Assignment2 实验报告

姓名：林筱涵
学号：22336141

基础任务：

1.实现Bresenham直线光栅化算法。

Bresenham算法：该算法是一种基于整数运算的高效线段光栅化算法，适用于将直线段从一个点绘制到另一个点。它通过计算误差来决定每次应该在 x 方向还是 y 方向进行步进，从而确保光栅化的直线接近理想的真实直线。
该算法通过在每次迭代中选择一个新的像素点，使得光栅化的线段最小化了像素与实际直线的偏差。

代码如下：

```
    //rasterized_points.push_back(to);
    // 获取起点和终点的屏幕空间坐标
    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);

    // 计算 dx 和 dy
    int dx = x1 - x0;
    int dy = y1 - y0;

    // 确定步长方向
    int stepX = (dx > 0) ? 1 : -1; // 水平方向的步长
    int stepY = (dy > 0) ? 1 : -1; // 垂直方向的步长
    dx = std::abs(dx); // 取绝对值
    dy = std::abs(dy);

    // 初始化误差
    int err = dx - dy; // 初始误差

    // 光栅化直线，从起点到终点
    while (true) {
        // 检查当前像素是否在窗口内
        if (x0 >= 0 && x0 < screen_width && y0 >= 0 && y0 < screen_height) {
            // 计算插值因子
            float weight = (dx + dy > 0) ? static_cast<float>(std::abs(x0 - x1) + std::abs(y0 - y1)) / (dx + dy) : 0.0f;
            // 使用线性插值生成当前像素的 VertexData，并将其加入到 rasterized_points 中
            rasterized_points.push_back(VertexData::lerp(from, to, weight));
        }

        // 检查是否到达终点
        if (x0 == x1 && y0 == y1) break;

        // 计算新的误差
        int err2 = err * 2;

        // 更新 x 和 y 坐标
        if (err2 > -dy) {
            err -= dy; // 更新误差
            x0 += stepX; // 水平移动
        }
        if (err2 < dx) {
            err += dx; // 更新误差
            y0 += stepY; // 垂直移动
        }
    }
```

具体步骤如下：

1.首先获取起点（x0, y0）和终点坐标（x1, y1）

2.计算差值 dx 和 dy ， 其分别是水平方向和垂直方向的差值，表示起点到终点在 x 和 y 方向上的距离。

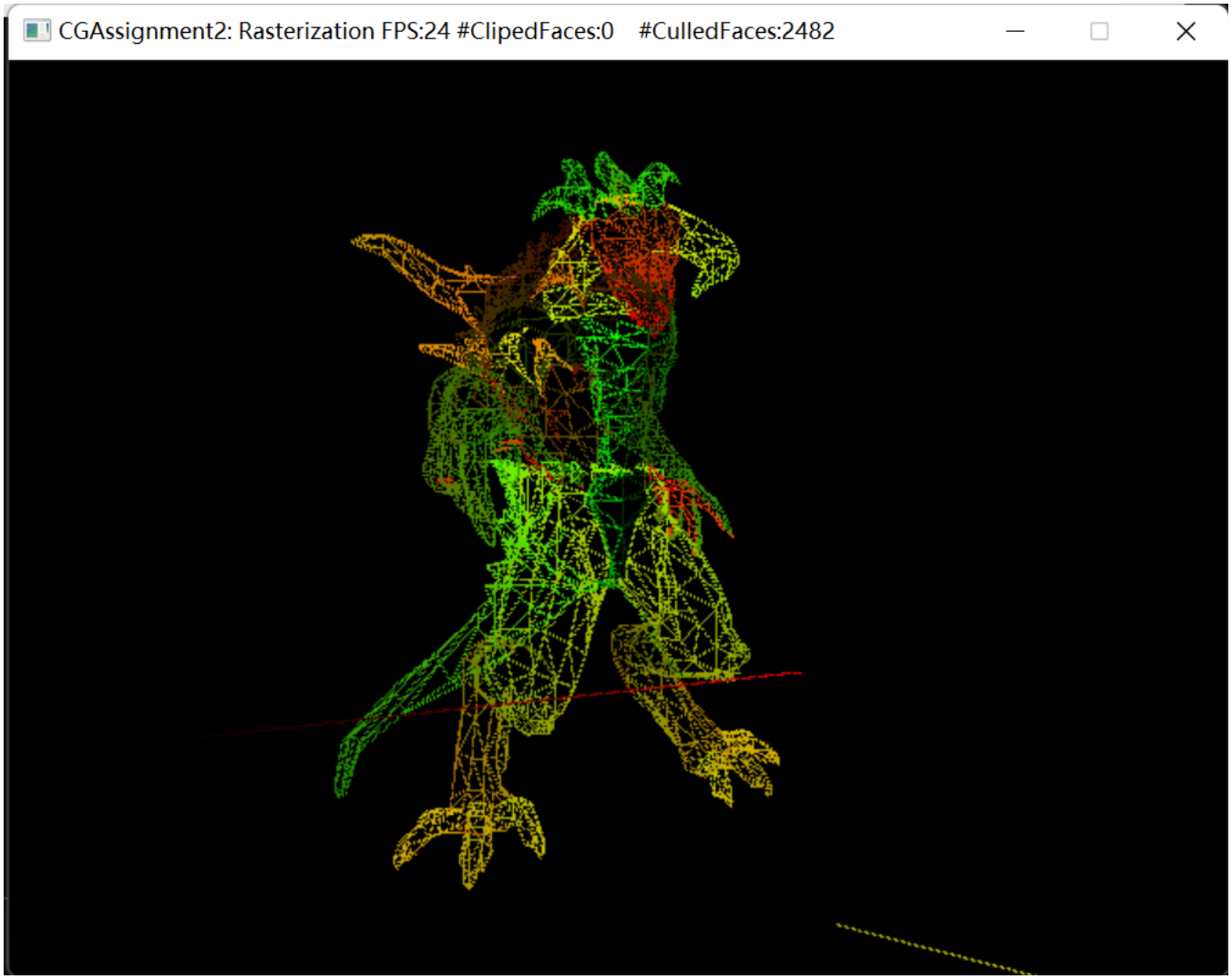
3.计算步长： stepX 和 stepY 决定了光栅化时在 x 和 y 方向上的步进方向。若 dx > 0，则步长为 1；若 dy > 0，则步长为 1，否则为 -1。

