Assignment 3: Rasterization与Z-buffering

Computer Graphics Teaching Stuff, Sun Yat-Sen University

- Due Date: 10月21号晚上12点之前，提交到zhangzk3@mail2.sysu.edu.cn邮箱。

> 在上次作业中，我们了解并实现了模型变换与投影变换，最终将给定的三角形顶点变换到屏幕空间，并调用我们给出的线框光栅化代码在屏幕上绘制出了一个线框三角形。单纯的线框三角形看起来并不是那么有趣，所以这一次作业我们将继续推进一步——在屏幕上画一个实心三角形，这需要你去实现三角形的光栅化算法。

1、总览

在上次作业中，在视口变化之后，提供的代码框架调用了函数`rasterize_wireframe(const Triangle &t)`，该函数实现了线条光栅化算法，在屏幕上绘制出了三角形线条。这一次，你需要实现的是三角形填充光栅化算法`rasterize_triangle(const Triangle &t)`。你将要实现的内容如下：

- 创建二维三角形的轴向包围盒；
- 遍历此轴向包围盒中的所有像素（使用其整数下标索引），然后使用像素中心的屏幕空间坐标来判断该像素是否位于三角形内；
- 如果像素位于三角形内部，则将其位置处的**插值深度值**（interpolated depth value）与深度缓冲区（depth buffer）中的相应值进行比较，执行Z-buffering算法；
- 如果当前的像素点深度值更靠近相机，请设像素颜色并更新深度缓冲区（depth buffer）。

你需要修改的函数主要是下面两个：

- `rasterize_triangle()`：执行三角形光栅化算法
- `static bool insideTriangle()`：判断给定的点是否在三角形内。

我们输入的是三角形的三个顶点，以及这三个顶点对应属性值（例如颜色值），经过变换之后我们也只知道三角形三个顶点的深度值，对于三角形内部像素的深度值，我们需要通过**线性插值**的方法得到其深度值。因为有关这方面的内容尚未在课程中涉及，我们已经为你处理好了这一部分，插值的深度值被存储在变量`z_interpolated`中，请注意深度值得符号以及我们是如何初始化depth buffer得，为了方便同学们编写代码，我们将z值进行了反转，保证深度值都是整数，并且越大表示离视点越远。

除了实现光栅化算法和Z-buffering，我们还布置了一个**选做**的作业：用超采样的方法处理反走样现象，简单来说就是提高分辨率。这个作业难度大了一点，给的分数将作为平时作业的额外分数，同学们可做可不做。如果喜欢挑战的话，就尽情尝试吧！遇到任何困难都可以问TA。

2、代码框架

代码的框架基本上与上次的作业一样，这里不再赘述。

3、编译

如果使用自己的系统而不是之前提供的虚拟机，本次程序作业中使用到的库为`Eigen`与`OpenCV`，请确保这两个库的配置正确。如果使用我们提供的虚拟机并用`CMake`进行编译，请在终端(命令行)以此输入以下内容：

```
mkdir build  // 创建build文件夹以保留的工程文件
cd build     // 进入build文件夹
cmake ..     // 通过提供CMakeLists.txt文件中的路径作为参数来运行cmake
make         // 通过make编译代码
./Rasterizer // 运行编译生成的可执行文件
```

`cmake`之后，每次修改之后重新编译代码只需执行`make`命令即可。

4、作业描述与提交

（1）、作业提交

将PDF报告文件和代码压缩打包提交到zhangzk3@mail2.sysu.edu.cn邮箱，邮件及压缩包命名格式为：hw2_姓名_学号。

（2）、作业描述

本次作业要求同学们完成的工作如下所示：

Task 1、在`rasterize_triangle`函数中实现计算二维三角形的轴向包围盒。（10分）

经过变换之后，三角形已经投影到了屏幕上，这是一个二维的三角形，请你计算该二维三角形的轴向包围盒，保存至`xmax`、`xmin`、`ymax`和`ymin`中：

```
    float xmin = FLT_MAX, xmax = FLT_MIN;
    float ymin = FLT_MAX, ymax = FLT_MIN;
    // TODO 1: Find out the bounding box of current triangle.
    {

    }
    // After you have calculated the bounding box, please comment the following
code.
    return;
```

实现好计算包围盒的代码之后，请注意把`return;`这一行注释掉。如果你正确计算了三角形包围盒，那么编译运行应该能够得到两个填充的长方形，这两个长方形就是三角形的包围盒。说说你是怎么做的，并贴上运行的结果。

Task 2、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）

经过前面的步骤我们绘制出来两个长方形，但这并不是我们想要的结果。我们的目的是绘制给定形状的三角形，这就需要我们判断包围盒内的一点是否在三角形，如果在三角形内部则绘制，如果不在内部则不绘制，这样三角形的形状就出来了。

```
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]

    return true;
}
```

`insideTriangle`函数在下面的双重循环中被调用，这个循环不难理解，就是对包围盒内的每一个像素处理。

```
    // 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.
            ...
        }
    }
```

