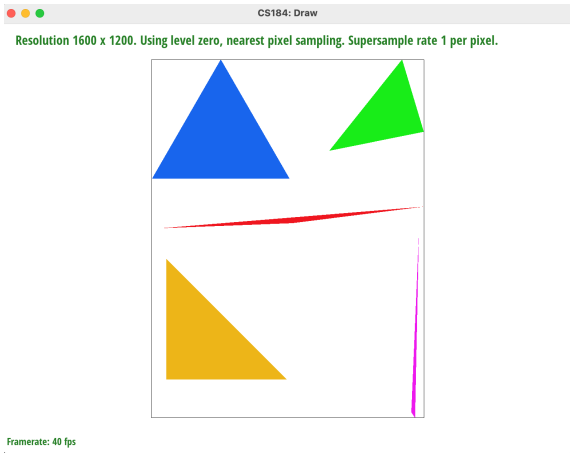
Task 1 (20 pts)

Walk through how you rasterize triangles in your own words.
Explain how your algorithm is no worse than one that checks each sample within the bounding box of the triangle.
Show a png screenshot of basic/test4.svg with the default viewing parameters and with the pixel inspector centered on an interesting part of the scene.
Extra credit: Explain any special optimizations you did beyond simple bounding box triangle rasterization, with a timing comparison table (we suggest using the c++ clock() function around the svg.draw() command in DrawRend::redraw() to compare millisecond timings with your various optimizations off and on).

I first determine the bounding box with min and max functions. Then, I sample points spaced by 1 in this region. Note that I account for the 0.5 offset by starting 0.5 less in the x direction. I use a helper function to do line tests. Then, I check for when all line checks have the same sign, in which case when they do I draw the color at the query point.



Task 2 (20 pts)

Walk through your supersampling algorithm and data structures. Why is supersampling useful? What modifications did you make to the rasterization pipeline in the process? Explain how you used supersampling to antialias your triangles.
Show png screenshots of basic/test4.svg with the default viewing parameters and sample rates 1, 4, and 16 to compare them side-by-side. Position the pixel inspector over an area that showcases the effect dramatically; for example, a very skinny triangle corner. Explain why these results are observed. ~~*Extra credit: If you implemented alternative antialiasing methods, describe them and include comparison pictures demonstrating the difference between your method and grid-based supersampling.*~~

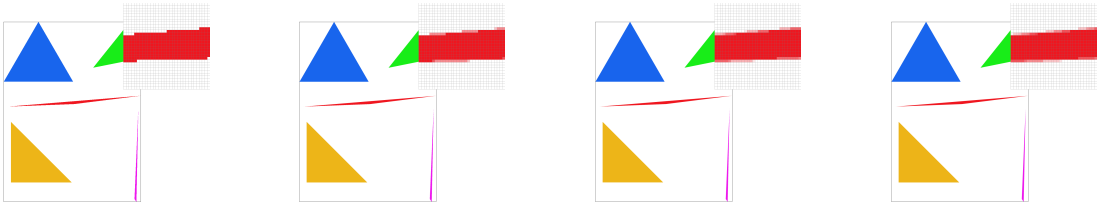
I resized the sample_buffer to be of size (width * height * sample_rate). This accounts for the extra factor of sample_rate samples to store. To index into the sample_buffer, I now do the following:

sqrt_sample_rate = sqrt(sample_rate) // so either 1, 2, 3, or 4
sample_buffer[floor(y * sqrt_sample_rate) * (width * sqrt_sample_rate) + floor(x * sqrt_sample_rate)] = color

This way the sample_buffer is row major and resembles the original dimensions without supersampling. Note that this layout could just as easily be implemented differently such as putting all supersamples next to each other in memory as opposed to on different "rows" as in this format.

Why is supersampling useful? It's useful for antialiasing by approximating the effect of a 1-pixel box filter.

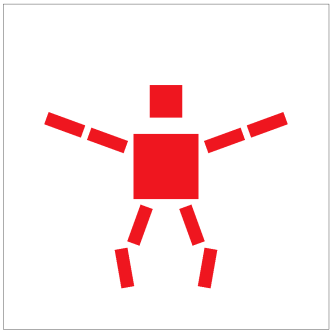
In addition to indexing differently, I also sample more points in the *rasterize_triangle* function and average colors in the *resolve_to_framebuffer* function.



From left to right, we show supersampling with 1, 4, 9, and 16 samples per pixel. Notice the "jaggies" without supersampling (leftmost) because this algorithm sets the color based on whether or not the center of the pixel is inside or outside the triangle. The jaggies start to go away from left to right and there are non-binary pixel values because of the averaging of different values where pixels and triangle boundaries/edges overlap.

