CS 184: Computer Graphics and Imaging, Spring 2020

Project 1: Rasterizer

Nico Deshler

Overview

In this project we were tasked with implementing a rasterizer - a program that renders images from svg files for which the underlying polygonal building block is the triangle. Three distinct functionalities of this rasterizer were central to the project: 1. Supersampling, 2. Barycentric Coordinate Interpolation, and 3. Texture mapping. In this write-up I present an implementation for this rasterizer and show results demonstrating its capacity to improve renderings through these three features. Supersampling directly conducts anti-aliasing, removing the appearance of jagged edges in the rendering. Barycentric coordinates enabled gradient coloring. Texture mapping made use of Mipmaps and bilinear interpolation to efficiently apply an anti-aliased target texture to a scene. Despite substantial debugging toils, I was deeply amazed and fulfilled with this project. These three features drastically improved the render quality and widened the artistic space for 2D image. The qualitative results presented herein are a testament to such claims.

Section I: Rasterization

Part 1: Rasterizing single-color triangles

Given a set of three screenspace coordinates corresponding to the vertices of a triangle and an rgb color vector as inputs, rasterizing involves identifying which pixels lie within the triangle and coloring them accordingly. A high level description of the procedure implemented here for triangle rasterization is as follows. Define a bounding box circumscribing the screenspace coordinates. For each pixel in this box, determine whether it lies within the triangle. If so, populate the pixel value with the color. To ensure that the input points systematically to adhered a clockwise enumeration scheme, I defined the 2D coordinate points in a 3D vector, taking advantage of the extra dimension and the property of vector cross-products to verify the orientation as per the right-hand rule. If a triplet of points did not follow this convention, I simply swapped two of them which guarantees proper enumeration. From here, determining whether a screenspace point in the search box was inside the triangle involved performing a line test for all three edges of the triangle, a technique discussed in lecture. The dot product between the screenspace coordinate (relative to some reference vertice) and an edge normal vector indicates whether the coordinate is located above, below, or on top of an edge. Performing this check for each edge determines whether the point lies within the triangle. Rasterization for a sparse collection of triangles is shown in Figure 1.

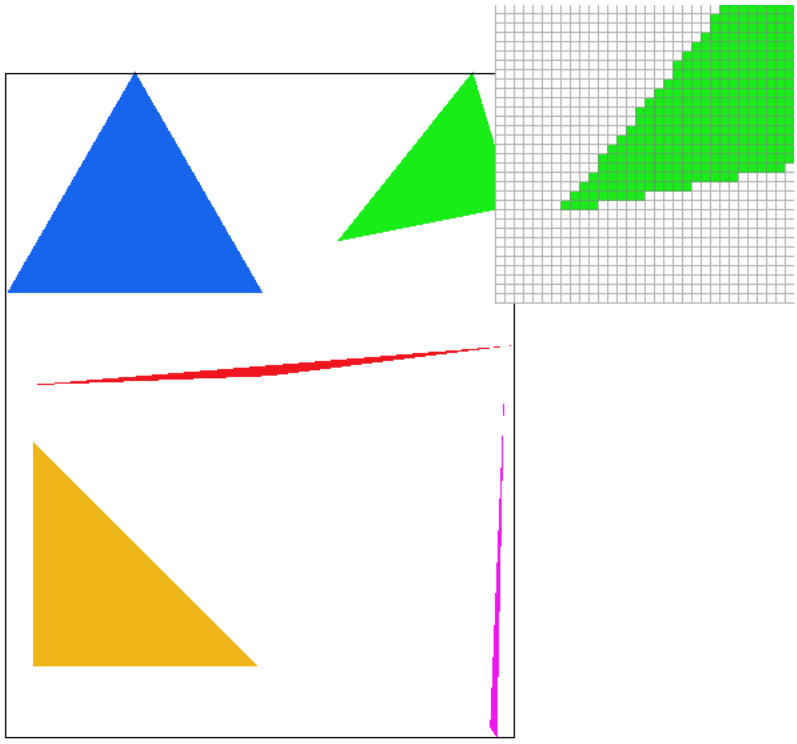

Figure 1: Triangle rasterization for solid triangles of different forms and colors. The toggle parameters are set to default (i.e. 1x sample rate, nearest pixel sampling, etc.). Note that this basic implementation suffers from aliasing (jaggies) evident in the zoomed view of the green triangle corner.