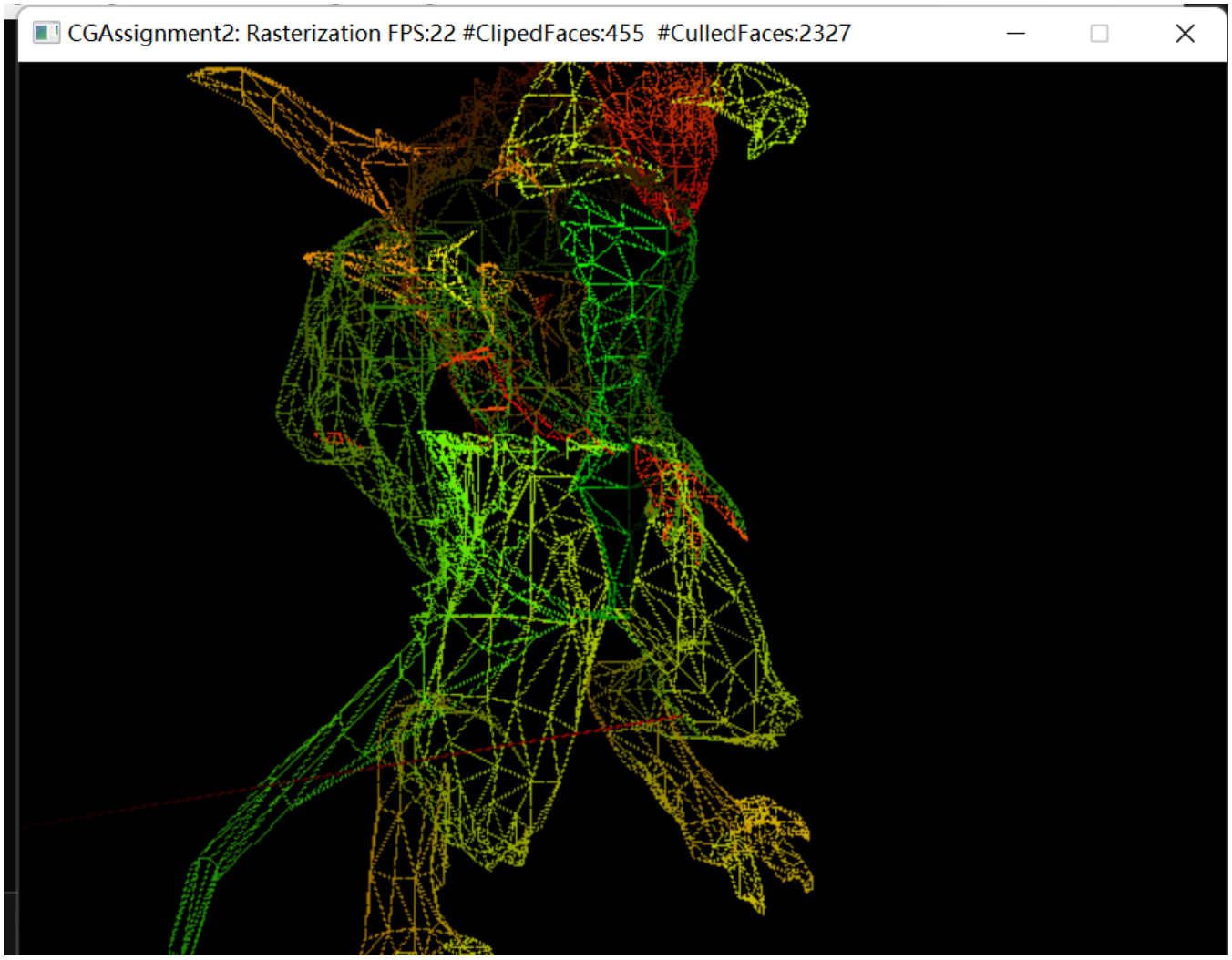
4.初始误差计算为 err = dx - dy，用于控制哪一方向的坐标变化更为优先。每次迭代时，会根据

