CSC4140 Assignment 3 and 4

Computer Graphics

March 19, 2022

Mid-term

Strict Due Date: 11:59PM, March 19^{th}th , 2022

Student ID: 119010344

Student Name: Xiao Nan

This assignment represents my own work in accordance with University regulations.

Signature: Xiao Nan

1 Overview

In this assignment I implemented basic triangle rasterizations and anti-aliasing operations. Specifically, I rasterized a single-colored triangle and performed anti-aliasing by super-sampling using SSAA(Super-Sampling Anti-Aliasing). I also implemented basic transformations to generate an image of robot in 2D plane. In the forth part, I implemented Barycentric color interpolation during the rasterization of a triangle. In the fifth part, I implemented different texture sampling methods, such as bi-linear sampling and nearest-pixel sampling, for the color of a given triangle during rasterization by utilizing the Barycentric coordinates of those points. In the sixth part, I implemented sampling in different mipmap levels using different level sampling methods, such as 0-level sampling, nearest-level sampling as well as tri-linear sampling.

2 Draw a single color triangle

In this part I rasterized a single-colored triangle on the frame buffer. The implementation is as efficient as checking within a bounding-box and for each pixel's center location, I performed point-in-triangle tests using Barycentric coordinates. The boundary points of the triangle are included by including the edge cases of the Barycentric coordinates (e.g. points with Barycentric coordinates (α, β, γ), satisfying $\alpha = 0$ or $\beta = 0$ or $\gamma = 0$, are included). Also, the tests are required only for points in the given bounding box containing the triangle, which reduces the time complexity for those tests. In addition, the effects of rasterizing order (either clock-wise or counter-clockwise) on point-in-triangle are avoided with the help of the Barycentric coordinates.

2.1 Point-in-triangle tests

In my implementation, I simply check through every pixel in the bounding box covered by the triangle obtained by taking floor() and ceil() to the input coordinates, respectively. I test whether or not the pixel center lies in the triangle using the non-negativity of the Barycentric coordinates. If yes then I fill the pixel in the supersample buffer (not using sampling rate yet) and later resolve to the frame buffer. Such greedy stretagy improves in time compared to scanning through the whole image by reducing the area to be scanned.

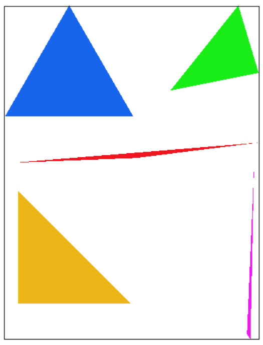


Figure 1: Triangle rasterization

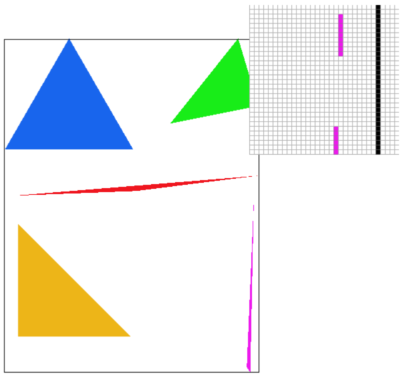
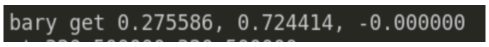


Figure 2: Triangle rasterization under inspector

The results for **basic/test4.svg** are shown in figure 1, 2, corresponding to the screenshot given default parameters and pixel inspector placed in the location where the triangle is not fully rasterized. The reason, after digging down into the implementation, shown in figure 3, is that the Barycentric coordinates computed for the edge of the sharp triangle is very closed to 0 (due to precision errors), making it hard to distinguish those points from points in triangle.



```
bary get 0.275586, 0.724414, -0.000000
```

Figure 3: Rounding error

In order to solve the precision problem, I bring in a degree of tolerance for the point-in-triangle test (Barycentric coordinates' test), which allows a Barycentric coordinate value that is closed enough to 0 to be considered as being inside the triangle.