如果正确实现了该函数的话，编译运行结果应该就能得到两个不同颜色的三角形，说说你是怎么做的，并贴上运行结果。

Task 3、在`rasterize_triangle`函数中实现Z-buffering算法。（30分）

在前一步骤，你正确实现得到的结果应该是浅蓝色的三角形在上面。但实际上我们输入的浅蓝色的三角形距离我们更远，按照前后的遮挡关系，浅蓝色的三角形应该在下面，这需要我们实现Z-buffering算法才能得到正确的遮挡结果。在双重循环中，请实现Z-buffering的逻辑（即下面的TODO 3）：

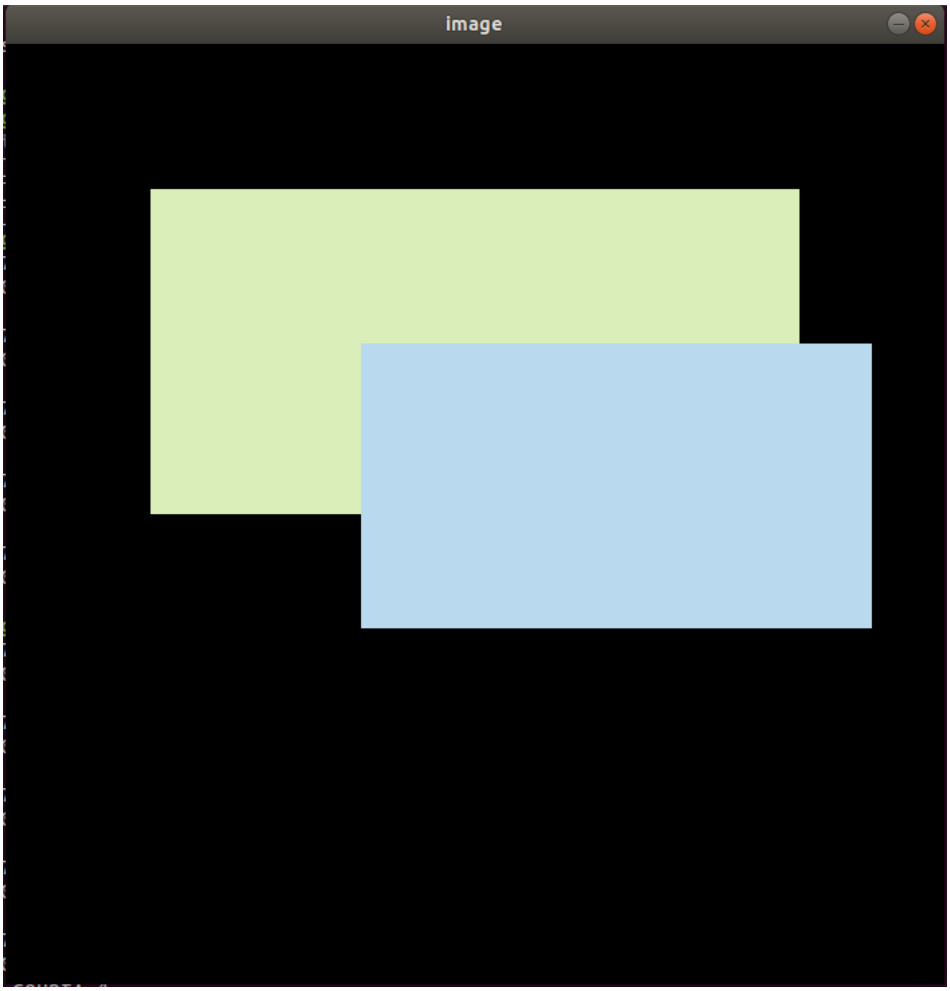
```
    // iterate through the pixel and find if the current pixel is inside the
triangle
    for(int x = static_cast<int>(xmin);x <= xmax;++x)
    {
        for(int y = static_cast<int>(ymin);y <= ymax;++y)
        {
            // if it's not in the area of current triangle, just do nothing.
            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
            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.


            // set the pixel color to frame buffer.
            frame_buf[get_index(x,y)] = 255.0f * t.color[0];
        }
    }
```

在实现Z-buffering之前，我们已经提供了计算当前像素对应的深度值代码，你只需把`z_interpolated`拿去做Z-buffering即可。请注意更新深度缓冲的值。正确实现得到的结果应该是浅蓝色的三角形在下面，说说你是怎么做的，并贴上运行结果。

Task 4、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。（15分）

Task 5、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。（15分）



Task 6、（选做）实现反走样算法——超采样抗锯齿方法。（50分）

一种减缓锯齿现象的最为暴力的方法就是提高采样率（称之为超采样方法，Super-sampling）。Super-sampling的方法思想其实非常简单，我们设置的窗口分辨率为700×700，即一共有700×700个像素，目前默认实现的是每个像素对应一个采样点。超采样方法就是一个像素对应多一个采样点，这里要求你实现一个像素对应2×2的采样点，比较前后结果的差异。每个像素使用2×2采样点进行采样，则总共有1400×1400个采样点，Super-sampling的过程你可以理解为先用1400×1400分辨率的缓冲去执行光栅化过程（即上采样），然后再缩小到700×700分辨并显示出来（即下采样）。在最终的下采样过程，你只需实现均值滤波即可，即将4个采样点的颜色值取平均作为最终的像素值。思路提示就到这里，具体实现由你自己灵活安排。

（注：选做的分数与必做的分数计算是独立的，不必担忧，必做分数满分已经满100了，选做分数只是额外提高你的平时作业分数）