Assignment 03

Name: An Zihang

Student ID: 121090001

0. Abstract

This project involves rasterization, shading and texture mapping. The program takes *.svg* files as input and uses GUI to show the result. Main contents are as follows:

1. **Rasterizing** and **shading** triangles on the screen
2. Performing **antialiasing** to the image
3. Performing **transforms** to the image
4. Using **barycentric coordinates** to color triangles via interpolation
5. **Texture** mapping
6. Optimization of texture mapping (**Mipmap**)
7. My tries and reflection in **optimizing antialiasing**

1. Rasterizing triangles onto the screen

Essentially, the key of rasterizing a triangle is to judge whether a pixel belongs to the triangle, should it be visible and what is its color. A basic, brute-force and effective way is to traverse the bound box of each triangle, and judge the pixels one by one:

```
float lBound, rBound, tBound, bBound;
lBound = min(min(x0, x1), x2), rBound = max(max(x0, x1), x2);
bBound = min(min(y0, y1), y2), tBound = max(max(y0, y1), y2);
for (int i = floor(bBound); i <= floor(tBound); i++)
    for (int j = floor(lBound); j <= floor(rBound); ++j)
        ...
```

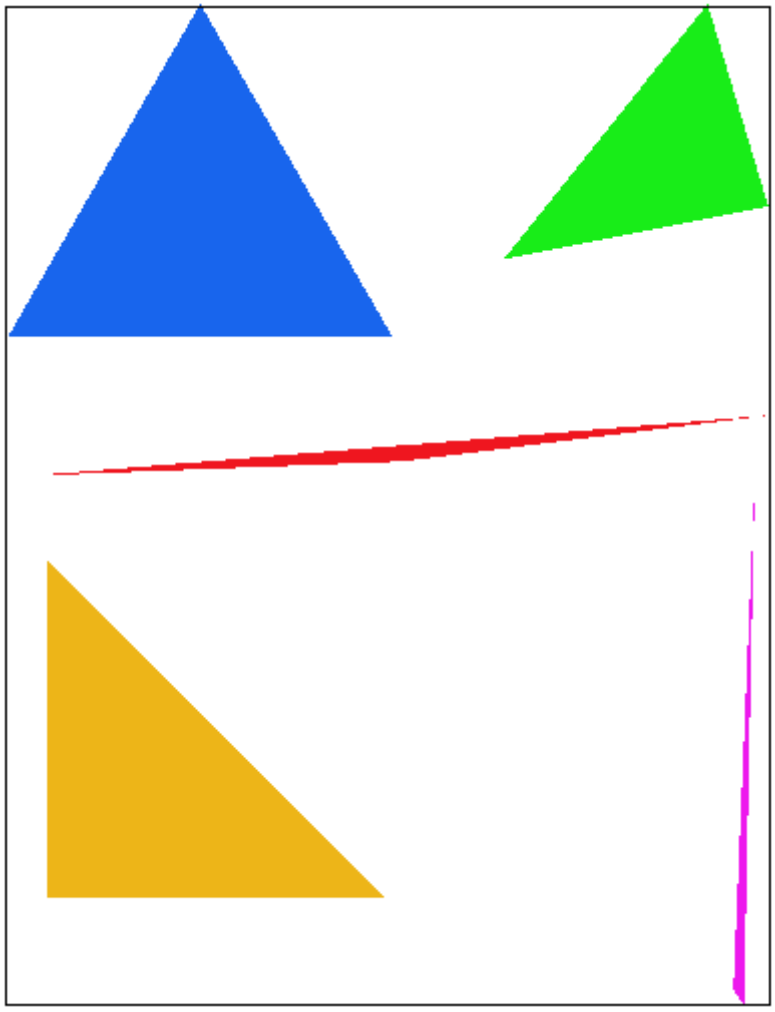
Also, when we want to judge some position, we need a function to take the position and the triangle as inputs, and give the conclusion:

```
bool RasterizerImp::p_in_triangle(float x, float y, float x0, float y0,
float x1, float y1, float x2, float y2) {
    CGL::Vector3D e0 = CGL::Vector3D(x1-x0, y1-y0, 0); // vector from point
0
    CGL::Vector3D e1 = CGL::Vector3D(x2-x1, y2-y1, 0); // vector from point
1
    CGL::Vector3D e2 = CGL::Vector3D(x0-x2, y0-y2, 0); // vector from point
2

    CGL::Vector3D v0 = CGL::cross(e0, CGL::Vector3D(x-x0, y-y0, 0));
    CGL::Vector3D v1 = CGL::cross(e1, CGL::Vector3D(x-x1, y-y1, 0));
    CGL::Vector3D v2 = CGL::cross(e2, CGL::Vector3D(x-x2, y-y2, 0));

    return (v0[2]*v1[2] >= 0 && v1[2]*v2[2] >= 0 && v2[2]*v0[2] >= 0);
}
```

This function uses vector cross product. If a point $P(x, y)$ is inside the triangle ABC, then $AB \times AP, BC \times BP, CA \times CP$ should have their z-coordinate of the same sign. Otherwise P is outside the triangle.

Antialiasing (SSAA)

when the sample rate is low, there are lots of jaggies on the edge of triangles:



jaggies can be clearly seen on the center and bottom-right triangles. So we use Super Sample Anti-Aliasing to make the edge smooth.

Specifically, by setting larger sample rate, each pixel in the bound box is partitioned into subpixels, and its final color is the average of all subpixels.

```
for (int i = floor(bBound); i <= floor(tBound); i++)
    for (int j = floor(lBound); j <= floor(rBound); ++j)
        for (float y = i; y < i+1; y += sample_unit)
            for (float x = j; x < j+1; x += sample_unit)
                if (p_in_triangle(x+0.5*sample_unit, y+0.5*sample_unit, x0,
y0, x1, y1, x2, y2))
                    fill_pixel(x, y, color);
```

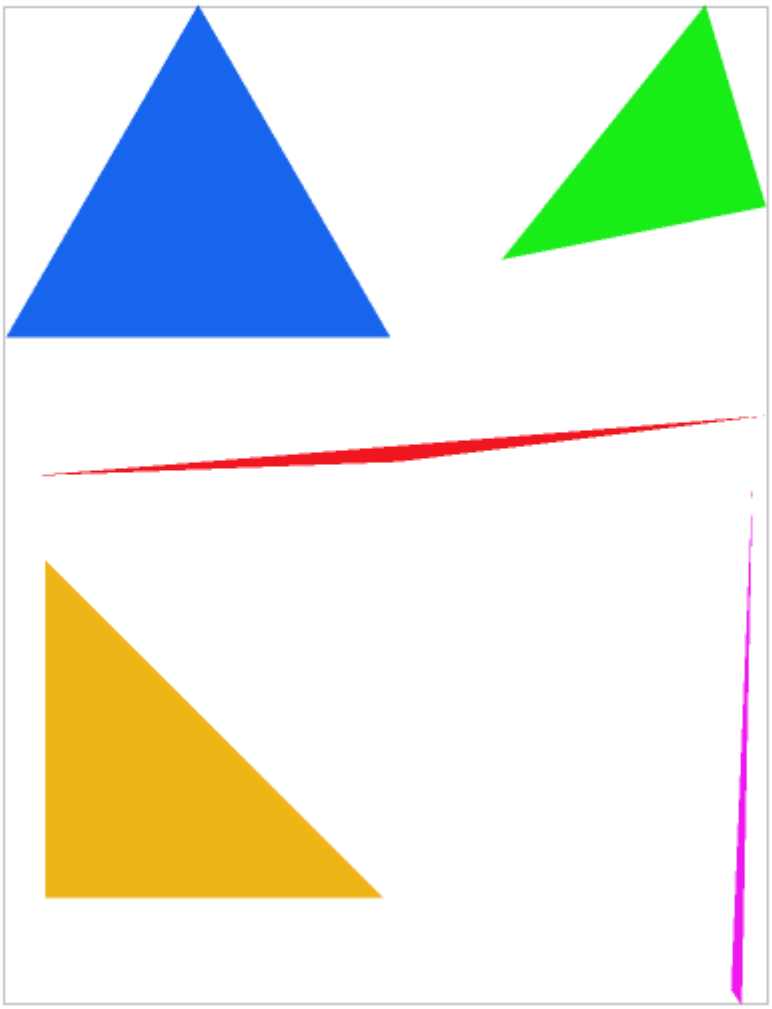
```
void RasterizerImp::resolve_to_framebuffer() {
    for (int x = 0; x < width; ++x) {
        for (int y = 0; y < height; ++y) {
            Color col;
```

```
            for (int i = 0; i < sample_sqrt; i++)
                for (int j = 0; j < sample_sqrt; j++) {
                    col += sample_buffer[(y*sample_sqrt + i) * width *
sample_sqrt + x*sample_sqrt + j];
                }
            col *= 1.0/sample_rate;
            for (int k = 0; k < 3; ++k) {
                this->rgb_framebuffer_target[3 * (y * width + x) + k] =
(&col.r)[k] * 255;
            }
        }
    }
}
```

