CSC4140 Assignment III & IV

Computer Graphics

March 19, 2022

Transformation

This assignment is 18%(with 5.4 extra credit) of the total mark.

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

Student ID: 118010141

Student Name: Jingyu Li

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

Signature: JINGYU LI

Contents

1 Overview

This midterm assignment aims to realize a rasterizer and finish a series of tasks. Rasterization means assigning the color values, RGB, to pixels stored in the frame buffer in order to display projected images on the screen. A screen is formed by several pixel, which is a square with certain property (color). In this project, the screen resolution is 800*600, so we have 480,000 pixels to check and fill for each image. As for rasterization, given a point, one can directly fill color to the point coordinate in the frame buffer, while given two points at the top of a line, one can also fill the color one by one. Triangle is the primitive geometry in computer graphics. All the beautiful images are formed by many little triangles. So our task focuses on how to deal with the triangles.

Starting from rasterizing a single color triangle, we move on to improve the rasterizing effect by anti-aliasing, and then perform some transformation, and then use barycentric coordinates to obtain a smooth color transition in an image. In texture mapping part, we make use of barycentric coordinates to map pixel and texel, filling color with that in texture. To further improve the effect, we use mipmapping to get better color average. We have six combinations of sampling methods to implement. This is the road-map of this assignment.

2 Task 1: Draw a single color triangles

2.1 Design and Implementation

Task 1 requires us to implement the basic triangle rasterization. Given three vertices of a triangle and a specific color, one needs to fill all pixels inside this triangle with that color.

To rasterize efficiently, I first limit the pixel range to a bounding box of the triangle, that is, find the maximum and minimum values of x-coordinate and y-coordinate, and iterate all the pixels in this square area. Then for each pixel, first locate the sample point, which is the center pixel (integer + 0.5), perform point-in triangle tests to check whether this point is inside the triangle. This test is done by making use of cross product: assume the triangle is ABC and the point is P, check the sign of $\boldsymbol{AP} \times \boldsymbol{AB}$ and $\boldsymbol{BP} \times \boldsymbol{BC}$ and $\boldsymbol{CP} \times \boldsymbol{CA}$, if all the three have the same sign (positive or negative), then the point lies inside the triangle. The equation sign = should be picked in the judgment since triangle edge also counts.

```
float L0 = -(x - x0) * (y1 - y0) + (y - y0) * (x1 - x0);
float L1 = -(x - x1) * (y2 - y1) + (y - y1) * (x2 - x1);
float L2 = -(x - x2) * (y0 - y2) + (y - y2) * (x0 - x2);
if ((L0 >= 0 && L1 >= 0 && L2 >= 0) || (L0 <= 0 && L1 <= 0 && L2 <= 0)) {
  return true;
}
```

Figure 1: Triangle Test

If the pixel passes the test, invoke the given rasterize_point() to fill color to pixel, this function will help round the float number and check image boundary. In task 1, we only need to sample once per pixel, so the code is relatively simple, and this part in **RasterizerImp::rasterize_triangle()** function is commented and will be covered by task 2. By now, we are able to rasterize a single color polygons formed by basic triangle.

The rasterization algorithm is as efficient as the bounding box approach (exactly it is), we only consider pixels in the smallest rectangle area covering the triangle. There is better algorithm that just requires to consider less pixel, row by row, starting from the left edge to the right edge, but I fail to implement it since it's quite complex and troublesome.

2.2 Results and Analysis

Here shows the results of triangle rasterization and several more complete images.

