CS 184: Computer Graphics and Imaging, Spring 2018

Project 1: Rasterizer

Jonathan Guo, Wentinn Liao, CS184-wl

Overview

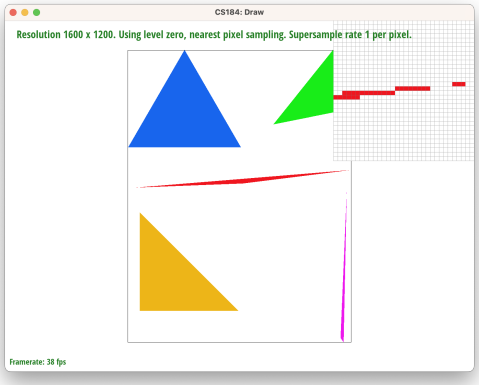
In this project, we implemented a basic rasterizer with efficient solutions to aliasing artifacts for jaggies and texture mapping. It was interesting to see how simple optimizations with algorithms and data structures can greatly improve the visual quality of triangle rasterization.

Section I: Rasterization

Part 1: Rasterizing single-color triangles

Our rasterization algorithm uses a helper function **inside_triangle** that determines if a point is inside a specified triangle. Its implementation uses cross product to determine which side the point is on for each side of the triangle. To avoid handedness issues and to ensure points on the edge are also considered inside the triangle, a point is considered inside if all three cross products are either nonnegative, or nonpositive.

We then compute the minimum and maximum x and y coordinates of each of the triangle vertices in order to compute the bounding box of the triangle. We make no optimizations beyond this and simply iterate through all coordinates in the bounding box checking whether they are inside the triangle or not. In the basic implementation, **fill_pixel** is called if the center of the pixel is within the triangle, and nothing is done otherwise.

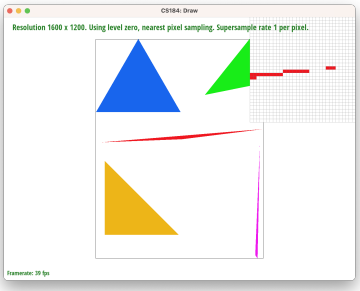


Default *basic/test4.svg*.

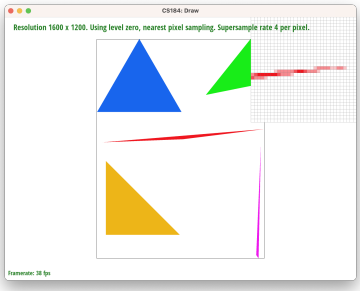
Part 2: Antialiasing triangles

Our supersampling algorithm modifies the basic rasterization algorithm and makes use of the **sample_buffer** variable, but no others. The amount of space allocated is modified for each pixel to have a number of colors equal to the sample rate, stored consecutively. To compute the dimensions of the supersample (n), we call **std::sqrt**, then calculate the increment between supersamples by 1 / n, then iterate through the centers of the supersample pixels accordingly. Supersampled pixels behave like normal pixels in that we write their color to the sample buffer and only modify **rasterize_buffer**.

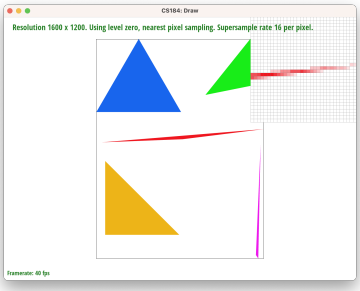
The modification to **rasterize_buffer** simply involves averaging all the color values of the supersampled pixels. This can be done by keeping a running sum of all color values, then dividing by the sample rate at the end. Because of this modification, we also need to adjust **fill_pixel** which is used to rasterize points, since the old implementation now only fills one supersampled pixel. The fix is to iterate through all supersampled pixels corresponding to the one pixel and filling in the same color accordingly.



Supersampling rate 1 *basic/test4.svg*.



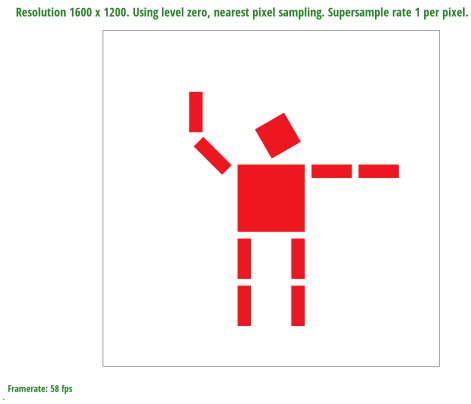
Supersampling rate 4 *basic/test4.svg*.



Supersampling rate 16 *basic/test4.svg*.

We can see that there is a great disconnection on the left when the sampling rate is 1, while 4 is blurred and 16 is more finely blurred. This small scale effect is due to the fact that not all supersamples of a pixel lie inside the triangle and the color assigned to the pixel now depends on the fraction of supersamples that lie in the triangle. On a large scale, this visually reduces aliasing artifacts. We do not see the blur from far away because at a single point, our retina sees an average of light reflecting from a small local region so a uniform blurred color achieves the same effect as would many smaller pixels without supersampling.

