Assignment 2: Rasterization & ZBuffering

22336226

王泓沣

基础任务

1. 实现Bresenham直线光栅化算法

该函数在`TRShaderPipeline.cpp`文件中，其中的`from`和`to`参数分别代表直线的起点和终点，`screen_width`和`screen_height`是窗口的宽高，超出窗口的点应该被丢弃。`rasterized_points`存放光栅化插值得到的点。

Source Code:

```
    void TRShaderPipeline::rasterize_wire_aux(
        const VertexData &from,
        const VertexData &to,
        const unsigned int &screen_width,
        const unsigned int &screen_height,
        std::vector<VertexData> &rasterized_points)
    {

        //1: Implement Bresenham line rasterization
        // Note: You shold use VertexData::lerp(from, to, weight) for
interpolation,
        //       interpolated points should be pushed back to rasterized_points.
        //       Interpolated points shold be discarded if they are outside the
window.

        //       from.spos and to.spos are the screen space vertices.

        int x0 = static_cast<int>(from.spos.x);
        int y0 = static_cast<int>(from.spos.y);
        int x1 = static_cast<int>(to.spos.x);
        int y1 = static_cast<int>(to.spos.y);

        int dx = abs(x1 - x0);
        int dy = abs(y1 - y0);

        int sx = x0 < x1 ? 1 : -1; // x 的前进方向
        int sy = y0 < y1 ? 1 : -1; // y 的前进方向

        // 判断斜率，决定主轴
        bool steep = dy > dx;
```

```
        if (steep) {
            // 交换 dx 和 dy
            std::swap(dx, dy);
            // 初始误差项
            int err = 2 * dx - dy;
            int x = x0;
            int y = y0;

            // 计算总步数用于插值
            int totalSteps = dy;
            int steps = 0;

            for (int i = 0; i <= dy; i++) {
                // 计算插值参数 t
                double t = totalSteps == 0 ? 0.0 : static_cast<double>(steps) /
totalSteps;
                VertexData temp = VertexData::lerp(from, to, t);

                if (temp.spos.x >= 0 && temp.spos.x < screen_width && temp.spos.y
>= 0 && temp.spos.y < screen_height) {
                    rasterized_points.push_back(temp);
                }

                y += sy;
                if (err >= 0) {
                    x += sx;
                    err -= 2 * dy;
                }
                err += 2 * dx;
                steps++;
            }
        }
        else {
            // 初始误差项
            int err = 2 * dy - dx;
            int x = x0;
            int y = y0;

            // 计算总步数用于插值
            int totalSteps = dx;
            int steps = 0;

            for (int i = 0; i <= dx; i++) {
                // 计算插值参数 t
                double t = totalSteps == 0 ? 0.0 : static_cast<double>(steps) /
totalSteps;
                VertexData temp = VertexData::lerp(from, to, t);

                if (temp.spos.x >= 0 && temp.spos.x < screen_width && temp.spos.y
>= 0 && temp.spos.y < screen_height) {
                    rasterized_points.push_back(temp);
                }

                x += sx;
                if (err >= 0) {
```

```
                    y += sy;
                    err -= 2 * dx;
                }
                err += 2 * dy;
                steps++;
            }
        }
    }
```

Parameter Analysis:

`x0`、`y0`：起点x坐标、y坐标；

`x1`、`y1`：终点x坐标、y坐标；

`dx`、`dy`：x方向上所差像素点个数、y方向上所差像素点个数；

`sx`、`sy`：从起点到终点的迭代方向

`err`：与理想直线的误差评价变量

`totalSteps`：步数总数，用于评价当前插值点在直线上的位置

`steps`：当前完成的步数

Algorithm

标准的Bresenham直线光栅化算法实现，通过评价x方向和y方向与理想直线偏移的差距选择下一个像素点。

给定两个点(x_0, y_0)和(x_1, y_1)，直线斜率为$k = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$，即直线为$y = kx + b$，$b$为截距。

设当前像素为(x_i, y_i)，则理想直线在$x_{i+1} = x_i + 1$处的真实y值为$k(x_i + 1) + b$，需要判断这一个值离y_i更近还是$y_i + 1$更近。

