计算机图形学

Rasterization 与 Z-buffering

Assignment 3

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1 作业任务与要求

(1) 实现计算二维三角形的轴向包围盒；

(2) 实现判断给定的二维点是否在三角形内部；

(3) 实现 Z-buffering 算法；

(4) 思考并回答：光栅化的输入和输出分别是什么，光栅化主要负责做什么工作；

(5) 思考并回答：走样现象产生的原因是？列举几个抗锯齿的方法；

(6)（选做）实现反走样算法——超采样抗锯齿方法；

2 实验过程与结果

2.1 Task1：实现计算二维三角形的轴向包围盒

2.1.1 实现思路

得到二维三角形的轴向包围盒的方法十分简单。我们只需要分别得到三角形三个顶点的 x 轴坐标最小值、x 轴坐标最大值、y 轴坐标最小值和 y 轴坐标最大值即可。根据这四个值，我们就可以得到二维三角形的轴向包围盒。

2.1.2 代码实现

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

    float xmin = FLT_MAX, xmax = FLT_MIN;
    float ymin = FLT_MAX, ymax = FLT_MIN;
    // TODO 1: Find out the bounding box of current triangle.
    {
        xmin=MIN(v[0][0],MIN(v[1][0],v[2][0]));// x方向上的最小值
        xmax=MAX(v[0][0],MAX(v[1][0],v[2][0]));// x方向上的最大值
        ymin=MIN(v[0][1],MIN(v[1][1],v[2][1]));// y方向上的最小值
        ymax=MAX(v[0][1],MAX(v[1][1],v[2][1]));// y方向上的最大值
    }
    // After you have calculated the bounding box, please comment the following code.
    // return;
    ......
}
```

2.1.3 编译运行结果

编译运行结果如图1所示。

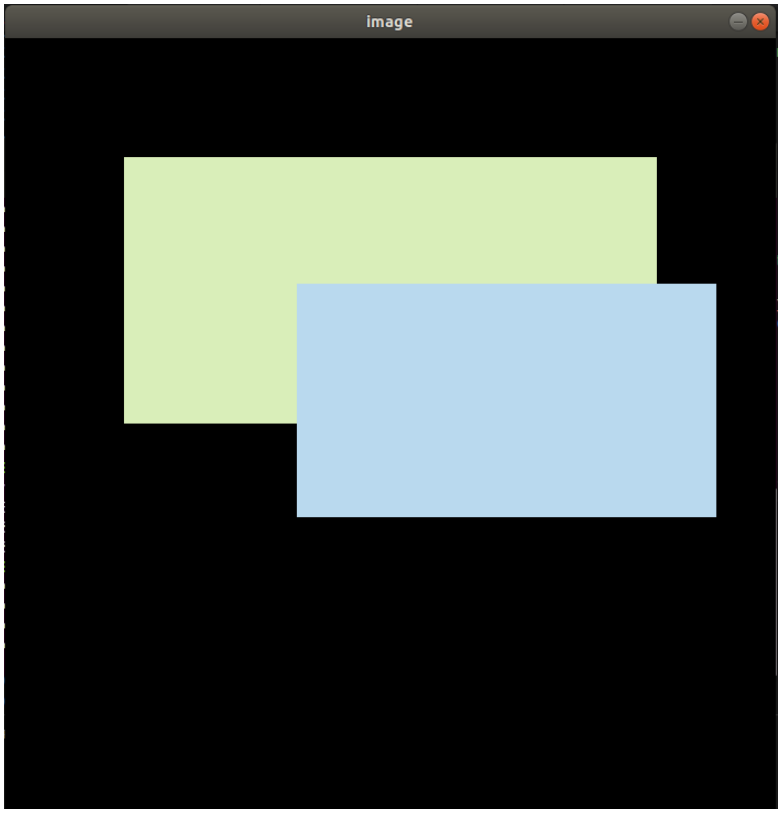

图 1: Task1 运行结果

2.2 Task2：实现判断给定的二维点是否在三角形内部

2.2.1 实现思路

得到三角形的轴向包围盒后，我们需要判断包围盒内的某一个二维点是否在三角形内部，以便于判断是否需要绘制该点。我们可以使用向量叉乘来判断一个点是否在三角形内部。假设该点为 P，三角形的三点分别为 A, B 和 C，那么我们分别计算以下向量：

$$\begin{aligned}\overrightarrow{v_1} &= \overrightarrow{AP} \times \overrightarrow{AB} \\ \overrightarrow{v_2} &= \overrightarrow{BP} \times \overrightarrow{BC} \\ \overrightarrow{v_3} &= \overrightarrow{CP} \times \overrightarrow{CA}\end{aligned}$$

如果 $\overrightarrow{v_1}, \overrightarrow{v_2}$ 和 $\overrightarrow{v_3}$ 方向相同，那么 P 在三角形内部，否则 P 在三角形外部。

因为我们需要判断的是二维点是否在二维三角形内部，故我们只需要判断 $\overrightarrow{v_1}, \overrightarrow{v_2}$ 和 $\overrightarrow{v_3}$ 的 z 值是否同号即可。

2.2.2 代码实现

