CS 184: Computer Graphics and Imaging, Spring 2019

Project 1: Rasterizer

Khadijah Flowers, CS184

Section I: Rasterization

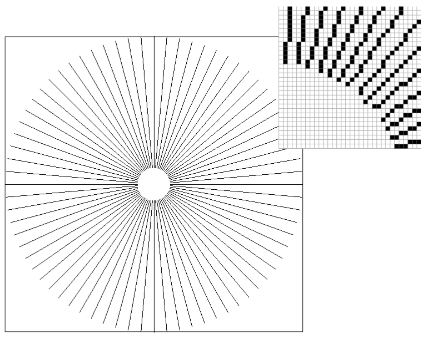
Part 1: Rasterizing single-color triangles

To rasterize a triangle, we choose a sample point consistently among all pixels and check to see whether or not there is a part of the triangle that intersects or overlaps the pixel at that point. To do this, we run a line test using the locations of the vertices in the triangle to determine our boundaries and we compute a normal vector of that line, and take the dot product with the point in question. A positive value tells us that the pixel is on the desired side on a line and running this with all three lines and getting a true value for all of them confirms that the point is in the line.

This point sample implementation places a tight bound on the number of pixels we need to use for sampling because it uses the lines of the triangles to determine what pixels inside of our triangle. Unlike this, *****Bounding Box Sampling produces a larger test margin and there are several different boxes that could be used to bound the triangle, which brings to question which box is the most optimal.

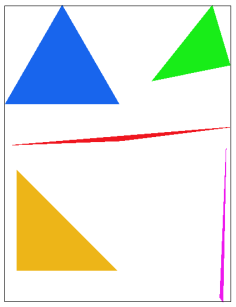
*****Bounding Box Sampling: Bounding the image with a box and sampling the pixels that fall within the box.

This part of the project helped to introduce the idea of taking an image and placing in on a grid of pixels so that it can be represented in a display. It also introduced issues that could arise when trying to determine the color of individual pixels when the frequency or quick changes in the image create aliasing artifacts such as jaggies. With single point sampling, we are not able to clearly capture these sharp changes in the image. Seen below in our Close Up image, we run into jaggies. However, we can fix this with supersampling.

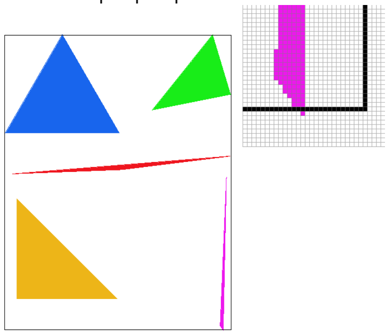

One sample per pixel.

Part 2: Antialiasing triangles

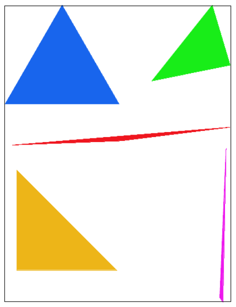
Now, we introduce supersampling. Supersampling is a way to get rid of aliasing artifacts that arise from single point sampling or, in general, sampling our image at too low a frequency such that it doesn't capture the quickly changing and high frequency parts of our image. Supersampling is a way to get rid o these high frequencies and make our approximation better. For every pixel, we take NxN samples and average the color among these pixels, setting the pixel to this averaged color. Then, after we filter, we sample and get a better approximation of what the image looks like abd we reduce the aliasing artifacts left behind when we did a single sample for each pixel. This process of supersampling before sampling causes the image to appear blurred, noticed mostly in high frequency areas of the image or, in other words, places in the image where there is a quick change in color or geometry. In my implementation, to attain this blur, I summed up the RGB color values from the sub pixels within the pixel, averaged the RGB values, and assigned the pixel color to the resulted average. Then, afterwards, I sampled and reduced the amount of jaggies.



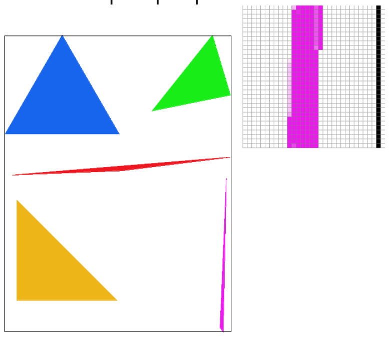
One sample per pixel.



