从计算图形学试验报告——第三次实验

杨伯宇 18340189

task1 在 rasterize_triangle 函数中实现计算二维三角形的轴向包围盒

代码如下

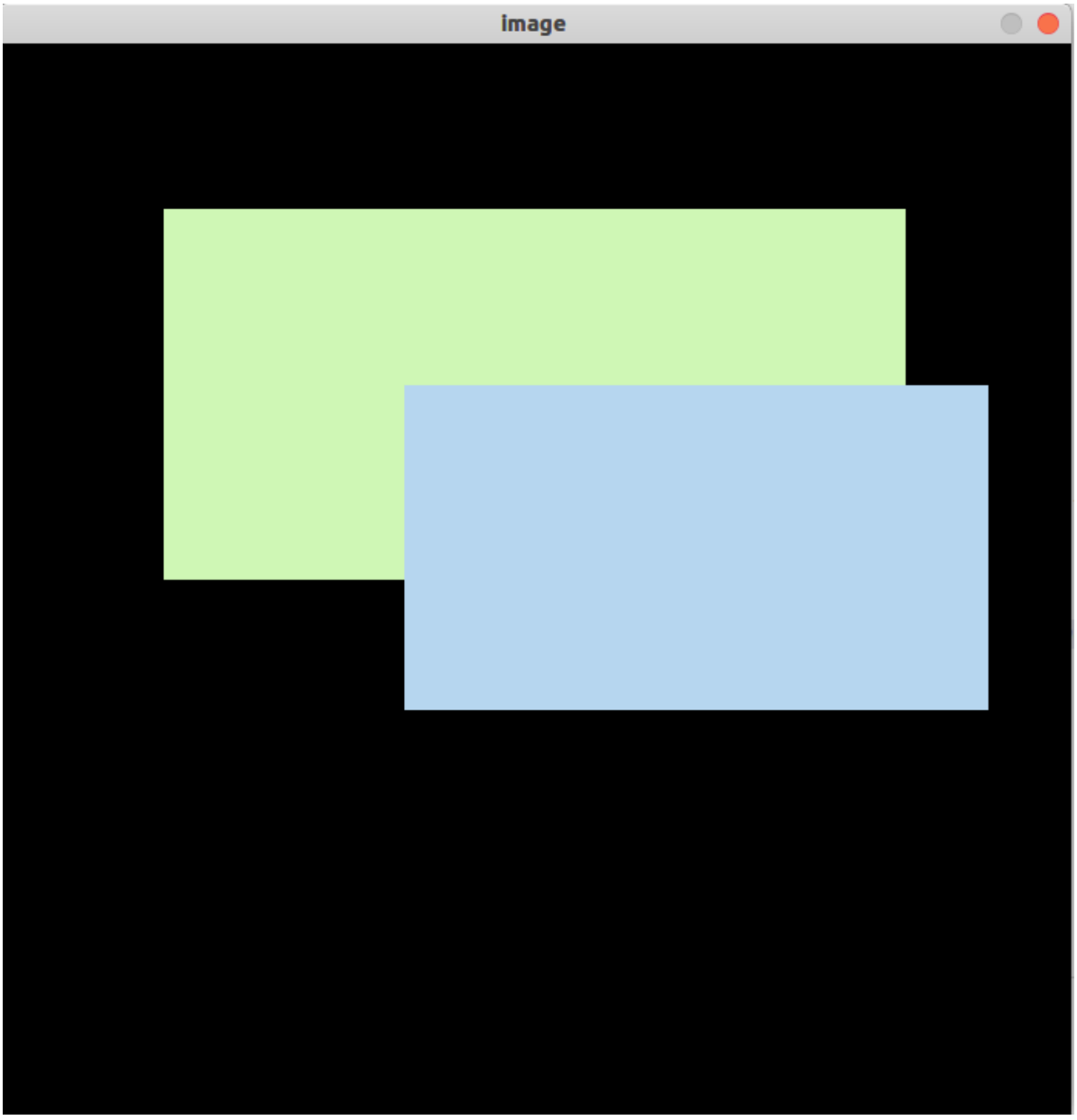
```
    // TODO 1: Find out the bounding box of current triangle.
    {

        for(int i=0;i<3;++i){
            if(t.v[i][0]<xmin){
                xmin=t.v[i][0];
            }
            if(t.v[i][0]>xmax){
                xmax=t.v[i][0];
            }
            if(t.v[i][1]<ymin){
                ymin=t.v[i][1];
            }
            if(t.v[i][1]>ymax){
                ymax=t.v[i][1];
            }
        }
    }
```