```
static bool insideTriangle(float x, float y, const Vector3f* _v)
{
    // TODO 2: Implement this function to check if the point (x, y) is inside the triangle
            represented by _v[0], _v[1], _v[2]
    Eigen::Vector3f p(x,y,0.0f);
```

```
    Eigen::Vector3f v01=_v[1]-_v[0];
    Eigen::Vector3f c01=v01.cross(p-_v[0]);// 叉乘结果1
    Eigen::Vector3f v12=_v[2]-_v[1];
    Eigen::Vector3f c12=v12.cross(p-_v[1]);// 叉乘结果2
    Eigen::Vector3f v20=_v[0]-_v[2];
    Eigen::Vector3f c20=v20.cross(p-_v[2]);// 叉乘结果3
    // 如果方向相同，返回真
    if ((c01[2]>0 && c12[2]>0 && c20[2]>0)||(c01[2]<0 && c12[2]<0 && c20[2]<0))
    return true;
    else
    return false;
}
```

2.2.3 编译运行结果

编译运行结果如图2所示。

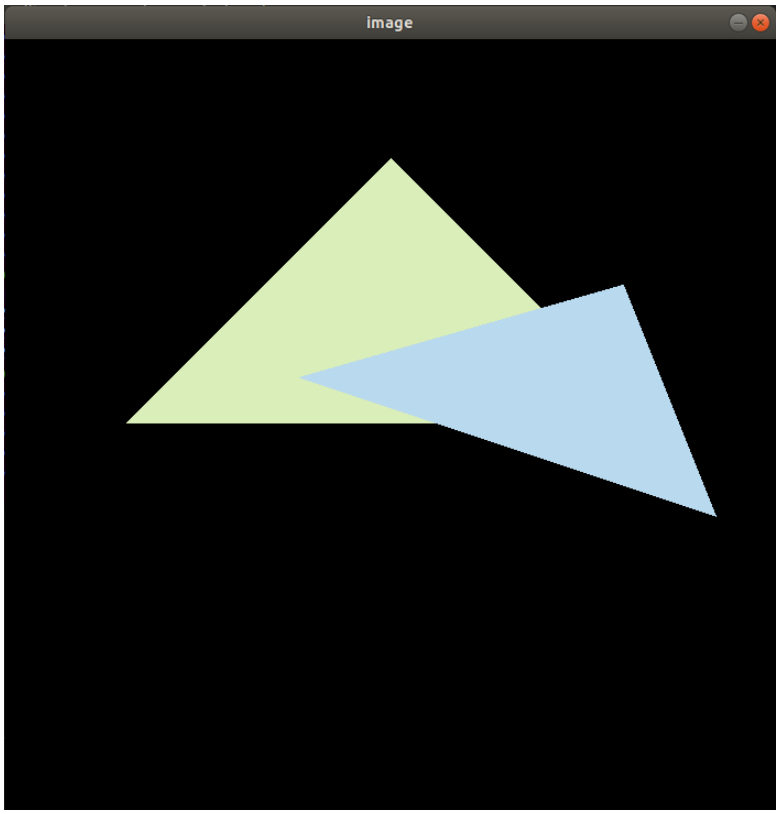

图 2: Task2 运行结果

可见，我们绘制出了两个三角形而不是方框。

2.3 Task3：实现 Z-buffering 算法

2.3.1 实现思路

在代码框架中，我们已经有了 $frame\_buf$ 和 $depth\_buf$，此外还有当前像素对应的深度值 $z\_interpolated$。因此，我们将 $z\_interpolated$ 与 $depth\_buf$ 中该像素对应的深度值进行比较。如果

$z\_interpolated$ 比当前 $depth\_buf$ 的深度值大，则不进行更新；反之，我们更新当前 $depth\_buf$ 对应的深度值为 $z\_interpolated$，同时更新 $frame\_buf$ 中该点的颜色信息。对所有像素点遍历一遍，我们就可以得到正确的 $frame\_buf$，将其显示出来就是准确的图像。而这也是 Z-buffering 算法的主要思想体现。

2.3.2 代码实现