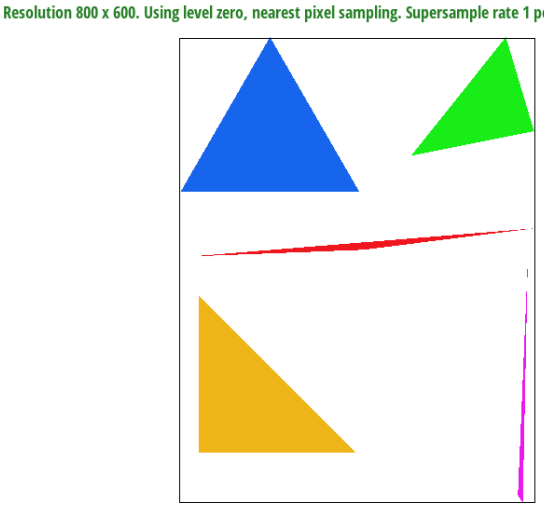

Figure 2: Test4 Task1

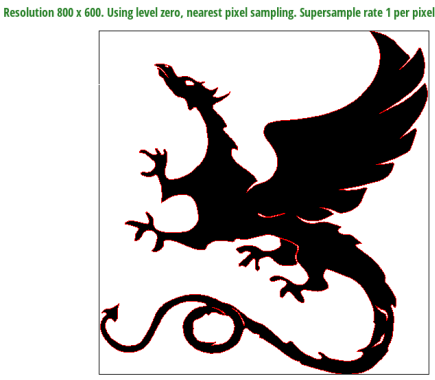

Figure 3: Test3 Task1

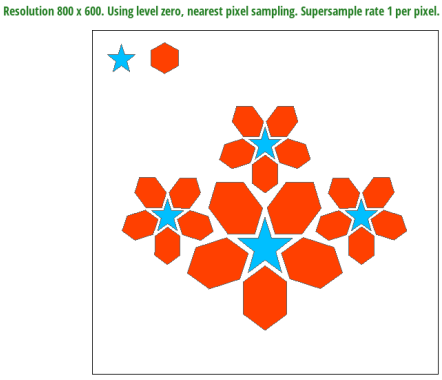

Figure 4: Test6 Task1

3 Task 2: Anti-aliasing

3.1 Design and Implementation

Task 2 requires us to use super-sampling method for anti-aliasing. The main idea of super sampling is first sample with a higher resolution and then down-sampling to the frame buffer. We divide a pixel into several grid area and sample each grid, then average the color according to the ratio of grid number to the sample rate. The purpose is to obtain a smooth edge and to reduce the jaggies effect.

The algorithm is as follows, as task 1, first obtain the rectangle bounding box of the triangle, then iterate pixel by pixel, in each pixel, iterate **sample_rate** times, calculating each grid center coordinates according to $\sqrt{sample\_rate}$, for example, if the sample_rate is 9, then go through the 3×3 area in the pixel, for each sample point, perform point-in triangle test, record the number of points **count** passing the test. If the number is larger than 0, it means this pixel should be rasterized instead of keeping its background color. Calculate the averaging ratio, which is $\frac{count}{sample\_rate}$, then obtain the color using this ratio to multiply given color and $(1 - ratio)$ to multiply background color (original value in the sample_buffer), add these two we can get the super sampling color result.

The super sampling makes sense since it blurred the triangle edge by averaging the color and background color, so when visualising, the jaggies effect will reduce. We can use + and - to adjust the sample_rate (1,4,9,16), the higher the rate, the more vague the edge will be.

```
if (count > 0) {
  Color c;
  Color bg = sample_buffer[y * width + x];
  if (bg != Color::White) {…
  float ratio = count / sample_rate;
  c.r = color.r * ratio + bg.r * (1 - ratio);
  c.g = color.g * ratio + bg.g * (1 - ratio);
  c.b = color.b * ratio + bg.b * (1 - ratio);
  rasterize_point(x, y, c);
}
```

Figure 5: Super Sampling

The rasterization pipeline is as follows:

i. SVG parser parses the .svg file to SVG class representation.

ii. The renderer calls SVG::draw to redraw the graph.

iii. SVG::draw calls rasterization functions (point, line, triangle) to draw primitive by primitive. **Redraw will clear the buffer contents, erasing all the values first to update / resize**

image size dynamically

iv. Redraw function calls the line rasterization to draw 4 lines as the boundary of the graph.

v. Resolve the color value to frame buffer. **Convert the complete color class to the impact data type of 8-bit**

vi. Display on the screen or write to a file.

Super-sampling makes changes in the *iii* step, which is the rasterization triangle process.

3.2 Results and Analysis

Figure 6: Test4 Task2 Rate 1

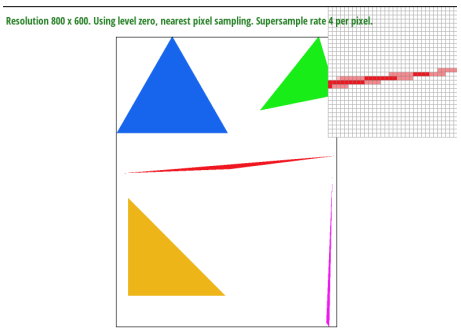

Figure 7: Test4 Task2 Rate 4

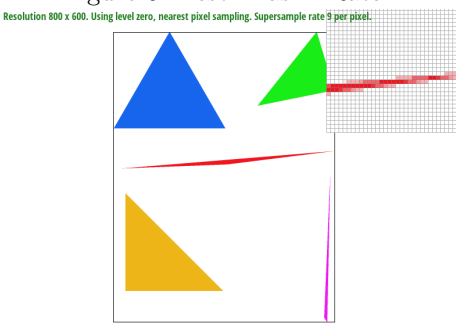

Figure 8: Test4 Task2 Rate 9

Figure 9: Test4 Task2 Rate 16

As can be seen from these images, when the sample rate is 1 (original state), the jaggies effect is obvious, and the inspector shows the very steep skinny triangle corner. As the sample rate increases, the corner becomes smoother and smoother, generating a better rasterization result. Seeing from the far, one can hardly recognize the jaggies. The higher the sample rate, the better the result. The reason is super sampling, where jaggies pixel color is averaged, making its surrounding pixels filled with shallower color instead of empty, thus it can give out the effect of a smooth edge.

