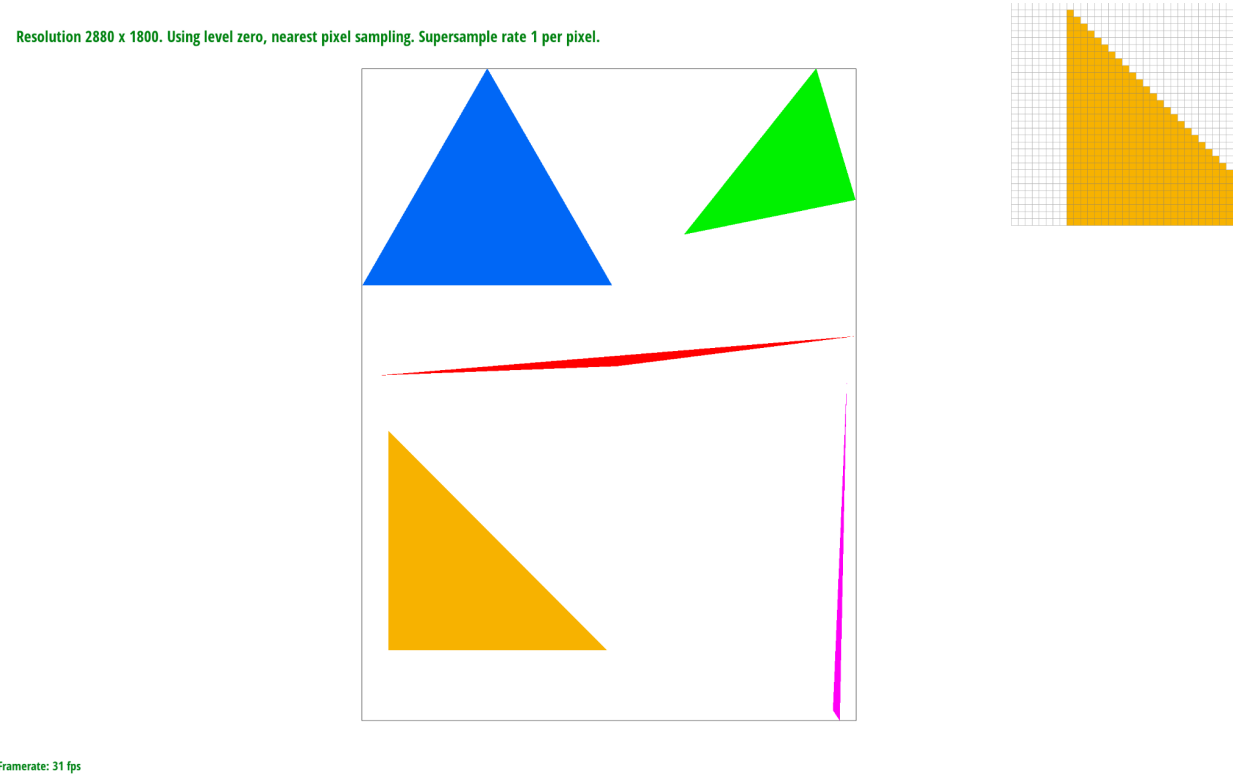
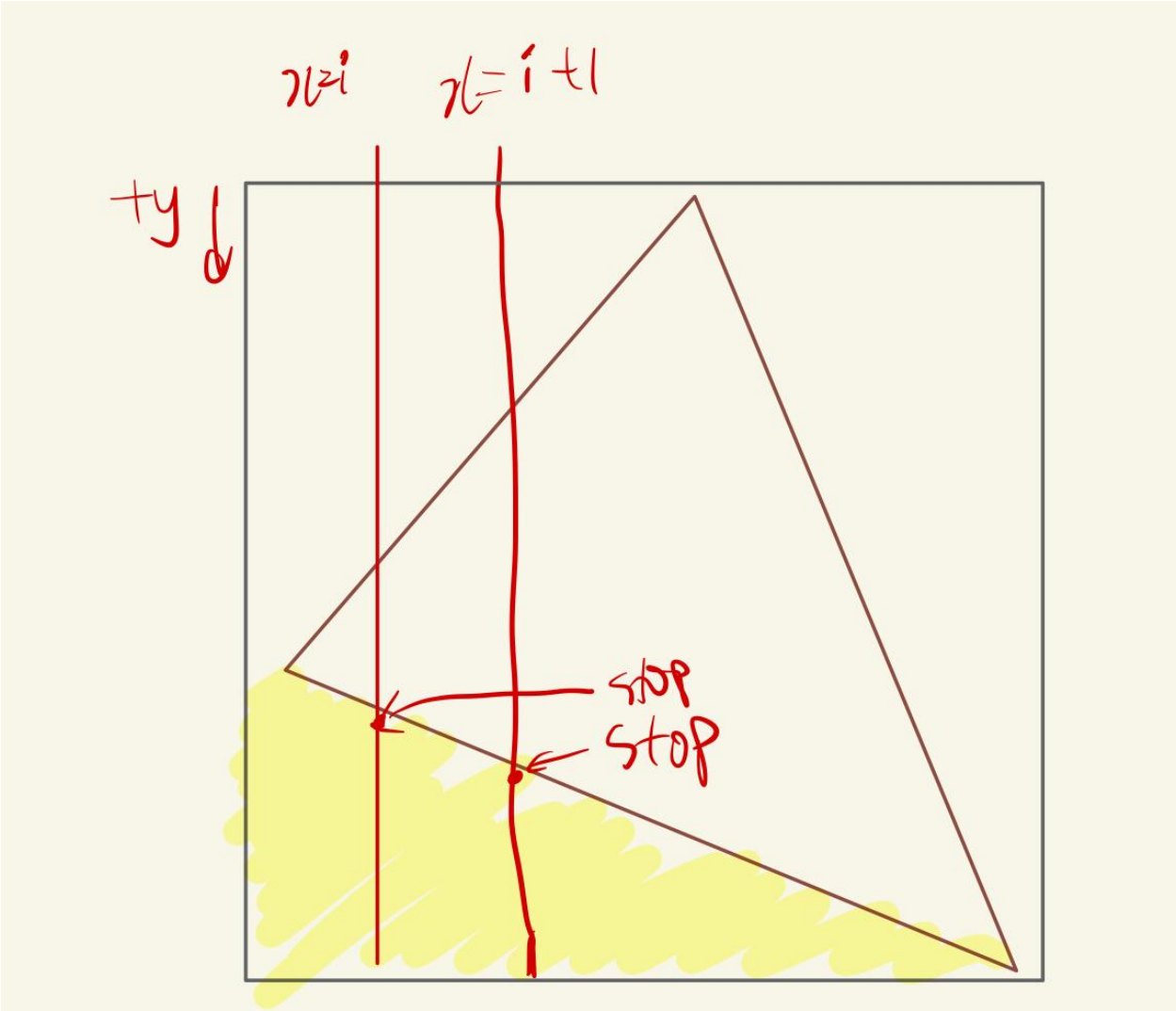
Project 1 Write-up
David Zuo, Bohan Yu

