Rasterizing SVG Images

by Jonah Bedouch and Brandon Wong

2025-02-22

This report is also available online at https://cs184-zen.vercel.app/projects/hw01.
The GitHub repository for this project can be found at
https://github.com/cal-cs184-student/sp25-hw1-zen

Project Overview

In the project, we implemented basic rasterization of triangles and then steadily added features to improve the images generated in a variety of ways. The first improvement was the addition of antialiasing via grid super sampling. We also looked at jittering for anti-aliasing, but it proved to have its own distinct issues. After that, we implemented transforms so when rasterizing the images, elements from the svg files could be translated, scaled, or rotated. We also added a couple of keybindings that would allow the viewer to rotate what they were looking at by 90 degrees clockwise or counterclockwise for any generated image. Then, we implemented a function that can rasterize triangles that interpolates colors throughout based on barycentric coordinates. Finally, we implemented pixel sampling to generate textures in the rasterized images and then improved it by adding level sampling with mipmaps to help deal with aliasing in the textures added to the images.

Task 1: Drawing Single-Color Triangles

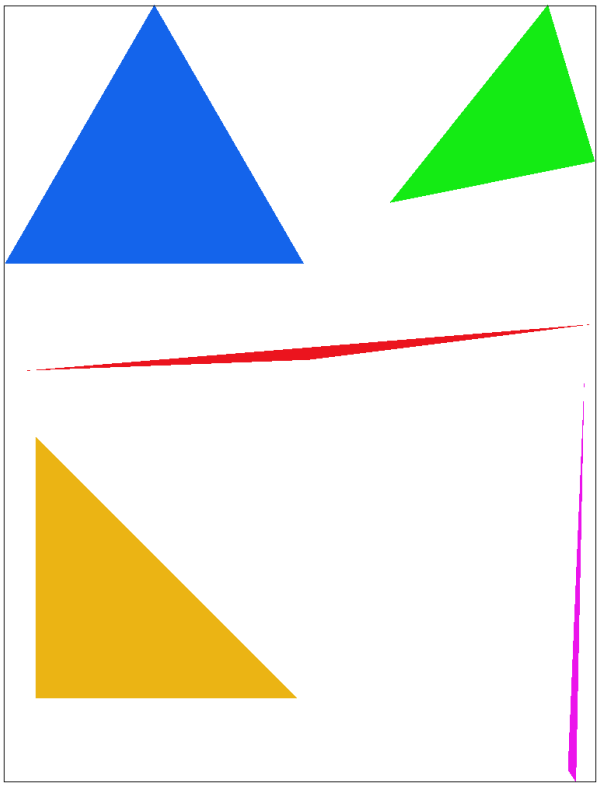
We first begin by finding the cross product of two vectors, one formed by the first and second points and one formed by the second and third points. This represents two adjacent edges of the triangle, and if the resulting value is positive, we know that the winding order is counter-clockwise. As such, if it's *negative*, we switch two of the points (arbitrarily points 0 and 1), so that the winding order becomes counterclockwise. From there, we define each of the three points as 2D vectors P0, P1, and P2. We also define the vectors that are normal to each edge of the triangle at points P0, P1, and P2 as N0, N1, N2.

After we've found this, we evaluate our points to find the minimum x, maximum x, minimum y, and maximum y value. This allows us to create a bounding box around the triangle itself by rounding down the minimum and rounding up the maximum. We do a nested for loop in order to check whether each pixel in this box is within the bounds of the triangle. By adding 0.5 to the minimum values before this loop, we are able to perform each check in the center of each pixel.

To do a check for any given pixel, we define 4 vectors – P, which represents the pixel itself; V0, which represents the vector from P0 to P; V1, which represents the vector from P1 to P; and

V2, which represents the vector from P2 to P. We then find the dot product between these V* vectors and their corresponding normal vectors. This works because geometrically, performing this operation finds the projection of V* onto N*. Because we've ensured that all lines are counter clockwiise, this normal vector is pointing inwards into the triangle. As such, if the vector pointing at our point is inside of the triangle, it creates an acute angle with the normal vector and the resulting dot product is positive. If the vector is on the line itself, it will be perpendicular and have a dot product of 0, and if the vector is outside of the triangle, it will form an obtuse angle which has a negative dot product. As such, we check the angle formed between the normal of each of the three edges of the triangle and each of the vectors to the points are all greater than or equal to zero (which indicates that the point is inside the triangle), and then we rasterize that point. If any of the three are negative, the point is outside the triangle and we do nothing.