Here shows two magnification images' comparison.

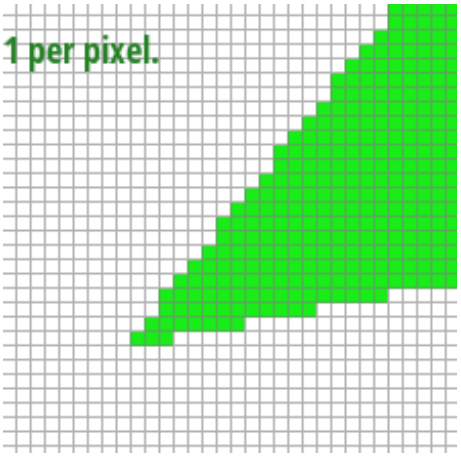

Figure 10: Green corner 1

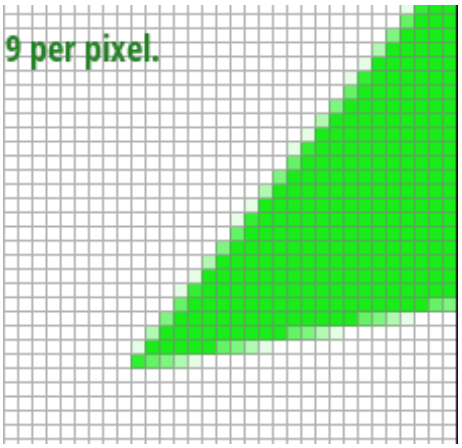

Figure 11: Green corner 9

3.3 Extra

The extra credit here lies in the implementation of SSAA, **box filter**, **convolution approach**. The main idea is similar with the super sampling, but averaging across pixel. First perform rasterization once (the same as task 1) to obtain a primitive image with jaggies (sample rate is 1). Then copy a buffer of the current sample buffer, iterate the image again, for each pixel, calculate the sum of the RGB values of surrounding sample_rate pixels (a square area), then divide by sample_rate, which is the same as performing convolution operation and averaging the pixel with its neighbors. Here shows the effect of box filter.

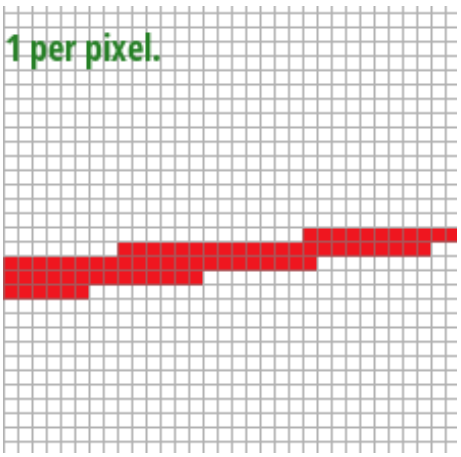

Figure 12: Box filter 1

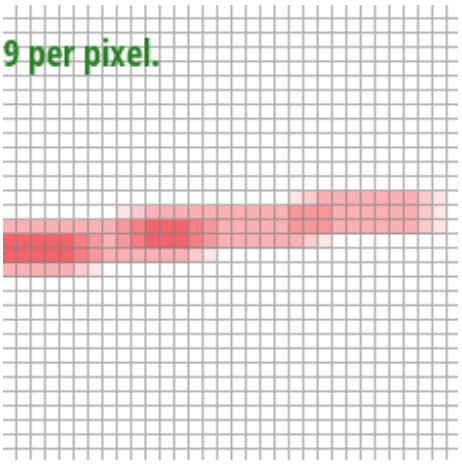

Figure 13: Box filter 9

Figure 14: Box filter triangle 1

Figure 15: Box filter triangle 16

Comparing with grid-based super sampling, box filter can have greater blurring effect, more powerful averaging ability. Since each pixel will be mixed with its neighbor, losing its unique color