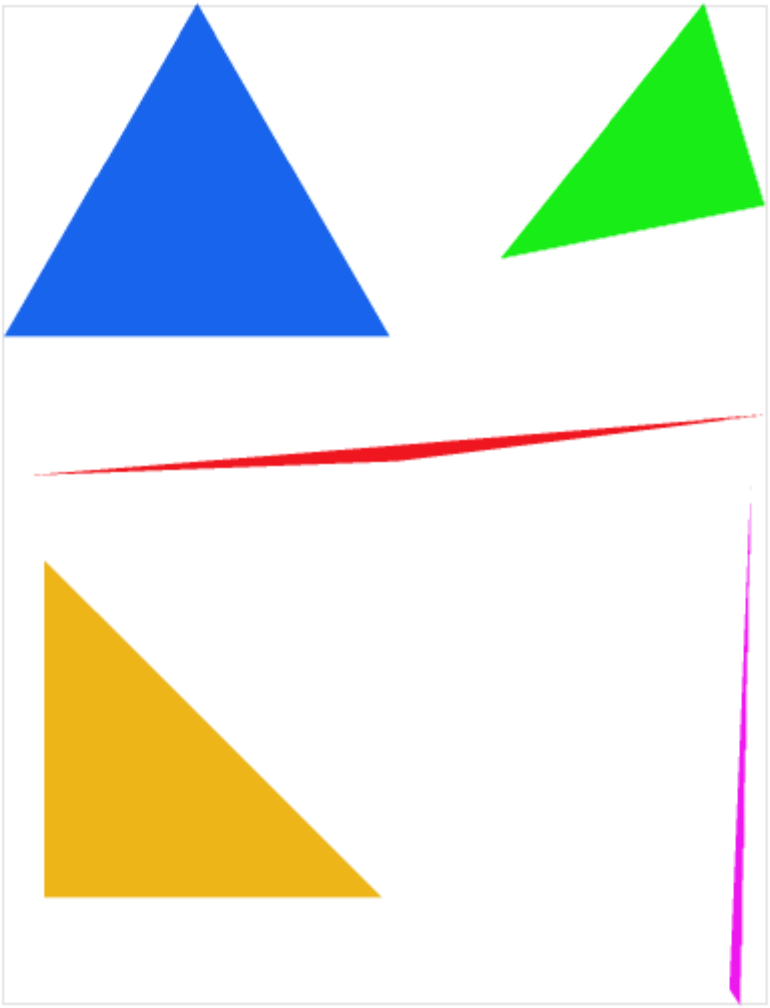
Here *sample_unit* equals to $\sqrt{sample\_rate}$, because we use a grid distribution to place the subsamples.

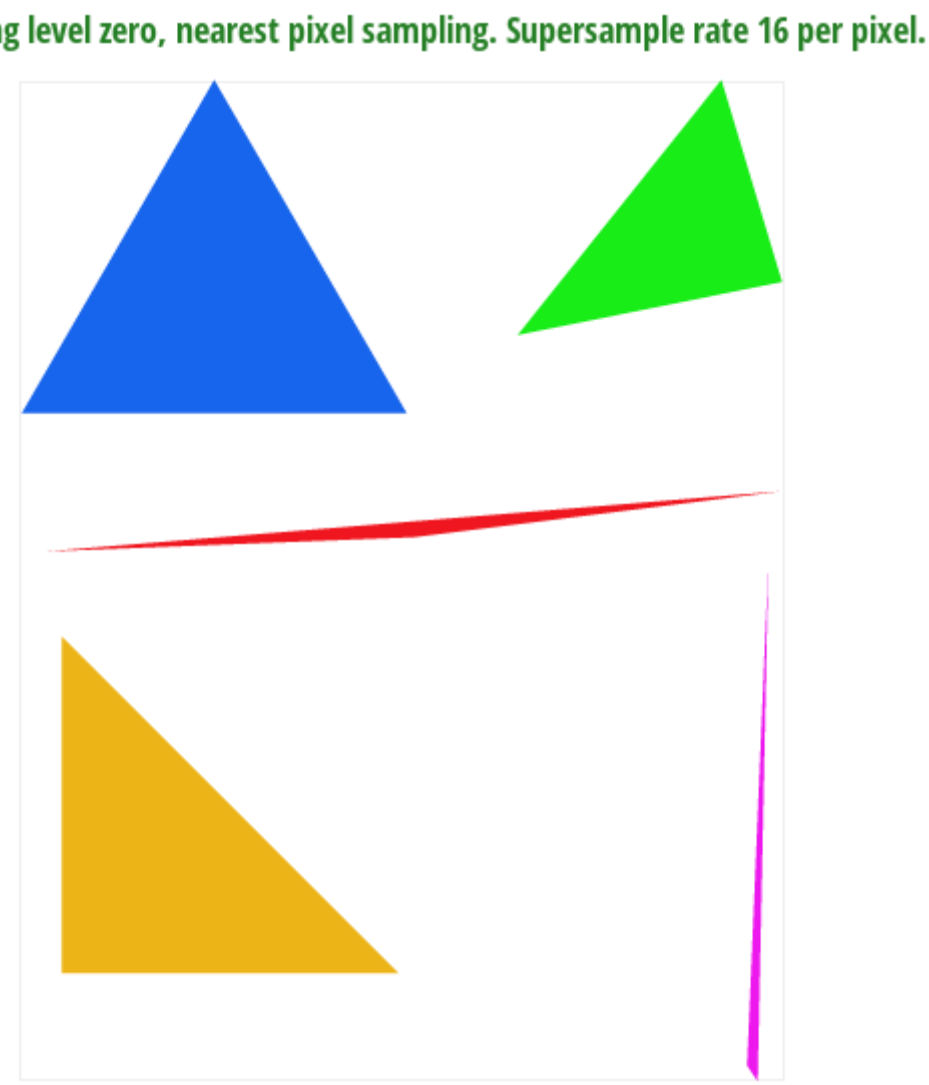
The following screenshots shows the result under different sample rates:

Using level zero, nearest pixel sampling. Supersample rate 4 per pixel.



Using level zero, nearest pixel sampling. Supersample rate 9 per pixel.





With increment of sample rate, the jaggies are reduced. When zooming in, we can see the averaged color of edge points (16x SSAA)

Note that the black border is also being lighter. That is because in this project, lines are **not** anti-aliased, so the black line will be averaged by the white subsamples around it.

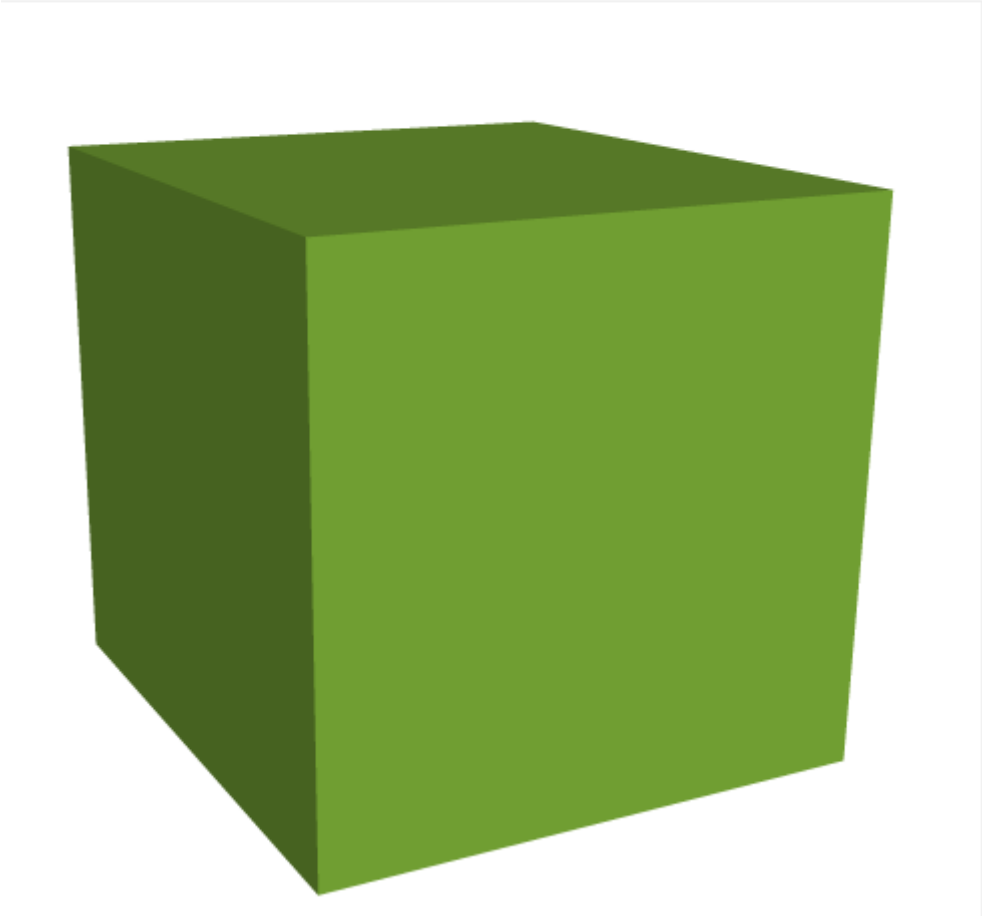
SSAA is the most effective way of anti-aliasing. But for some leaning, narrow and long triangles, there will be extremely much **redundant computation**. Also, there is meaningless computation inside pure-color triangles. We also have to store all information of **all subsamples**, like color, depth, so we need to extend all buffers' size by **many times**!