The algorithm is no worse than one that checks each sample within the bounding box of the triangle because the only section of the code that scales is the size of the box covered by the dual for loop within the code. This dual for loop is limited to covering the bounding box and nothing else.



Task 1 Test 4 Result

Extra Credit

This task's extra credit was done after the completion of Task 4. Speed ups were on a mix of task 1 and 2 code due to how connected they are.

There were two speed ups clearly on task 1 code. The biggest speed up was by ending checks after the bottom and right ends of the triangles were reached for columns and rows respectively. Three booleans were used for the checks, two detecting if the last pixel in the row or column (depending on direction being tracked, although rows needed a list of booleans, one for each row, due to x being the outer loop) to see if the last pixel in the row or column was a triangle and one to decide whether to stop if the next pixel was not. Once it's detected an end was reached, a break was put in the y loop for columns and an if statement skips all the internal checking logic for rows.

The other task 1 speed up was moving the calculations for the L values to decide whether or not a pixel was in the triangle into the if statement deciding it, allowing some of the calculations to not be run if an earlier one is clearly wrong. This gave a small speed up.

The rest of the speed ups were mainly for task 2 code. To speed these up, one, the calculations adjusting the x, y, width, and height values for deciding locations in the sample buffer were moved out of as many loops as possible, outright out for the width and height. Two, the calculations for the shifts for the internal for loops deciding where in the pixel the super sampling was done was put in a separate standalone double for loop, as these additions do not change.

The average original time (the average of the triangles in task four, run three times) was 4887797.2. The average time after the optimizations was 3446249.933. This was an approximately 29.5% improvement in speed.

Task 2: Antialiasing by Supersampling

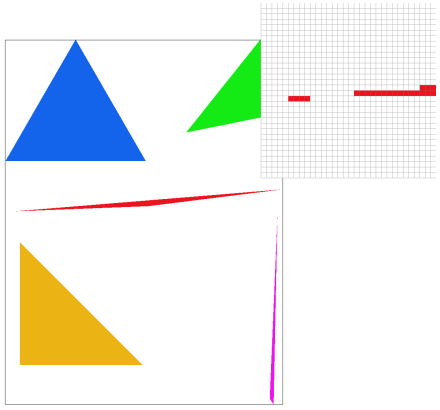
The supersampling algorithm works by using a sample_buffer that has been scaled up by the sample rate, done in the set_sample_rate and set_framebuffer_target functions. This allows each pixel to be represented by a sqrt(sample_rate) by sqrt(sample_rate) square.

With the sample_buffer size set based on the sample rate, when rasterizing each triange we loop through each pixel n times, with n being the sample rate. This takes the form of two loops that repeat sqrt(n) times, one in the x direction and one in the y direction. Each of these two inner loops shift the point being looked at in the pixel by 1 / sqrt(n) in their respective direction, starting from half that value into the pixel from the starting corner represented by the actual x and y values of the pixel. The check done in the previous part occurs and the corresponding point within the pixel in the sample buffer is assigned the color if the point in the pixel is within the triangle. Before this color is assigned, a check is done to make sure that the point in the sample buffer being modified is within the current width and height of the area being looked at to ensure that only area within the sample buffer is accessed and modified.

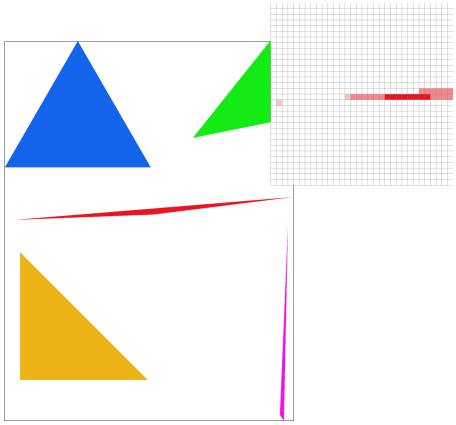
Once the sample buffer is done being filled by the rasterize triangle function, the resolve to framebuffer function has to average out all the samples within a pixel to decide the final color value of the pixel. This is done by using two loops in the x and y directions to access the color value for each sample in the buffer for a pixel and add them together. After the color values have been summed, they are divided by the sample rate to obtain the average color values for the pixel, which is then converted to the color datatype for the rgb framebuffer target and assigned to its respective point in it.