Task 1:
Our naive solution (without optimization) checks every sample point within the bounding box of the triangle: if the point is inside the triangle, then we update its color in the sample buffer, else we do nothing. We do the inside-triangle testing by following the logic described in the lecture, with a little twist to generalize it. Let's say we have ABC as the three vertices of the triangle. We randomly pick one, say A, and then construct vectors AB, BC, CA, where AB means a vector that points from A to B. This ensures that AB, BC, CA are either counterclockwise or clockwise, and we compute the L(x, y) of a sample point on each of the edges. This gives us three conditions, and we determine if the point is inside the triangle by checking that either they are all greater than 0 (in the case of a clockwise loop) or they are all smaller than 0 (in the case of a counterclockwise loop).



Extra credit: We did a same for loop for the X coordinate like the naive solution. Instead of checking every Y coordinate point within the same x, we break out of the Y loop after reaching a y coordinate when we noticed a pixel that is completely outside of the triangle, and continues with the next X coordinate. This would save us from processing pixels that is under the triangle but in the bounding box.

As shown in the example image below, the black rectangle is the bounding box of the brown triangle. When we are looping through the Y coordinate(going from top to bottom) while fixing X to be i(see the red line), we would break out of the Y loop after reaching the point that is under the triangle instead of continuing looping until the edge of the bounding box. The area highlighted in yellow would be the pixels that we processed in the naive solution but not the optimized one.



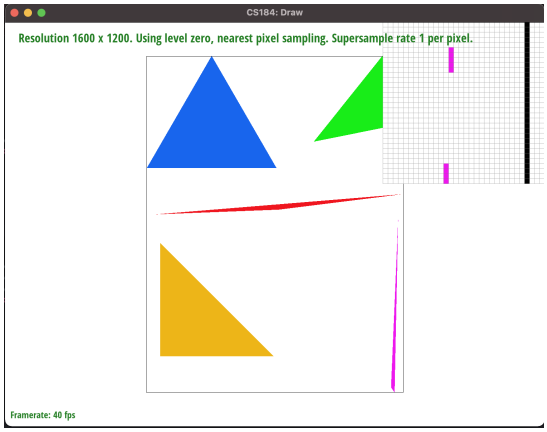
Time comparison table for test picture 6: (the time unit is in 1e-6 s).