$$d1 = k(x_i + 1) + b - y_i$$
$$d2 = y_i + 1 - k(x_i + 1) - b$$

当$d1 > d2$时，$d1 - d2 > 0$，取$y_{i+1} = y_i + 1$，否则取$y_{i+1} = y_i$

$$d1 - d2 = 2k(x_i + 1) - 2y_i + 2b - 1$$

两边同时乘Δx得

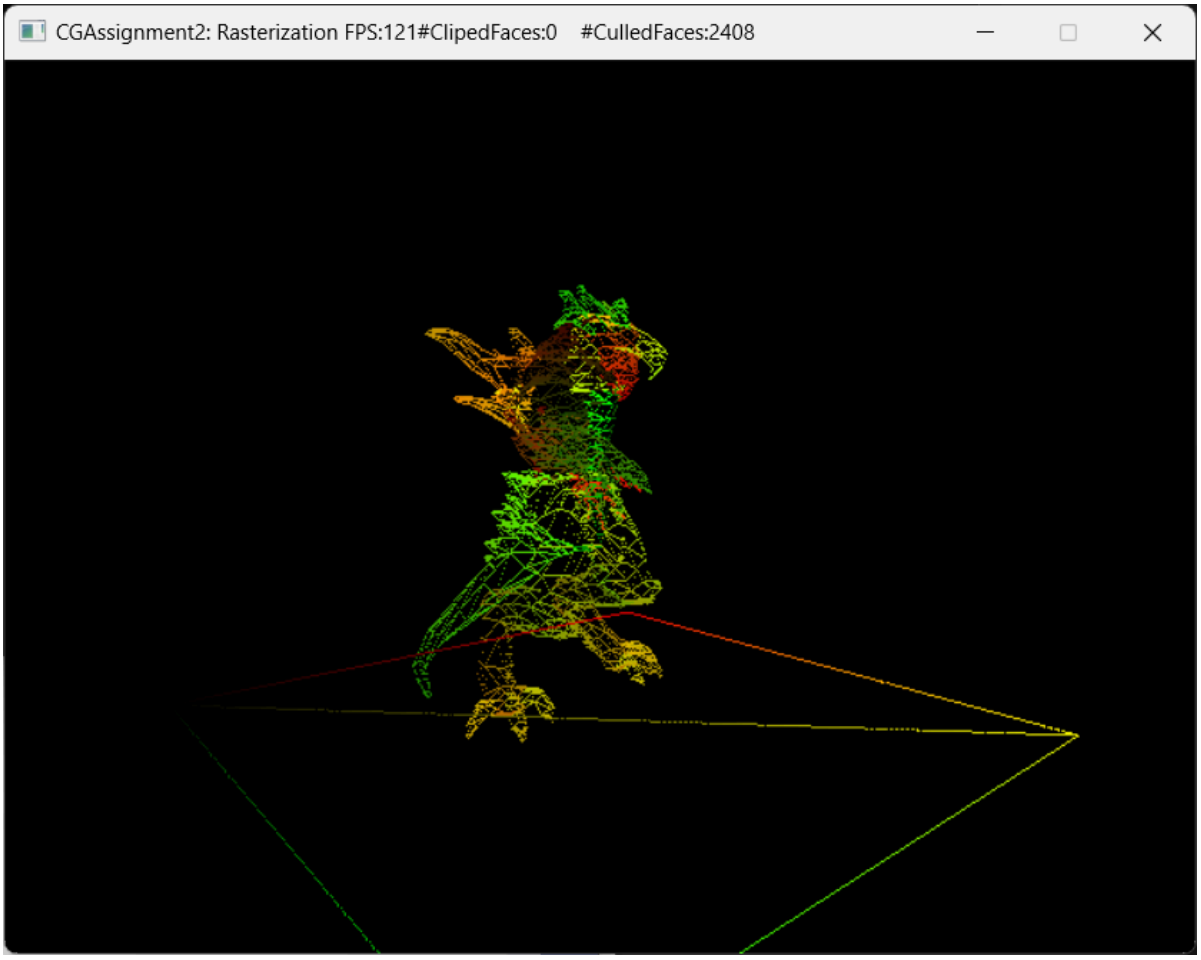
$$\Delta x(d1 - d2) = 2\Delta y \cdot x_i - 2\Delta x \cdot y_i + (2b - 1)\Delta x + 2\Delta y$$

令$p_i = \Delta x(d1 - d2)$，则p_i的符号可以作为选择标准，对于p_i的迭代计算有：

$$p_0 = 2\Delta y - \Delta x$$
$$p_{i+1} - p_i = 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$
$$p_{i+1} = \begin{cases} p_i + 2\Delta y, p_i \le 0 \\ p_i + 2\Delta y - 2\Delta x, p_i > 0 \end{cases}$$