For rasterizing lines and points, the fill_pixel function was modified to loop through the samples in the sample buffer for each pixel and assign the same color to every one of them. The function for rasterizing triangles was modified to not require it.

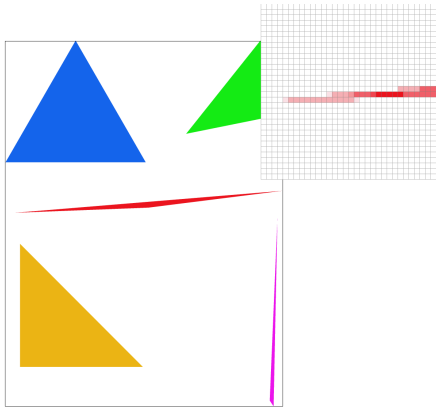
The final results are shown below, with supersampling resulting in edge pixels of the triangles holding averaged out values that make the triangles loop sharper and smoother than before, removing aliasing from pixels that decide their color based on points within them that do not match all the colors within the pixel.



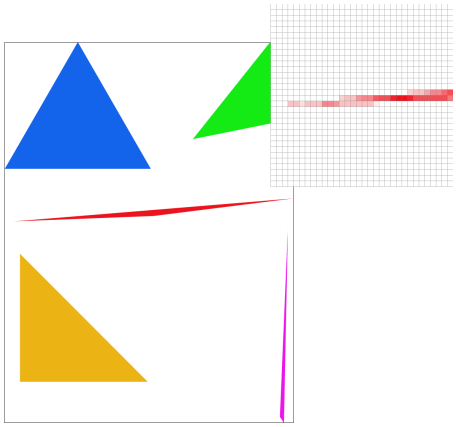
Result for Sample Rate 1



Result for Sample Rate 4



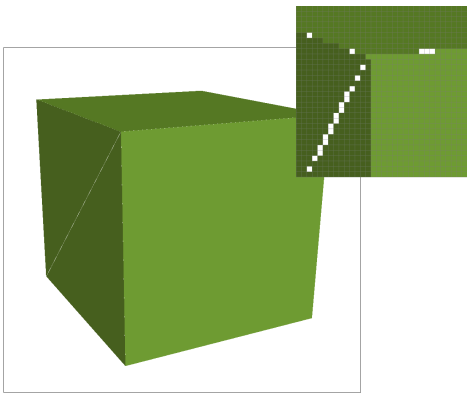
Result for Sample Rate 9



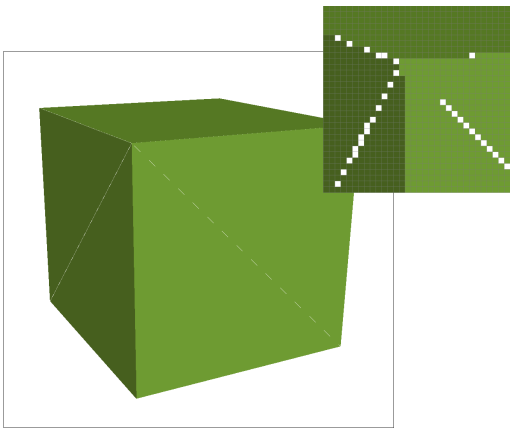
Result for Sample Rate 16

Extra Credit

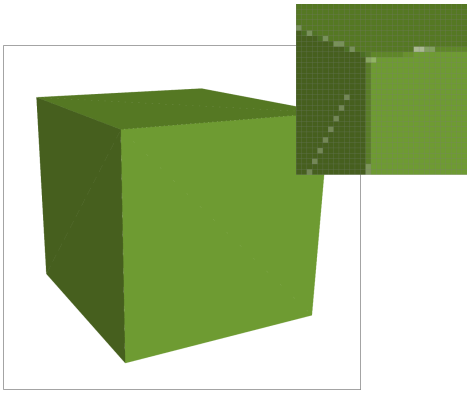
The additional sampling pattern we implemented was jittered sampling. Instead of taking the value for each sample, or partial sample, from the center of each one, we "jittered" the point we looked at slightly to be within the box in the grid for the samples. The points for each pixel or subpixel were off center by a random amount and direction. Unfortunately, this generally made aliasing worse, especially at lower sampling rates. Due to randomness, two results for jittered sampling at sampling rates of 1 and 4 are shown below, followed by the better results from super sampling. The worse results are likely due to the random point status per pixel resulting in many pixels having their points in the smaller non-triangle portion. This is likely made worse due to the implementation using the same jittered values for each subpixel due to previous optimizations, which is just each pixel for the sampling rate of one, but it is clear that jittered sampling has worse results, especially at lower sampling rates.