Part 2: Antialiasing triangles

As discussed in lecture, supersampling is equivalent to convolving our image with a 2D box function. This in turn is equivalent to filtering out high-frequency content in our rasterized scene. Consequently, the effect of supersampling is to antialias the image. The process of supersampling can be described as follows. Begin by subdividing the coordinate increment between each pixel by some some given supersampling rate. For each sub-pixel coordinate in this uniform division perform the inside-triangle test. The final color assigned to the pixel becomes the pixel's original color weighted by the proportion of sub-pixel coordinates that were found to fall inside the triangle. Following recommendations from the project spec, I decided to implement supersampling by creating a dedidcated data structure (vector of color objects) for storing the upsampled scene. While there is a memory usage and write cost associated with including this supersample buffer compared to simply computing the color weighting on-the-fly for each pixel as they are traversed, having the supersample buffer proved convenient for expanding the capabilities of the rasterizer in later parts of the project like pixel color interpolation. Additionally, the supersample buffer required extra maintenance. For instance, the rasterizer provides a GUI through which the supersample rate parameter can be updated. Thus the rasterization pipeline had to support dynamic memory allocation for the buffer. Finally, the resolve_to_framebuffer() method was necessary for transferring the averaged supersamples into the frame buffer.

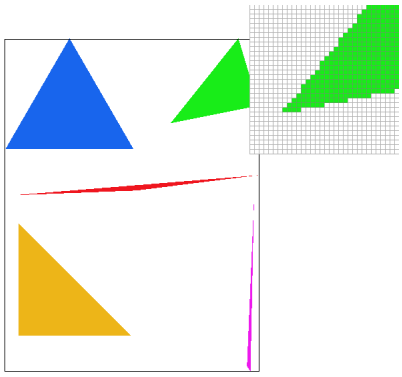

Figure 2.1a: Sample Rate = 1. 'Jaggies' clearly present in zoomed view of green triangle corner.

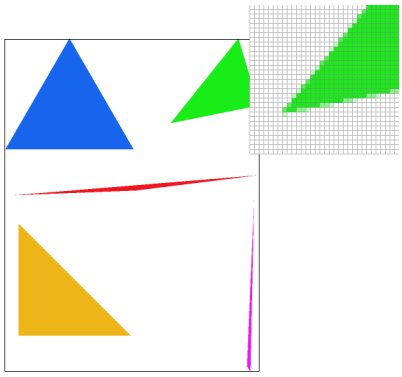

Figure 2.1b: Sample Rate = 4. Introducing some supersampling immediately softens the jagged edges.

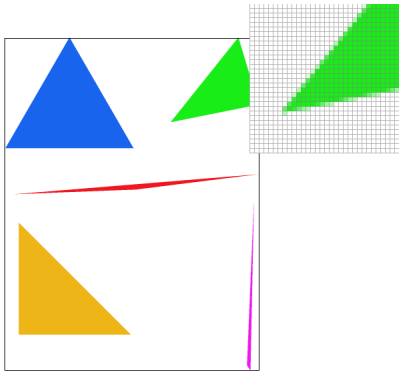

Figure 2.1c: Sample Rate = 9. Higher supersampling further reduces aliasing.

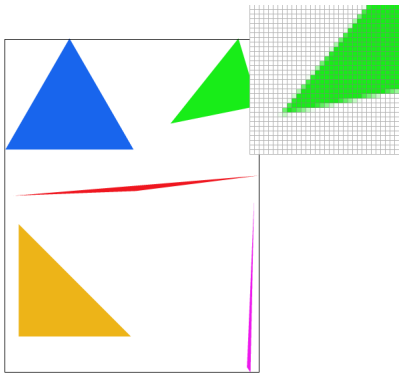

Figure 2.1d: Sample Rate = 16.