In the last part of this report, I will talk about some other ways I tried to optimize anti-aliasing. (They are not eventually used in this project because I failed to make them perfect due to lack of time)

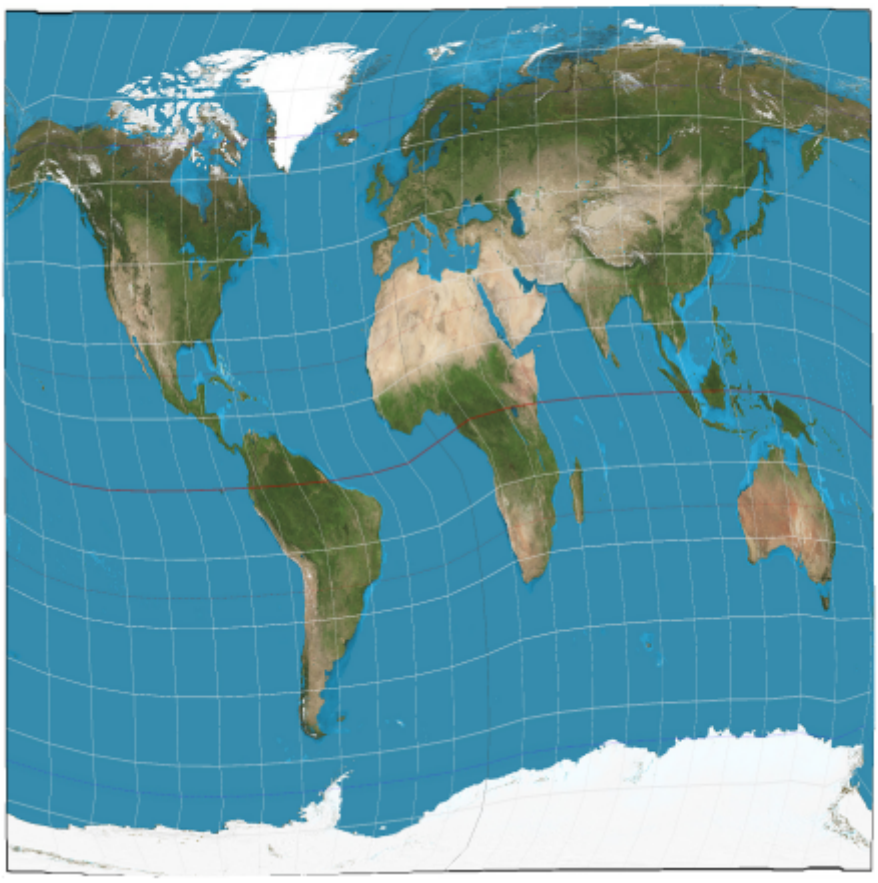
Screenshots below are more results with SSAA, some other artifacts like moire pattern can be solved (the last screenshot, which is hardcore 01 svg):

0. Using level zero, nearest pixel sampling. Supersample rate 16 per pixel.

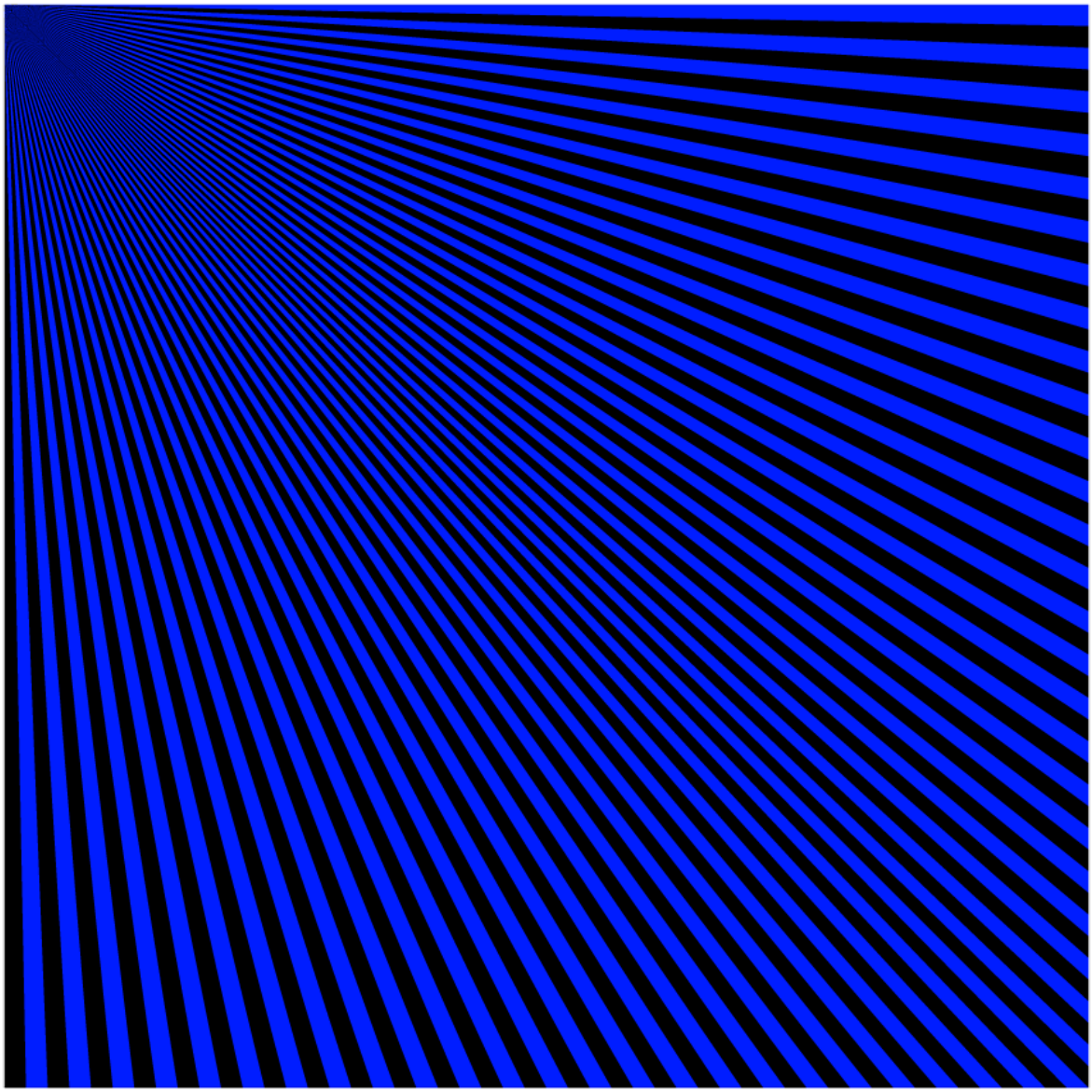
. Using level zero, nearest pixel sampling. Supersample rate 16 per pixel.



. Using level zero, nearest pixel sampling. Supersample rate 16 per pixel.



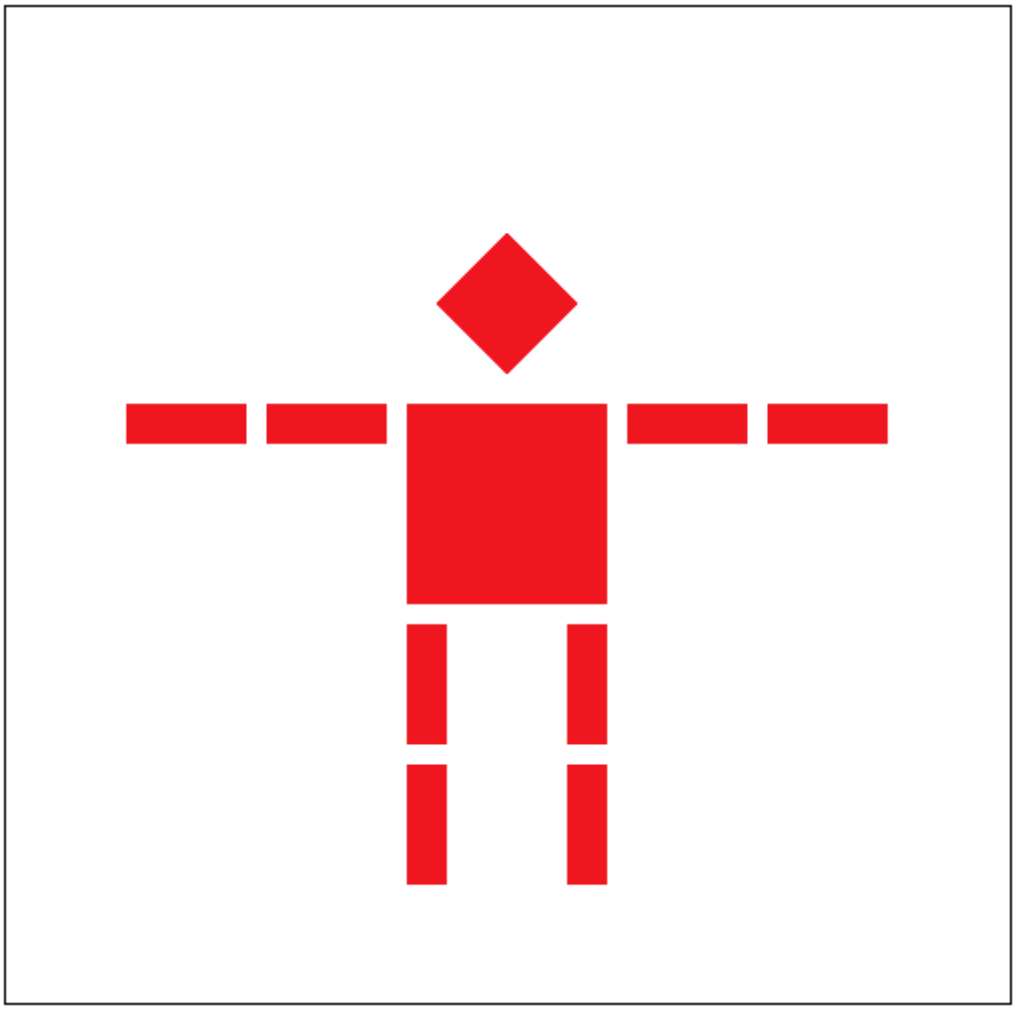
Supersample rate 16 per pixel.