One sample per pixel (Close Up).



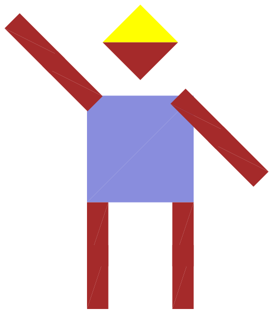
Sixteen samples per pixel.



Sixteen samples per pixel. (Close Up).

The most important difference between what we did before with single point sampling and what we are doing now with supersampling is that we first reduced all of the high frequency areas by supersampling, causing a blur, and sampling afterwards to reduced jaggies.

Part 3: Transforms

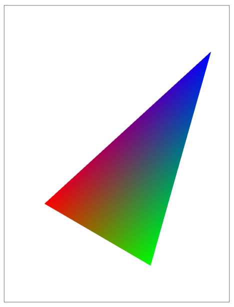


Casual Berkeley student.

This is meant to be a rendition of a mild-mannered, waving Berkeley student. I did this by using rotations to move the arm and I experimented with a few (x, y) coordinates until I was satisfied with the positioning of the limbs. The current rotation of the head was useful and made it easy to give this Berkeley student a snazzy hat.

Section II: Sampling

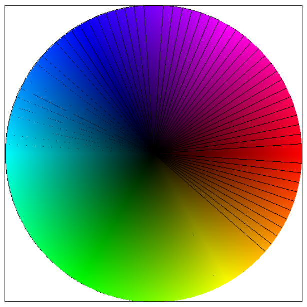
Part 4: Barycentric coordinates



Example/Visualization of barycentric coordinates.

Barycentric coordinates are a tuple of three numbers (alpha, beta, gamma) that correspond to an **(x, y)** pair and these coordinates map this pair to a color within a color triangle. These coordinates are determined by mapping **(x, y)** onto a color triangle and looking at the distance from this point to all vertices of the triangle to determine the ultimate color of the point. If we imagine that we have two xy-planes, one with (x, y) points and another with a triangle meshed texture map, we can better understand what is happening. We take a point from the normal xy-plane and look for the corresponding point in the texture's plane. It will fall into one of the triangles in the mesh. Each vertex in the triangle is associated with an absolute color (nt necessarily Red, Green, or Blue) and the barycentric coordinates of the point are computed by looking at the distance from the point to all lines in the triangle. We then take a *****linear combination of the vertices and the barycentric coordinates to get the color.

*****If A, B, and C are the vertices of the triangle, then this linear combination is computed using the formula: (alpha)*A + (beta)*B + (gamma)*C



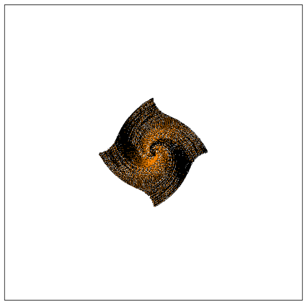
Color Wheel generated by algorithm.

Part 5: "Pixel sampling" for texture mapping

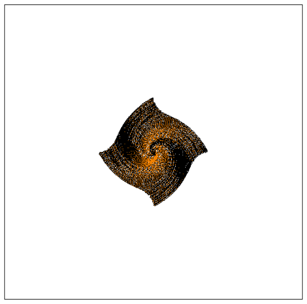
Now, we look at how we can combine Barycentric Coordinates, Textures, and Pixel Sampling. Given a texture with a triangle mesh, we place it on an xy-plane and for any **(x, y)** pair, we look for the triangle where this point lies in the texture, compute the Barycentric Coordinates to find the color/texture at that point.

In my own implementation, for every point, I compute the Barycentric Coordinates for that point, look at thet triangle where this point is in the texture, take a linear combination of the Barycentric Coordinates and the vertices, and I return the color that is obtained from taking this linear combination.