这个很简单，依次访问三角形的每个点，迭代更新最小值和最大值，结果如下。这里如果使用if-else if可能会有问题，假如写成这样

```
    // TODO 1: Find out the bounding box of current triangle.
    //error code
    {

        for(int i=0;i<3;++i){
            if(t.v[i][0]<xmin){
                xmin=t.v[i][0];
            }
            else if(t.v[i][0]>xmax){
                xmax=t.v[i][0];
            }
            if(t.v[i][1]<ymin){
                ymin=t.v[i][1];
```

```
            }
            else if(t.v[i][1]>ymax){
                ymax=t.v[i][1];
            }
        }
    }
```

因为在第一次迭代中,如果选择的恰好是x的最大值，xmax就不会被赋值，所以这样写是错的。结果如下



task2 在 insideTriangle 函数中实现判断给定的二维点是否在三角形内部。（30分）

就像老师上课讲的那样，使用外积,对于一个三角形 ABC 和一个点P，如果

$$(\vec{PA} \times \vec{PB})(\vec{PB} \times \vec{PC}) > 0$$
$$(\vec{PB} \times \vec{PC})(\vec{PC} \times \vec{PA}) > 0 \quad (1)$$

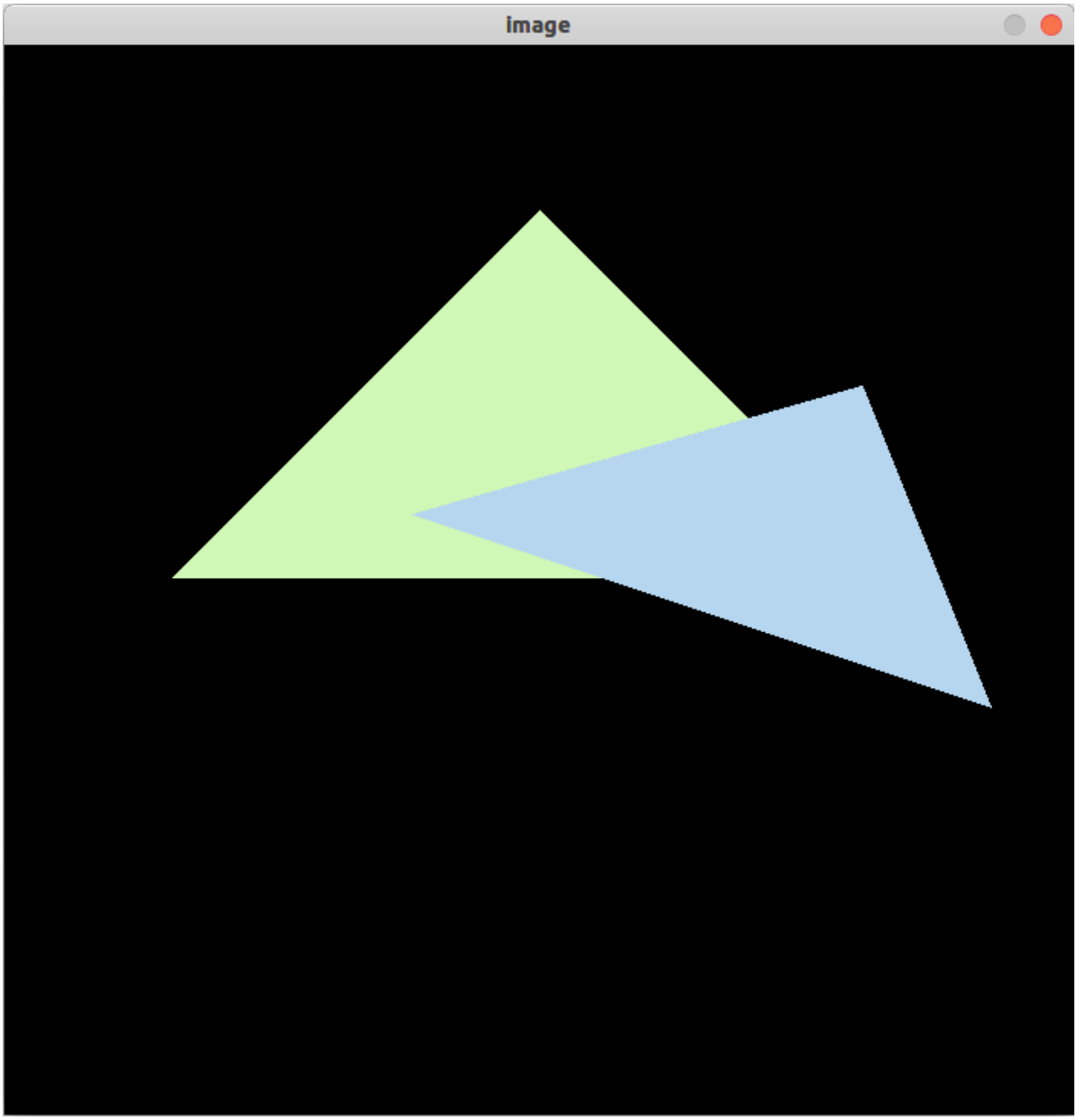
那么点P就在三角形内部，所以代码如下

```
static bool insideTriangle(int x, int 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]
    Vector3f point(x,y,_v[0](2)); //把(x,y)变成三维点
    Vector3f one=_v[0]-point;//计算PA
    Vector3f two=_v[1]-point;//计算PB
    Vector3f three=_v[2]-point;//计算PC
    return one.cross(two).transpose()*two.cross(three)>0 &&
two.cross(three).transpose()*three.cross(one)>0;
}
```

这样还可以优化，因为得到的三个外积向量的方向要么相同，要么相反，所以只用比较向量中相同位置的一个值就可以知道它们是否同向，这里使用z值。

```
static bool insideTriangle(int x, int 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]
    Vector3f point(x,y,_v[0](2)); //把(x,y)变成三维点
    Vector3f one=_v[0]-point;//计算PA
    Vector3f two=_v[1]-point;//计算PB
    Vector3f three=_v[2]-point;//计算PC
    return one.cross(two)(2)*two.cross(three)(2)>0 && two.cross(three)
(2)*three.cross(one)(2)>0;
}
```