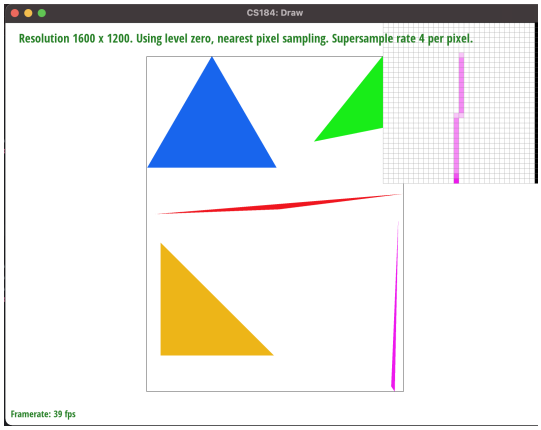
Sample rate	Total time taken by naive solution	Total time taken by optimized solution	how many percentage faster than naive solution
1	3958	3474	12.228398
16	34186	26286	23.108875

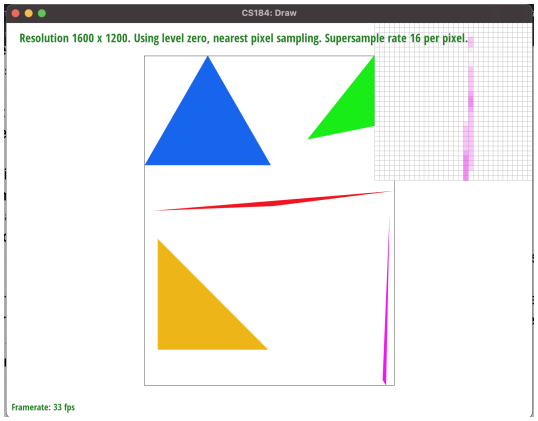
As we can see, the optimized solution is 23.1% faster than the naive solution when the sample rate is 16.

Task 2:
We used grid supersampling. A twist that we made to our naive solution (as described above) is that now, for each pixel center in the bounding box of the triangle, we divide it into X little squares (where X = sample rate), and we loop over these little squares, perform inside-triangle test, and compute the final color value for that pixel. Supersampling is useful because it smoothes jaggies by having a color gradient at the edges of the triangle, instead of a sharp change in the color (e.g. from pure red to pure white). By increasing the sample rate, we are able to anti-alias our triangles as for pixels whose center are not inside the triangle, we can still update its color by calculating the amount of little squares of this pixel that are inside the triangle. A change that we made to our rasterization pipeline is that we clear the buffer each time we change the sample rate or set the framebuffer target. We also updated the fill_pixel function to accommodate supersampling.

Screenshots:



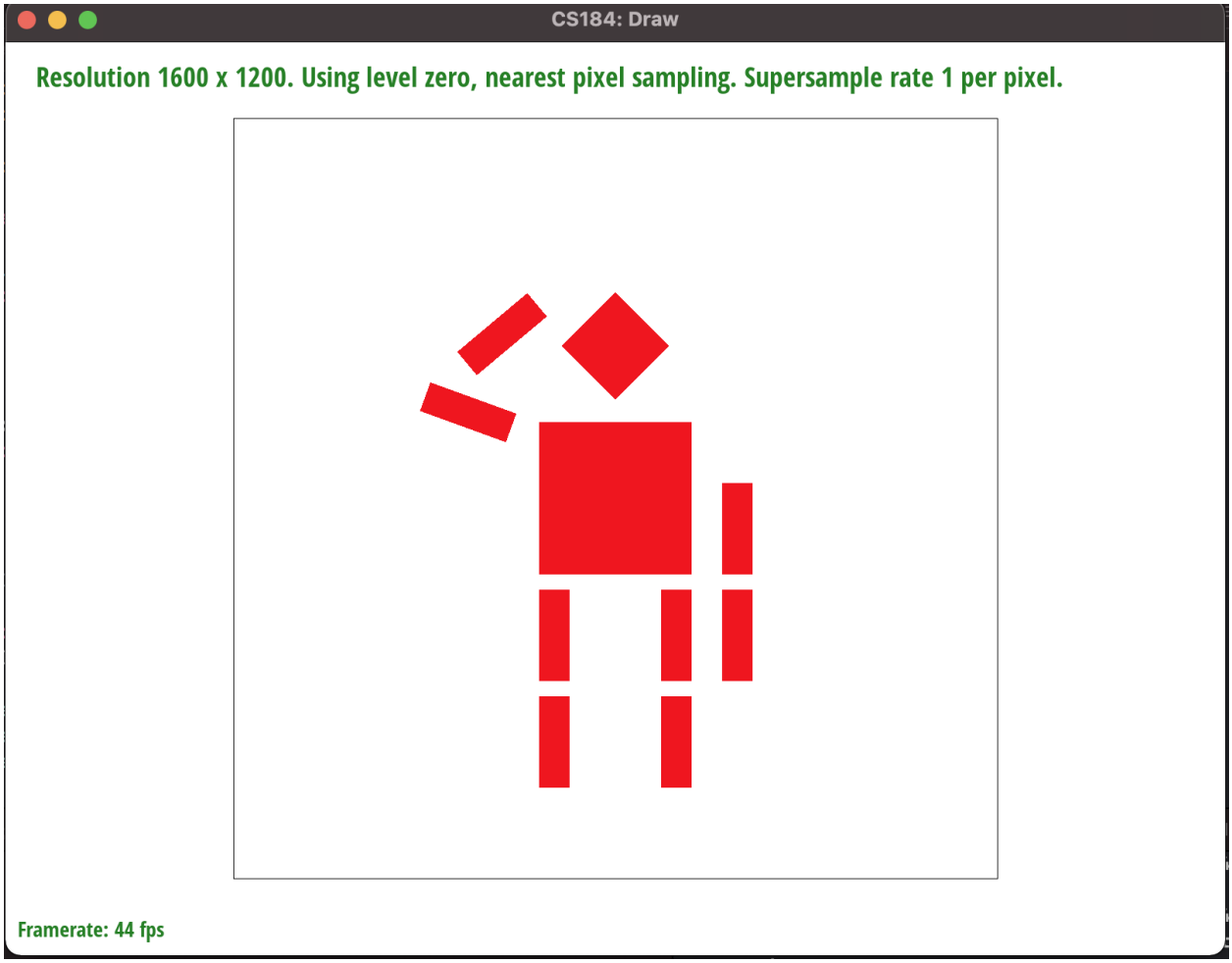




We can notice that some pixels that don't have color when sample rate is 1 become a little bit pink when the sample rate goes up. This is because the center of these pixels (x+0.5, y+0.5) are not in the triangle, but after dividing the pixels into grids, the little squares' centers are in the triangle, updating the color of that pixel to be faint pink.
We also notice that some pixels at the upper corner that have color when sample rate is 1 become white when the sample rate is 4. That is because the pixel centers are inside the triangle, but all four centers of its grid happened to be not inside the triangle.
Still, at a first glance, when the sample rate goes up, the triangle appears to be more smooth and "clear" to the human eye.

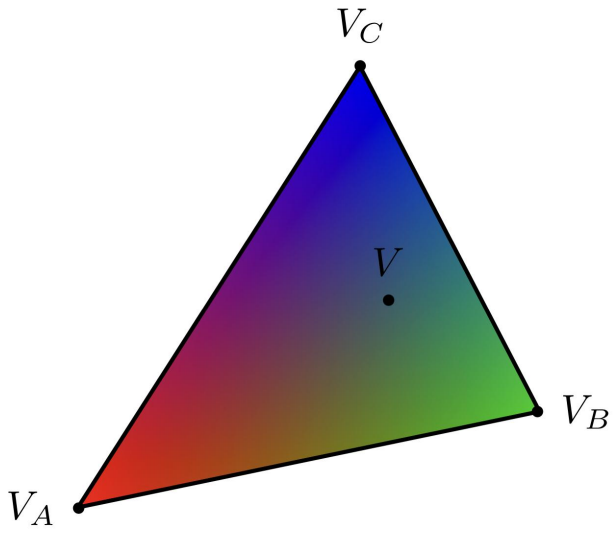
No extra credit: We didn't implement other antialiasing for this question.

Task3:
We transformed the robot to salute to CS184 course staff!!! (See screenshot below).
We rotated the right arm down and the left arm and elbow up. The biggest challenge encountered in this part is to figure out the order of transformation. My partner and I didn't know the importance of the order of transformation until we spent an hour debugging at the wrong place and were finally rescued by lecture slides.