3 Anti-aliasing by supersampling

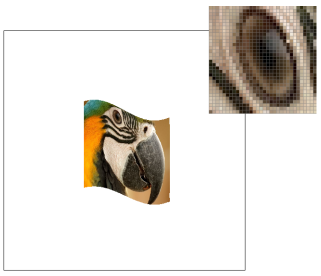
In this part I implemented SSAA(Super-Sampling Anti-Aliasing). Specifically, I first rasterize an svg image with a higher resolution, and then downsample from the supersampling buffer using the box filter in order to obtain the frame.

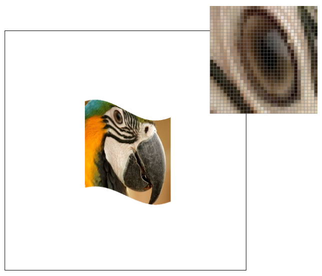
Super-sampling is a common technique used for performing image anti-aliasing. It is useful in that it increases the sampling frequency, hence filtering out the low-frequency contents in the original input signal and reduces the aliasing artifacts caused by the aliasing spectrum in the frequency domain. It essentially is a low-pass filter that removes the high-frequency part of the signal that causes aliasing easily.

In the rasterization pipeline, I added additional "for" loops inside the size-1 pixel iteration in order to sample sub-pixels within an original pixel. I used a supersampling buffer to temporarily store the resulting color values for the sub-pixels obtained by super-sampling. After the sampling process is finished, I call the **resolve-to-buffer** function to box-filter the supersampling buffer and update the results in the displayed frame buffer.

3.1 Results

The comparisons among different screenshots under different sampling rates are shown in figure 4, where generally the displayed pixel image becomes more and more smooth at objects' edges as the sampling rate increases. This can be due to the fact that box-filtering improves the continuity of the pixel color values, especially at the edges where the color values had a great jump before filtering. The eye figures in 4 provided by the pixel inspector further illustrated the phenomena, where the pixel values around the originally sharp edge were averaged by the box filter and hence making it seem more smooth.


(a) Sample rate = 1


(b) Sample rate = 4

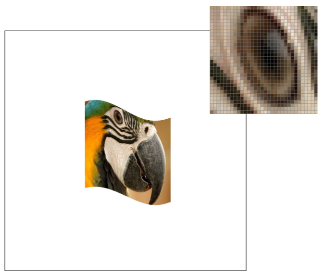

(c) Sample rate = 16

Figure 4: Results under different sampling rates

4 Transformations

In transformation I implemented the transformation matrices for rendering a robot with basic transformations on its building blocks. The results of the waving and running robots are shown in figure 5 and 6, respectively. In order to render the waving robot, I tried adding some rotations and translations to the robot's right leg blocks in the svg file. Also, in order to render the little yellow boy wearing a yellow clothe with red sleeves, I tried adding translations and rotations on head, arm and legs. I modified the color of the little boy's legs, forearms and body in order to render the effect of the Minion. In addition, the boy's head is rotated a bit to show his happiness.