误差的变化来调整 x 或 y 坐标。

5.误差更新：根据误差值 err 来判断是否需要沿着 x 轴或 y 轴步进，其中err2 = err * 2 通过将误差放大两倍来检查是否需要调整坐标。
若err2 > -dy，在 x 方向上步进；若err2 < dx ， 则在 y 方向上步进。

运行后结果如下图：





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

其原理是基于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。

代码如下：

```
    // 获取三角形的屏幕空间坐标
    int x0 = static_cast<int>(v0.spos.x);
    int y0 = static_cast<int>(v0.spos.y);
    int x1 = static_cast<int>(v1.spos.x);
    int y1 = static_cast<int>(v1.spos.y);
    int x2 = static_cast<int>(v2.spos.x);
    int y2 = static_cast<int>(v2.spos.y);

    // 计算三角形的包围盒
    int minX = std::max(0, std::min({ x0, x1, x2 }));
    int maxX = std::min(int(screen_width) - 1, std::max(x0, std::max(x1, x2)));
    int minY = std::max(0, std::min({ y0, y1, y2 }));
    int maxY = std::min(int(screen_height) - 1, std::max(y0, std::max(y1, y2)));

    // 遍历包围盒内的像素点
    for (int y = minY; y <= maxY; ++y) {
        for (int x = minX; x <= maxX; ++x) {
            glm::vec2 p(x + 0.5f, y + 0.5f); // 当前像素的中心坐标

            float w0 = edgeFunction(v1.spos, v2.spos, p);
            float w1 = edgeFunction(v2.spos, v0.spos, p);
            float w2 = edgeFunction(v0.spos, v1.spos, p);

            // 检查是否在三角形内部
            if (w0 >= 0 && w1 >= 0 && w2 >= 0) {
                // 计算重心坐标
                glm::vec3 weights = glm::vec3(w0, w1, w2);
                weights /= (w0 + w1 + w2); // 归一化

                // 使用重心插值生成当前像素的 VertexData，并将其加入到 rasterized_points 中
                rasterized_points.push_back(VertexData::barycentricLerp(v0, v1, v2, weights));
            }
        }
    }
```

具体步骤如下：

1、先获取三角形的屏幕空间坐标，将三角形三个顶点的屏幕空间坐标转换为整数，表示屏幕上的像素位置。

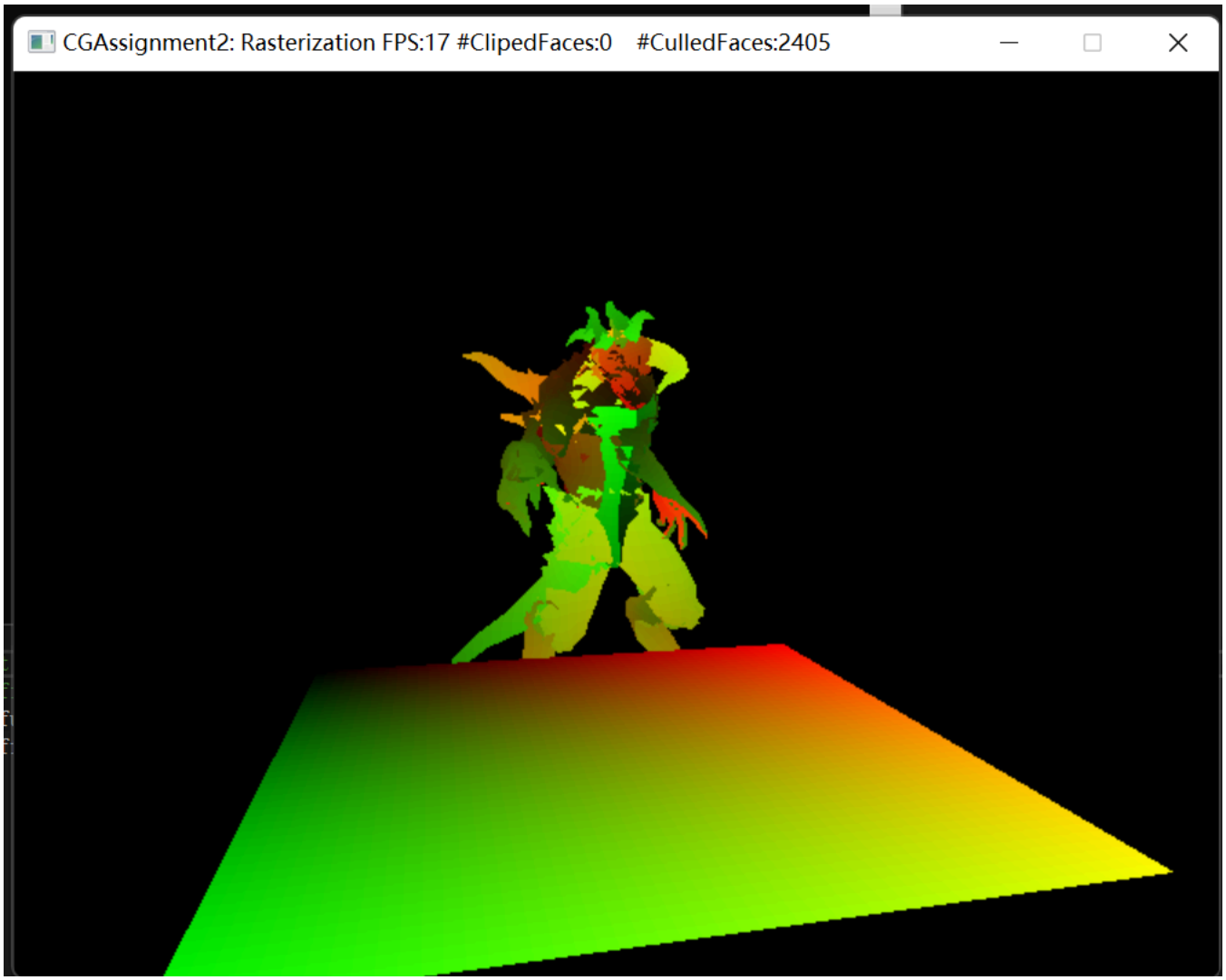
2、计算三角形包围盒，即一个矩形区域，包含了三角形的所有像素；通过 min 和 max 函数找到三角形顶点的最小和最大x、y 坐标。最后用minX、maxX、minY 和 maxY 定义三角形的边界。