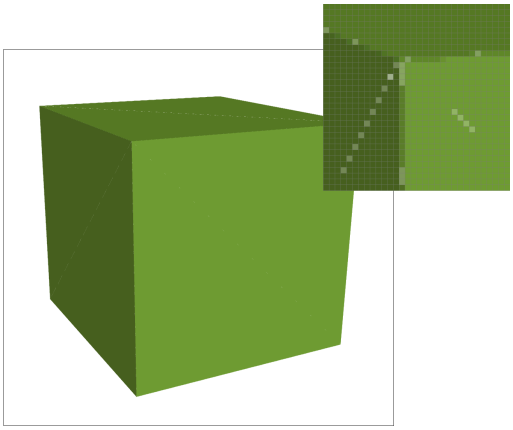
Result with Jittering Sample Rate 1 (1st Trial)



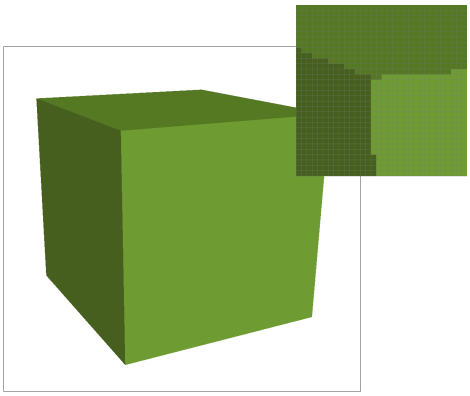
Result with Jittering Sample Rate 1 (2nd Trial)



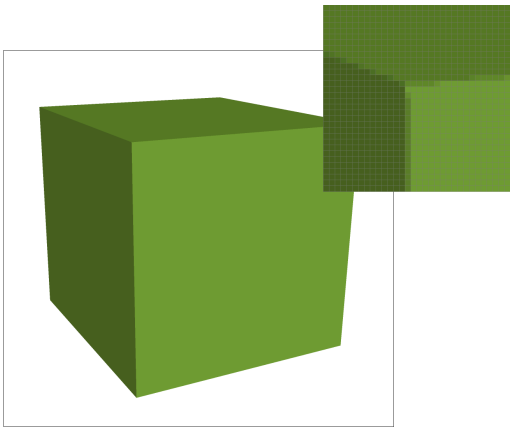
Result with Jittering Sample Rate 4 (1st Trial)



Result with Jittering Sample Rate 4 (2nd Trial)