3. Transforms

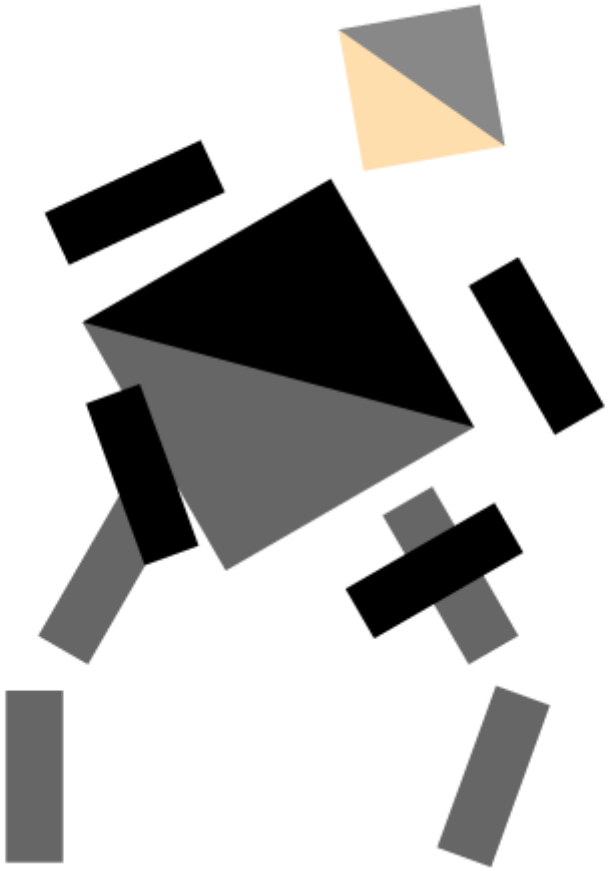
In this part, we support the transformations of the image. Using 2D model matrix, we can do **translation, scaling and rotation** to the image elements.

The image below is a robot:



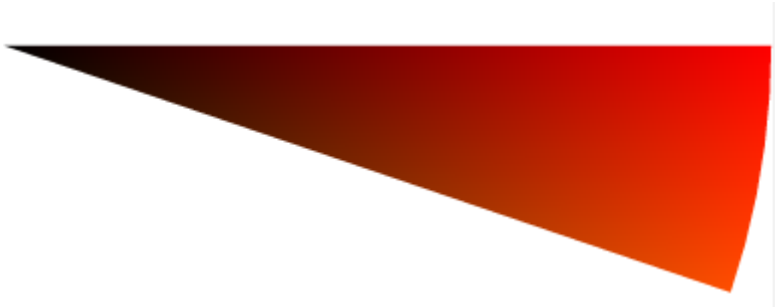
by setting the parameters in the .svg file, we can transform the robot, rotate his body, arms and legs, change the color of the tiles to be clothes, now he is dancing like a idol!!

Using level zero, nearest pixel sampling. Supersample rate 16 per pixel.



4. Barycentric coordinates and interpolated color triangle

Color can change inside a triangle. When only the colors of vertices are given, we need to use interpolation to calculate the color of the triangle.



Basically, interpolation is a way to estimate the value of an arbitrary point in a discrete function. An image is like a 2D discrete function, so we can use interpolation to estimate the color of any point. Specifically, the position of any point inside a triangle can be expressed by a positive linear combination of the coordinates of the three vertices, and the coefficients add up to be 1. i.e.

$$\forall P(x,y) \in \triangle ABC, \exists \alpha, \beta, \gamma > 0$$
$$x = \alpha x_A + \beta x_B + \gamma x_C, \;\; y = \alpha y_A + \beta y_B + \gamma y_C$$
$$\wedge \alpha + \beta + \gamma = 1$$

```
if (p_in_triangle(x+0.5*sample_unit, y+0.5*sample_unit, x0, y0, x1, y1, x2, y2)) {
    float alpha = ( -(x-x1)*(y2-y1) + (y-y1)*(x2-x1) )/
        ( -(x0-x1)*(y2-y1) + (y0-y1)*(x2-x1) );
    float beta = ( -(x-x2)*(y0-y2) + (y-y2)*(x0-x2) )/
        ( -(x1-x2)*(y0-y2) + (y1-y2)*(x0-x2) );
    float gamma = 1 - alpha - beta;

    color = alpha*c0 + beta*c1 + gamma*c2;
    fill_pixel(x, y, color);
}
```

We can use this to finish test 7:

600. Using level zero, nearest pixel sampling. Supersample rate 16 per pixel.

5. Texture mapping

Besides color given in *.svg* files, the color information can also be determined by texture image. Like rendering a photo of someone, or the texture of bricks.

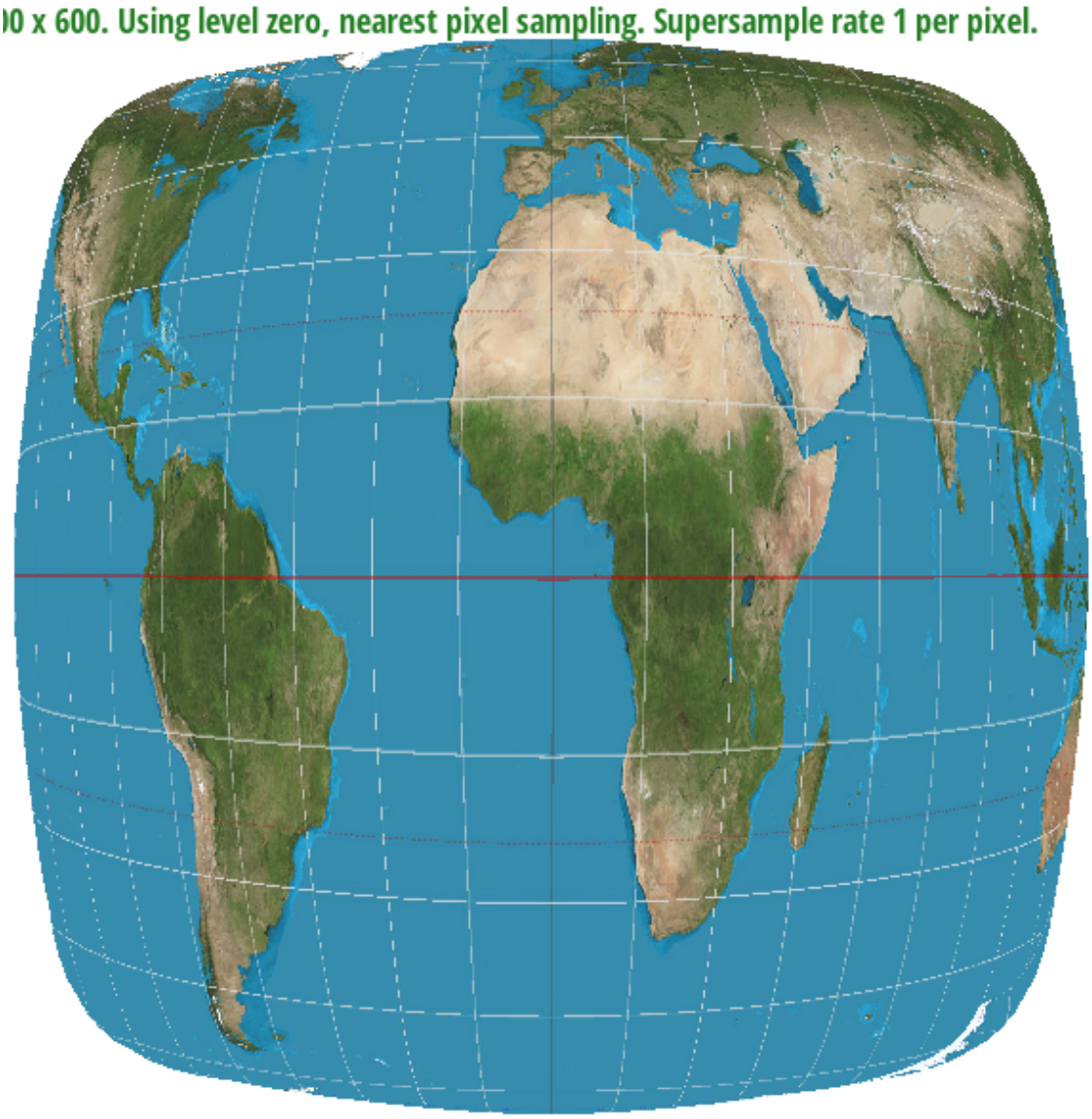
Since the texture is obviously static, there should a **mapping relationship** between the pixels we will display and the texels on the texture, so that we can know which part of texture a region on the screen corresponds to.

Assume we have got the mapping relationship, and we have known the corresponding coordinates of each triangle vertex, we have to determine the colors of our pixels using interpolation. There are 2 ways to do this sampling.

Nearest pixel sampling

When the texture is of **small size**, which means many pixels may be covered by the same texel. If we don't care about this, and just take the nearest texel as the sampling result, the resolution will be reduced.

```
auto& mip = mipmap[level];
int tx = round(uv[0] * mip.width);
int ty = round(uv[1] * mip.height);
return mip.get_texel(tx, ty);
```



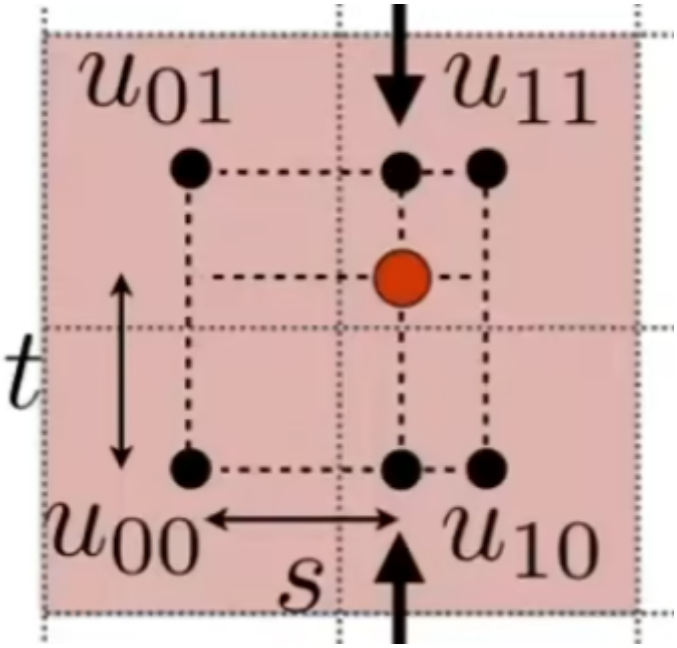
we can see lots of artifacts in the image.

Bilinear interpolation sampling

So how to estimate the color among discrete texel color values? Interpolation!