property and becomes more in major. The following shows the main code logic.

```
// rasterize once
for (float x = floor(xMin); x <= floor(xMax); x++) {
  for (float y = floor(yMin); y <= floor(yMax); y++) {
    if (inside_tri(x + 0.5, y + 0.5, x0, y0, x1, y1, x2, y2)) {
      rasterize_point(x, y, color);
    }
  }
}

vector<Color> copy_buffer = this->sample_buffer;
int length = sqrt(sample_rate);

for (float x = floor(xMin); x <= floor(xMax); x++) {
  for (float y = floor(yMin); y <= floor(yMax); y++) {
    Color c;
    float red = 0.0, green = 0.0, blue = 0.0;
    for (int idx = 0; idx < sample_rate; idx++) {
      float xSample = x - (length - 1) / 2 +  idx / length;
      float ySample = y - (length - 1) / 2 + idx % length;
      red += copy_buffer[ySample * width + xSample].r;
      green += copy_buffer[ySample * width + xSample].g;
      blue += copy_buffer[ySample * width + xSample].b;
    }

    c.r = red / sample_rate; c.g = green / sample_rate; c.b = blue / sample_rate;
    rasterize_point(x, y, c);
  }
}
```

Figure 16: Box filter

The second extra point here is reduced memory consumption for implementing super sampling. The sample_buffer is still in the same size as the image. The method is not difficult, which is calculating the ratio and multiply by the color and directly write to the frame buffer when iterating the pixel, instead of using another bigger buffer to store each grid result. More exactly, recording the count of grid that lies inside the triangle. The trade-off is more computation during the iteration process. No extra operation or modification is required for related function like clearing buffer contents or fill pixels, which is more convenient. Besides, to eliminate the appearance of white line at the edge of triangle, I use a tricky method: check whether a pixel is rasterized before, if it is, fill this pixel with the current color directly but not multiplying by the ratio, in this way the edge color will not be averaged. The following shows the code.

```
float step = 1.0 / sqrt(sample_rate);
int length = sqrt(sample_rate);
// cout << color.r << " " << color.g << " " << color.b << " ";

for (float x = floor(xMin); x <= floor(xMax); x++) {
  for (float y = floor(yMin); y <= floor(yMax); y++) {
    float count = 0;
    for (int idx = 0; idx < sample_rate; idx++) {
      float xSample = x + step / 2 + step * (idx / length);
      float ySample = y + step / 2 + step * (idx % length);
      if (inside_tri(xSample, ySample, x0, y0, x1, y1, x2, y2)) count++;
    }
    if (count > 0) {
      Color c;
      Color bg = sample_buffer[y * width + x];
      if (bg != Color::White) {
        rasterize_point(x, y, color);
        continue;
      }
      float ratio = count / sample_rate;
      c.r = color.r * ratio + bg.r * (1 - ratio);
      c.g = color.g * ratio + bg.g * (1 - ratio);
      c.b = color.b * ratio + bg.b * (1 - ratio);
      rasterize_point(x, y, c);
    }
  }
}
```

Figure 17: Super sample

4 Task 3: Transforms

4.1 Design and Implementation

This task aims to implement three basic transformation matrix for svg file. The screen is in 2-dimension, so a 3*3 homogeneous coordinate is used. Just use the formula of translation, scaling and rotation to fill the matrix. Here shows the matrix in the code.

```
Matrix3x3 translate(float dx, float dy) {
    // Part 3: Fill this in.
    return Matrix3x3(1, 0, dx, 0, 1, dy, 0, 0, 1);
}

Matrix3x3 scale(float sx, float sy) {
    // Part 3: Fill this in.
    return Matrix3x3(sx, 0, 0, 0, sy, 0, 0, 0, 1);
}

// The input argument is in degrees counterclockwise
Matrix3x3 rotate(float deg) {
    // Part 3: Fill this in.
    float sinDeg = sin(deg * PI / 180);
    float cosDeg = cos(deg * PI / 180);
    return Matrix3x3(cosDeg, -sinDeg, 0, sinDeg, cosDeg, 0, 0, 0, 1);
}
```

Figure 18: Transform

4.2 Results and Analysis

Here shows the original robot and an updated version of the robot with some changes on color and body structure. The robot's head becomes black, representing wisdom, and he wears a blue

clothes, with left leg curved and two hands expanding to give out Kamehameha (GuiPaiQiGong).