3、遍历包围盒内的像素点

4、用edgeFunction函数计算每个像素点到三角形三条边的有向面积

5、计算重心坐标并进行插值

运行后结果如下图：



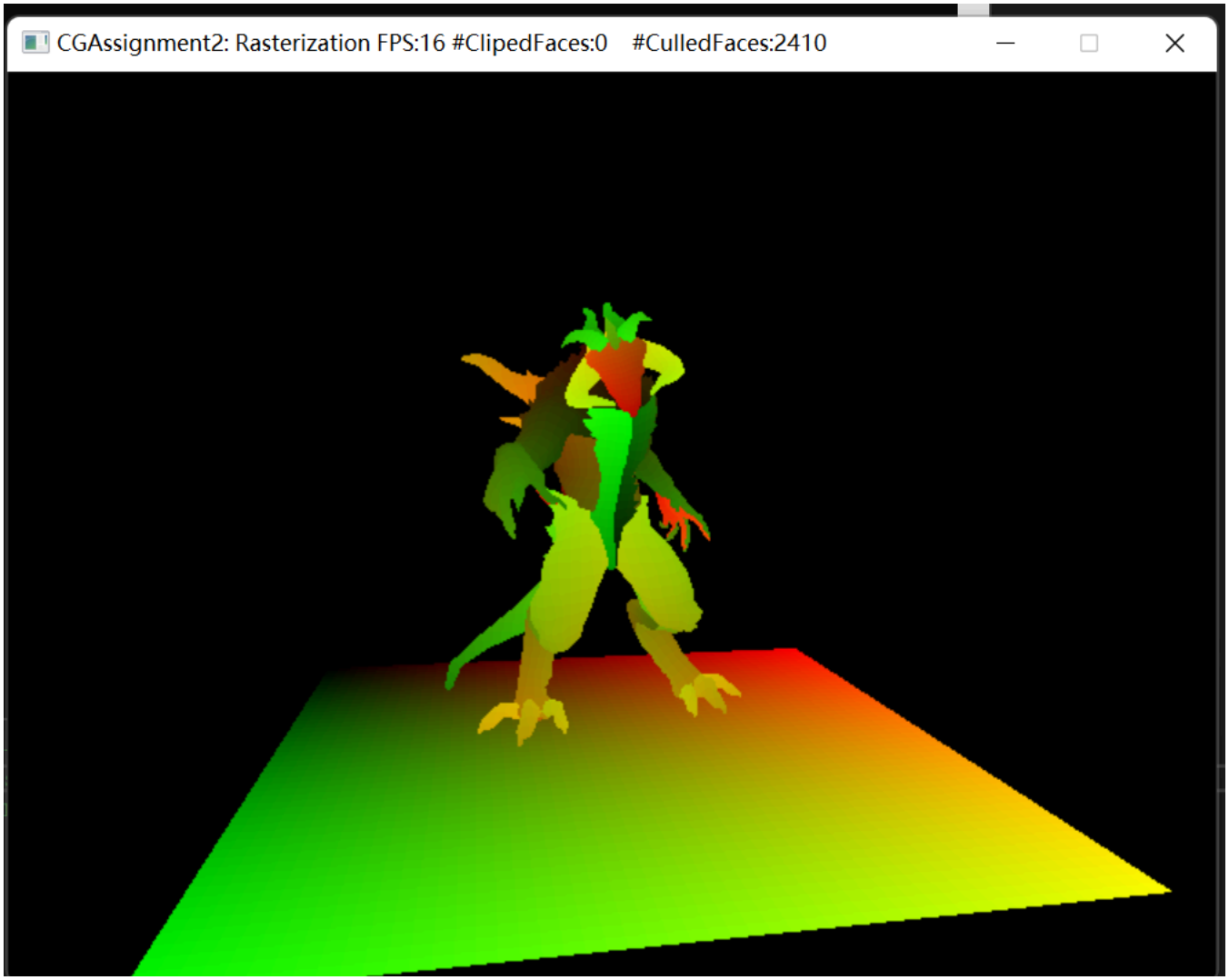
3、实现深度测试

深度测试的目标是确保只有最近的片元会被渲染，即如果当前片元距离视点更近，它就覆盖之前的片元；反之如果当前片元距离视点更远，它就被丢弃，不更新颜色和深度缓冲区。
代码如下：

```
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
        TRShaderPipeline::VertexData::aftPrespCorrection(points);
        glm::vec4 fragColor;
        m_shader_handler->fragmentShader(points, fragColor);
        //-------------------------------------------------------------以下是修改

        // 从深度缓冲中读取当前深度值
        float currentDepth = points.cpos.z; // 当前片元的深度
        float bufferDepth = m_backBuffer->readDepth(points.spos.x, points.spos.y); // 缓冲中的深度值

        // 深度测试

        if (currentDepth < bufferDepth) {
            // 如果当前片元更近，则更新颜色和深度缓冲
            m_backBuffer->writeColor(points.spos.x, points.spos.y, fragColor);
            m_backBuffer->writeDepth(points.spos.x, points.spos.y, currentDepth);
        }

        //-------------------------------------------------------------------------------
    //  m_backBuffer->writeColor(points.spos.x, points.spos.y, fragColor);
        //m_backBuffer->writeDepth(points.spos.x, points.spos.y, points.cpos.z);
    }
}
```

即通过比较当前片元的深度值和深度缓冲中的深度值，判断是否应该更新该片元的颜色和深度。
如果当前片元距离更近（深度值更小），则更新颜色和深度缓冲。
以此实现正确的遮挡关系，确保场景中离视点最近的物体可见，远离视点的物体被遮挡。

运行后结果如下图：



提升任务：为了帮助更好地掌握光栅化与Z-buffer算法，希望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。

代码如下：

```
//  提升任务---------------------------- -
// 将 toVector4 的结果赋值给 vertices
auto vertices = t.toVector4();

// 找到三角形的包围盒
int minX = std::min({ static_cast<int>(vertices.at(0).x()), static_cast<int>(vertices.at(1).x()), static_cast<int>(vertices.at(2).x()) });
int minY = std::min({ static_cast<int>(vertices.at(0).y()), static_cast<int>(vertices.at(1).y()), static_cast<int>(vertices.at(2).y()) });
int maxX = std::max({ static_cast<int>(vertices.at(0).x()), static_cast<int>(vertices.at(1).x()), static_cast<int>(vertices.at(2).x()) });
int maxY = std::max({ static_cast<int>(vertices.at(0).y()), static_cast<int>(vertices.at(1).y()), static_cast<int>(vertices.at(2).y()) });

//遍历包围盒内的像素
for (int x = minX; x <= maxX; x++) {
    for (int y = minY; y <= maxY; y++) {
        if (insideTriangle(x, y, t.v)) {
            ///计算重心坐标
            auto [alpha, beta, gamma] = computeBarycentric2D(x, y, t.v);
            //计算插值后的深度值
            float w_reciprocal = 1.0f / (alpha / vertices.at(0).w() + beta / vertices.at(1).w() + gamma / vertices.at(2).w());
            float z_interpolated = alpha * vertices.at(0).z() / vertices.at(0).w() + beta * vertices.at(1).z() / vertices.at(1).w() + gamma * vertices.at(2).z() / vertices.at(2).w
            z_interpolated *= w_reciprocal;
            //深度测试和颜色更新
            int index = get_index(x, y);

            if (z_interpolated < depth_buf[index]) {
                set_pixel(Eigen::Vector3f(x, y, z_interpolated), t.getColor());
                depth_buf[index] = z_interpolated;
            }
        }
    }
}
```