```
void rst::rasterizer::rasterize_triangle(const Triangle& t)
{
    ......
    // iterate through the pixel and find if the current pixel is inside the triangle
    for(int x = static_cast<int>(xmin);x <= xmax;++x)
    {
        for(int y = static_cast<int>(ymin);y <= ymax;++y)
        {
            // if it's not in the area of current triangle, just do nothing.
            if(!insideTriangle(x, y, t.v))
            continue;
            // otherwise we need to do z-buffer testing.
            // use the following code to get the depth value of pixel (x,y), it's stored
                in z_interpolated
            // 计算子采样的深度
            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 3: perform Z-buffer algorithm here.
            // 如果当前深度大于depth_buf深度，不更新
            if (z_interpolated>=depth_buf[get_index(x,y)])
            continue;
            // 如果当前深度小于depth_buf深度，更新depth_buf和preframe_buf
            depth_buf[get_index(x,y)]=z_interpolated;
            // set the pixel color to frame buffer.
            frame_buf[get_index(x,y)] = 255.0f * t.color[0];
        }
    }
}
```

2.3.3 编译运行结果

编译运行结果如图3所示。

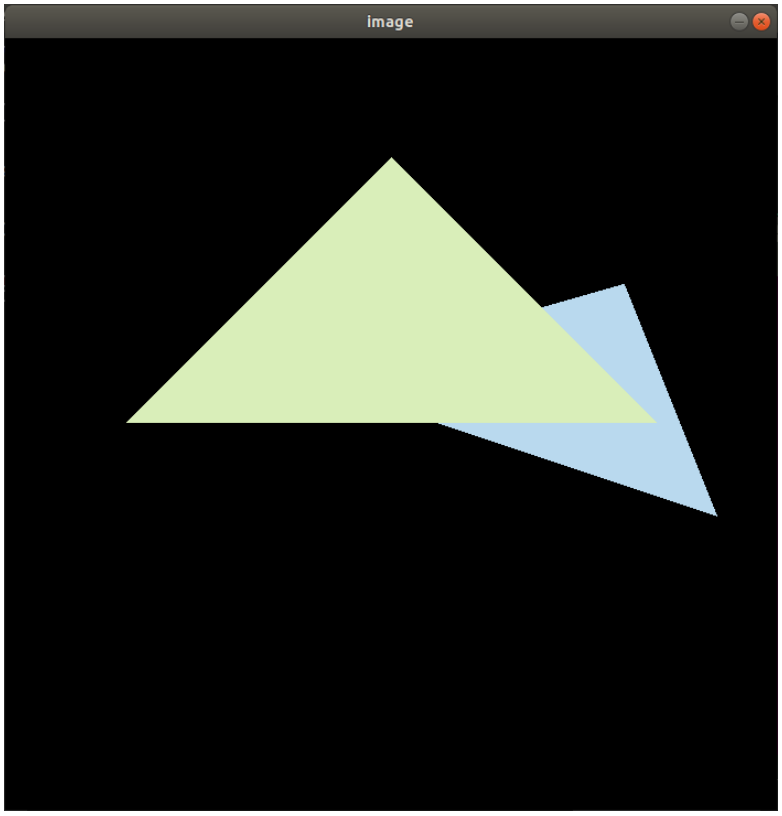

图 3: Task3 运行结果

可见，两个三角形处于正确的位置，深度更浅的黄色三角形位于深度更深的浅蓝色三角形上方。

2.4 Task4：思考并回答：光栅化的输入和输出分别是什么，光栅化主要负责做什么工作

光栅化就是把一个图元转换成一个二维图像的过程。该图像上的每个点都包含了颜色、深度等数据。对光栅化而言，其输入是一个几何单元的顶点数据，输出就是屏幕上每个像素点/栅格的对应数据，以此实现几何图形在电脑屏幕上的显示。

光栅化主要负责的工作如下：

(1) 获取图形的顶点数据；

(2) 根据图形的顶点，确定由哪些像素点构成图形；

(3) 根据深度、图形颜色等信息确认像素的颜色值等数据；

(4) 输出整个屏幕像素的对应数据，以便在屏幕上显示；

我们可以发现，光栅化的本质就是离散化，将连续的图形映射到由栅格组成的图像上。

2.5 Task5：思考并回答：走样现象产生的原因是？列举几个抗锯齿的方法

走样现象的产生是离散化导致的。显示设备的像素点是离散的，而需要显示的图形是连续的。在光栅化过程中对图形进行点采样时，如果采样率不够高，就会导致细节失真，一些信息出现丢失与错误。

抗锯齿的方法有以下几种：

(1) 提高图像的分辨率；

(2) 使用滤波器对图像预滤波，去除高频信息，然后再进行采样；

(3) 对像素数值进行平均化计算；

(4) 使用超采样算法对图像进行采样；

2.6 Task6：（选做）实现反走样算法——超采样抗锯齿方法

2.6.1 实现思路

超采样技术就是提高图像的采样率。对于原图像，每个像素对应 1×1 个采样点，而采用超采样技术后，每个像素对应 2×2 个采样点。在得到 4 个采样点的颜色值后，我们取平均即可得到像素的颜色值。

对于本次实验，图像分辨率为 700×700，那么我们设置一个大小为 1400×1400 的 $depth\_buf$ 和 1400×1400 的 $preframe\_buf$。按照上述过程，我们分别检测每个采样点是否在三角形内部、使用 Z-buffering 算法得到 $preframe\_buffer$ 中对应点的颜色值。最后，我们对 $preframe\_buf$ 进行取平均、合成，得到 700×700 的 $frame\_buf$，而这也是我们最终输出的图像。

2.6.2 代码实现

首先，我们要在 rasterizer.hpp 中添加对变量 $preframe\_buf$ 和用于得到原始采样点索引的函数 $get\_subindex()$ 的支持：

