CS 184: Computer Graphics and Imaging, Spring 2023

Project 1: Rasterizer

Johannes Fung

Overview

Rasterization is one of the two core paradigms of computer graphics alongside ray tracing. Understanding how complex images can be broken down into countless primitve polygons and then how each polygon made (sometimes) different from the other to form a homogeneous image is both fun and important!

Section I: Rasterization

Part 1: Rasterizing single-color triangles

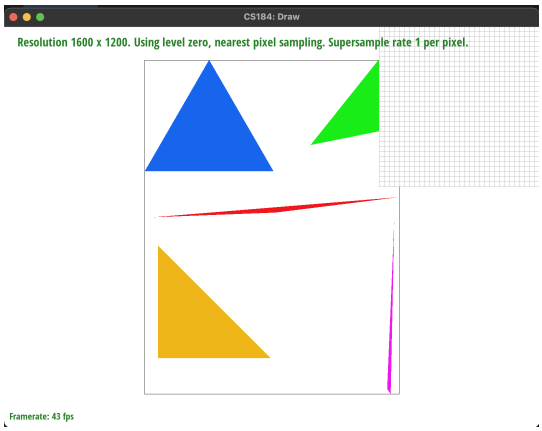
Finding a point is in a triangle or how I learned to stop worrying calculating line equations and love barycentric coordinates.

A simple way of rasterizing triangles onto a screen requires a few things: the points of a triangle and the resolution of the screen space. Using the width and height to form a nested loop, one can iterate across the screen space and determine through sampling how much of a pixel is within the target triangle. My first implementation involved the center point sampling approach and the line equation algorithm presented in class. This worked well for test4.svg and test5.svg, but not for test6.svg which tested whether or not my rasterization approach was winding-order agnostic (spoiler: it wasn't).

This had me stumped for a bit until I came across this article https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage.html which presents a thorough, easy to read breakdown on how line equations worked to detect if a point was in a triangle in the first place. It also goes on to elaborate on the intuition behind barycentric coordinates, which seems like a really neat system to work with when dealing with triangular polygons. Replacing my point checker to use barycentric coordinates solved my issue, so my test .svgs are now nice and happy to be rendered.

Unfortunately, it was still quite slow, since I haven't change the bounds of the nested for loop which were the width and height of the screen. By making the nested for loop iterate over the frame that a triangle forms, the rasterization processed much more quickly!

Here is an example 2x2 gridlike structure using an HTML table. Each `tr` is a row and each `td` is a column in that row. You might find this useful for framing and showing your result images in an organized fashion.



A picture of five triangles being rendered with a sample rate of 1.

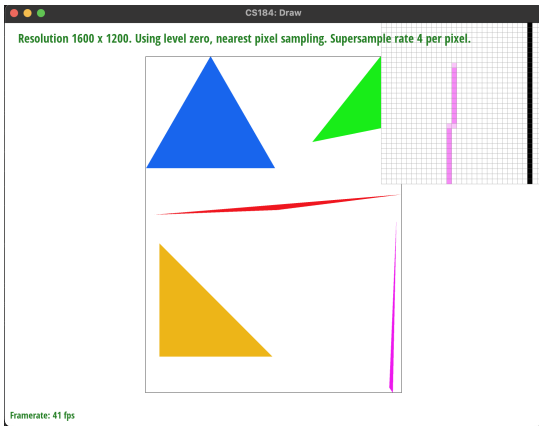
Part 2: Antialiasing triangles

In some of the images, particularly the one with the five different sized triangles, jaggies are quite noticeable. One way to deal with these visual artifacts is through supersampling. To put it simply, supersampling on a high level can be thought of as artificially increasing the resolution over the image through increasing the sample buffer size to allow for more point-in-triangle checking. Then,

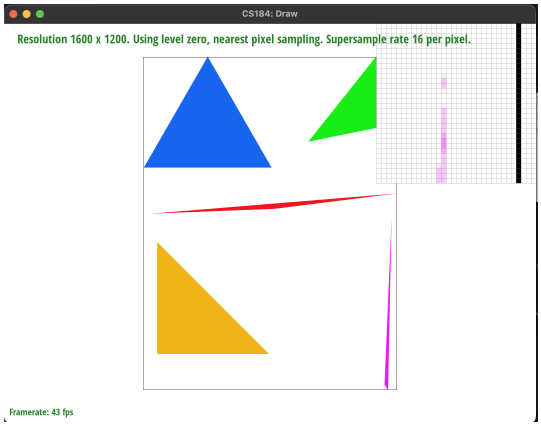
once all the triangles have been sampled, supersampling finishes by averaging the RGB values in perfect square chunks i.e. 1, 4, 9, 16. This averaged value represents the color to display for the (x,y) point that corresponds to the perfect square chunk.