以上为$k \le 1$时的推导，$k > 1$时交换x轴与y轴即可。

Result



2. 、实现基于Edge-function的三角形填充算法

基于Edge-function的三角形填充算法首先计算三角形的包围盒，然后遍历包围盒内的像素点，判断该像素点是否在三角形内部，这就是它的基本原理。

Source Code:

```
    void TRShaderPipeline::rasterize_fill_edge_function(
        const VertexData &v0,
        const VertexData &v1,
        const VertexData &v2,
        const unsigned int &screen_width,
        const unsigned int &screene_height,
        std::vector<VertexData> &rasterized_points)
    {
        //Edge-function rasterization algorithm

        // 2: Implement edge-function triangle rassterization algorithm
```

```
        // Note: You should use VertexData::barycentricLerp(v0, v1, v2, w) for
interpolation,
        //       interpolated points should be pushed back to rasterized_points.
        //       Interpolated points shold be discarded if they are outside the
window.

        //       v0.spos, v1.spos and v2.spos are the screen space vertices.

        int max_x = std::max({ v0.spos.x,v1.spos.x,v2.spos.x });
        int max_y = std::max({ v0.spos.y,v1.spos.y,v2.spos.y });
        int min_x = std::min({ v0.spos.x,v1.spos.x,v2.spos.x });
        int min_y = std::min({ v0.spos.y,v1.spos.y,v2.spos.y });
        for (int i = min_x; i <= max_x; i++) {
            for (int j = min_y; j <= max_y; j++) {
                glm::vec3 x = glm::vec3(v1.spos.x - v0.spos.x, v2.spos.x -
v0.spos.x, v0.spos.x - i);
                glm::vec3 y = glm::vec3(v1.spos.y - v0.spos.y, v2.spos.y -
v0.spos.y, v0.spos.y - j);
                glm::vec3 n = glm::cross(x, y);
                glm::vec3 w = { 1.0f - n.x / n.z - n.y / n.z,n.x / n.z,n.y / n.z
};

                if (w.x >= 0 && w.y >= 0 && w.x + w.y <= 1) {
                    VertexData p = VertexData::barycentricLerp(v0, v1, v2, w);
                    p.spos.x = i;
                    p.spos.y = j;
                    rasterized_points.push_back(p);
                }
            }
        }
    }
```

Parameter Analysis

`max_x`、`min_x`、`max_y`、`min_y`：三角形中x的最大最小值、y的最大最小值；

`x`：向量$v_0 \rightarrow v_1$；

`y`：向量$v_0 \rightarrow v_2$；

`n`：x向量和y向量构成平面的法向量

`w`：目标像素点坐标由三角形三个顶点坐标构成的系数

Algorithm

设目标像素点为P，若其在三角形内，则：

$$\overrightarrow{AP} = u\overrightarrow{AB} + v\overrightarrow{AC}$$
$$P = (1-u-v)A + uB + vC$$

其中$0 \leq u, v \leq 1$

对于二维三角形，有

$$[u,v,1]\begin{bmatrix}\overrightarrow{AB}_x\\ \overrightarrow{AB}_x\\ \overrightarrow{PA}_x\end{bmatrix}=0$$

$$[u,v,1]\begin{bmatrix}\overrightarrow{AB}_y\\ \overrightarrow{AB}_y\\ \overrightarrow{PA}_y\end{bmatrix}=0$$

所以$\overrightarrow{n}=(u,v,1)$为二维三角形x平面和y平面的法向量，由于计算中包含浮点数，所以$\overrightarrow{n}$的z分量不一定等于1.0.

