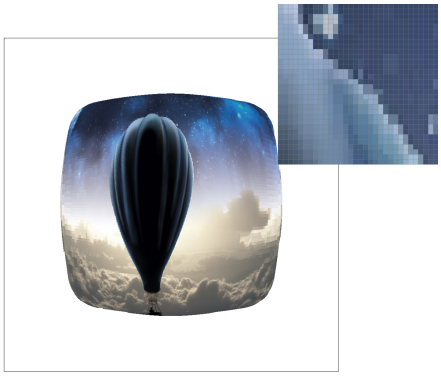

Fig 21. task6_nearest_nearest

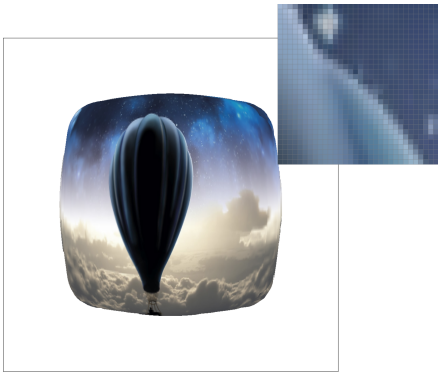

Fig 22. task6_nearest_bilinear

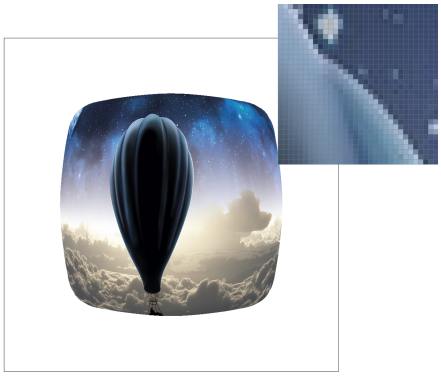

Fig 23. task6_level0_bilinear

Pixel sampling is the fastest technique because we only have to sample color from the nearest texel. It requires the least processing and memory among these three methods, but it can result in undesirable visual artifacts like jaggies and aliasing.

Level sampling with mipmaps requires more memory, specifically $\frac{1}{3}$ memory overhead from the multiple lower-resolution texture images, than pixel sampling. However, it provides improved antialiasing and thus less visual artifacts. However, because more linear interpolations are done, the speed of level sampling is slower than pixel sampling.

Supersampling provides the accurate antialiasing, but it requires the a lot of processing power with bad scalability. It also introduces memory overhead because we use larger sample frames to store the intermediate supersample.

In practice, a combination of techniques may be used to achieve a balance between performance and quality.