```
namespace rst
{
    ......
    class rasterizer
    {
    public:
        ......
    private:
        ......

    private:
        ......

        std::vector<Eigen::Vector3f> preframe_buf;
        std::vector<Eigen::Vector3f> frame_buf;
        std::vector<float> depth_buf;
        int get_index(int x, int y);
        int get_subindex(int x, int y, int subx, int suby);

        ......
    };
}
```

然后,我们在 rasterizer.cpp 中实现超采样。在设置三个 buffer 的大小时,$preframe\_buf$ 和 $depth\_buf$ 设置为 1400×1400，而 $frame\_buf$ 保持为 700×700：

```
rst::rasterizer::rasterizer(int w, int h) : width(w), height(h)
{
    preframe_buf.resize(4*w*h);
    frame_buf.resize(w*h);
    depth_buf.resize(4*w*h);
}
```

同样，在每轮迭代，我们要对三个 $buffer$ 都进行初始化：

```
void rst::rasterizer::clear(rst::Buffers buff)
{
    if ((buff & rst::Buffers::Color) == rst::Buffers::Color)
    {
        std::fill(preframe_buf.begin(), preframe_buf.end(), Eigen::Vector3f{0, 0, 0});
        std::fill(frame_buf.begin(), frame_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
            ());
    }
}
```

至于关键函数 $rasterize\_triangle()$，其修改如下：

```
void rst::rasterizer::rasterize_triangle(const Triangle& t)
{
    ......
    // iterate through the pixel and find if the current pixel is inside the triangle
    for(int x = static_cast<int>(xmin);x <= xmax;++x)
    {
        for(int y = static_cast<int>(ymin);y <= ymax;++y)
        {
            // 对每个像素的子采样点进行遍历
            for (int subx=0; subx<2; subx++)
            {
                for (int suby=0; suby<2; suby++)
                {
                    float x1=x+0.5*subx;// 子采样点的x坐标
                    float y1=y+0.5*suby;// 子采样点的x坐标
                    // if it's not in the area of current triangle, just do nothing.
                    if(!insideTriangle(x1, y1, t.v))// 判断子采样点是否在三角形内
                    continue;
                    // otherwise we need to do z-buffer testing.
                    // use the following code to get the depth value of pixel (x,y), it's
                        stored in z_interpolated
```