This time we take 4 nearest texels around the pixel, first do linear interpolation horizontally, then do linear interpolation vertically, then we can get the approximate color of this pixel. More specifically,

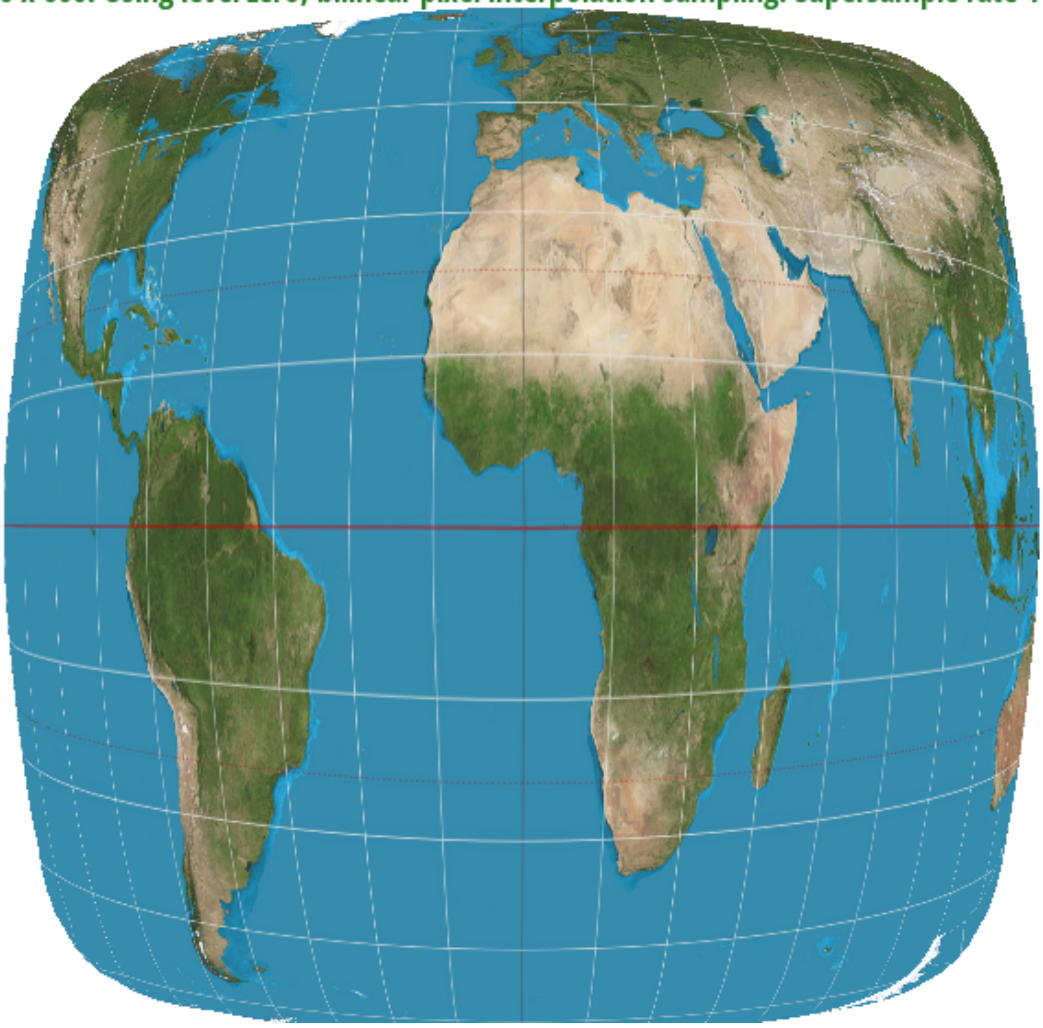
$$u_{up} = s \times u_{11} + (1 - s) \times u_{01}$$
$$u_{down} = s \times u_{10} + (1 - s) \times u_{00}$$
$$Color(P) = t \times u_{up} + (1 - t) \times u_{down}$$




```
auto& mip = mipmap[level];
float cx = uv[0]*mip.width, cy = uv[1]*mip.height;
float l = round(cx) - 0.5, r = round(cx) + 0.5;
float b = round(cy) - 0.5, t = round(cy) + 0.5;
Color u0 = mip.get_texel(l-0.5, b-0.5) * (r-cx) + mip.get_texel(r-0.5, b-0.5) * (cx-l);
Color u1 = mip.get_texel(l-0.5, t-0.5) * (r-cx) + mip.get_texel(r-0.5, t-0.5) * (cx-l);
Color u = u0 * (t-cy) + u1 * (cy-b);
return u;
```

The latter performs better

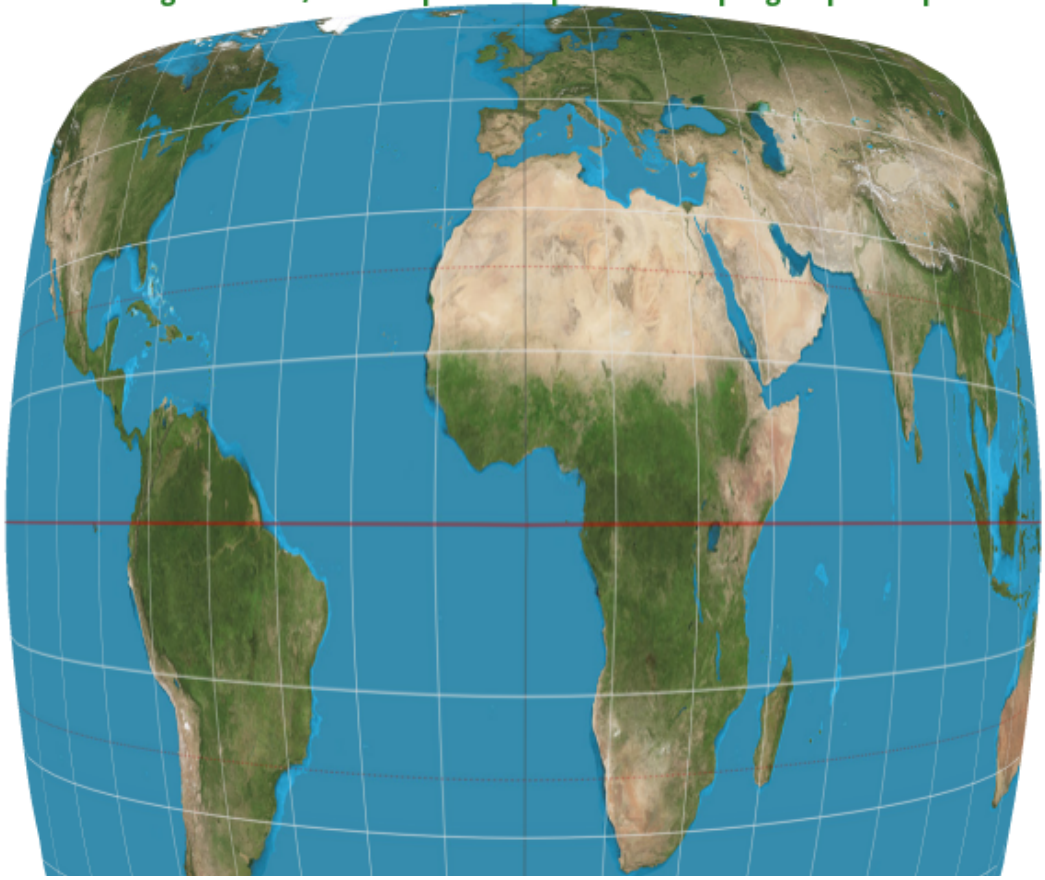
)0 x 600. Using level zero, bilinear pixel interpolation sampling. Supersample rate 1 per pixel.

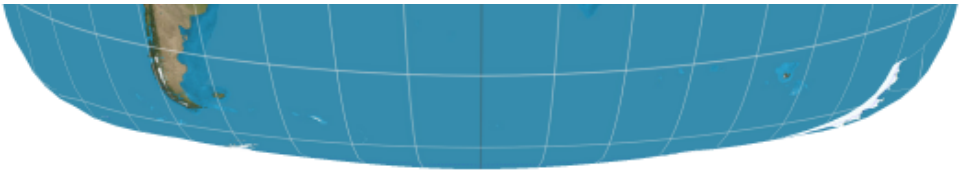


but not enough. We can combine them with 16x SSAA:

0 x 600. Using level zero, nearest pixel sampling. Supersample rate 16 per pixel.

0 x 600. Using level zero, bilinear pixel interpolation sampling. Supersample rate 16 per pixel.





6. Mipmap

When the texture is of **large size**, one pixel will cover many texels, then too high frequency will cause artifacts, like jaggies and moire. So we have to desample the texture.

Mipmap generates a level of 1/4 size of the last level each time using a 2*2 filter. This will make a pixel cover less "texel" on higher level. there are $log_2(max\{width, height\})$ levels in total. i.e., the highest level will be 1*1.

each time we sample, we calculate the approximate area of the pixel, and find a level where the area equals to 1. To **compat with the small-sized textures**, we need to use $max(0, log_2 L)$ rather than L.