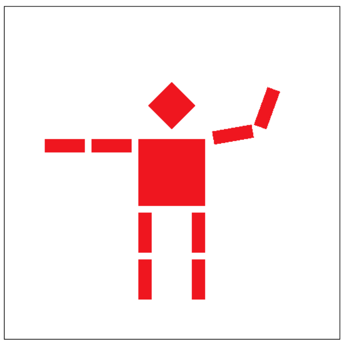


Figure 5: Waving boy

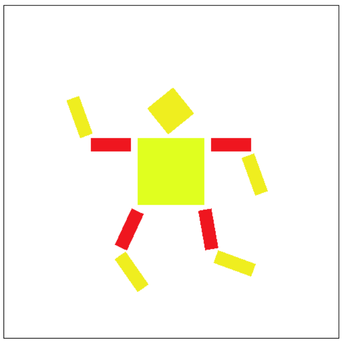


Figure 6: Running boy

5 Barycentric color interpolation

In this part I implemented color interpolation for a given triangle using Barycentric coordinates. Barycentric coordinates is a coordinate system in which a point's coordinate is determined by the weights in the weighted sum of the three vertices' interpolated values. Physically, it can indicate the mass to put on the vertices in order to make the point the center of gravity of the triangle. Mathematically, it can also indicate the ratio between the triangular area of the triangle formed by the point together with the other two opposite vertices, and the area of the whole triangle. Figure 7 gives an example of a triangle interpolated using Barycentric coordinates, where the color values of the vertices A, B, C are (0, 0, 1), (0, 1, 0), (1, 0, 0) in normalized RGB form, respectively (A: the upper-left vertex, B: the upper-right vertex, C: the lower vertex). We may see that the closer the pixels are to a given vertex (say, vertex A), the more likely that the pixel would have a color (say blue) similar to the vertex. This is because in Barycentric coordinates, such pixel has a greater color component on the corresponding vertex (A, that is to say, $\frac{\Delta DBC}{\Delta ABC}$ is large enough if we denote the point to be D).

The results of rendering a colored circle are shown in figure 8 with sample rate 1 and default viewing parameters.

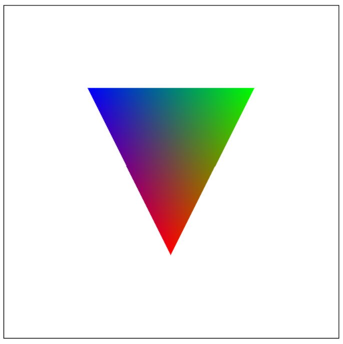


Figure 7: Colored triangle

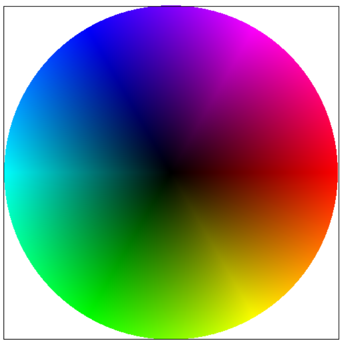


Figure 8: Colored circle

6 Pixel sampling for texture mapping

In this part I implemented pixel sampling using texture mapping. Specifically, I sampled the RGB color values of the pixels by using texture mapping to map the pixel center points (x, y) to a given coordinate in the texture space (u, v) and sampling the value at the 0-th level mip (for this task). I implemented two pixel sampling methods: nearest sampling and bilinear sampling. The nearest sampling method samples from the texel (pixel in the texture space) that is closest to the point in texture space mapped from the screen space pixel center point. In this method, the RGB color values of the texel is treated as the sampled color values directly. The bilinear sampling method samples from the 4 nearest texels in the texture space and performs bilinear interpolation: The color values from the 4 nearest texels interpolated to the sample point have weights proportional to the distance between the sample point to those texels. The bilinear sampling produces smoother results in that it averages the sampled texel values nearby, making the color displayed in the screen space more continuous. The nearest sampling method may result in aliasing artifacts since different pixels may map to the same texel (by nearest rounding), resulting in jumps on color values in the screen space.

Figure 9 given by the pixel inspectors shows how the resulting pixel image sampled using bilinear sampling method defeats that sampled using nearest sampling method. We may observe that the "Ber" letters in the bilinear sampling resulting image are more clear and continuous (more balanced in color) than that given by the nearest pixel sampling method.