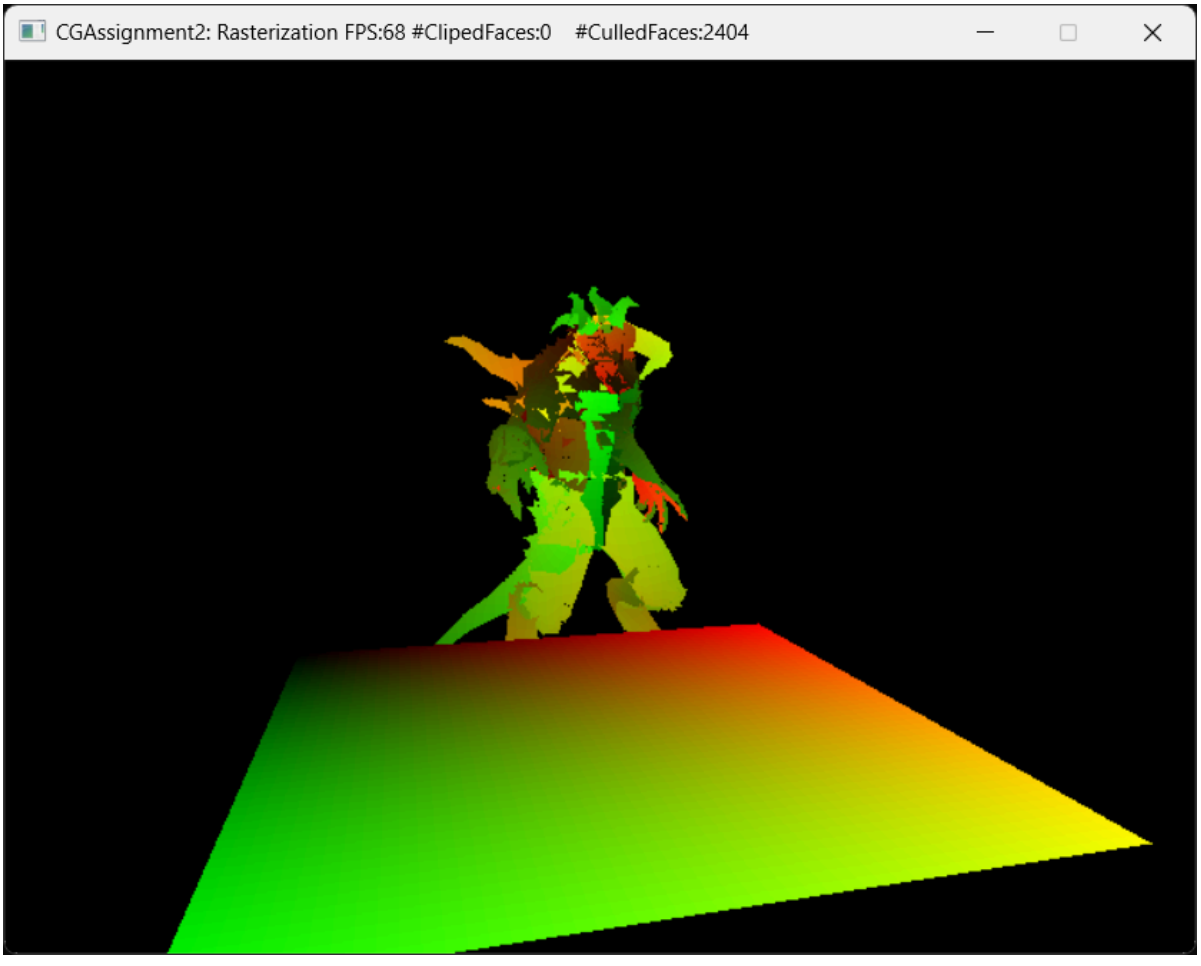
令$\overrightarrow{n}=(a,b,c)$，则

$$a\overrightarrow{AB}+b\overrightarrow{AC}+c\overrightarrow{PA}=0$$

$$P=(1-\frac{a}{c}-\frac{b}{c})A+\frac{a}{c}B+\frac{b}{c}C, c\neq 0$$

若$0\leq\frac{a}{c}\leq 1, 0\leq\frac{b}{c}\leq 1, (1-\frac{a}{c}-\frac{b}{c})\geq 0$，则P在三角形内，否则P在三角形外部。