Part 3: Transforms

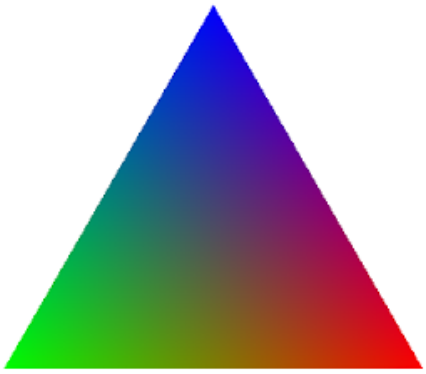


Robot Waving.

Here we tried making our robot wave. This was achieved by rotating the robot's head as well as the robot's arm by various degrees. However, with the rotation of the head/arm, we also had to translate those back to their original position as the rotation took them out of position.

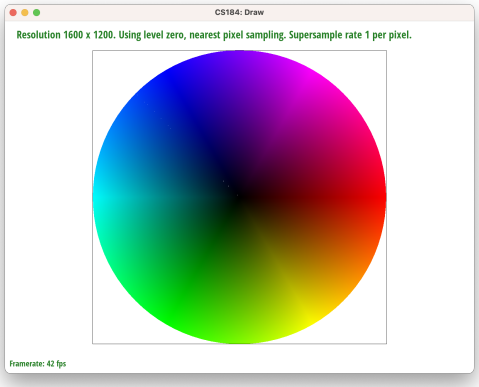
Section II: Sampling

Part 4: Barycentric coordinates



Barycentric interpolation of colors.

Barycentric coordinates express any point as the weighted average of the vertices of a polygon (or polyhedron in multiple dimensions) which allows two dimensional interpolation with three points. Furthermore, the interpolation is also linear, because the line connecting any two points is colored with a linear gradient. This is useful for both color and texture interpolation of the triangle interior.

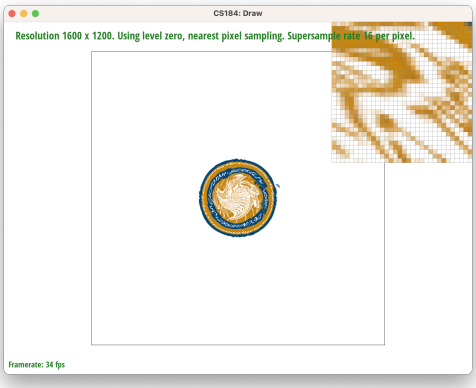


Barycentric interpolation *basic/test7.svg*.

Part 5: "Pixel sampling" for texture mapping

Pixel sampling is a way of obtaining or approximating the desired texture for intermediate pixels. Both nearest and bilinear start with barycentric interpolation of the texel coordinates of the vertices to obtain the appropriate texel coordinates of the desired pixel. Nearest pixel sampling rounds the texel coordinates to the nearest integer and matches the texture at those integer coordinates. In contrast, bilinear pixel sampling takes the texture values at the vertices of the unit square containing the pixel coordinates and makes a linear interpolation in the u and v directions (texel coordinates). This allows a smoother gradient for "large" pixels, i.e. pixels containing many texel lattices.

Nearest pixel sampling rate 1 *texmap/test5.svg*.



Nearest pixel sampling rate 16 *texmap/test5.svg*.

Bilinear pixel sampling rate 1 *texmap/test5.svg*.

Bilinear pixel sampling rate 16 *texmap/test5.svg*.

As seen above, bilinear texture sampling already nearly approximates rate 16 antialiasing of nearest pixel. Bilinear interpolation visually outperforms nearest neighbor when few texture pixels correspond to many pixels, i.e. when texels are stretched across an obtuse triangle. Nearest pixel will show a significant dropoff between texture lattice points, while bilinear interpolation will produce a smooth linear gradient.

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

Level sampling allows us to compute pixel sampling faster and allow for more antialiasing. Instead of going pixel by pixel, level sampling may in some cases jump by many pixels at once (by taking the average which has been precomputed). This allows us to render the images faster. Increasing the number of samples per pixel decreases speed, increases memory usage, and increases antialiasing power since we sample a lot more. Level sampling, on the other hand, is the opposite: as the level number increases, the image becomes blurrier, so it is faster, uses less memory, but also has decreased antialising power. Finally, pixel sampling nearest is slightly faster than bilinear but also is susceptible to aliases.

Level 0, Nearest Pixel *./test6.svg*.

Level 0, Bilinear Pixel *./test6.svg*.

Bilinear Level, Nearest Pixel *./test6.svg*.

Bilinear Level, Bilinear Pixel *./test6.svg*.

Above show the effects of the various pixel and level sampling methods. As seen above, bilinear pixel causes the image to become more blurred, similar to bilinear levels. However, the bilinear level has a way larger effect than the bilinear pixel. This is because the bilinear pixel is only affected by its neighboring pixels, while the bilinear level could cause each pixel to be affected by pixels dozens of pixels away.

Section III: Art Competition

If you are not participating in the optional art competition, don't worry about this section!

Part 7: Draw something interesting!