The **Figure 2.1** quad-chart shows how supersampling reduces jaggies to display a more natural rendering of edges that are not aligned with the cartesian pixel grid. This is qualitatively apparent in the zoomed view of the green triangle corner. As the sample rate is increased from 1 to 16 the pixelation of the edge dwindles and we resolve a more natural-looking edge. Assuming the sampling frequency is 1 sample per pixel, the Nyquist rate for any scene becomes 1/2 cycles per pixel. Mathematically, the Fourier decomposition of an edge contains frequencies well above the Nyquist rate. This is the origin of aliasing. By smoothing the sharpness via supersampling (i.e. making the change in color more gradual) the high-frequency content originally describing the edge becomes supressed. Further improvements from supersampling are shown in **Figure 2.2** below. Specifically, we observe the elimination of visual artifacts such as Moire patterns with increased sample rates.

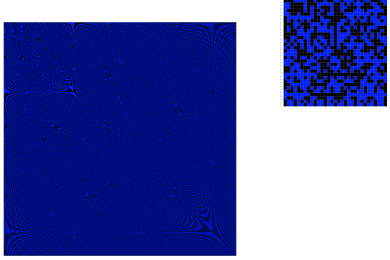

Figure 2.2a: Sample Rate = 1. Without supersampling Moire patterns are abundant.

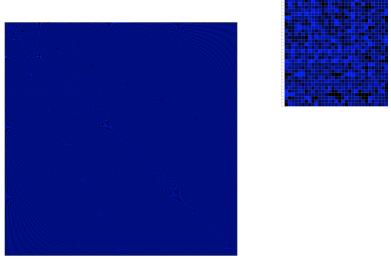

Figure 2.2b: Sample Rate = 4. Introducing some supersampling begins filtering the high-frequency content and reduces aliasing

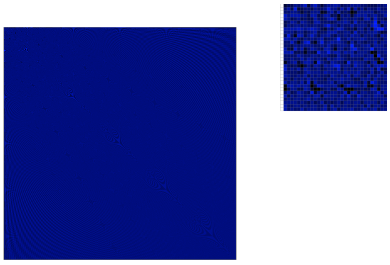

Figure 2.2c: Sample Rate = 9. Higher supersampling further reduces aliasing.

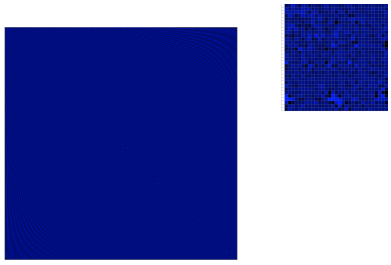

Figure 2.2d: Sample Rate = 16. Highest supersampling largely eliminates Moire Patterns and image artifacts.

Part 3: Transforms

Here I simply decided to expose cubeman's true identity: SPIDERCUBEMAN! To do this I added 16 additional triangles comprising the 4 middle legs. For each of these legs I create a new group in the hierarchy of transformations so as to preserve the relative positions of each leg segment with respect to each other and the torso. Finally I shaded the limbs for artistic effect. In the extra credit portion of this project, I extend the use of these transformations in a generative script for making SVG files.