Result



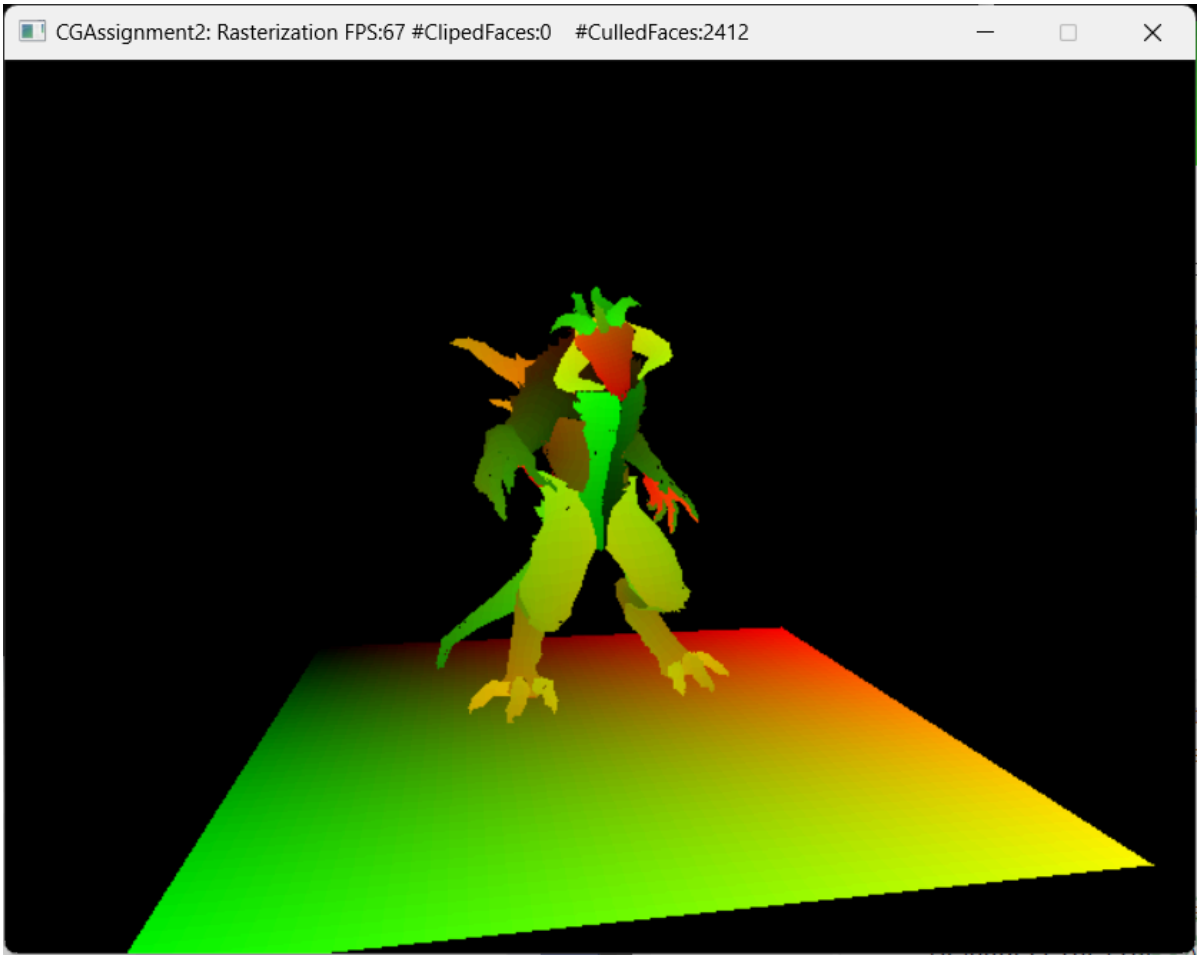
3. 实现深度测试

为了体现正确的三维前后遮挡关系，我们实现的帧缓冲包含了一个深度缓冲，用于存储当前场景中最近物体的深度值，前后的。三角形的三个顶点经过一系列变换之后，其z 存储了深度信息，取值为[-1,1]，越大则越远。经过光栅化的线性插值，每个片元都有一个深度值，存储在cpos.z 中。在着色阶段，我们可以用当前片元的cpos.z 与当前深度缓冲的深度值进行比较，如果发现深度缓冲的取值更小（即更近），则应该直接不进行着色器并写入到帧缓冲。

Source Code:

```
//Fragment shader & Depth testing
{
    for (auto &points : rasterized_points)
    {
        // 3: Implement depth testing here
        // Note: You should use m_backBuffer->readDepth() and points.spos to read
the depth in buffer
        //       points.cpos.z is the depth of current fragment
        {
            //Perspective correction after rasterization
            if (m_backBuffer->readDepth(points.spos.x, points.spos.y) <
points.cpos.z) {
                continue;
            }
            TRShaderPipeline::VertexData::aftPrespCorrection(points);
            glm::vec4 fragColor;
            m_shader_handler->fragmentShader(points, fragColor);
            m_backBuffer->writeColor(points.spos.x, points.spos.y, fragColor);
            m_backBuffer->writeDepth(points.spos.x, points.spos.y,
points.cpos.z);
        }
    }
}
```

Result



提升任务

正确绘制两个三角形的前后关系

为了帮助你更好地掌握光栅化与Z-buffer算法，我们希望你能正确光栅化三角形，并通过深度缓冲区，正确绘制两个三角形的前后关系。
为此你需要正确填写并调用在文件rasterizer.cpp 的rasterize_triangle 函数，该函数的内部工作流程如下：
1、创建三角形的2维包围盒。
2、遍历此包围盒内的所有像素（使用其整数索引）。然后，使用像素中心的屏幕空间坐标来检查中心点是否在三角形内。
3、如果在内部，则将其位置处的插值深度值与深度缓冲区中的相应值进行比较。
4、如果当前点更靠近相机，请设置像素颜色并更新深度缓冲区。
因为我们只知道三角形三个顶点处的深度值，所以对于三角形内部的像素，我们需要使用插值的方法得到其深度值。插值的深度值被存储在z_interpolated 中。请注意我们是如何初始化 depth buffer 和注意 z values 的符号。为了方便同学们写代码，我们将 z 进行了反转，保证都是正数，并且越大表示离视点越远。

Source Code:

```
//Screen space rasterization
void rst::rasterizer::rasterize_triangle(const Triangle& t) {
    auto v = t.toVector4();

    // Task_improve: Find out the bounding box of current triangle.
    // iterate through the pixel and find if the current pixel is inside the
triangle

    // If so, use the following code to get the interpolated z value.

    //auto[alpha, beta, gamma] = computeBarycentric2D(x, y, t.v);
    //float w_reciprocal = 1.0/(alpha / v[0].w() + beta / v[1].w() + gamma /
v[2].w());
    //float z_interpolated = alpha * v[0].z() / v[0].w() + beta * v[1].z() /
v[1].w() + gamma * v[2].z() / v[2].w();
    //z_interpolated *= w_reciprocal;

    // TODO : set the current pixel (use the set_pixel function) to the color of
the triangle (use getColor function) if it should be painted.
    float min_x = std::min({ v[0].x(), v[1].x(), v[2].x() });
    float max_x = std::max({ v[0].x(), v[1].x(), v[2].x() });
    float min_y = std::min({ v[0].y(), v[1].y(), v[2].y() });
    float max_y = std::max({ v[0].y(), v[1].y(), v[2].y() });

    int x_min = static_cast<int>(std::floor(min_x));
    int x_max = static_cast<int>(std::ceil(max_x));
    int y_min = static_cast<int>(std::floor(min_y));
    int y_max = static_cast<int>(std::ceil(max_y));

    for (int i = x_min; i <= x_max; i++) {
        for (int j = y_min; j <= y_max; j++) {
            if (insideTriangle(i+0.5f, j+0.5f, t.v)) {
                auto[alpha, beta, gamma] = computeBarycentric2D(i+0.5f, j+0.5f,
t.v);
                float w_reciprocal = 1.0f/(alpha / v[0].w() + beta / v[1].w() +
gamma / v[2].w());
                float z_interpolated = alpha * v[0].z() / v[0].w() + beta *
v[1].z() / v[1].w() + gamma * v[2].z() / v[2].w();
                z_interpolated *= w_reciprocal;

                int index = get_index(i, j);
                if (z_interpolated < depth_buf[index]) {
                    depth_buf[index] = z_interpolated;
                    set_pixel(Eigen::Vector3f(i, j, z_interpolated),
t.getColor());
                }

            }
        }
    }
```

```
}
```