However, because supersampling takes the average of a perfect squared region, this means that the higher the sample rate, the greater the divisive term. As an illustration, imagine singular black point surrounded by white space. If we took the average of the region surrounded by black point, the resulting color value would lean heavily towards a white-greyish color. Consequently, my supersampling algorithm initially practically erased singular points or lines that were surrounded by a dominating color.

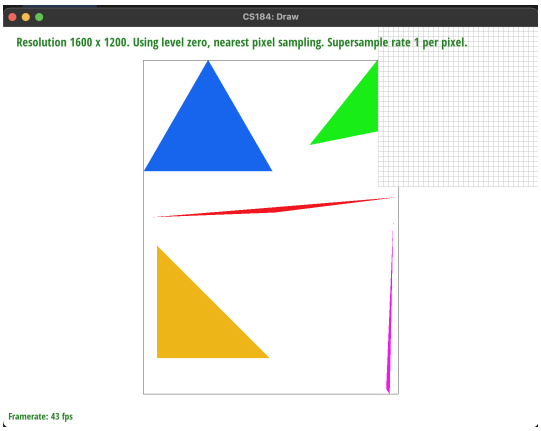
To compensate for this, when I drew lines or rendered points, I made sure that each supersampled block would all have the same color as the original 1x sample.



Sample rate of 4



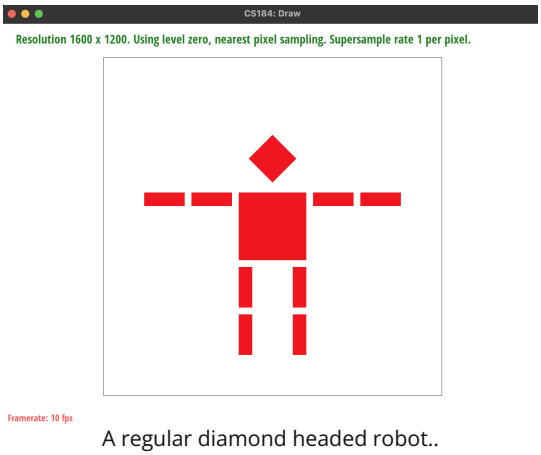
Sample rate of 16.



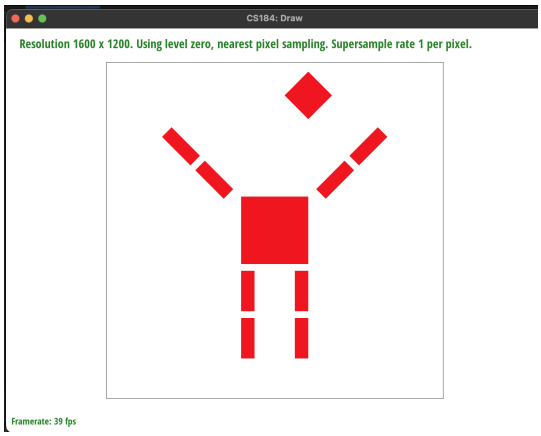
Sample rate of 2

If you zoom in on the thin, pink triangle on the right of each image, you can notice how as the sample rate increases, the triangle looks more contiguous and smooth, which is what supersampling does!

Part 3: Transforms



A regular diamond headed robot..



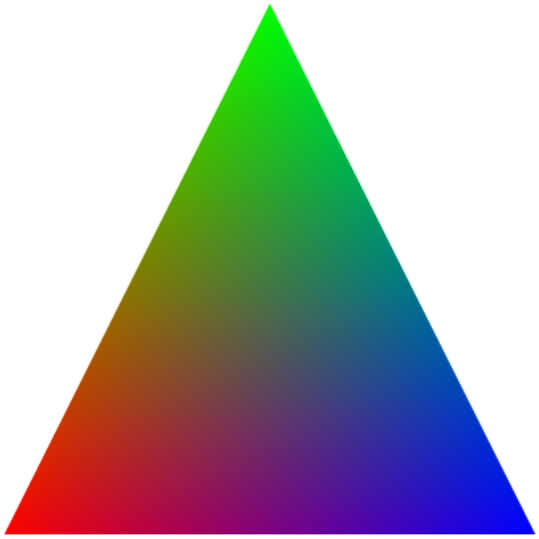
My head!

.