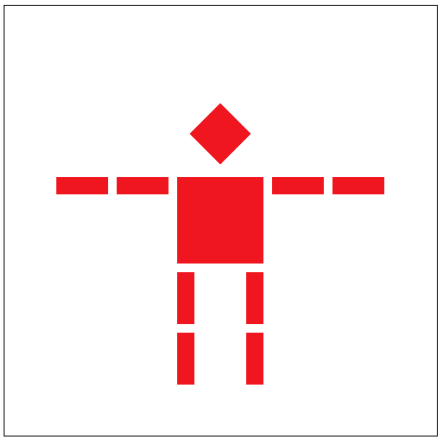

Figure 19: Robot

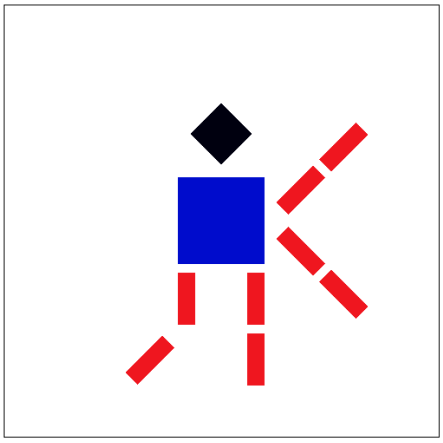

Figure 20: Box filter triangle 16

5 Task 4: Barycentric coordinate

5.1 Design and Implementation

This task aims to draw a triangle with smooth color transition using barycentric coordinate. The colors are defined at the three vertices and interpolated in the middle, going through different area. In the triangle, the three vertices are colored red, blue and green, respectively, and transit gradually. From the picture we can view how colors are blended and what's the resulting color, for example, red plus blue is purple.

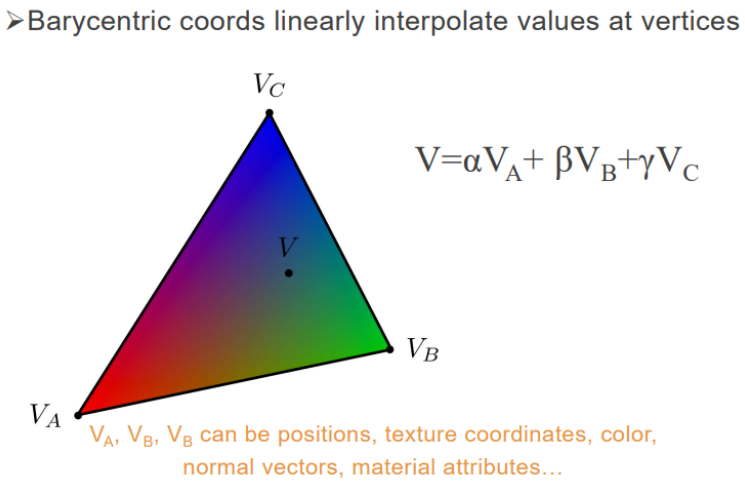

Figure 21: Triangle color

Barycentric coordinate is a 3-dim vector with α, β, and γ, representing the ratio of each vertex. The main idea is to use three vertex's coordinates to represent any point inside the triangle. For

example, (1, 0, 0) represents the A vertex, and (1/3, 1/3, 1/3) represents the center of gravity. To obtain the α, for example, we use the ratio of its opposite triangle connecting to the gravity to the whole triangle area, using cross product. The main code is shown as follows.

```
Color c;
float xMip = x + 0.5;
float yMip = y + 0.5;
float alpha = (-(xMip - x1) * (y2 - y1) + (yMip - y1) * (x2 - x1)) / (-(x0 - x1) * (y2 - y1) + (y0 - y1) * (x2 - x1));
float beta = (-(xMip - x2) * (y0 - y2) + (yMip - y2) * (x0 - x2)) / (-(x1 - x2) * (y0 - y2) + (y1 - y2) * (x0 - x2));
// float gamma = (-(x - x0) * (y1 - y0) + (y - y0) * (x1 - x0)) / (-(x2 - x0) * (y1 - y0) + (y2 - y0) * (x1 - x0));
float gamma = 1 - alpha - beta;
c.r = alpha * c0.r + beta * c1.r + gamma * c2.r;
c.g = alpha * c0.g + beta * c1.g + gamma * c2.g;
c.b = alpha * c0.b + beta * c1.b + gamma * c2.b;
```

Figure 22: Triangle color code

5.2 Results and Analysis

Here shows the color wheel in test7.svg.