Result with Supersampling and Sample Rate 1

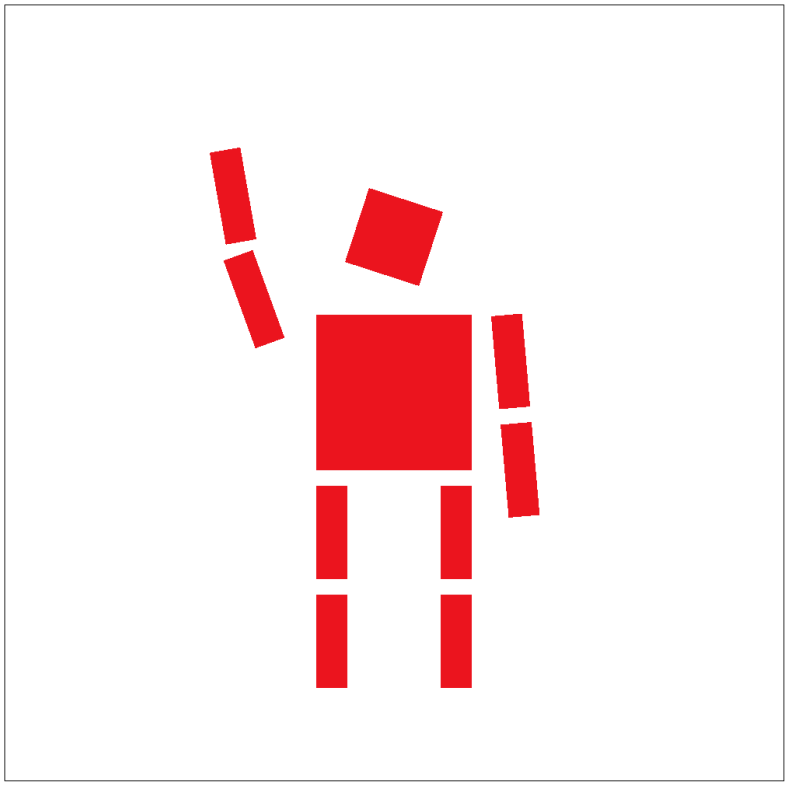


Result with Supersampling and Sample Rate 4

Task 3: Transforms

This task was much simpler than previous ones. To complete this task, the 3D transformation matrices from lecture had to be manually and made based on the inputs to the function and returned. For rotate, the degree input had to be converted to radians before being put into sine and cosine functions to obtain the values for the matrix.

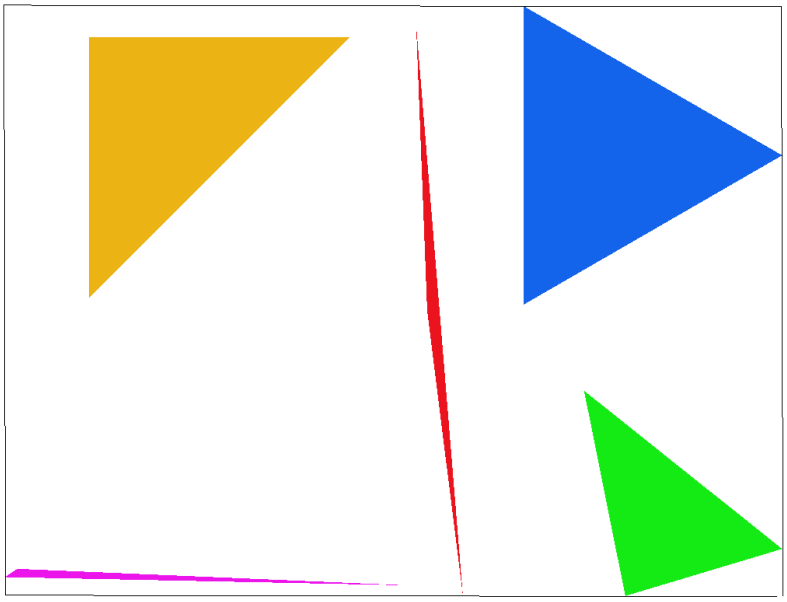
To finish the task, the cubeman was modified to be waving directly at the screen. Its left arm (relative to the viewer) is raised up high with and angle at the elbow to look like waving, while its right arm has been rotated down to show that it isn't doing anything. The head has been tilted away from the left arm to better give the appearance of waving as people tend to angle their head away from their waving arm.



Task 3 Result of Waving Robot

Extra Credit

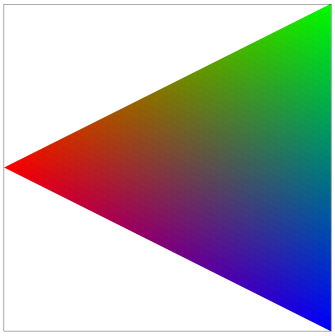
The extra feature added to the GUI was rotation via pressing the bracket keys. Left bracket is counterclockwise and right bracket is clockwise. This was done by adding two new functions, one for clockwise and one for counterclockwise. The two functions work by first translating the current svg so that it's center aligns with the overall center, then multipling it with the 90 or −90 degree rotation matrix depending on direction, then translating it back so it's centered in the viewport again. These functions are called in response the their corresponding key presses.



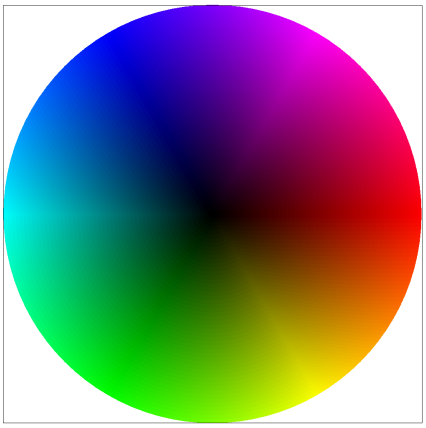
Task 3 Rotation

Task 4: Barycentric Coordinates

Barycentric coordinates are coordinates based off the points of a triangle. Points on a plane can be found by assigning weights to the coordinates of the vertices, and these weights form the barycentric coordinates for these points on the plane for the specific triange the weights were generated for. This system of assigning weights to vertices of a triangle allows for interpolation of values; in the case of the first example shown below, by finding the weights and therefore the barycentric coordinates of each point based on the triange below, the colors in the triangle at those points can also be weighted in the direction of the color of a corner. The corner a point is closest too would have the largest barycentric coordinate value, and as a result the color at that point would be closest to that closest corner.