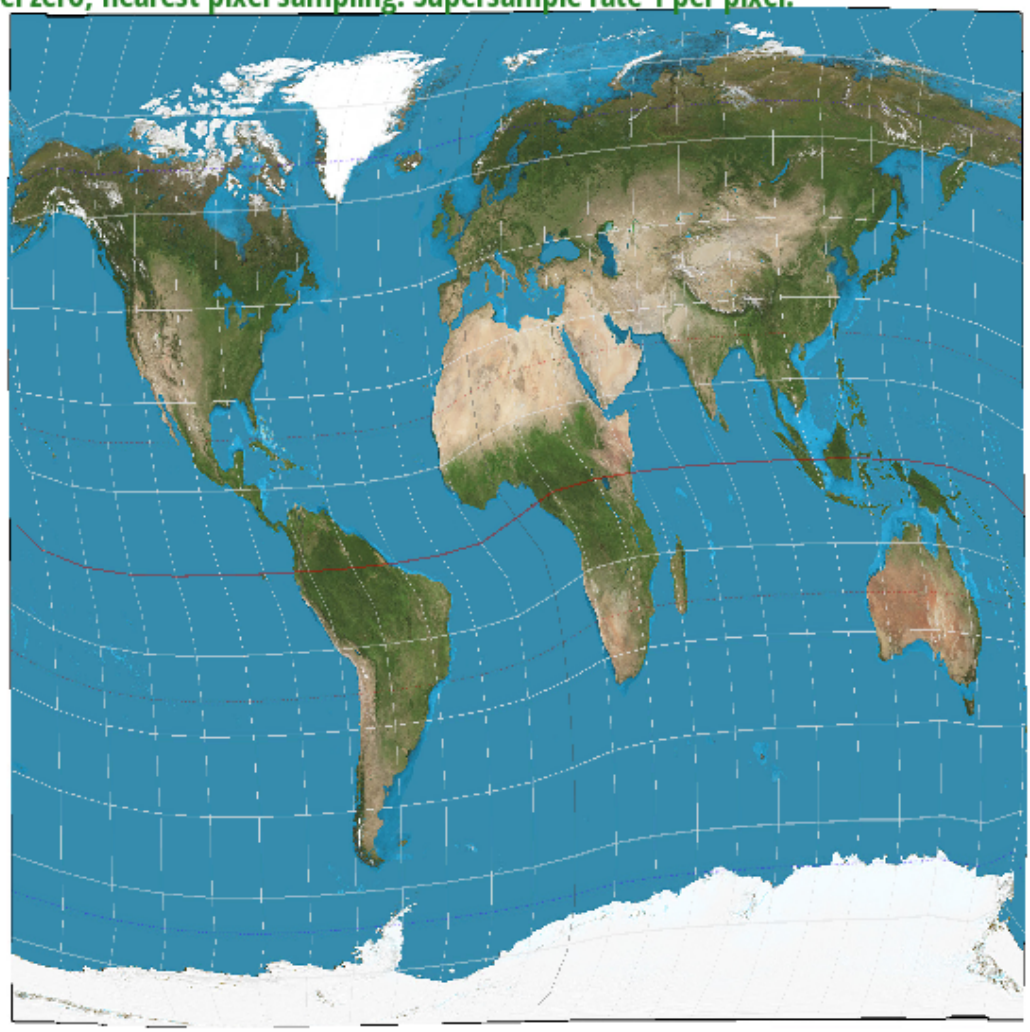
```
float du_dx = (sp.p_dx_uv[0] - sp.p_uv[0]) * width;
float du_dy = (sp.p_dy_uv[0] - sp.p_uv[0]) * height;
float dv_dx = (sp.p_dx_uv[1] - sp.p_uv[1]) * width;
float dv_dy = (sp.p_dx_uv[1] - sp.p_uv[1]) * height;
float L = max(sqrt(du_dx*du_dx + dv_dx*dv_dx),
              sqrt(du_dy*du_dy + dv_dy*dv_dy));
return max(0.0f, log2f(L));
```

What if we get a level **1.666**? We can of course choose the nearest level, but there is a better choice: interpolation again! This time we interpolate between two levels linearly.

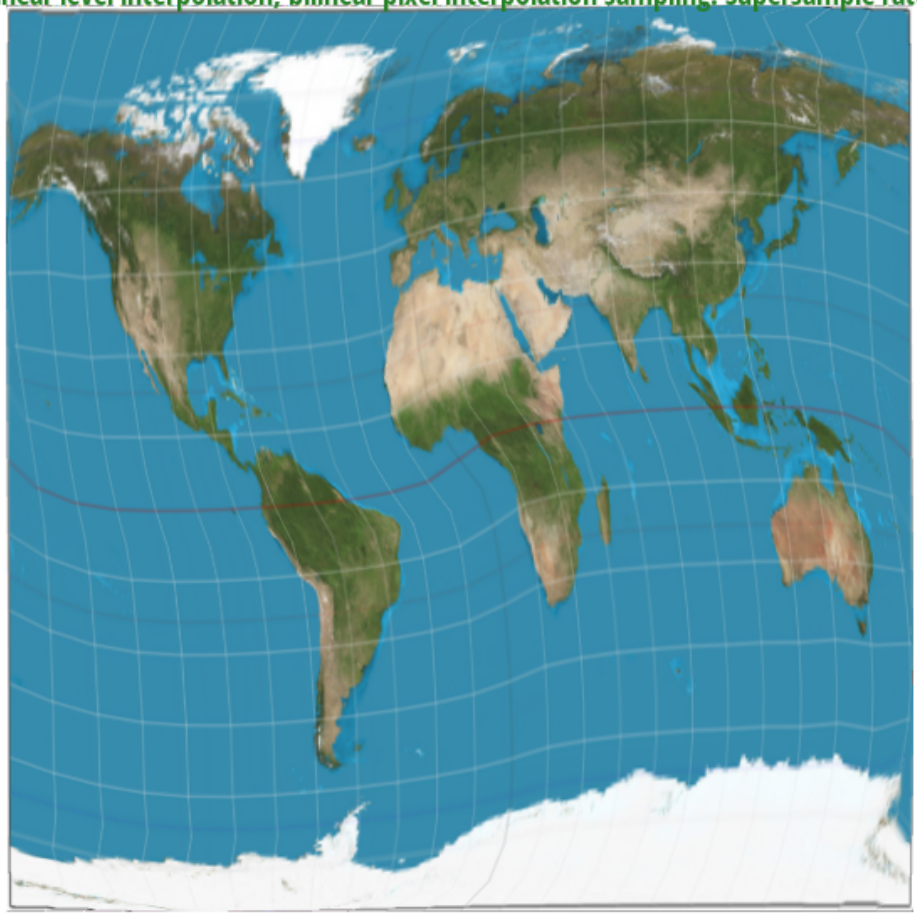
```
else if (sp.psm == CGL::P_LINEAR) {
    Color c1 = sample_bilinear(sp.p_uv, ceil(level));
    if (ceil(level) == floor(level))
        return c1;

    Color c2 = sample_bilinear(sp.p_uv, floor(level));
    return c1 * (level-floor(level)) + c2 * (ceil(level)-level);
}
```

Now lets compare the original image, **trilinear interpolation** and **16x SSAA**

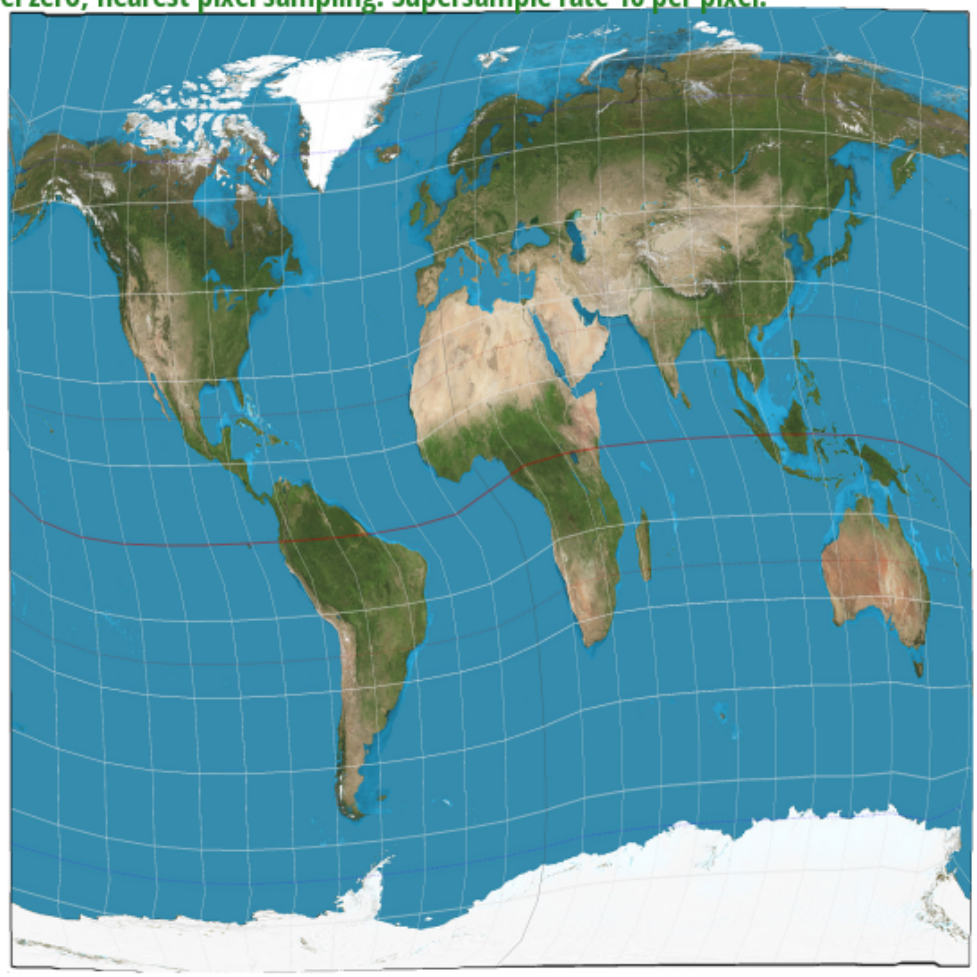
Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.