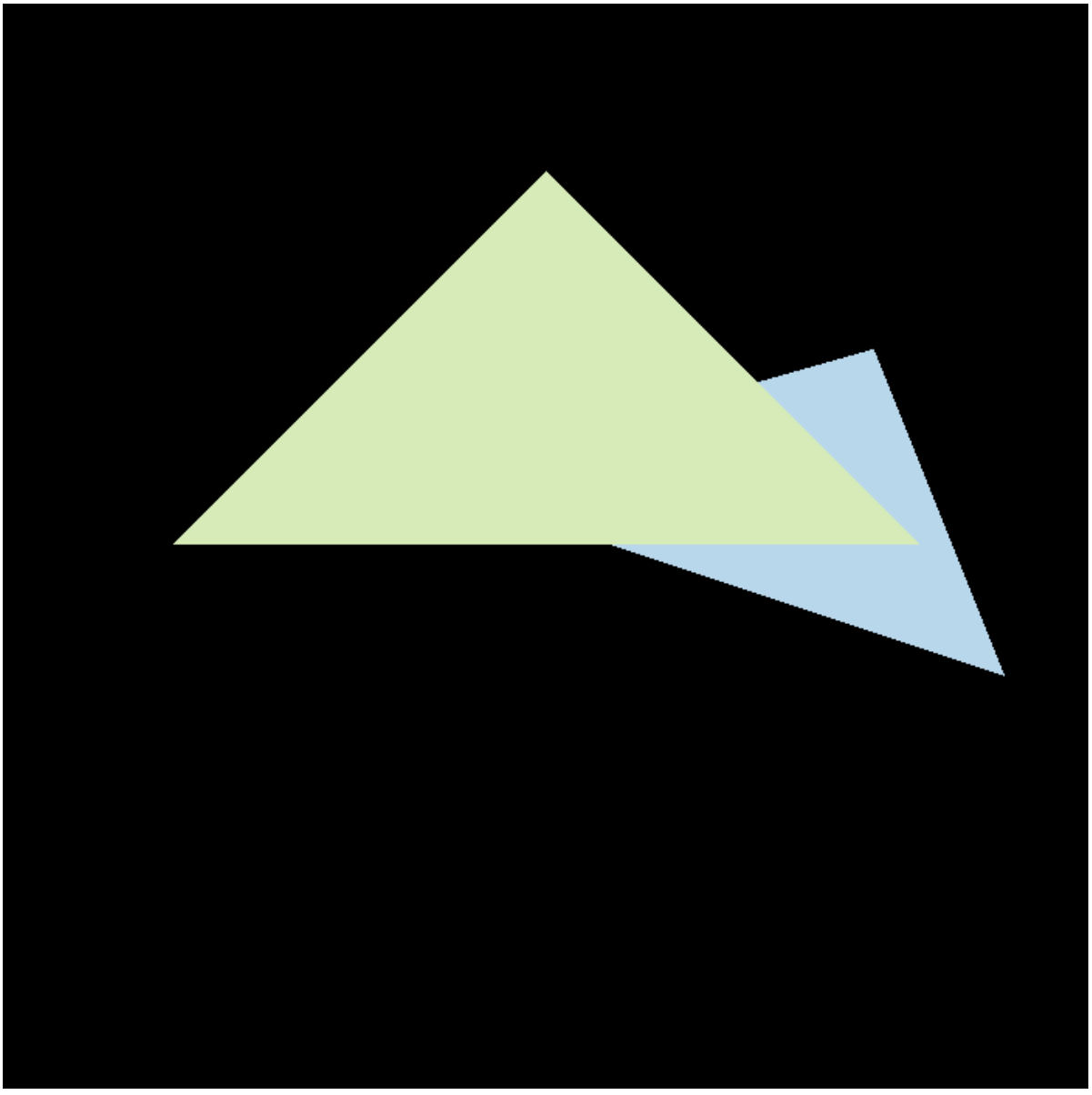
Parameter Analysis

`min_x`、`max_x`、`min_y`、`max_y`：三角形的二维包围盒的四个顶点

`x_min`、`x_max`、`y_min`、`y_max`：三角形的二维包围盒的四个顶点整数索引

`z_interpolated`：所遍历的像素中心点的深度值

2. 提升任务实现效果



挑战任务

超采样抗锯齿

用 super-sampling 处理 Anti-aliasing : 对每个像素进行 2 * 2 采样，并比较前后的结果 (这里并不需要考虑像素与像素间的样本复用)。需要注意，对于像素内的每一个样本都需要维护它自己的深度值，即每一个像素都需要维护一个 samplelist。
为此，需要在对应位置添加超采样相关的数组及函数，主要修改仍然是在文件`rasterizer.cpp`的`rasterize_triangle`函数中。
最后，如果你实现正确的话，你得到的三角形不应该有不正常的黑边。