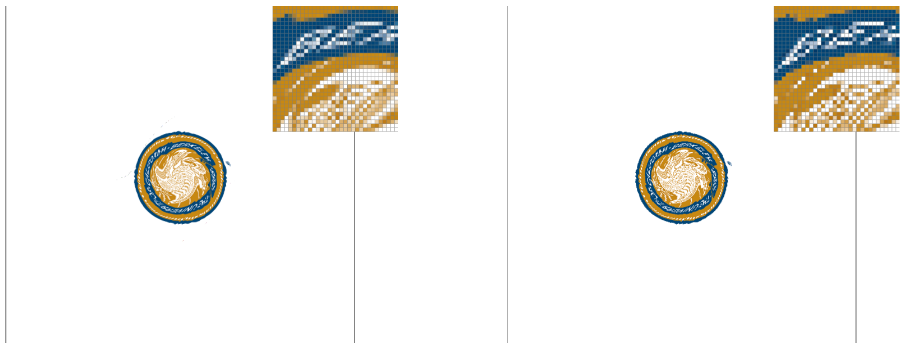
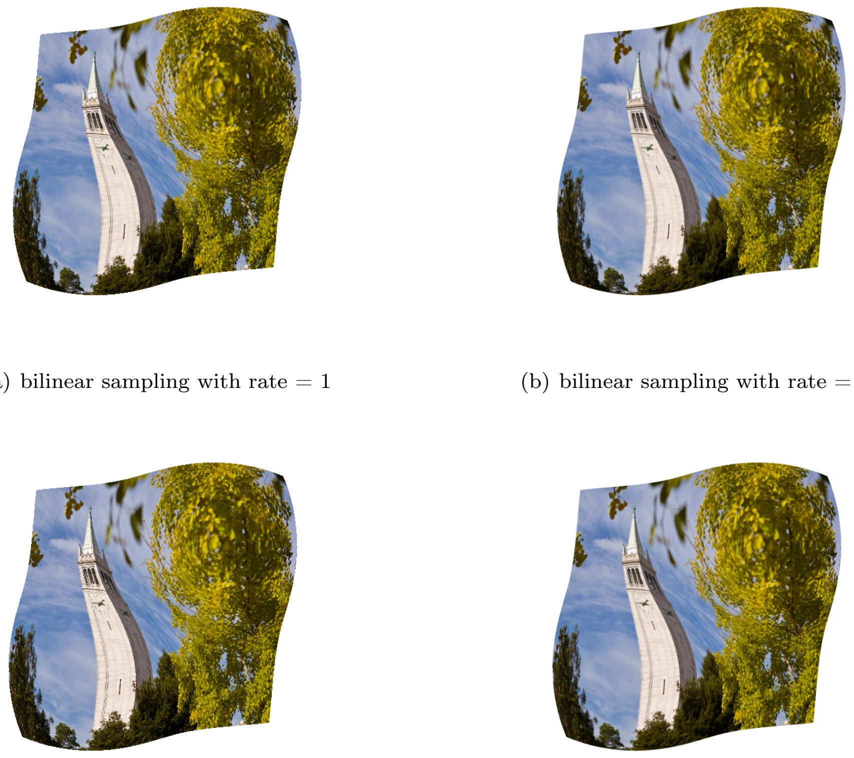

(a) bilinear sampling (b) nearest-pixel sampling

Figure 9: Different pixel sampling modes, where bilinear wins

Figure 10 shows 4 different resulting images in different sampling modes: One using nearest sampling method with sampling rate 1, one using nearest sampling method with sampling rate 16, one using bilinear sampling method with sampling rate 1, and one using bilinear sampling method with sampling rate 16. We may observe from the comparion between figure 10(c) and 10(d) that when the nearest sampling method is used, as the sampling rate increases, the resulting image becomes smoother (in that the details in the clock tower becomes more vague and the aliasing artifacts are removed). The comparion between 10(a) and 10(b) yields similar results (removing clock tower aliasing artifacts by calculating averaged values nearby). We may also compare 10(c) and 10(a) to see that bilinear sampling yields smoother results than that given by nearest sampling under sampling rate 1. The two methods may output very differently (matching or not matching what we want) when the object we're trying to observe has different characteristics: On the one hand, the bilinear sampling works better when we want to observe continuous objects from an image. For example, the cloud in figure 10 in its nature has a closed color to the sky and a continuous, vague edge under the sky, and in this case bilinear sampling produces an image that is far more continuous than that produced by nearest sampling (which contains obvious jaggies, and is unnatural). On the other hand, when we want to display an object with high contrast to the environment (e.g. a red box on the white floor), then the nearest sampling may be more useful in that it preserves the high contrast and makes the object more real than that given by bilinear sampling.