结果如下



可以看到两个三角形

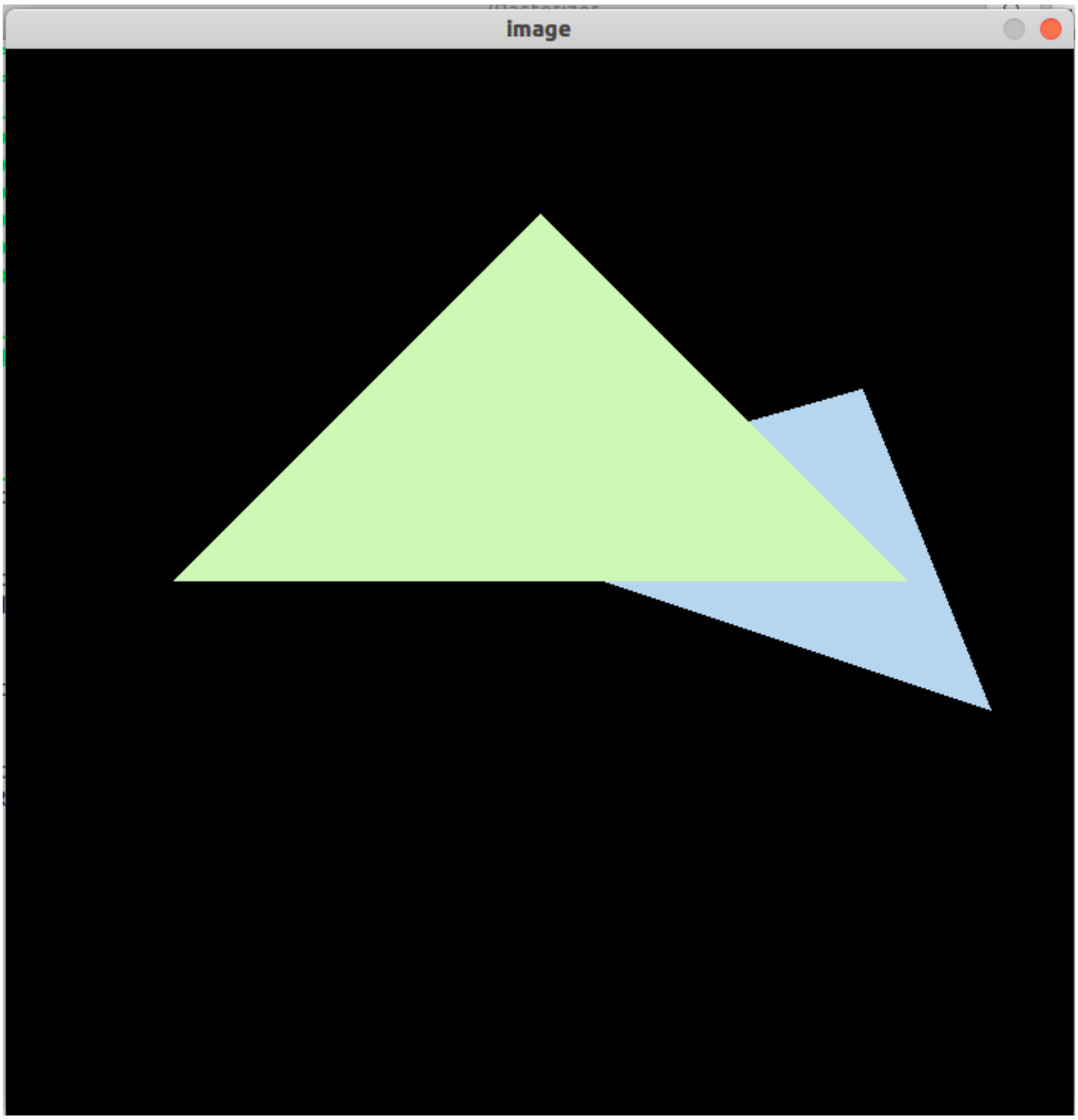
task3 在 rasterize_triangle 函数中实现Z-buffering算法

Z-buffering算法的思路其实很简单，就是对于每个屏幕上的像素，记录它在三维空间中距离平面的距离，当要处理新的像素时，比较他和原来像素的距离的，谁小就用谁。

但是这里我找了半天才找到记录距离的向量，是depth_buf，使用get_index 来获得(x,y)在该向量中对应的下标，进行记录即可

```
// TODO 3: perform Z-buffer algorithm here.
if(z_interpolated>depth_buf[get_index(x,y)]){
    continue; //如果原来的距离小，就直接舍弃，处理下一个像素
}
else{
    //否则就更新深度值，并且更新颜色值
    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];
```