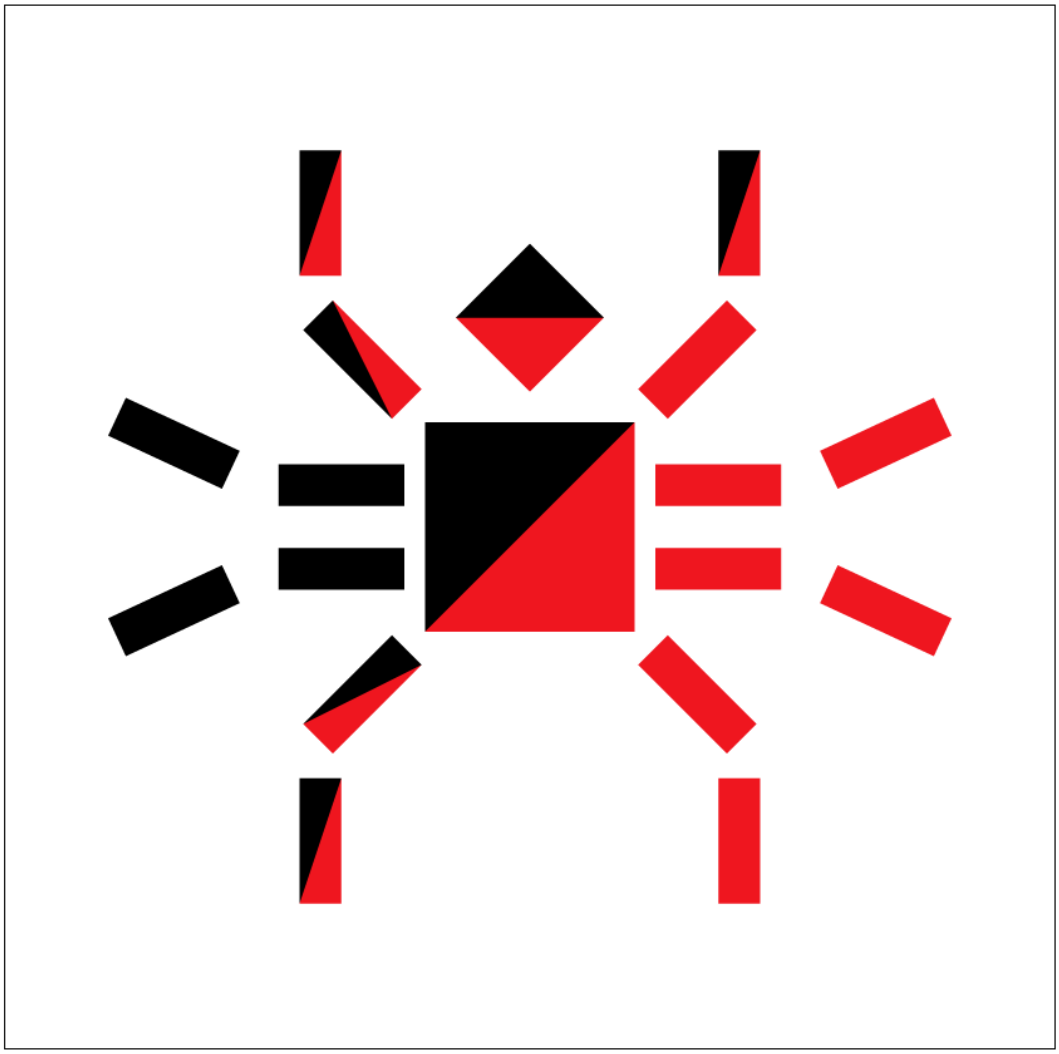

Figure 3: A basic geometric assembly of a spider with shadowing generated using translations, rotations, and scalings in homogenous coordinates. Body part locations for the insect are specified in a hierarchal manner so as to impose relative placement constraints.

Section II: Sampling

FIXME

Part 4: Barycentric coordinates

Barycentric coordinates, as used here, are a coordinate system defined through three reference values. To introduce a barycentric coordinate is to define a weighted sum of the reference values. Moving along the alpha, beta, and gamma axes individually has the effect of assigning greater weight to reference values A, B, and C respectively. Since we constrain the sum of the barycentric coordinates to be unity, we are left with 2 degrees of freedom. As such this scheme is particularly useful for smoothly assigning values to the interior of a triangle defined in some feature space (e.g. rgb color space, uv texture space, etc.) like in Figure 4a:.

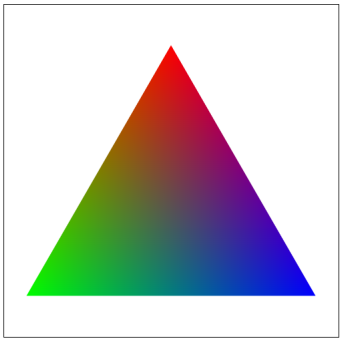

Figure 4a: A single triangle colored using barycentric coordinates. The reference values are the unitary RGB channels (1,0,0), (0,1,0), (0,0,1) located at each of the vertices.