(a) bilinear sampling with rate = 1

(b) bilinear sampling with rate = 16

(c) nearest sampling with rate = 1

(d) nearest sampling with rate = 16

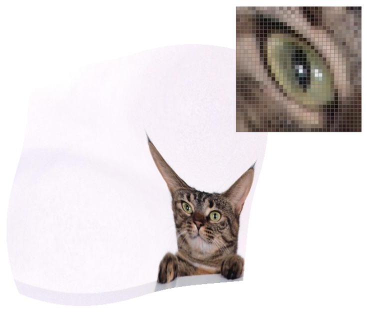
Figure 10: Results for sampling in different pixel modes

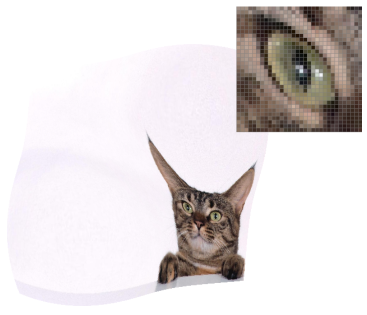
7 Mipmapping

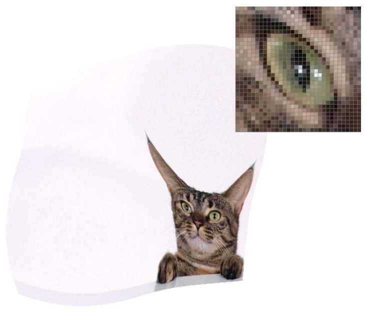
In this part I implemented mip-map level sampling for rasterizing a textured triangle. In level sampling, a number of mipmap levels (textures of different resolutions) are generated from the full-resolution texture in advance, and the texture sampling in this case takes place by sampling values from the nearest or from different levels (depending on level sampling stretagy) and averaging the values in order to avoid aliasing artifacts and increase rendering speed. I implemented it by first calculating and comparing the norm of the differential vectors along u, v directions to update the sampling parameters, and then checking the appropriate sampling level in $get-level$ function using the Sample parameters passed in. After getting the appropriate level, I determine the corresponding sampling methods (whether to sample from 0-th level or from the nearest appropriate level) by checking $sp.lsm$ and $sp.psm$ respectively. Specifically, when the level is not an integer, I sample the value using the weighted sum of the sampled values from the adjacent two levels nearby.

In practice, level sampling methods are good ways for reducing aliasing artifacts since it allows sampling from different levels, making the filtering process more averaged. It does brings costs on speed and memory space usage (for storing different mipmap level textures), but the extra costs are affordable (in that the costs are increased in logarithmic level, which compared to linear level is acceptable). Pixel sampling methods are friendly in memory usage and speed in that it only interpolates within one particular level, but the disvantage is that its anti-aliasing ability is worse in that a single-leveled filter is limited in filtering ability. Increasing supersampling rate is the most effective one in its anti-aliasing power in that it samples more pixels, but the speed and memory usage costs are also significant compared to the previous two techniques.

The results after rendering my png file 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, respectively, are shown in figure 11. comparing 11(a) and 11(b) we may observe that using the same nearest level sampling method, the bilinear pixel sampling method yields an smoother image with better continuity. The comparison between figure 11(c) and 11(d) yield the same conclusion when the sampling level is set 0. On the other hand, we observe from the comparisons between figure 11(d) and 11(b) that sampling from the nearest level does yields better results than always sampling from level 0, though it seems that the effect brought by using the bilinear pixel sampling method is larger than that brought by using nearest level sampling method.


(a) L-NEAREST and P-LINEAR


(b) L-NEAREST and P-NEAREST


(c) L-ZERO and P-LINEAR

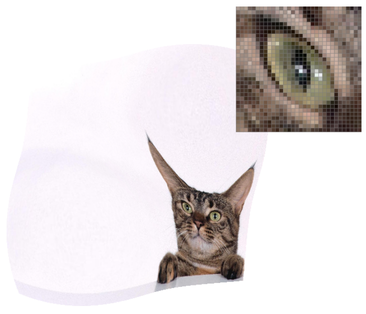

(d) L-ZERO and P-NEAREST

Figure 11: Results for sampling in different modes