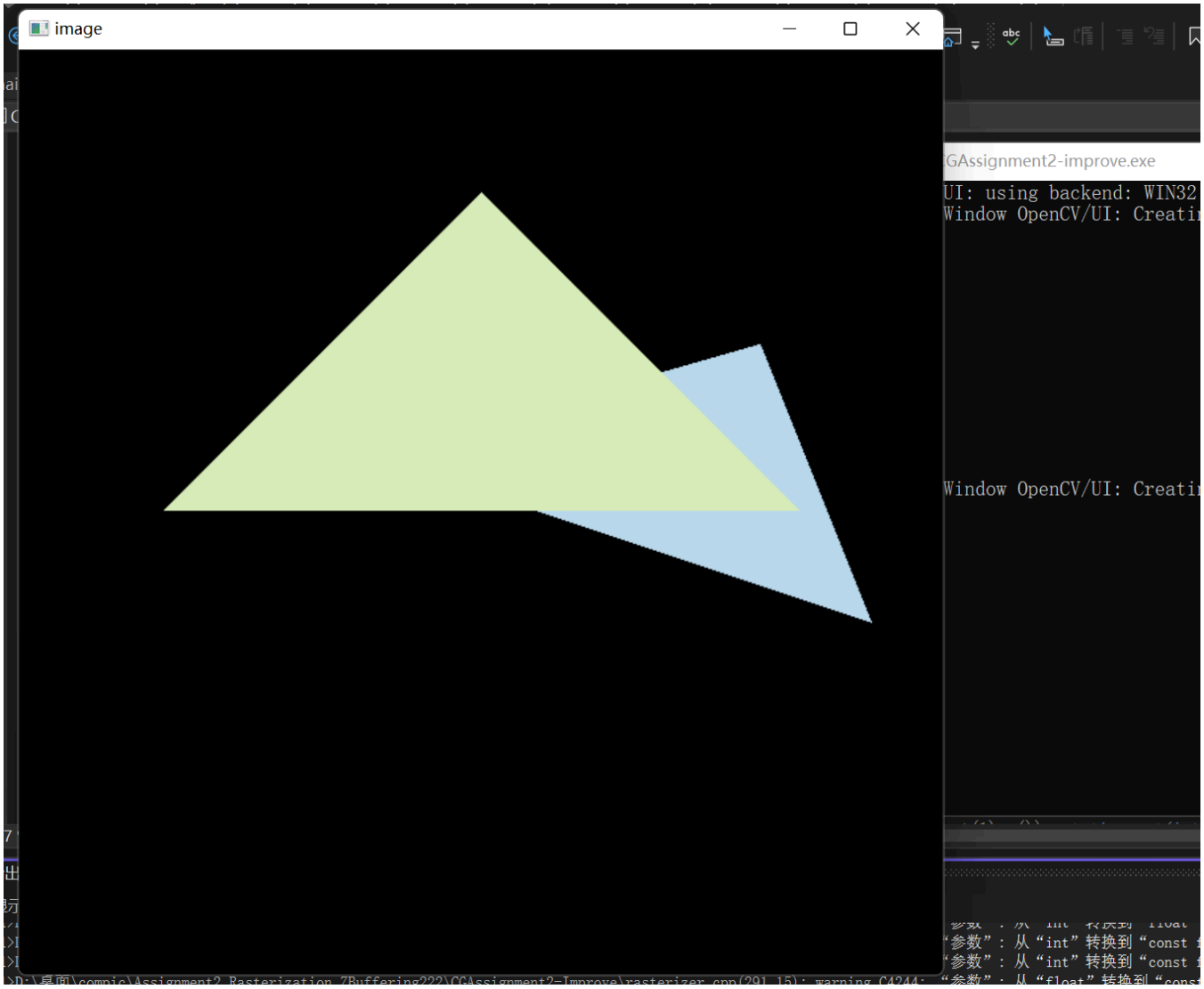
光栅化三角形：

将三角形的顶点投影到屏幕空间; 计算三角形的包围盒后遍历，检查每个像素是否位于三角形内部。

深度测试：

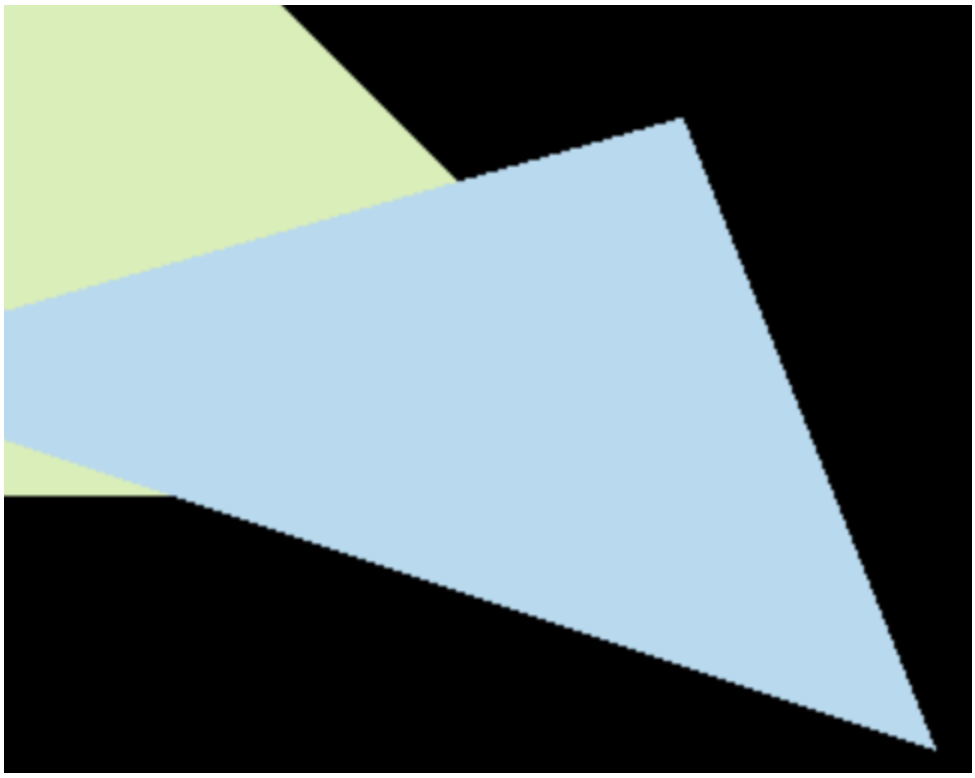
计算三角形重心坐标后，计算其插值后的深度值；用depth_buf[index] 存储像素 (x, y) 在深度缓冲区中的当前深度值，如果插值后的深度值z_interpolated小于缓冲区中的深度值，说明当前像素更接近视点，则更新缓冲区中的深度值，并绘制该像素的颜色；最后对通过深度测试的像素，更新其颜色值和深度值，渲染最终的三角形。