Transformations are a really useful way to manipulate polygons. The image on the left represents an .svg file where the torso, limbs, and head have been rotated to form a t-posing robot. The image on the right has been manipulated such that it has been translated to the top right, while its arms have been translated and rotated up in protest against its head being lobbed against its will

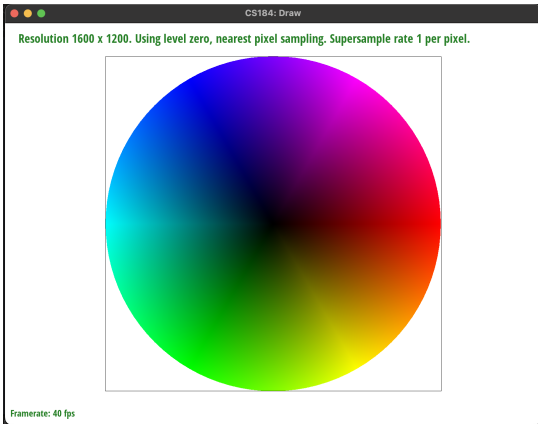
Section II: Sampling

Part 4: Barycentric coordinates



RGB triangle

Barycentric coordinates is a way of determining where a point is in a triangle through using three parameters, alpha, beta, and gamma, which represent the ratio of the point's distance from each of the three lines that make up the triangle. This is incredibly useful because one can use these ratios to linearly interpolate across colors or textures. As a simple example, the triangle above has been colored through barycentric coordinates. Note how each corner of the triangle is dominantly either red, green, or blue. The red corner blends towards a more purple hue as it moves towards the blue corner, and the green corner blends towards a more brownish hue.



A fun color wheel!

Here's a more complicated picture that uses the same principles as barycentric coordinates to render a colorful circle!

Part 5: "Pixel sampling" for texture mapping

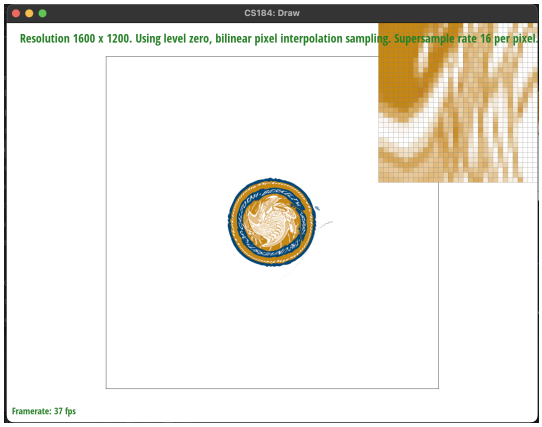
In the previous task, we found a way to linearly interpolate RGB values using barycentric coordinates to weight each color. This is great because it lends itself very nicely to texture mapping: a way to take an existing texture and have it wrap around a collection of contiguous polygons to provide it more visual appeal. Important to note is that the implementation of texture mapping isn't necessarily parameterizing the texture map to fit the mesh, but rather to fit the mesh to the texture.

In pixel sampling, we point sample our triangles to determine which texture to yield. Recall that in point checking, we used barycentric coordinates to determine if a point was in a triangle. However, we need to perform point checking for not just the (x,y) positions, but also the (x+1,y) and (x, y+1) positions. If these three points are found to be in the triangle, we can then fetch the respective texture for a given triangular texture that corresponds to those aforementioned three points using the barycentric parameters to linearly interpolate the texture.

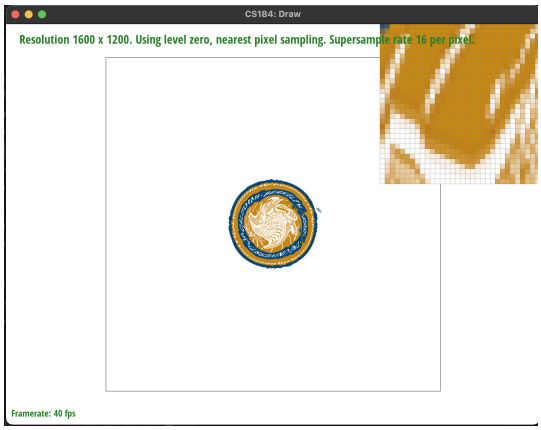
Bilinear sampling with sample rate of 1

Nearest sampling with sample rate of 1

.



Bilinear sampling with sample rate of 16



Nearest sampling with sample rate of 16

.

There are two basic pixel sampling methods: nearest and bilinear. The former simply checks which grid 's center the sample is closest to, while bilinear involves using linear interpolation (also known as lerping) to produce a smoother texture. To sidetrack a little, nearest point sampling is a simple way to implement image magnification while bilinear interpolation can be used to perform image minification.

In the images above of UC Berkeley's Campanile (sample rate of 1), if you zoom in near the top of the clock tower where there are three arches, the nearest sampling approach has obvious artifacts around the left arch, while bilinear sampling has a much smoother rendering of the same problematic area. This is an example of when there can be a large difference between the two, where nearest sampling will generate obvious artifacts since the surrounding problem area vary significantly in hue.

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

Level sampling is a way to reduce aliasing on polygons that are in the background of an image. For instance, in an older video game during the mid 2000s, one might notice that the floor or walls is well defined in contrast to its counterpart further away. To combat these unwanted visual artifacts, we can leverage mipmaps.