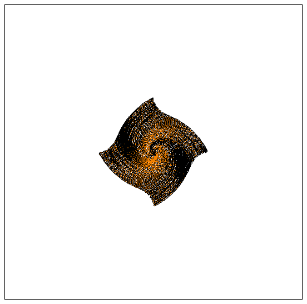
There are two types of sampling techniques; bilinear and nearest. Nearest sampling corresponds to taking an **(x, y)** coordinate and assigning the color or texture of the closest pixel location to this point. Bilinear sampling corresponds to taking the closest 4 pixels, and averaging them to get a value for the point.



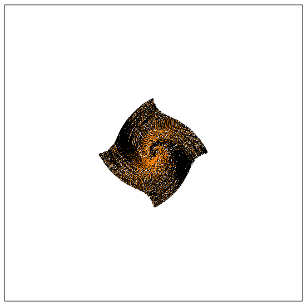
Nearest Sampling at 1 per pixel rate.



Nearest Sampling at 16 per pixel rate.



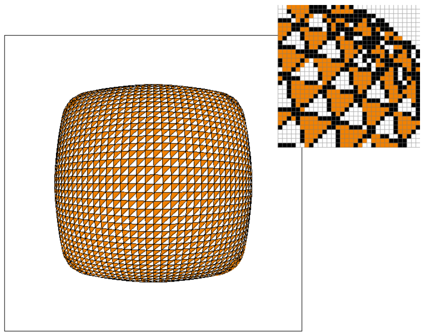
Bilinear Sampling at 1 per pixel rate.



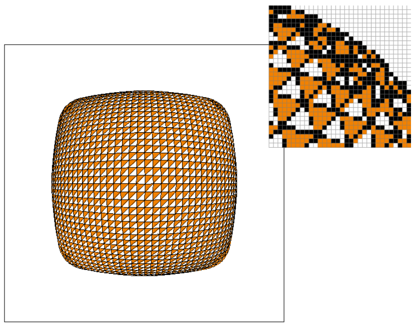
Bilinear Sampling at 16 per pixel rate.

In these pictures, the nearest sampling produces a darker image where there are darker textures. This happens because the textures are taking from the closest pixel, and the closest pixels are darker. This is in contrast to bilinear sampling where the average of many different textures, light and dark, are contributing to the texture of the pixel.

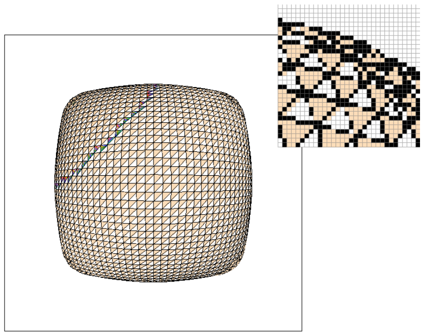
Part 6: "Level sampling" with mipmaps for texture mapping



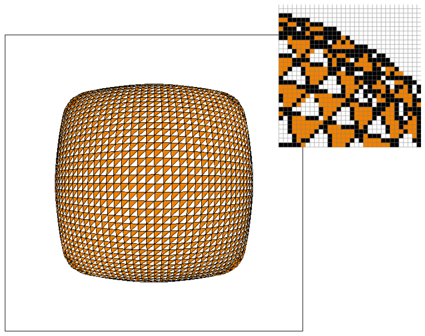
Bilinear at Level 0.



Nearest at Level 0.



Bilinear at Nearest Level.



Nearest at Nearest Level.

Level sampling is another way to avoid aliasing. This idea arises because we find that, sometimes, the sampling rate for one part of the scene is not always the best for every part of the scene. This can result in aliasing in some places and none in others. In my iplementation, I use the position vectors of the pixel to determine what level to sample from, using the formula **max(sqrt((du/dx)^2 + (dv/dx)^2), sqrt((du/dy)^2 + (dv/dy)^2))**. Once I have the level where the sampling will be the best, I go to that mipmap level and apply that texture.

This form of antialiasing is expensive in both memory and time, but yields very exciting results. It allows us to experiment with different sampling rates and zooms, but this experimentation is constly in time as we are comparing the quality of different sampling rates and mipmap levels. There are ways to quickly determine the mipmap level needed, but this requires the storage of position vectors for each pixel, and we'd have to compute and store these vectors for every pixel.