结果如下



task4、回顾光栅化所处的阶段，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。

光栅化的输入是坐标和其对应的颜色，输出是屏幕中每个点的像素值，

主要负责的工作是读取顶点，画成三角形（因为每个图形都是由三角形组成的），并计算屏幕像素点的值，在三角形里面且距离屏幕最近的点被上色，其他点被设为背景色，不断循环，直到处理完所有三角形。当然这里没有考虑到光线，阴影等，通俗来讲，就是负责确定屏幕中的像素值。

task5 、仔细观察光栅化的结果，你可以看到如下的锯齿走样现象，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。

原因就是采样点不够密集，或者说屏幕的像素点不够密集，导致“直线”中有锯齿，实际上直线是有折线近似成的，如果采样点足够多，锯齿就会很小，于是很难看出来，而采样点不够多时，折线看起来就会很明显。除了直接增加像素以外，抗锯齿的方法有超采样，就是把高分别率的图形使用低分辨率显示，低分辨率的每个像素都是高分辨率中对应的几个像素的近似；还有多采样方法，这种方法的思想与超采样基本相同，但是它会忽略掉不可能产生锯齿的内部像素，故效率更高。还有计算平均颜色值的方法。

task6、（选做）实现反走样算法——超采样抗锯齿方法。

先采样一个高清的图片，在这之前，要修改其他的类函数

```
//因为三角形的坐标没有改变，但是采样点个更多了，所以传进来的点不一定是整数，将其修改为浮点数
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]
    Vector3f point(x,y,_v[0](2));
    Vector3f one=_v[0]-point;
    Vector3f two=_v[1]-point;
    Vector3f three=_v[2]-point;
    return one.cross(two)(2)*two.cross(three)(2)>0 && two.cross(three)
(2)*three.cross(one)(2)>0;
}
```

```
//需要一个更大的像素缓存frame_buf_enlarged，和深度信息缓存depth_buf_enlarged，所以要对其进行初始化
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(frame_buf_enlarged.begin(), frame_buf_enlarged.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(depth_buf_enlarged.begin(), depth_buf_enlarged.end(),
std::numeric_limits<float>::infinity());
    }
}

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