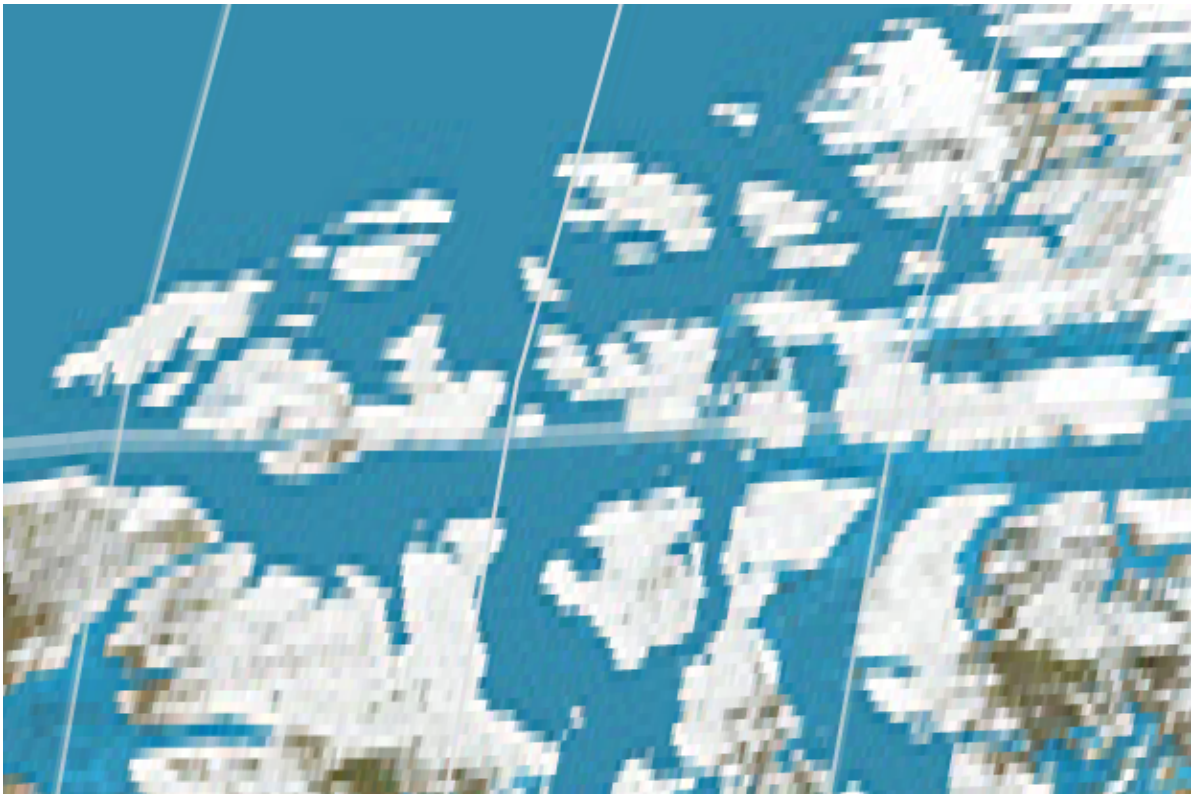
. Using bilinear level interpolation, bilinear pixel interpolation sampling. Supersample rate 1 per pixel.



Using level zero, nearest pixel sampling. Supersample rate 16 per pixel.



We can see very good effect of trilinear interpolation. Overall it is not as clear as16x SSAA, but there can be even less jaggies than SSAA:



It seems that mipmap uses lots of space, but the total memory cost is a power series, which add up to be $\frac{4}{3} \times width \times height$, which is just $\frac{1}{3}$ more than the original cost, far less than SSAA.

Mipmap can only estimate square regions on texture, but often, a pixel covers a rectangular region on the texture, then **overblur** will occur.

Then we can consider using anisotropic filtering. If we use Anisotropic filtering, the memory cost will be $3 \times width \times height$. So **better quality always comes with more cost**.

The screenshots below are different level sampling and texture mapping modes:

800 x 600. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.

800 x 600. Using level zero, bilinear pixel interpolation sampling. Supersample rate 1 per pixel.

800 x 600. Using nearest level, nearest pixel sampling. Supersample rate 1 per pixel.

800 x 600. Using nearest level, bilinear pixel interpolation sampling. Supersample rate 1 per pixel.

7. My tries and reflections in optimizing antialiasing

My main aim is to lover the memory cost, and traverse less redundant pixels.

7.1 coverage buffer

At first I found it difficult to know how much is a pixel covered by a particular triangle, unless I record all subsamples' colors, which is actually SSAA. To avoid extending the buffers by many times, I tried using a "**coverage buffer**", which stores "how much in total is the pixel covered by all triangles", if $coverage\_buffer[y * width + x] == 1$, that means the pixel is fully covered; if it equals to 0.6, that means 60% area of the pixel is covered by some triangles.

What is the meaning of the buffer? I wanted to **change from "6 out of 16 subsamples are covered by an orange triangle" to "this pixel contains $\frac{6}{16}$ orange"**. Then the memory cost is not extended. For those whose coverage is less than 1 after rasterizing all triangles, interpolate using the background color (white).

This was a effect try, but I didn't realize that there can be triangles covering each other, and I can't know what colors are replaced, because I lost detailed information of the subsamples.

7.2 breadth-first-search

I learned that MSAA is a kind of optimization that only detects the edge of triangles. Then I thought how could I only super sample the edge of triangles?

First, I want to start rasterization inside the triangle, rather than traversing the whole bound box. I chose the barycenter as the starting point because it is always inside the triangle. Then I used **breadth-first-search** to expand to the 4 or 8 pixels around the pixel. **When at least one of them is outside the triangle, the current pixel is an "edge pixel"**. Then I declared a map named **"SSP_buffer"** which maps $pair < float, float >$ coordinates with $vector < Color >$ color buffers.

```
// in rasterizer.h / RasterizerizeImp / private
struct super_sample_pixel {
    std::vector<Color> color_buffer;
};

std::map<std::pair<int, int>, super_sample_pixel> SSP_buffer;
```

In this way, I can detect the "**edge pixels**", super sample them, and insert them into the "**SSP_buffer**". When resolving, I just need to check whether a pixel is a "super sampled pixel", **if yes, I use the average of its color buffer, if not, I just use the value in sample_buffer**.

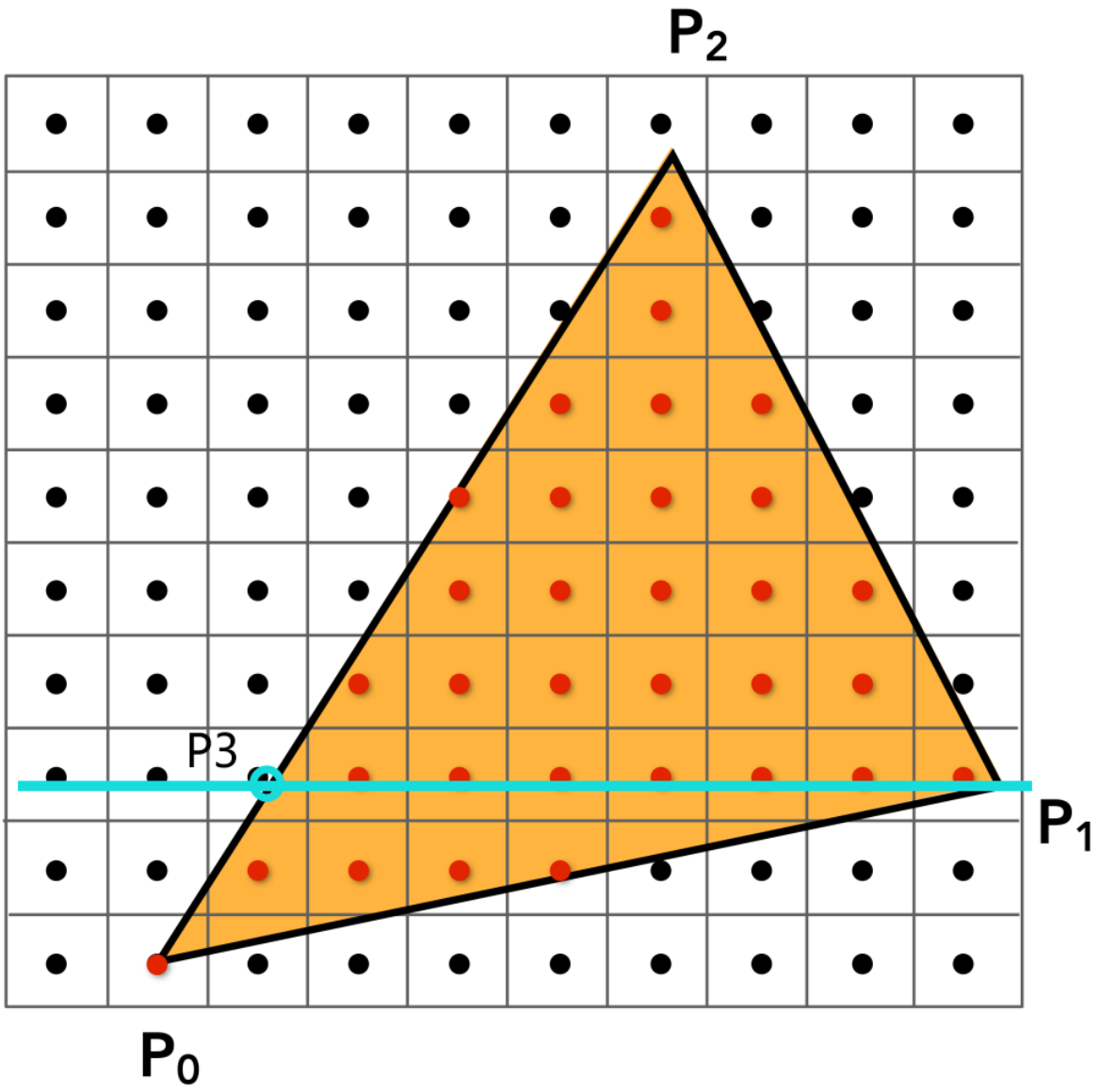
I thought this should work, but there were lots of white lines on edges of triangles:

This is probably because my edge judgement is not perfect, then some pixels are judged as "internal point" by one triangle, while as "edge point" by another, then it is not well super-sampled. After hours of debugging, I finally gave up due to lack of time.