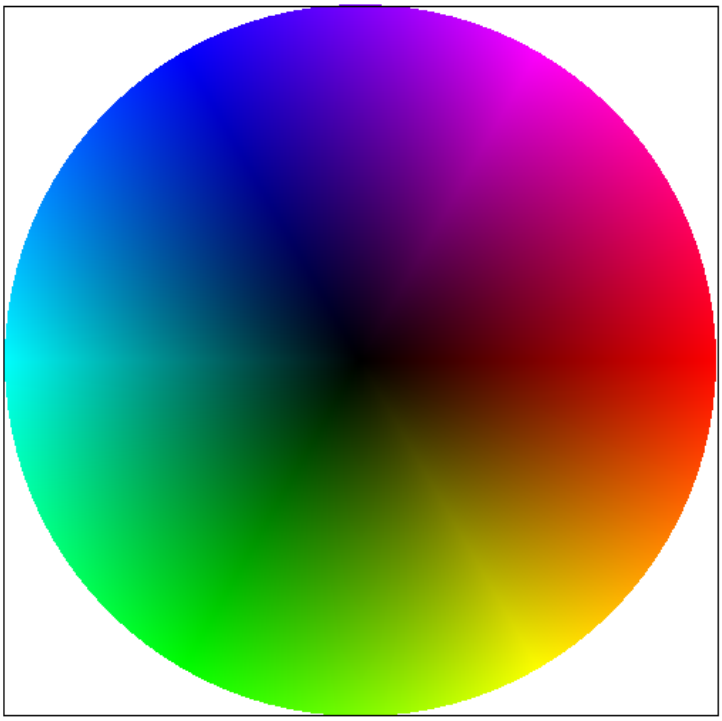

Figure 23: Color wheel

6 Task 5: Texture mapping

6.1 Design and Implementation

This task aims to implement pixel sampling for texture mapping. A texture image is an image with certain pattern on it. A texel is a basic element in the image with RGB format. Pixel sampling is to map pixel to a texel on the texture image, filling the pixel with the texture image's color. Here we do not care about the exact relationship or say design between of the map, but just need

to find their relationship using barycentric coordinate and implement the mapping computation. In task 5 there is only zero-th level full resolution but no mipmapping.

There are two kinds of pixel sampling method, the first one is nearest method, which is to choose the nearest texel in the texture image as the return color, and the second is called linear method, which is calculating the ratio between the four neighboring texels and return the average color. Here we make use of the simple idea of linear function (lerp). User can toggle two methods with the 'P' key. Since the code is covered by task 6, the code logic will be talked in the next section.

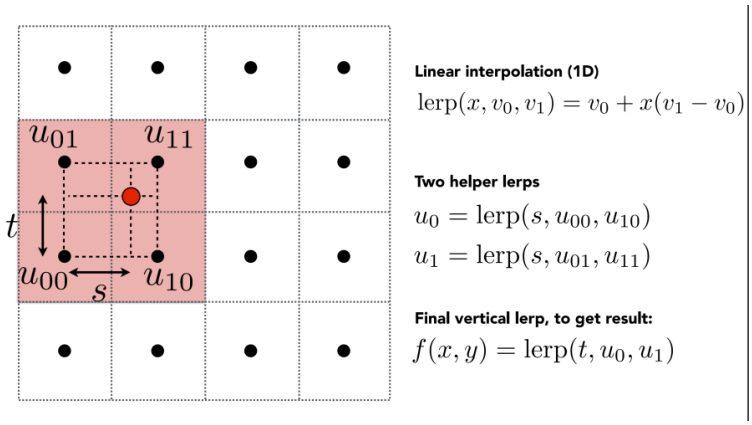

Figure 24: Linear method

6.2 Results and Analysis

Here shows two general texture map images.

Figure 25: Robot

Figure 26: Box filter triangle 16

Check the image of the map, the four pictures represent nearest - sample rate 1, nearest - 16, bilinear - 1, and bilinear - 16. The super sampling method only makes minor influence on the edge of the map, as described in task 2. As for the nearest and bilinear method, bilinear perform better than the former one that the warp and weft of the earth is more continuous (blurred) instead of

discrete, thus having a better visual effect. Since bilinear pixel sampling consider the neighboring texture colors, getting the average, so it should have smoother visualization. The major effect will take place at: the color transition area and the edges.