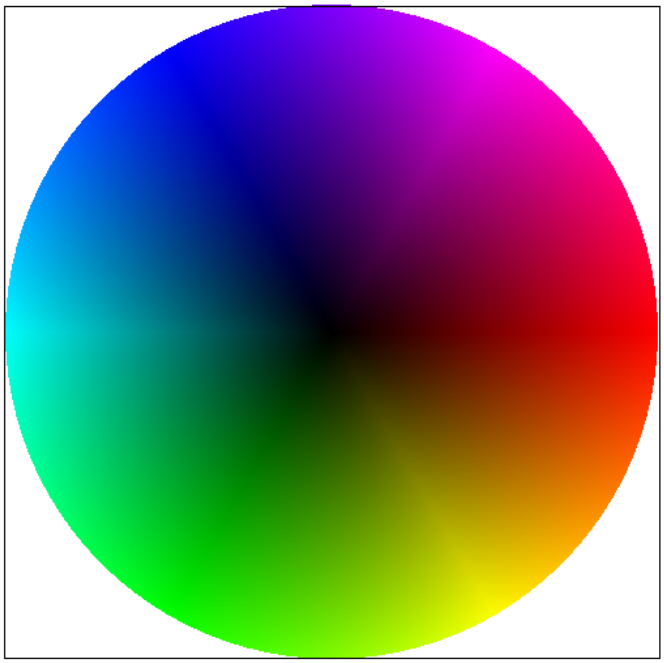
Task4:

Barycentric coordinates of a point allows us to interpolate the “value” of a point (could be color, texture, etc) by the weighted average of the vertices’ value, where the three weights are the three coordinates respectively. The closer the point is to one vertex, the larger the weight of that vertex would take. In this image below, because point V is the closest to Vertex B, it would have a larger weight in the color green, and thus would appear green.



PNG screenshot of test7.svg:

Resolution 800 x 600. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.



Framerate: 45 fps

Task 5:

Explain pixel sampling in your own words and describe how you implemented it to perform texture mapping.

Given a screen space and a texture space, we want to figure out which texture to apply to the current pixel. We first converted the pixel coordinate (x,y) in the triangle to

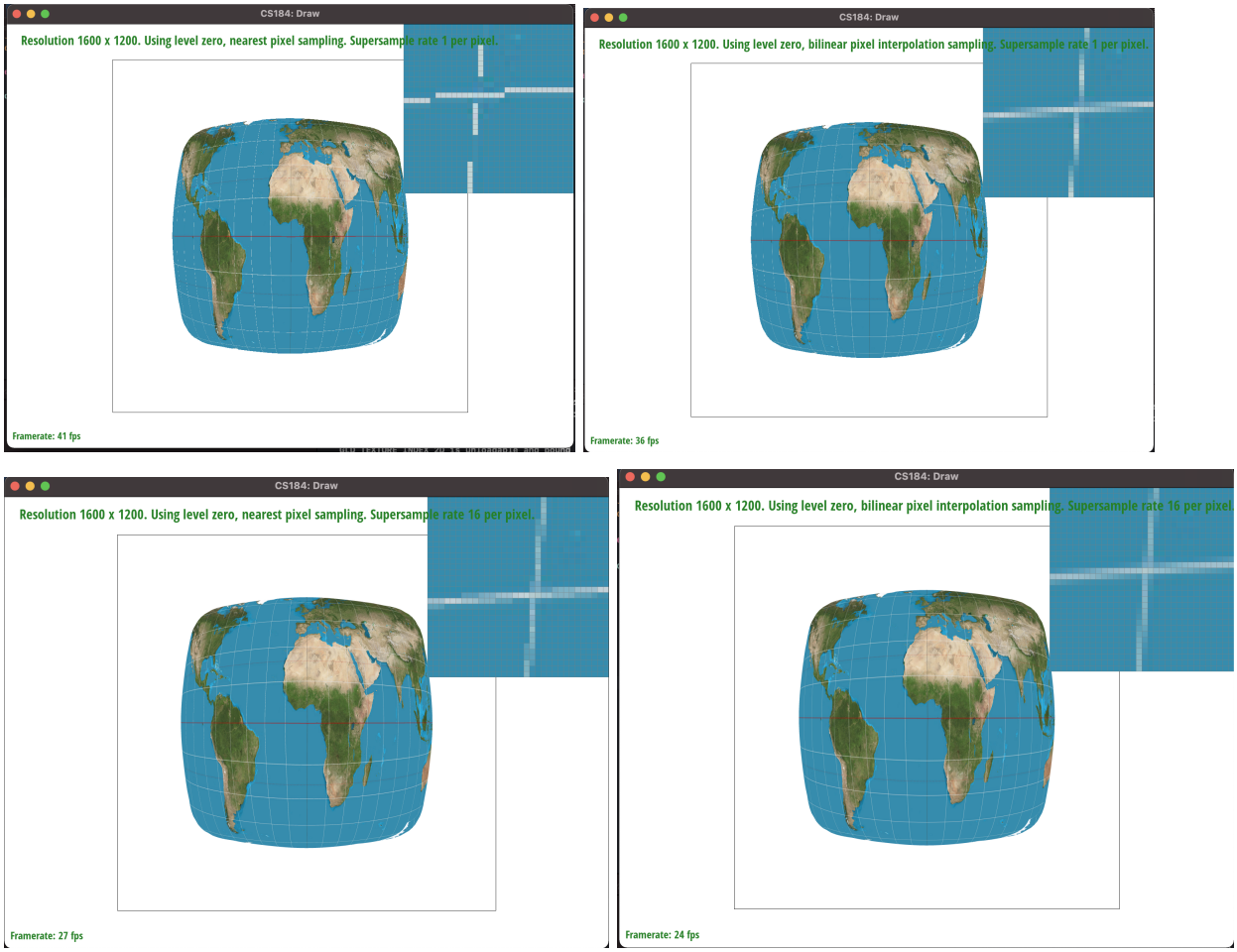
barycentric coordinates, and using those coordinates, we calculated the corresponding uv coordinates that are used for the texture space. For each pixel, we compute the corresponding uv coordinates and pass that as a parameter to the sampling method—nearest or bilinear—which will return the texture for that pixel. We then update the pixel with the returned texture.