Task 3 (10 pts)

Create an updated version of svg/transforms/robot.svg with cubeman doing something more interesting, like waving or running. Feel free to change his colors or proportions to suit your creativity. Save your svg file as my_robot.svg in your docs/ directory and show a png screenshot of your rendered drawing in your write-up. Explain what you were trying to do with cubeman in words.

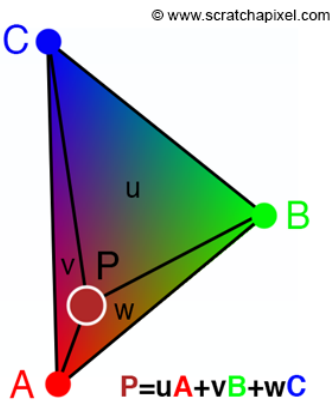


Cubeman just landed a super high jumping jack in this image, which is why the knees look quite bent/strange because the impact was so strong. Hopefully he'll be okay. I modified the svg file to have more rotations at different joints.

Task 4 (10 pts)

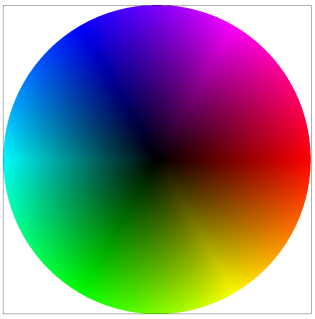
Explain barycentric coordinates in your own words and use an image to aid you in your explanation. One idea is to use a svg file that plots a single triangle with one red, one green, and one blue vertex, which should produce a smoothly blended color triangle.

Show a png screenshot of svg/basic/test7.svg with default viewing parameters and sample rate 1. If you make any additional images with color gradients, include them.



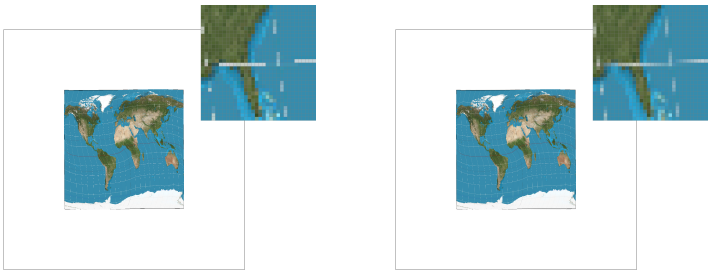
Credit for the above image is from https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates. Here I explain what barycentric coordinates are in my own words in reference to this figure. Essentially, they are used to linearly interpolate between three values at vertices in proportion to the distances to the vertices. If you drop a point P inside a triangle, you can compute the distance to each vertex A, B, and C. Then you can find a weighting u + v + w = 1 that can be used to interpolate the values at the vertices. Note that u, v, and w can also be computed by proportional areas of triangles inside the full triangle--as shown in the diagram.

Next, we show our result from the code.



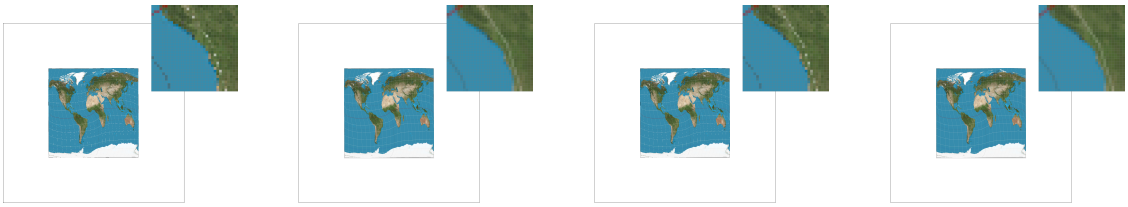
Task 5 (15 pts)
Explain pixel sampling in your own words and describe how you implemented it to perform texture mapping. Briefly discuss the two different pixel sampling methods, nearest and bilinear.
Check out the svg files in the svg/texmap/ directory. Use the pixel inspector to find a good example of where bilinear sampling clearly defeats nearest sampling. Show and compare four png screenshots using nearest sampling at 1 sample per pixel, nearest sampling at 16 samples per pixel, bilinear sampling at 1 sample per pixel, and bilinear sampling at 16 samples per pixel. Comment on the relative differences. Discuss when there will be a large difference between the two methods and why.

Nearest sampling will sample the nearest pixel in the texture map, while bilinear will do bilinear sampling in the texture map for the query texture coordinate.