Task 4 Result



Task 4 Test 7 Result

Task 5: 'Pixel Sampling' for Texture Mapping

Pixel sampling is a technique by which we can render an image onto an SVG by scaling the coordinate system of the render frame to the coordinate system of the image itself, and sampling points from the image that correspond to the point being rendered in order to figure out which color to render at each sample point. We implemented this technique through triangle rasterization by taking the coordinates of the pixel that each triangle vertex sits at and mapping it to the coordinates of the texel corresponding to that vertex. From there, we are able to interpolate the correct coordinate to sample in the texture by using barycentric coordinates. The alpha, beta, and gamma values multiply with the texel coordinates of each of the verticies to produce a single set of `(u, v)` coordinates that are scaled correctly to the texel, which we can then use to grab a color from the texture using any technique we choose.

We implement two techniques – nearest neighbor and bilinear filtering. The nearest neighbor technique simply takes the `(u, v)` pair and rounds `u` and `v` into integers, which are used as the coordinates in the image to sample. This method is simple, and only requires a single sample, but it can create more abrupt changes in the image (especially in viewports that are very different sizes to the image), since one pixel in the frame doesn't directly map to one pixel in the texel, creating weird effects. The bilinear filtering technique works by interpolating between the colors of the four surrounding pixels to `(u, v)`. In practice, this means finding the floored integer components of each value, and using them to sample four texels (the texels to the top left, top right, bottom left, and bottom right of the `(u, v)` pair). Then, we use the floating component to determine how much of the color of each texel we should blend into the final color. We create two intermediate colors made up of the top and bottom sets of texels (blending between left and right based on the position of `u` between the texels). Then, we blend the intermediate colors together based on the position of `v`. This gives us a single color that is representative of the four surrounding pixels that we sampled, which should give a smoother appearance to the overall image, especially in cases where the coordinate systems are not 1:1.

One spot where the difference between these modes is quite visible is at the top of the campanile in Test 6:

Nearest Neighbor with Sample Rate 1

Bilinear with Sample Rate 1

Nearest Neighbor with Sample Rate 16

Bilinear with Sample Rate 16

We see that especially in Sample Rate 1, the borders to the holes in the middle of the Campanile are much more jagged without Bilinear sampling, and the middle hole is much darker and has a very bright orange pixel in the center of it. The right hole also has a very large number of colors, making the image feel busier and making it harder to tell what you're looking at from close up. The bilinear one appears smoother, causing the image to feel slightly more homogenous and cohesive. In the Sample Rate 16 images, there are actually very few differences (and those that are present aren't better in either of the two). This, however, makes sense, since bilinear interpolation and supersampling both help with aliasing, so as we sample more per pixel, the effects of the difference in scale between the original image and the viewport begins to matter less, resulting in less of a difference between the two sampling methods. At 800x600 (the scale that we sampled), there wasn't really any point that stood out as far better when using a sample rate of 16.

Task 6: "Level Sampling" with mipmaps for texture mapping

Level sampling is the technique by which we select between different scaled versions of the image (called mipmaps) in order to sample from an image which has a closer display scale to the actual rendered output. Doing this allows our sample rate to be closer to 1:1, reducing problems with aliasing. We implement this by taking a step forward in each direction (x and y) and finding the coordinates in texture space. We take the difference between these coordinates and our starting coordinates in order to find a rough derivative, which we use to find the specific mipmap level we need by scaling it appropriately and plugging it in to the formula from lecture. Once we have the level, we have three choices of level sampling: not to use it (always use level 0, which is the full image), choose the level nearest to the number we get out of the formula (which is a floating point number), or use linear interpolation to sample both the mipmap level below and the level above, then interpolate between the two results based on how close the actual level was to either of them.