The “Nearest” sampling method simply returns the texture of the square where the sampling point uv lies. That is, given (uv.x, uv.y), we return the texture at (floor(uv.x), floor(uv.y)).

“Bilinear” sampling method, on the other hand, first finds the four nearest neighbors of the sampling point (including the square where the point lies). We return the weighted average of these four textures, where the weights are determined by the maze distance between the sampling point and the center of the four texture squares. The closer the sampling points to the center of a texture square, the higher the weights, and vice versa.

To compare these two methods, we find the second picture inside the folder textmap to be the one that most clearly shows the difference between bilinear and nearest sampling. See screenshots below: Pictures on the left are produced by nearest sampling, and pictures on the right are produced by bilinear sampling. The first row has a sample rate of 1, and the second row has a sample rate of 16. We can clearly see that when we use the nearest sampling at the sample rate of 1, the white lines on the map have jaggies and missing pixels, where the bilinear sampling method gives a much more smooth line, with gradual change between “blue” and “white.” The jaggies in the nearest sampling still persist when the sample rate is 16, though with a less sharp change in color thanks to the anti-aliasing effect of super sampling. The white lines in the bilinear method appear to be even more smooth not only on edges, also due to the antialiasing effect of supersampling.

There will be a large difference between the two methods when the neighboring pixel texture differs by a lot (e.g. a sudden change from white to black, which means a very high frequency). Since the nearest neighbor only samples one texture sample, it's likely that the resulting rendering has some wrong pixel values in it (as in the example above, the white line is broken into segments due to texture sampling of blue instead of white).



Task 6:

- Explain level sampling in your own words and describe how you implemented it for texture mapping.

Level sampling is a function that get the you the best mipmap level you should use for texture sampling. This level sampling would attempt to prevent aliasing and over-blurred images.

Implementation:
We first converted the pixel coordinate (x,y) in the triangle to barycentric coordinates, and using those coordinates, we constructed the same point into the texture coordinate

(u,v). We did the same conversion process from the pixel coordinates (x + 1, y) and (x, y + 1), to texture coordinatees (ux, vx), (uy, vy). Then, we subtracted the coordinates (ux, vx) from (u,v) to get (du/dx, dv/dx) and the coordinates (uy, vy) from (u, v) to get (du/dy, dv/dy). Since those coordinates are normalized to 1, we then multiplied the **width** of the mipmap to both du/dx and du/dy, and we multiplied the **height** by dv/dx and dv/dy. At last, we applied this formula in lecture to get the float D.