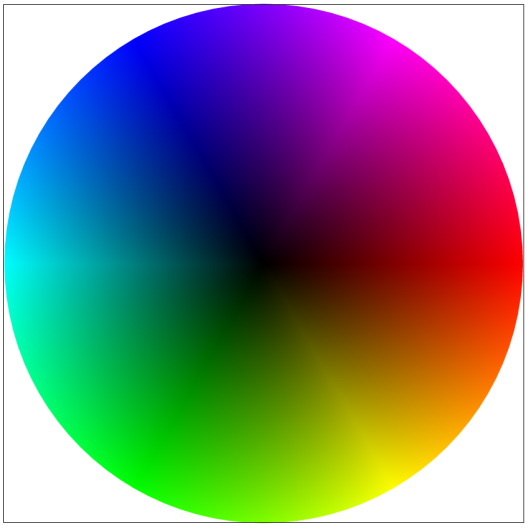

Figure 4a: The figure shows a colorwheel composed of radially fanning acute isosceles triangles. Barycentric coordinates are used to smoothly interpolate the color values within each triangle given that the colors at the triangle vertices are defined.

Part 5: "Pixel sampling" for texture mapping

For texture mapping, the task of identifying which pixel from the texture to sample for a given screenspace position is a challenge. This is because the map between the screenspace and the texturespace may be highly non-linear. Hence the appropriate value to pull from the texture for a given screenspace pixel can be unclear. Pixel sampling methods like nearest-neighbor sampling and bilinear sampling attempt to tackle this difficulty. To implement nearest-neighbor pixel sampling for a given UV normalized coordinate, I simply rescaled the normalized coordinate by the texture map dimensions and identified the closest texel. In contrast, to implement bilinear sampling I located the four nearest texels in the texture map and performed a series of 1D linear interpolations(LERP) over 2 axes. First I LERPed the four texels along the horizontal axis, collapsing them down to two vertical texels. Subsequently I LERPed these remaining two vertical texels to produce a color value that preserves information about variations in the immediate surroundings. Figure 5 demonstrates differences between these two pixel sampling methods. The zoomed view of high-frequency content in the texture mapping (narrow feathers on the neck of the parrot) illustrate how bilinear sampling surpasses nearest-neighbor. The pixelation of the nearest-neighbor method hides the streak-like characteristics of feathers evident in the bilinear sampling images. Supersampling improves the quality of both sampling methods. In theory, the largest differences between the two methods should be most prominent when the screen space is magnified relative to the texture space (i.e. small L metric) at the sample point. Then the nearest-neighbor approach will effectively magnify the pixelation in texel space into screen space since 1 texel maps to multiple pixels under a magnification regime.

Figure 5a: Nearest-Neighbor(sample rate = 1).

Figure 5b: Bilinear (sample rate = 1).

Figure 5c: Nearest-Neighbor (sample rate = 16).

Figure 5b: Bilinear (sample rate = 16).

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

Level sampling becomes important under the framework of Mipmaps and attempts to address issues associated with minification. This corresponds the regime where a single pixel in screen space spans several texels in texture space. This poses a serious challenge for mapping coherent visuals onto screen space. If the texture has sufficiently high frequency content in a minified region, moving by a single pixel could land us on a texel very distant from the starting texel with unrelated irreconcilable color content. Mipmap formulate a collection of downsampled (texel-averaged) textures from which sensible texel values can be passed back to screen space pixels. The process identifying the proper Mipmap level to retrieve texel values from for a given pixel in screen space is called level sampling. Findamentally, I implemented level sampling by using the columns of the map's Jacobian matrix. The magnitude of the column vectors of the Jacobian provide an approximate measure of the local pixel footprint in texture space. From this information, I was able to calculate which mipmap level was best suited coloring a certain part of screenspace based on the degree of minification induced by the map at that local region. Mipmaps are worthwhile implementation because they only require 1/3 the amount of additional memory compared to the texture map alone yet offer impressive antialiasing benefits and computational speedups. Anecdotally, the rasterizer performed comparatively slowly when the pixel sampling method was set to default on level 0. As shown in Figure 6 Nearest-neighbor sampling paired with level sampling demonstrates marked improvements in render quality through antialiasing.

Figure 6a: LSM = Level 0, PSM = Nearest.

Figure 6b: LSM = Level 0, PSM = Bilinear.

Figure 6c: LSM = Nearest, PSM = Nearest.

Figure 6d: LSM = Nearest, PSM = Bilinear.

Section III: Art Competition

Part 7: Draw something interesting!