```
    frame_buf_enlarged.resize(w * h*4);
    depth_buf.resize(w * h);
    depth_buf_enlarged.resize(w * h*4);
    // std::cout<<w*h<<'\t'<<w*h*4<<std::endl;
}
```

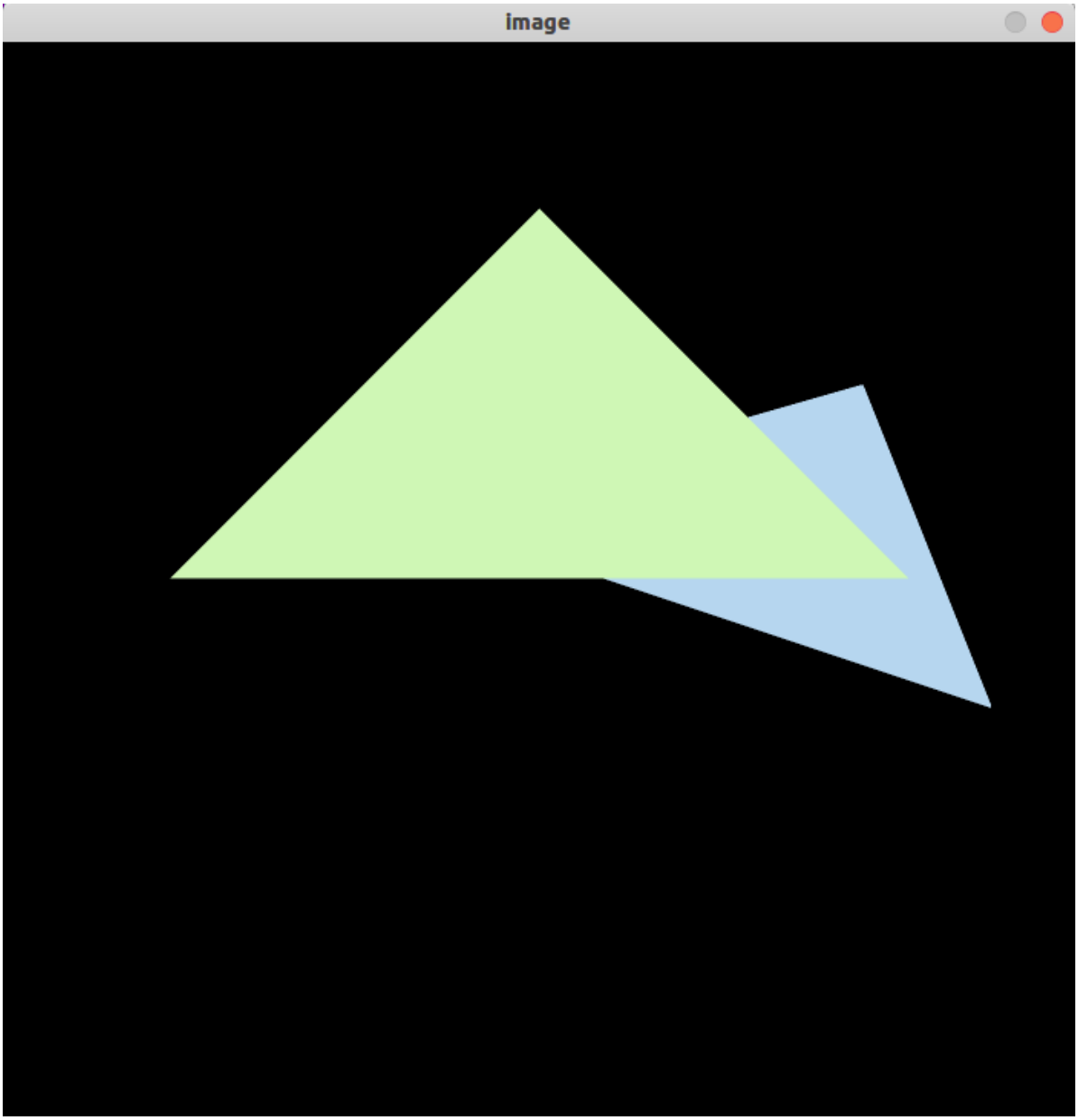
```
//对于更大的缓存，也需要一个求相应坐标的函数
int rst::rasterizer::get_index_enlarged(int x, int y)
{
    return (height*2-1-y)*width*2 + x;
}
```

接下来就是进行采样

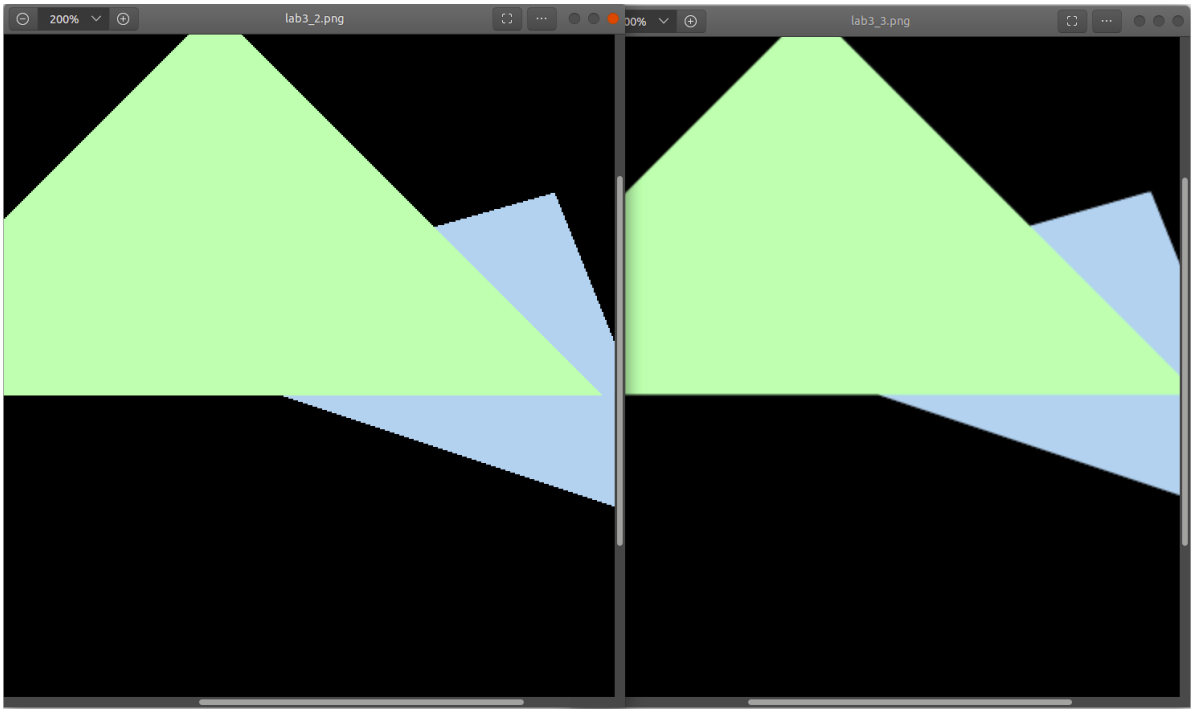
```
void rst::rasterizer::rasterize_triangle(const Triangle& t) {
    //.......
    //上面的代码和之前一样
    //这里进行更高像素的采样，这里要注意，原来的x<=xmax 等价于 x <= static_cast<int>(xmax)，但是当像素范围扩大一倍时，x<=xmax*2 不等价于 x <= static_cast<int>(xmax)*2，比如x<=1.7*2不等价于 x<=1*2，所以这里强制转化为整数来避免错误
    for(int x = static_cast<int>(xmin)*2;x <= static_cast<int>(xmax)*2;++x)
    {
        for(int y = static_cast<int>(ymin)*2;y <= static_cast<int>(ymax)*2;++y)
        {
            //这里的x,y对应到三角形中的点不一定是整数，所以要传入浮点数
            if(!insideTriangle((float)x/2, (float)y/2, t.v))
                continue;