```
                    // 计算子采样的深度
                    auto[alpha, beta, gamma] = computeBarycentric2D(x1, y1, 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 3: perform Z-buffer algorithm here.
                    // 如果子采样点的深度大于当前深度，不更新
                    if (z_interpolated>=depth_buf[get_subindex(x,y,subx,suby)])
                    continue;
                    // 子采样点的深度小于当前深度，更新depth_buf和preframe_buf
                    depth_buf[get_subindex(x,y,subx,suby)]=z_interpolated;
                    preframe_buf[get_subindex(x,y,subx,suby)]=t.getColor();
                }
            }
            // 计算像素点的平均颜色值
            Eigen::Vector3f newcolor(0.0f,0.0f,0.0f);
            for (int subx=0; subx<2; subx++)
            {
                for (int suby=0; suby<2; suby++)
                newcolor+=preframe_buf[get_subindex(x,y,subx,suby)];
            }
            // 对该像素点的颜色进行赋值
            set_pixel(Vector3f(x,y,1.0f),newcolor/4);
        }
    }
}
```

与此同时，我们要修改判断二维点是否在三角形内的函数 *insideTriangle*() 的输入参数，以实现对浮点参数的支持：

```
static bool insideTriangle(float x, float y, const Vector3f* _v)
{
    ......
}
```

此外，为了实现能在大小变大后的 *depth_buf* 和 *preframe_buf* 中索引相应的值，我们需要函数 *get_subindex*()：