Source Code:

添加颜色及深度缓冲定义及初始化

```
        std::vector<Eigen::Vector3f> my_col_buf;
        std::vector<float> my_depth_buf;
```

```
void rst::rasterizer::clear(rst::Buffers buff)
{
    if ((buff & rst::Buffers::Color) == rst::Buffers::Color)
    {
        std::fill(frame_buf.begin(), frame_buf.end(), Eigen::Vector3f{0, 0, 0});
        std::fill(my_col_buf.begin(), my_col_buf.end(), Eigen::Vector3f{ 0,0,0
});
    }
    if ((buff & rst::Buffers::Depth) == rst::Buffers::Depth)
    {
        std::fill(depth_buf.begin(), depth_buf.end(),
std::numeric_limits<float>::infinity());
        std::fill(my_depth_buf.begin(), my_depth_buf.end(),
std::numeric_limits<float>::infinity());
    }
}
```

抗锯齿三角光栅化

```
void rst::rasterizer::rasterize_triangle(const Triangle& t) {
    auto v = t.toVector4();

    // Task_improve: Find out the bounding box of current triangle.
    // iterate through the pixel and find if the current pixel is inside the
triangle

    // If so, use the following code to get the interpolated z value.

    //auto[alpha, beta, gamma] = computeBarycentric2D(x, y, t.v);
```

```
    //float w_reciprocal = 1.0/(alpha / v[0].w() + beta / v[1].w() + gamma /
v[2].w());
    //float z_interpolated = alpha * v[0].z() / v[0].w() + beta * v[1].z() /
v[1].w() + gamma * v[2].z() / v[2].w();
    //z_interpolated *= w_reciprocal;

    // TODO : set the current pixel (use the set_pixel function) to the color of
the triangle (use getColor function) if it should be painted.

    int super_sample_rate = 2;

    float min_x = std::min({ v[0].x(), v[1].x(), v[2].x() });
    float max_x = std::max({ v[0].x(), v[1].x(), v[2].x() });
    float min_y = std::min({ v[0].y(), v[1].y(), v[2].y() });
    float max_y = std::max({ v[0].y(), v[1].y(), v[2].y() });

    int x_min = static_cast<int>(std::floor(min_x));
    int x_max = static_cast<int>(std::ceil(max_x));
    int y_min = static_cast<int>(std::floor(min_y));
    int y_max = static_cast<int>(std::ceil(max_y));

    for (int x = x_min; x <= x_max; x++) {
        for (int y = y_min; y <= y_max; y++) {
            std::vector<float> sample_depths;
            std::vector<Eigen::Vector3f> sample_colors;
            for (int sx = 0; sx < super_sample_rate; sx++) {
                for (int sy = 0; sy < super_sample_rate; sy++) {
                    float sub_x = x + (sx + 0.5) / super_sample_rate;
                    float sub_y = y + (sy + 0.5) / super_sample_rate;
                    if (!insideTriangle(sub_x, sub_y, t.v)) {
                        continue;
                    }
                    auto [alpha, beta, gamma] = computeBarycentric2D(sub_x,
sub_y, t.v);
                    if (alpha >= 0 && beta >= 0 && gamma >= 0) {
                        float w_reciprocal = 1.0f / (alpha / v[0].w() + beta /
v[1].w() + gamma / v[2].w());
                        float z_interpolated = alpha * v[0].z() / v[0].w() + beta
* v[1].z() / v[1].w() + gamma * v[2].z() / v[2].w();
                        z_interpolated *= w_reciprocal;
                        int mind = get_index(x, y) + sx * 2 + sy;
                        if (my_depth_buf[mind] > z_interpolated) {
                            depth_buf[mind] = z_interpolated;
                            my_col_buf[mind] = t.getColor() / (super_sample_rate
* super_sample_rate);
                            my_depth_buf[mind] = z_interpolated;
                        }
                    }
                }
            }
            int index = get_index(x, y);
            Vector3f inserted_color = { 0,0,0 };
            for (int i = 0; i < super_sample_rate * super_sample_rate; i++) {
                inserted_color += my_col_buf[index + i];
            }
            Eigen::Vector3f point = Eigen::Vector3f(x, y, 1.0f);
```

```
                set_pixel(point,inserted_color);
            }
        }
    }
```