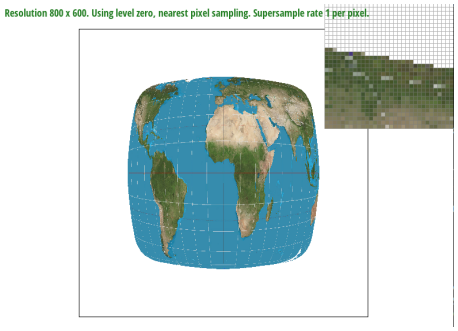

Figure 27: Near 1

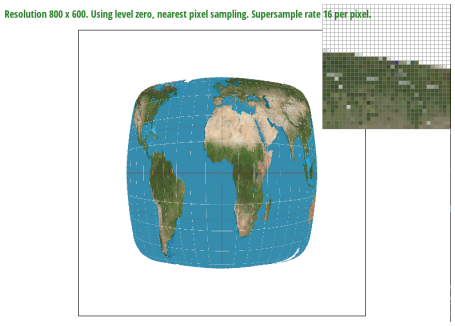

Figure 28: Near 16

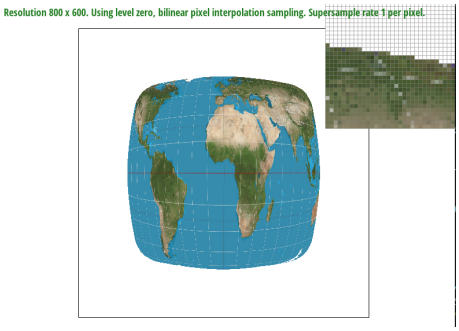

Figure 29: Linear 1

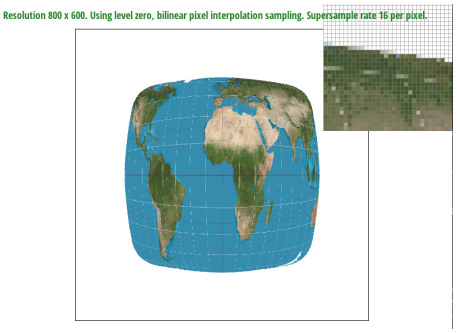

Figure 30: Linear 16

7 Task 6: Mipmapping

7.1 Design and Implementation

This task aims to perform mipmapping to deal with inconformity of pixel and texel. Sometimes when a pixel maps to a texel, the pixel will be either too small or too large, generating an unreasonable color to make the image messy. To deal with the case, some smaller version of the original images are generated for texture mapping picking (divide by 2 every time). We calculate the ration between a unit of length in pixel and its mapping length to texture, and find corresponding mip level to sample. We have three level mapping method, the first one is just as task 5, zero-th level; The second one is nearest level, that is to round the level result to nearest integer level; and the third one is linear level, calculate the weighted average color between two continuous mipmap level and return the color. One can use "L" key to toggle.

The code logic is as follows: Iterate the pixel in boudning box, calculate barycentric coordinates of (x, y), (x+1, y) and (x, y+1), store the three u-v coordinates in the vector. Invoke

tex.sample(The parameter structure). According to the pixel sampling and level method, we have 2*3 = 6 methods, implement according to method. The following code shows how to get the mipmapping level (Use the difference) and bilinear sampling method (Use lerp()).

```
Color Texture::sample_bilinear(Vector2D uv, int level) {
  // TODO: Task 5: Fill this in.
  // return magenta for invalid level
  if (level < 0) return Color(1, 0, 1);

  auto& mip = mipmap[level];
  float u = uv[0] * mip.width;
  float v = uv[1] * mip.height;

  float s = u - round(u) + 0.5;
  float t = v - round(v) + 0.5;
  Color c00 = mip.get_texel(round(u) - 1, round(v) - 1);
  Color c10 = mip.get_texel(round(u), round(v) - 1);
  Color c01 = mip.get_texel(round(u) - 1, round(v));
  Color c11 = mip.get_texel(round(u), round(v));
  Color c0 = lerp(s, c00, c10);
  Color c1 = lerp(s, c01, c11);
  return lerp(t, c0, c1);
}
```

Figure 31: Bilinear method

```
float Texture::get_level(const SampleParams& sp) {
  // TODO: Task 6: Fill this in.
  float du_dx = (sp.p_dx_uv[0] - sp.p_uv[0]) * this->width;
  float du_dy = (sp.p_dy_uv[0] - sp.p_uv[0]) * this->width;
  float dv_dx = (sp.p_dx_uv[1] - sp.p_uv[1]) * this->height;
  float dv_dy = (sp.p_dy_uv[1] - sp.p_uv[1]) * this->height;

  float L = max(sqrt(du_dx * du_dx + dv_dx * dv_dx), sqrt(du_dy * du_dy + dv_dy * dv_dy));
  return log2f(L);
}
```

Figure 32: Level

Now we can adjust sampling technique by selecting pixel sampling, level sampling, or the number of samples per pixel, the trade-off between speed memory usage, anti-aliasing power is obvious, the higher the speed or the larger the memory use, the stronger the anti-aliasing power.