运行结果如下：



挑战任务：用 super-sampling来解决图像边缘有锯齿感的问题

放大提升任务中获得的图片，可以发现边缘有明显锯齿感



计算三角形包围盒的代码和提升任务一样，在遍历时对每个像素进行 2 * 2 采样
代码如下：

```
for (int subx = 0; subx < 2; subx++) // 在一个像素内进行 2x2 超采样，遍历子像素
{
    for (int suby = 0; suby < 2; suby++)
    {

        float x1 = x + 0.5 * subx; // 子像素的x坐标，偏移0.5实现超采样
        float y1 = y + 0.5 * suby; // 子像素的y坐标，偏移0.5实现超采样

        // 判断子采样点是否在当前三角形内
        if (!insideTriangle(x1, y1, t.v))
            continue; // 如果不在三角形内，跳过该点的处理

        // 计算该点的重心坐标
        auto [alpha, beta, gamma] = computeBarycentric2D(x1, y1, t.v);

        // 计算逆深度加权因子，用于插值矫正
        float w_reciprocal = 1.0 / (alpha / vertices[0].w() + beta / vertices[1].w() + gamma / vertices[2].w());

        // 计算插值后的深度值（z坐标），并应用归一化
        float z_interpolated = alpha * vertices[0].z() / vertices[0].w() +
            beta * vertices[1].z() / vertices[1].w() +
            gamma * vertices[2].z() / vertices[2].w();
        z_interpolated *= w_reciprocal;

        // 深度测试：检查当前深度值是否小于深度缓冲中的值
        if (z_interpolated < depth_buf[get_subindex(x, y, subx, suby)]) {
            depth_buf[get_subindex(x, y, subx, suby)] = z_interpolated; // 更新深度缓冲
            preframe_buf[get_subindex(x, y, subx, suby)] = t.getColor(); // 更新颜色缓冲
        }


    }
}
```