$$D = \log_2 L$$

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

When it is lsm_nearest, we would round the float D and access the corresponding level. When it is lsm_linear, we would get both the sample from DLower=floor(D) and DUpper=floor(D) + 1. Then we would linearly interpolate the color from the samples of floor(D) and floor(D) + 1, based on the float D from level sampling. The formula is like below:

Final_color = color_from_lower * (DUpper - D) + color_from_upper * (D - DLower)

- **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.**

In terms of speed and memory, the 16 samples per pixel has the highest, as you are sampling 16 times per pixel, causing 16x of memory and time. Then level sampling is the second highest in memory and lowest on time, as you will be storing about 2x of data vs the original(each mipmap is divided by two in the next level – n + n/2 + … + 2 + 1 = 2n) and 2x more time(as you would need to be sampling twice and linearly interpolating between the two levels). The one that has the least memory and second highest on time is selecting pixel sampling, as you would only need to store one copy of the image, but since you would be doing bilinear-sampling on the 4 nearest pixels, you would be using about 4x of time.

In terms of anti-aliasing power, supersampling has the best power, as not only do you eliminate the alias, you also keep the original feature as much as possible without impacting the visuals. Level sampling is the second best, it eliminates aliasing just as

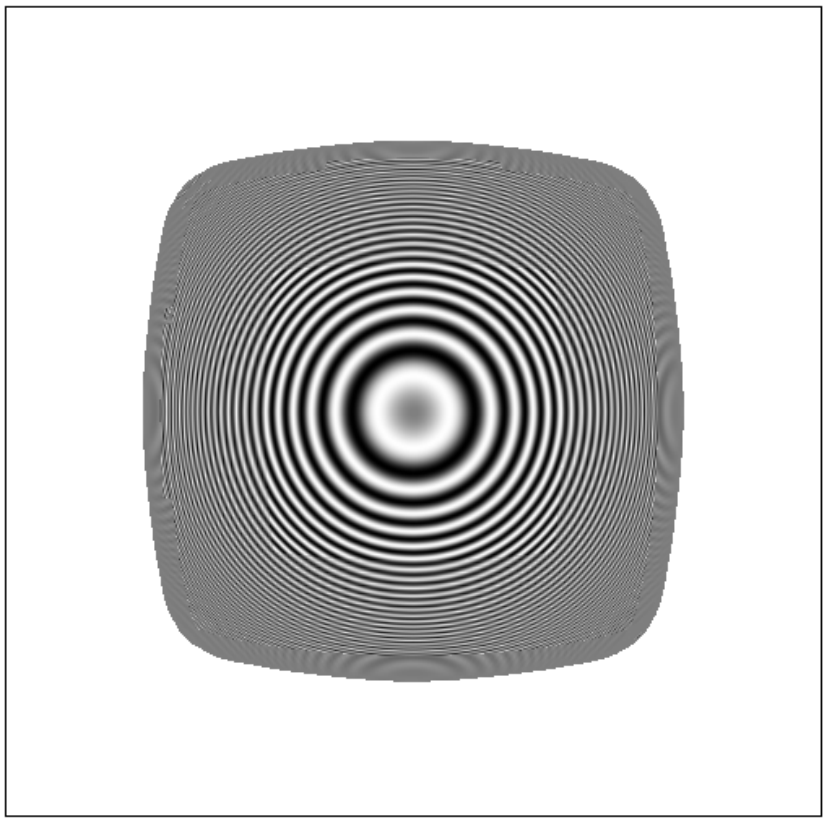
well as supersampling at 16 sample rate, but the small details are either lost or very slightly colored(like the white lines on the world map). Then the worst is pixel sampling, as it doesn't eliminate all of the aliasing in the world map.