7.2 Results and Analysis

Here shows the selecting results of zero + nearest; zero + linear, nearest + nearest, and nearest + linear. The sampling effect is better and better. The more averaging method and mipmapping method used, the better the visualization, clear image shown. Here also shows some of my own texture images:

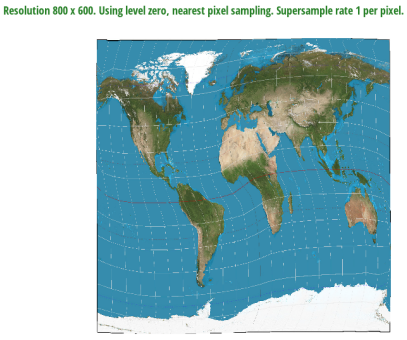

Figure 33: zero + nearest

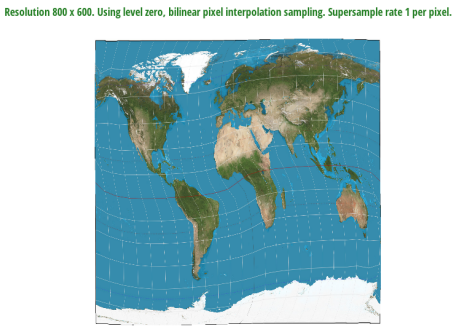

Figure 34: zero + linear

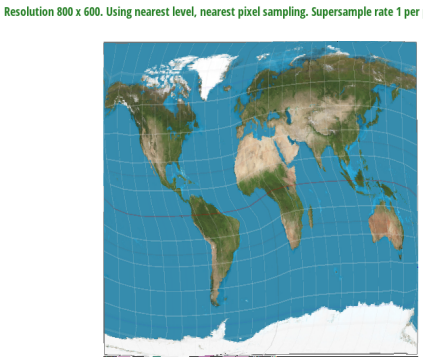

Figure 35: nearest + nearest

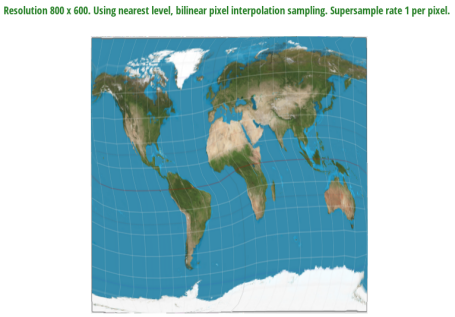

Figure 36: nearest + linear

Figure 37: Original whale

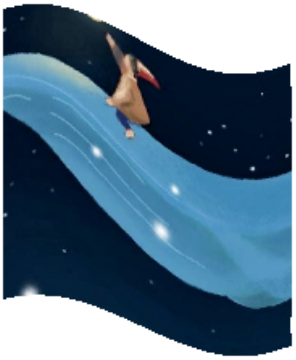

Figure 38: Whale 1

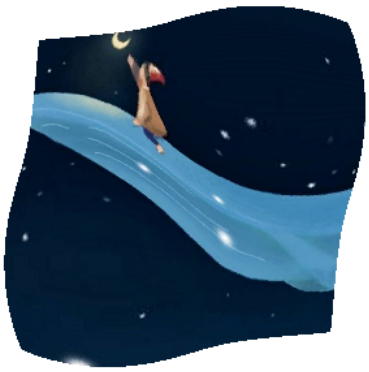

Figure 39: whale2

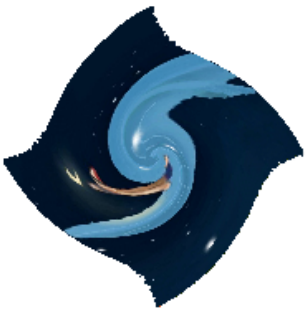

Figure 40: whale 3

8 Execution

(a) After build, use the following command to rasterize:

$$./draw \quad xxx.png$$

(b) Can use the testall.py to test the correctness. Notice that not all the cases can pass since there will be error when performing float comparison. The result is: 23 passed. 9 failed.

(c) Many results are not shown but the graph can be generated properly.

(d) SVG files created by myself are placed in the "Own image" folder

(e) Comment on and out the code in task 2 to execute another algorithm if needed.

9 Summary

Here I will summarize what I have learned when writing this assignment.

(a) Understanding of rasterization in computer graphics. For example, how to rasterize a triangle and how to perform texture mapping, mipmapping, the rasterization pipeline and so on.

(b) The many programming problems to be noticed: float-integer transformation(1 / 2 = 0 !!!), boundary judgment (or core dump will happen), using z to have inspector, considering background color when averaging the pixel color, **multiplying the mipmap's width and height to the u and v between 0 1**...

(c) How to edit a svg file and create my artifacts: I firstly click to open but actually it should be opened as text file to edit.

(d) Object-Oriented programming paradigm, use class to encapsulate module.

(e) Some CG related library usage

(f) Meaningful time on debugging and coding skill improvement, write-up report writing.

(g) The ability to use searching engine and discussion.

(h) Virtual machine usage, Linux operations.

(i) The ability to finish a large project in 20 days. Time planning.

(j) By the way, the discussion in lecture/tutorial/wechat group is helpful, Kudos!

That's all.