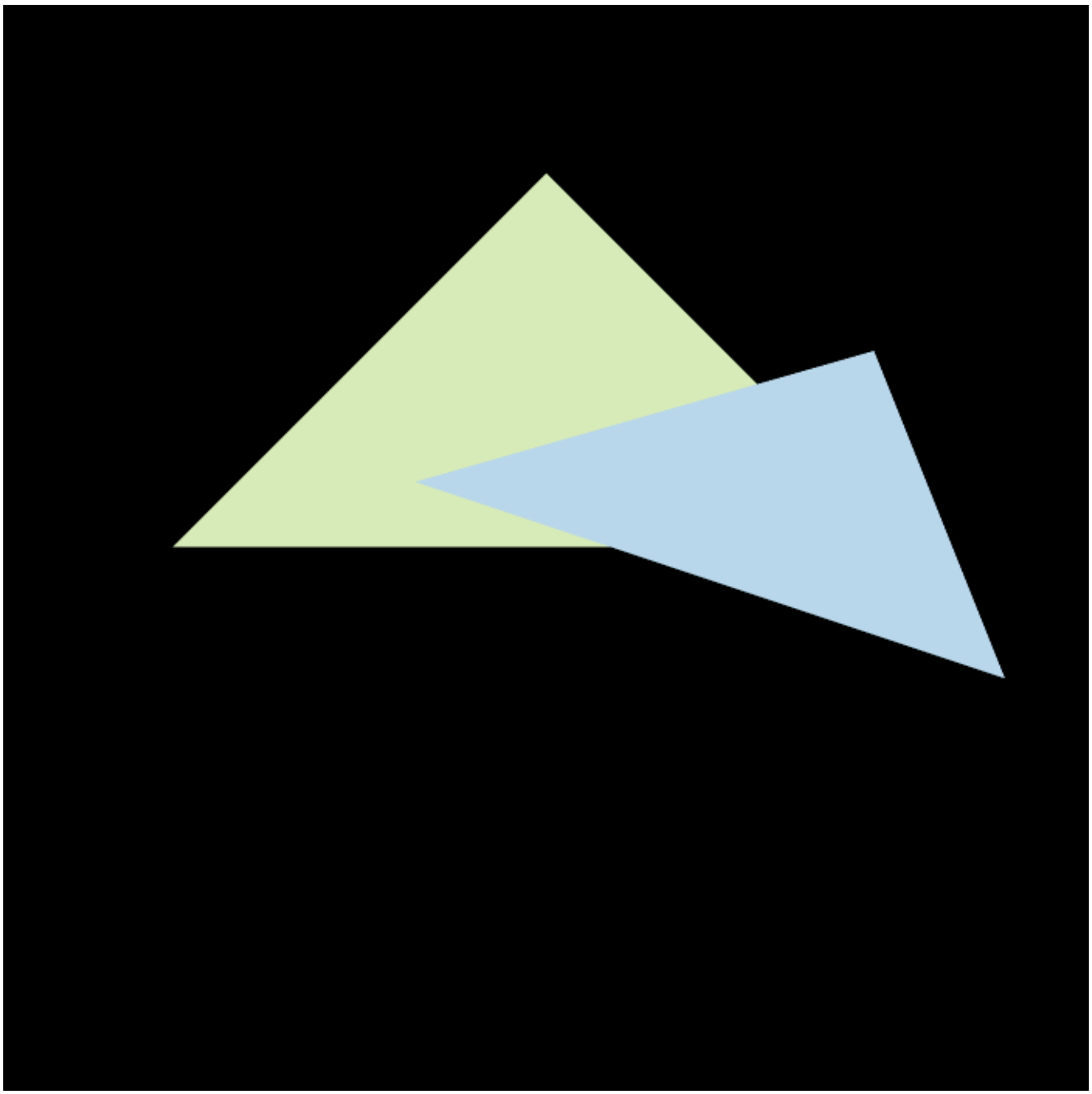
其中逆深度矫正插值：考虑齐次坐标的 w 分量，确保插值结果的精确性

在最终像素颜色计算时遍历 4 个子像素的颜色值，将它们累加后取平均值，生成像素点的最终颜色，通过平均多个子像素的颜色平滑像素的边缘：

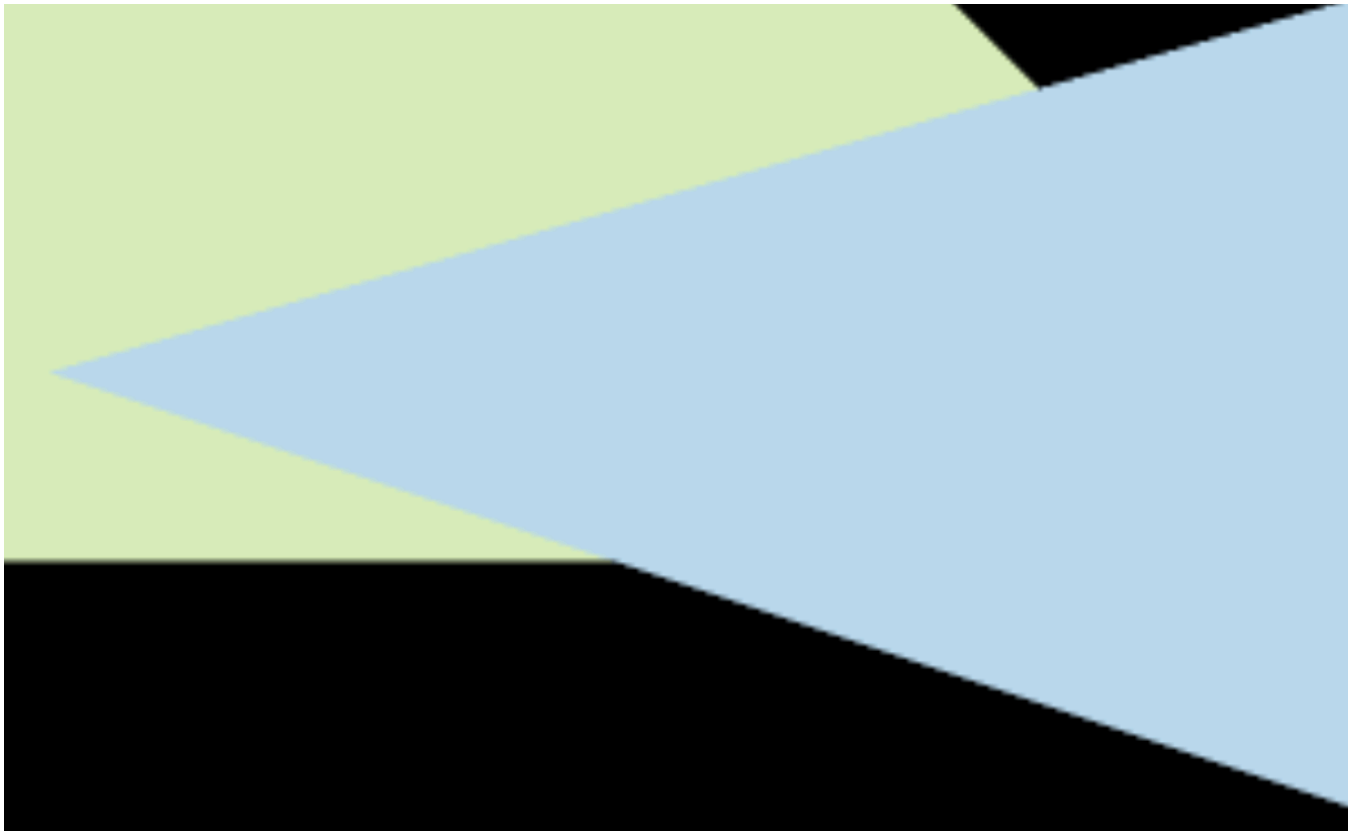
代码如下：

```
// 初始化像素点的颜色
Eigen::Vector3f newcolor(0.0f, 0.0f, 0.0f);

// 合并子采样点的颜色值，计算像素点的最终颜色
for (int subx = 0; subx < 2; subx++)
{
    for (int suby = 0; suby < 2; suby++)
        newcolor += preframe_buf[get_subindex(x, y, subx, suby)]; // 累加子采样点颜色
}

// 将像素的最终颜色设置为子采样点颜色的平均值
set_pixel(Vector3f(x, y, 1.0f), newcolor / 4); // 平均 4 个子像素的颜色
```

运行后结果如下图：



可以看出三角形边缘变平滑，放大后如下：



可以看出虽然三角形边缘不是完全光滑但是比提升任务中的锯齿状好了很多。