Mipmaps are created by sampling the original picture and downsizing the resolution by a half for each level in the mipmap. An example could be where the original resolution is something like 1024x1024. This would be called the 0th level mip map, and every subsequent level would cut the resolution by a factor of 2. One great thing about mipmaps is that they not only reduce aliasing, but can also speed up the texturing process.

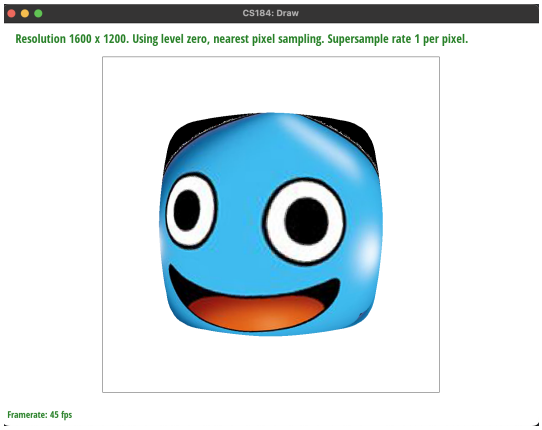
My implementation of level sampling involved calculating the mipmap level as a float. There are different ways to choose which mipmap level to sample from, and in this project I implemented both nearest mipmap level as well as trilinear interpolation. Nearest mipmap level is basically what it sounds like: rounding the float to the nearest integer, while trilinear interpolation is linear interpolation between the texture yielded from the ceil(float) and the floor(float) values.

Between the three sampling techniques, we can break down their tradeoffs as follows.

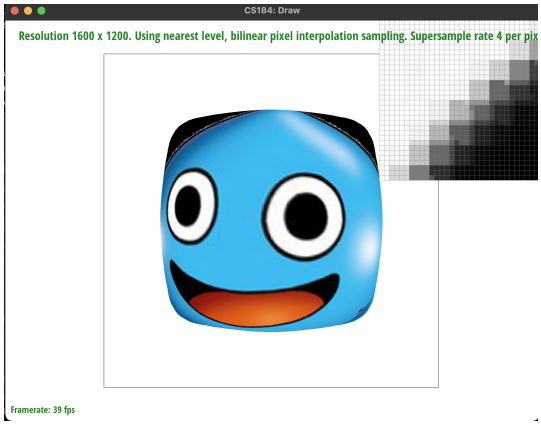
Pixel Sampling: It's the fastest method and requires little memory since it doesn't need to refer to any mipmaps, but it can also yield a lot of jaggies!

Level Sampling: This method is faster than using supersampling, but requires more memory usage to account for the varying level of mipmaps.

Number of samples per pixel: This is basically supersampling, and while it has been shown to render the best quality images, we know that it's computationally expensive and requires a lot of memory on the fly to render high resolution 2D buffers!

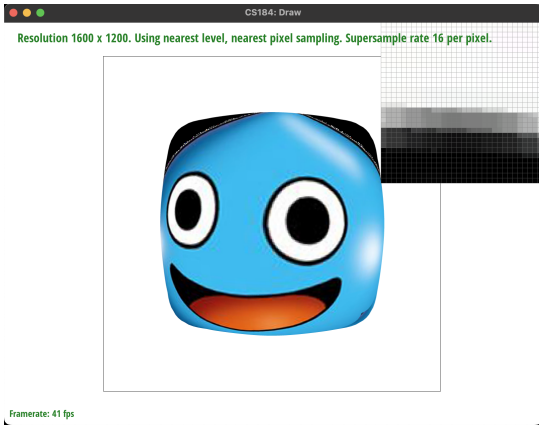


Level 0 nearest point sampling rate of 1

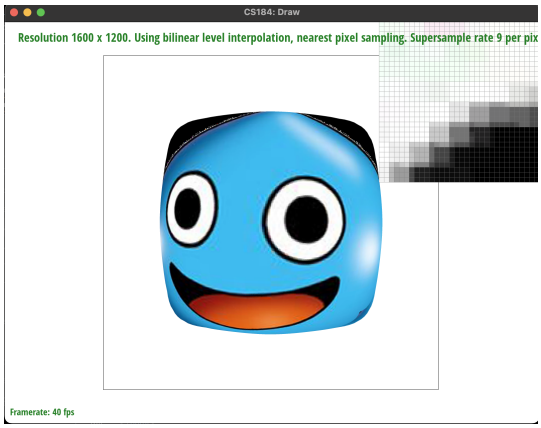


Nearest level bilinear sampling sampling rate of 4

.



Nearest level nearest point sampling rate of 16



Bilinear level nearest point sampling rate of 9

.