```
int rst::rasterizer::get_subindex(int x, int y, int subx, int suby)
{
    int X=2*x+subx;
    int Y=2*y+suby;
    return (2*height-1-Y)*2*width + X;
}
```

以上就是实现超采样的修改内容。

2.6.3 编译运行结果

编译代码后运行程序，结果如图4所示：

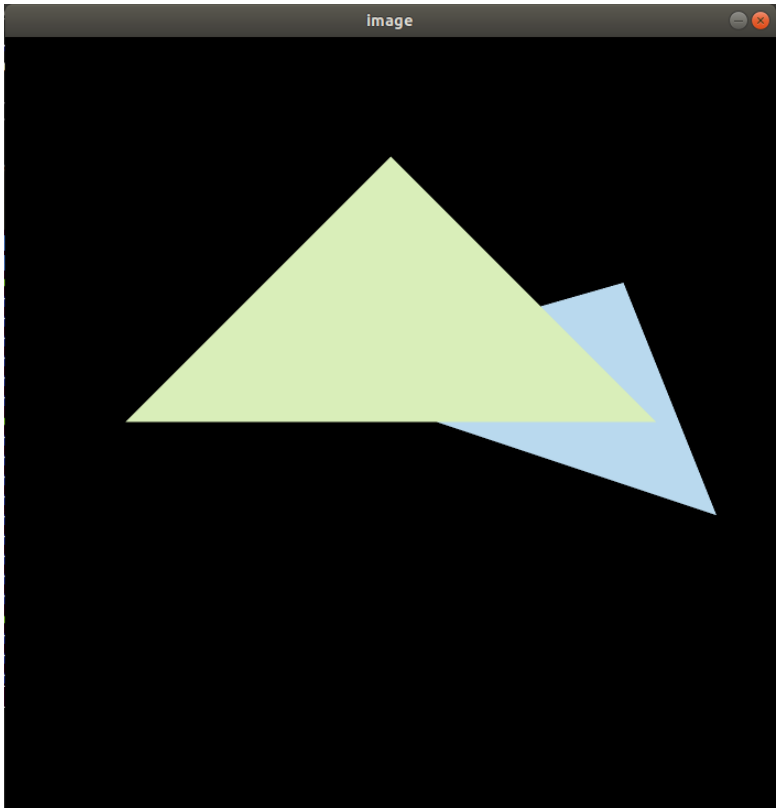

图 4: Task6 运行结果

可见，相比于图3，三角形边缘的锯齿感明显减弱。
我们可以放大边缘，可以看到超采样算法起到了良好的效果。

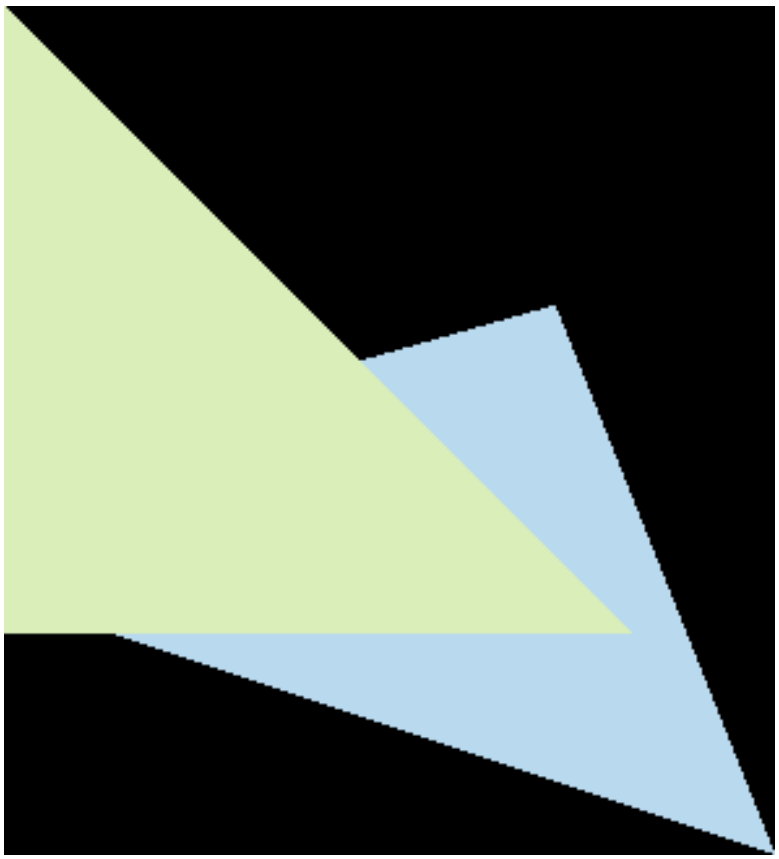

图 5: 图形边缘（未使用超采样）

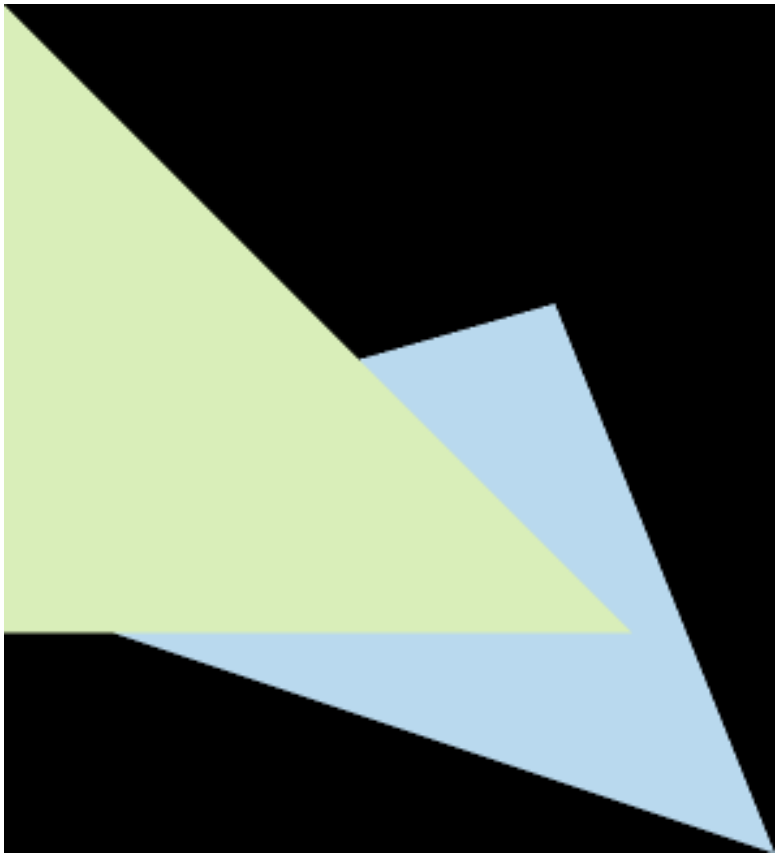

图 6: 图形边缘（使用超采样后）

3 作业总结与感想

本次作业主要是实现两个三角形的光栅化，涉及到轴向包围盒的获取、图形的绘制、Z-buffering 算法和超采样等内容。相比于以前的作业，本次作业难度较大，需要理解整个代码框架的内容和运作方式。在查阅了一些资料后，我花了几天的时间弄懂了整个框架中各部分代码的含义，并对框架进行了一些修改，实现了超采样算法。虽然使用超采样后代码运行速度明显降低，但抗锯齿的效果还算不错。

在此感谢老师和助教提供的帮助，希望我能顺利完成下一次实验。