            //.....
            //求z_interpolated的代码和之前一样
            //这里也原来一样，只不过把depth_buf换为depth_buf_enlarged，frame_buf换为frame_buf_enlarged,get_index_enlarged换为get_index_enlarged
            if(z_interpolated>depth_buf_enlarged[get_index_enlarged(x,y)]){
                continue;
            }
            else{
                depth_buf_enlarged[get_index_enlarged(x,y)]=z_interpolated;
            }

            frame_buf_enlarged[get_index_enlarged(x,y)] = 255.0f * t.color[0];
        }
    }
    //现在把超采样的结果映射到原来清晰度的像素向量中
    for(int x = static_cast<int>(xmin);x < static_cast<int>(xmax);++x)
    {
        for(int y = static_cast<int>(ymin);y < static_cast<int>(xmax);++y)
        {
            //取均值
            frame_buf[get_index(x,y)] = 1/(float)4*(frame_buf_enlarged[get_index_enlarged(2*x,2*y)]+frame_buf_enlarged[get_index_enlarged(2*x+1,2*y)]+frame_buf_enlarged[get_index_enlarged(2*x,2*y+1)]+frame_buf_enlarged[get_index_enlarged(2*x+1,2*y+1)]);

        }
    }
```

```
}
```

结果为



看着好像区被不大，当放大后



左侧是原答案，右侧是超采样后的结果，可以看到左侧的锯齿情况严重，而右侧的边缘应该有的锯齿部分被模糊化了，所以没有很强的锯齿感。