Now, we have developed three different methods of adjusting our sampling technique – pixel sampling, level sampling, and the number of samples per pixel. It's worth spending some time to compare them. Our slowest tool by far is adjusting the number of samples per pixel, since we incur a multiplicative increase in the number of samples on a per-pixel basis. This method is also not incredibly ideal for memory usage in the way that we have implemented it, since it requires our staging array to scale with the size of the number of samples we need, but there are ways to implement this with constant time memory, so this isn't as big of a deal. However, with high cost comes good results, and this often produces the best looking images. The middle child of our techniques is pixel sampling. Using bilinear sampling offers a moderate reduction in aliasing, but it does so quite cheaply, as it only requires us to multiply the number of points we sample by 4, and in turn improves the accuracy of each rendered sample (instead of trying to sample more per pixel). It also allows us to more effectively map textures onto faces, which is harder to do without it. It doesn't use substantially more memory. Finally, level sampling uses a decent amount of memory unavoidably, because it requires multiple different mipmaps to be created. However, it is not much more expensive computationally than pixel sampling and can produce far better results, since it attempts to minimize the difference between the actual size of the image being sampled from and the size of the buffer being outputted to.

For some examples, the image that we chose to focus on is a selfie from Jonah's camera roll from a conference in LA. This image was interesting because the logo in the background has a lot of glare, which we see each of these methods render quite differently.

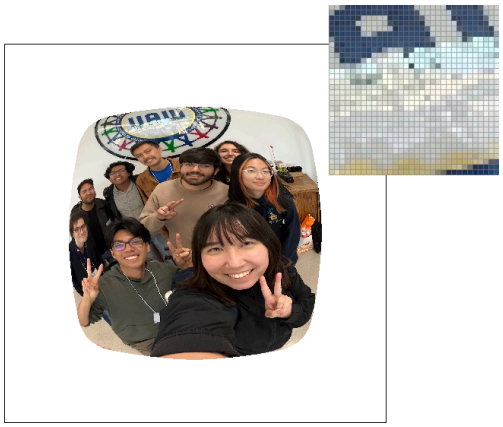


Photo rasterized with L_ZERO P_NEAREST

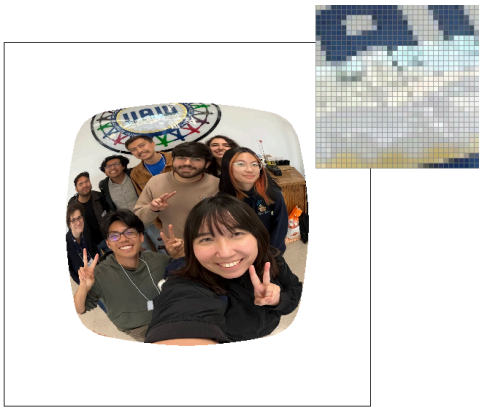


Photo rasterized with L_ZERO P_LINEAR

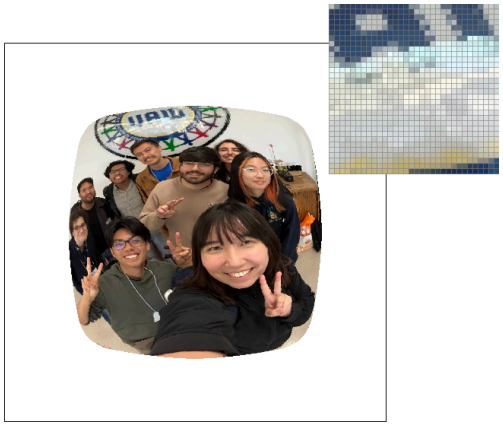


Photo rasterized with L_NEAREST P_NEAREST

Photo rasterized with L_LINEAR P_LINEAR

We see that the “amount of antialiasing” increases from top left to bottom right, where there is very visible aliasing within the glare itself in the top left. There are many different individual pixel colors, and we see some blue from the background cutting through. In the top right, this background is still cutting through, but we see that especially the lower gray color is quite a bit more homogenous. As we move to the bottom right and turn on level sampling, we see a significant improvement. The background is not shining through nearly as much, and while it’s still kind of blocky there are some pretty distinct layers of color emerging. Finally, when we up the level sampling again to linear, we see a very strong smoothing effect, making the entire thing actually feel quite homogenous and subtle. If we zoom out from the pixel view, we see that this is the only one that sort of resembles glare, while the other ones feel like white spots on the wall.