- **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`.**

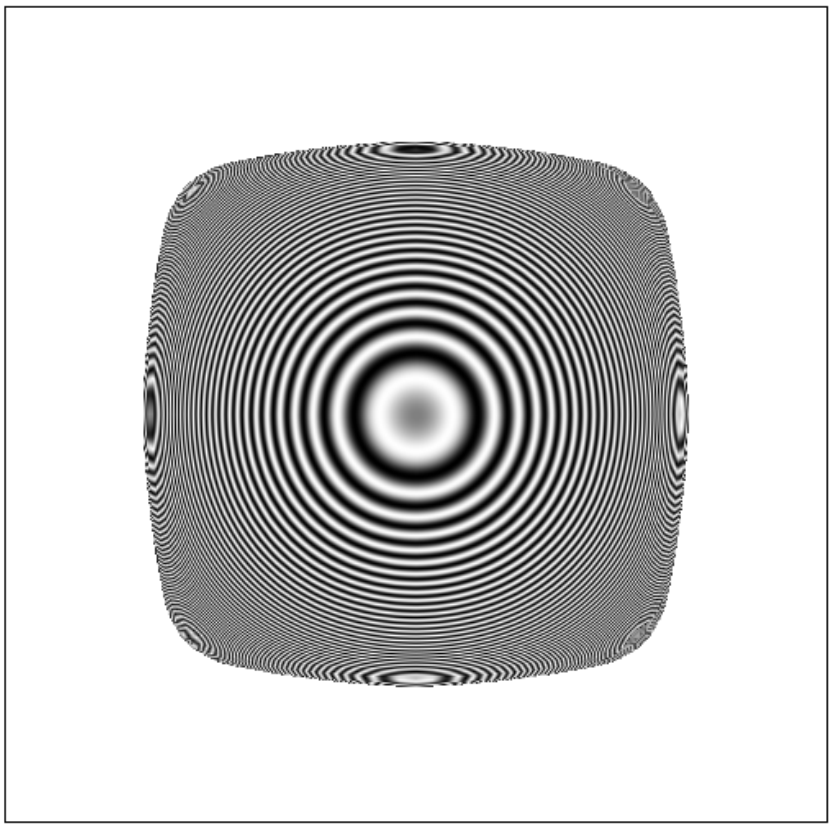
As one can see, there are some aliasing of circles on the top and the side of the screenshots below when it is level zero. Although it is a bit better on the bilinear pixel sampling, I think the one that is the best is nearest level.

(See the description in each of the screenshots)

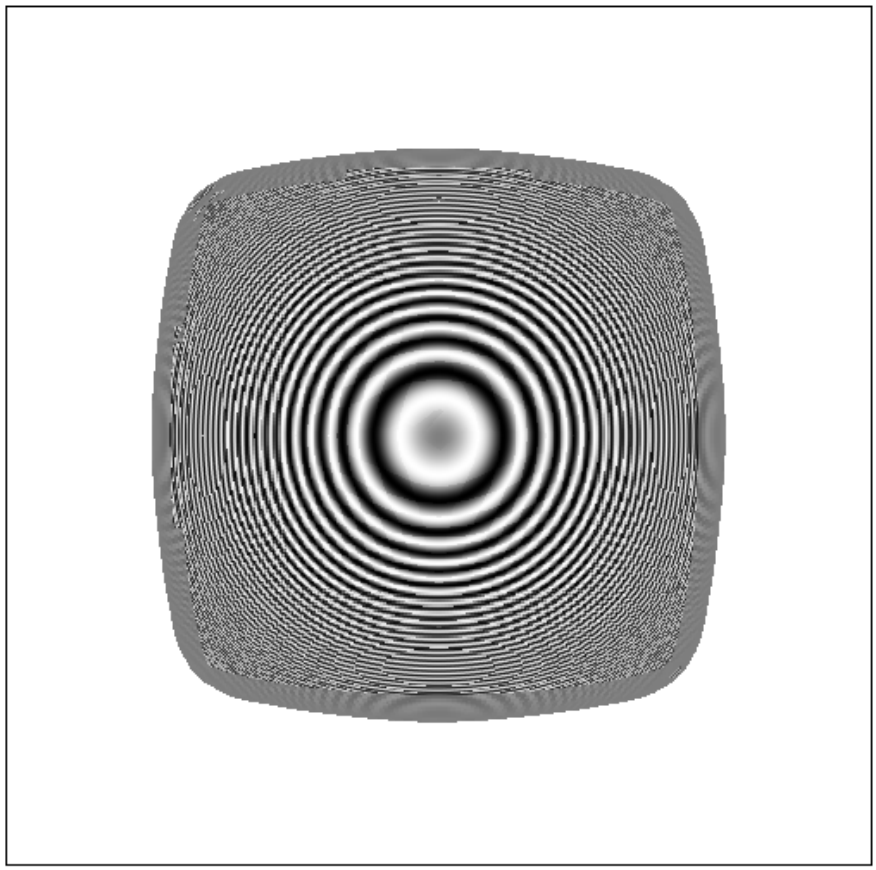
Resolution 800 x 600. Using nearest level, bilinear pixel interpolation sampling. Supersample rate 1 per pixel.



Resolution 800 x 600. Using level zero, bilinear pixel interpolation sampling. Supersample rate 1 per pixel.



Resolution 800 x 600. Using nearest level, nearest pixel sampling. Supersample rate 1 per pixel.



Resolution 800 x 600. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.