7.3 Analytic geometry

I still want to keep the traversing inside the triangle rather than the bound box. So I thought I can use the functions of the lines to compute the left-most and the right-most of each row of pixel in the triangle.

First, I sort the vertices by y coordinates and partitioned a triangle into two parts, one with flat bottom and one with flat top



Then I implemented a function to rasterize the "flat triangles". I have to know the $\frac{1}{slope}$ of two lines ($\frac{1}{k_{P0P1}}$ and $\frac{1}{k_{P0P3}}$ for example), and calculate the left-most and right-most point of each row. For some point $P(x_n, y_n)$ in the n^{th} row, if $left\_most\_x_{n-1} < x_n < right\_most\_x_{n-1}$ and $left\_most\_x_{n+1} < x_n < right\_most\_x_{n+1}$, it is an "internal point", otherwise it is an "edge point", which should be super-sampled.

This algorithm also produced white lines on the edge of triangles, unfortunately. After hours of adjustment, I gave up and went back to SSAA.

```
void RasterizerImp::rasterize_flat_triangle(float x0, float y0,
                                            float x1, float y1,
                                            float x2, float y2,
                                            Color color) {
    if (y0 == y1) {
        if (x0 > x1) { swap(x0, x1); }

        float m02 = (x2-x0) / (y2-y0);
        float m12 = (x2-x1) / (y2-y1);
        for (int y = floor(y0); y <= floor(y2); ++y) {
            int upper_xl, upper_xr, xl, xr, lower_xl, lower_xr;
            if (x2 < x0) {
```

```
                upper_xl = max(floor(x2), floor(x0 + m02*(y+2-y0)));
                upper_xr = min(floor(x1), floor(x1 + m12*(y+1-y0)));
                xl = max(floor(x2), floor(x0 + m02*(y+1-y0)));
                xr = min(floor(x1), floor(x1 + m12*(y-y0)));
                lower_xl = y == floor(y0) ? 0 : max(floor(x2), floor(x0 + m02*(y-y0)));
                lower_xr = y == floor(y0) ? 0 : min(floor(x1), floor(x1 + m12*(y-1-y0)));
            }
            else if (x2 >= x0 && x2 <= x1) {
                upper_xl = max(floor(x0), floor(x0 + m02*(y+1-y0)));
                upper_xr = min(floor(x1), floor(x1 + m12*(y+1-y0)));
                xl = max(floor(x0), floor(x0 + m02*(y-y0)));
                xr = min(floor(x1), floor(x1 + m12*(y-y0)));
                lower_xl = y == floor(y0) ? 0 : max(floor(x0), floor(x0 + m02*(y-1-y0)));
                lower_xr = y == floor(y0) ? 0 : min(floor(x1), floor(x1 + m12*(y-1-y0)));
            }
            else {
                upper_xl = max(floor(x0), floor(x0 + m02*(y+1-y0)));
                upper_xr = min(floor(x2), floor(x1 + m12*(y+2-y0)));
                xl = max(floor(x0), floor(x0 + m02*(y-y0)));
                xr = min(floor(x2), floor(x1 + m12*(y+1-y0)));
                lower_xl = y == floor(y0) ? 0 : max(floor(x0), floor(x0 + m02*(y-1-y0)));
                lower_xr = y == floor(y0) ? 0 : min(floor(x2), floor(x1 + m12*(y-y0)));
            }

            for (int x = xl; x <= xr; ++x) {
                if (x > lower_xl && x > upper_xl && x < lower_xr && x < upper_xr) {
                    rasterize_point(x, y, color);
                }
                else {
                    //rasterize_point(x, y, color);
                    int sx = floor(x), sy = floor(y);
                    super_sample_pixel new_ssp;
                    if (SSP_buffer.find(make_pair(sx, sy)) != SSP_buffer.end())
                        new_ssp = SSP_buffer[make_pair(sx, sy)];
                    else
                        new_ssp.color_buffer.resize(sample_rate, Color::White);

                    int ind = 0;
                    for (float i = sy + 0.5*sample_unit; i < sy+1; i += sample_unit) {
                        for (float j = sx + 0.5*sample_unit; j < sx+1; j += sample_unit) {
                            if (p_in_triangle(j, i, x0, y0, x1, y1, x2, y2)){
                                new_ssp.color_buffer[ind] = color;
                            }
```

```
                            ind++;
                        }
                    }
                    SSP_buffer[make_pair(sx, sy)] = new_ssp;
                }
            }
        }
    }
    else { // y1 == y2
        if (x1 > x2) { swap(x1, x2); }
        //cout << (y1 == y2) << " " << y0 << " " << y1 << " " << y2 << endl;
        float m01 = (x1-x0) / (y1-y0);
        float m02 = (x2-x0) / (y2-y0);
        for (int y = floor(y1); y > floor(y0); --y) {
            int upper_xl, upper_xr, xl, xr, lower_xl, lower_xr;
            if (x0 < x1) {
                upper_xl = y == floor(y1) ? 0 : max(floor(x0), floor(x1 + m01*(y+1-y1)));
                upper_xr = y == floor(y1) ? 0 : min(floor(x2), floor(x2 + m02*(y+2-y1)));
                xl = max(floor(x0), floor(x1 + m01*(y-y1)));
                xr = min(floor(x2), floor(x2 + m02*(y+1-y1)));
                lower_xl = max(floor(x0), floor(x1 + m01*(y-1-y1)));
                lower_xr = min(floor(x2), floor(x2 + m02*(y-y1)));
            }
            else if (x0 >= x1 && x0 <= x2) {
                upper_xl = y == floor(y1) ? 0 : max(floor(x1), floor(x1 + m01*(y+2-y1)));
                upper_xr = y == floor(y1) ? 0 : min(floor(x2), floor(x2 + m02*(y+2-y1)));
                xl = max(floor(x1), floor(x1 + m01*(y+1-y1)));
                xr = min(floor(x2), floor(x2 + m02*(y+1-y1)));
                lower_xl = max(floor(x1), floor(x1 + m01*(y-y1)));
                lower_xr = min(floor(x2), floor(x2 + m02*(y-y1)));
            }
            else {
                upper_xl = y == floor(y1) ? 0 : max(floor(x1), floor(x1 + m01*(y+2-y1)));
                upper_xr = y == floor(y1) ? 0 : min(floor(x0), floor(x2 + m02*(y+1-y1)));
                xl = max(floor(x1), floor(x1 + m01*(y+1-y1)));
                xr = min(floor(x0), floor(x2 + m02*(y-y1)));
                lower_xl = max(floor(x1), floor(x1 + m01*(y-y1)));
                lower_xr = min(floor(x0), floor(x2 + m02*(y-1-y1)));
            }

            for (int x = xl; x <= xr; ++x) {
                if (x > lower_xl+100 && x > upper_xl+100 && x < lower_xr-100 && x < upper_xr-100) {
                    rasterize_point(x, y-1, color);
                }
                else {
                    //rasterize_point(x, y, color);
                    int sx = floor(x), sy = floor(y)-1;
```

```
                    super_sample_pixel new_ssp;
                    if (SSP_buffer.find(make_pair(sx, sy)) !=
SSP_buffer.end())
                        new_ssp = SSP_buffer[make_pair(sx, sy)];
                    else
                        new_ssp.color_buffer.resize(sample_rate,
Color::White);

                    int ind = 0;
                    for (float i = sy + 0.5*sample_unit; i < sy+1; i +=
sample_unit) {
                        for (float j = sx + 0.5*sample_unit; j < sx+1; j +=
sample_unit) {
                            if (p_in_triangle(j, i, x0, y0, x1, y1, x2,
y2)) {
                                new_ssp.color_buffer[ind] = color;
                            }
                            ind++;
                        }
                    }
                    SSP_buffer[make_pair(sx, sy)] = new_ssp;
                }
            }
        }
    }
}
void RasterizerImp::rasterize_triangle(float x0, float y0,
                                       float x1, float y1,
                                       float x2, float y2,
                                       Color color) {
    // TODO: Task 1: Implement basic triangle rasterization here, no
supersampling
    // TODO: Task 2: Update to implement super-sampled rasterization

    // make y0 <= y1 <= y2
    if (y0 > y1) { swap(x0, x1); swap(y0, y1); }
    if (y1 > y2) { swap(x1, x2); swap(y1, y2); }
    if (y0 > y1) { swap(x0, x1); swap(y0, y1); }

    if (y0 == y1 || y1 == y2)
        rasterize_flat_triangle(x0, y0, x1, y1, x2, y2, color);
    else {
        // cout << x0 << ", " << y0 << "  " << x1 << ", " << y1 << "  " <<
x2 << ", " << y2 << endl;
        float xm = x0 + (y1-y0) * (x2-x0)/(y2-y0);
        // cout << xm << endl;
        rasterize_flat_triangle(x1, y1, xm, y1, x2, y2, color);
        rasterize_flat_triangle(x0, y0, x1, y1, xm, y1, color);
    }
}
void RasterizerImp::resolve_to_framebuffer() {
    // TODO: Task 2: You will likely want to update this function for
supersampling support
    for (int x = 0; x < width; ++x) {
        for (int y = 0; y < height; ++y) {
```

```
            Color col = Color::Black;
            if (SSP_buffer.find(make_pair(x, y)) == SSP_buffer.end())
                col = sample_buffer[y*width + x];
            else {
                super_sample_pixel ssp = SSP_buffer[make_pair(x, y)];
                for (int i = 0; i < sample_rate; i++)
                    col += ssp.color_buffer[i];
                col *= 1.0 / sample_rate;
            }

            for (int k = 0; k < 3; ++k) {
                this->rgb_framebuffer_target[3 * (y * width + x) + k] =
(&col.r)[k] * 255;
            }
        }
    }

}
```

Even though I didn't successfully optimize antialiasing on my own, I'm still glad to have learned a lot from learning about the techniques :)