Parameter Analysis

`super_sample_rate`：超采样率

`sub_x`、`sub_y`：子采样点坐标；

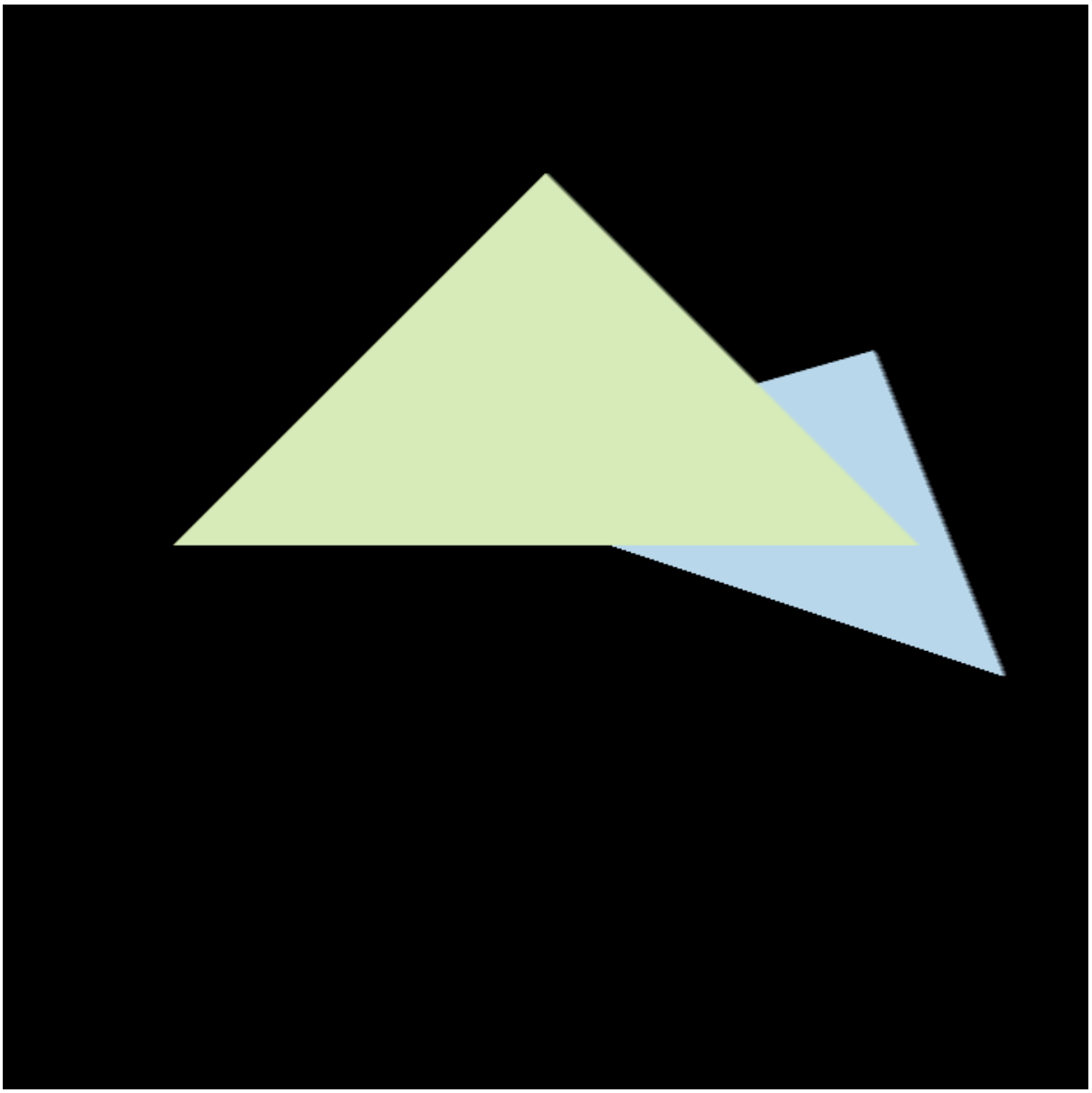
`mind`：一个超采样像素内的子采样点的深度缓冲区索引；

Algorithm

超采样即将每个像素划分为多个子采样点，通常以均匀的方式分布在像素内。对于每个子采样点，记录其颜色值与深度值，最终对所有子采样点的像素进行平均计算像素的最终颜色，获得更平滑的颜色过渡。

1. 定义超采样率，指定每个像素内的子采样次数
2. 计算三角形在屏幕空间中的2D包围盒
3. 遍历包围盒中像素，在每个像素内进行超采样循环，计算子采样点的坐标，将每个子采样点平均到像素中心。之后，检查每个子采样点是否在三角形内部，若在，计算重心坐标`alpha`、`beta`、`gamma`。执行深度插值计算，并根据深度值与颜色将子采样点的深度和颜色加权存储在缓冲区中。遍历一个像素的所有子采样点后对该像素内所有子采样点颜色进行平均，得到最终颜色。

2. 挑战任务实现效果



Reference (挑战任务)

https://github.com/jjL357/SYSU_Computer-Graphics/blob/main/assignment2