On the left is nearest sampling and on the right is bilinear sampling. Notice that on the boundary of the United States, the bilinear sampling algorithm is much better.

Here are four screenshots from left to right: nearest 1 sample, nearest 16 samples, bilinear 1 sample, bilinear 16 samples per pixel.

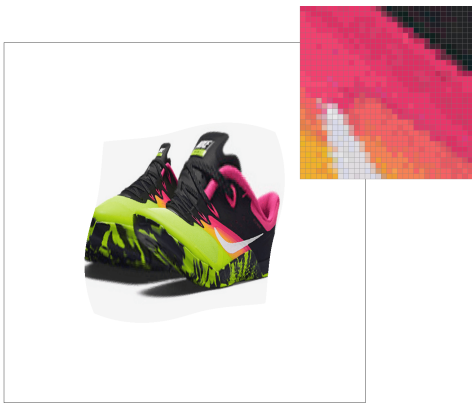


You can see that more samples and using bilinear leads to the best results. The largest difference will be in high frequency regions. Using more samples and using bilinear interpolation will have similar effects, though more samples is always going to be the best option at the cost of speed. This has to deal with the number of operations required for more dense sampling. Bilinear sampling isn't too computational expensive, as it's just a weighting on the nearby texture vertices.

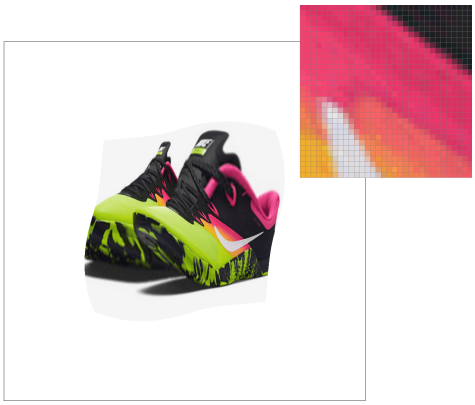
Task 6 (25 pts)
Explain level sampling in your own words and describe how you implemented it for texture mapping.
You can now adjust your sampling technique by selecting pixel sampling, level sampling, or the number of samples per pixel. Describe the tradeoffs between speed, memory usage, and antialiasing power between the three various techniques.
Using a png file you find yourself, show us four versions of the image, using the combinations of L_ZERO and P_NEAREST, L_ZERO and P_LINEAR, L_NEAREST and P_NEAREST, as well as L_NEAREST and P_LINEAR.
To use your own png, make a copy of one of the existing svg files in svg/texmap/ (or create your own modelled after one of the provided svg files). Then, near the top of the file, change the texture filename to point to your own png. From there, you can run ./draw and pass in that svg file to render it and then save a screenshot of your results.
Note: Choose a png that showcases the different sampling effects well. You may also want to zoom in/out, use the pixel inspector, etc. to demonstrate the differences. ~~Extra credit: If you implemented any extra filtering methods, describe them and show comparisons between your results with the other above methods.~~

Pixel sampling, level sampling, and the number of samples will affect the speed, memory usage, antialiasing power w/ these three techniques. Pixel sampling has trade offs related to speed. Using nearest pixel sampling, it will be quick but lead to aliasing artifacts because it must choose which pixel to grab from when in fact some vertices will be between color values and hence, bilinear interpolation could be used--however, it will cost more operations. Level sampling is using a mipmap, which costs a bit more memory than storing the texture image only at its highest resolution. Antialiasing power will be best when doing supersampling and bilinear sampling, but it won't be as fast as using a mipmap and sampling at different layers of different resolutions.

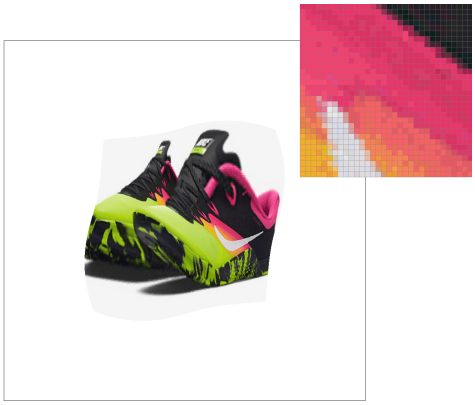
Below I show some examples with different level and pixel sampling, as specified in the instructions. I'm using a Nike shoe as the texture image with the pointer inspector towards the top left of the swoosh.
L_ZERO and P_NEAREST



L_ZERO and P_LINEAR



L_NEAREST and P_NEAREST



L_NEAREST and P_LINEAR