作业 1：Rasterization

1 总览

到目前为止，我们已经学习了光栅化的基本知识。在本次作业中，你需要模拟一个基于 CPU 的光栅化渲染器的简化版本，并在屏幕上画出两个实心三角形，换言之，栅格化两个三角形。两个三角形有相互遮挡关系，你需要正确实现 z-buffer 算法，将三角形按顺序画在屏幕上。最后，用 super-sampling 处理 Anti-aliasing，减轻三角形的锯齿感。

2 代码框架

在本次作业中，你需要修改 **rasterizer.cpp** 中的的函数：

- rasterize_triangle(): 输入三角形顶点，执行三角形栅格化算法。
- static bool insideTriangle(): 输入坐标(x, y)与三角形顶点，测试坐标是否在三角形内。你可以修改此函数的定义，这意味着，你可以按照自己的方式更新返回类型或函数参数。

光栅化器类中，你可能会用到的成员变量与函数如下。

- vector<Vector3f> frame_buf: 帧缓冲对象，用于存储需要在屏幕上绘制的颜色数据。
- vector<float> depth_buf: 深度缓冲对象，用于存储点的深度值。
- set_pixel(Vector2f point, Vector3f color): 将屏幕像素点 (x, y) 设为 (r, g, b) 的颜色，并写入相应的帧缓冲区位置。
- get_index(int x, int y): 输入坐标，输出缓冲索引。Eg. depth_buf[get_index(x, y)]
- computeBarycentric2D(float x, float y, const Vector3f* v): 重心坐标插值

三角形类中，你可能会用到的成员变量与函数如下。

- toVector4(): 获得三角形三个点坐标。Eg.auto v = t.toVector4(); v[0].x;v[0].y...
- getColor(): 获得三角形颜色。

具体的，**rasterize_triangle** 函数的内部工作流程如下：

- 创建三角形的 2 维 bounding box。
- 遍历此 bounding box 内的所有像素（使用其整数索引）。然后，使用像素中心的屏幕

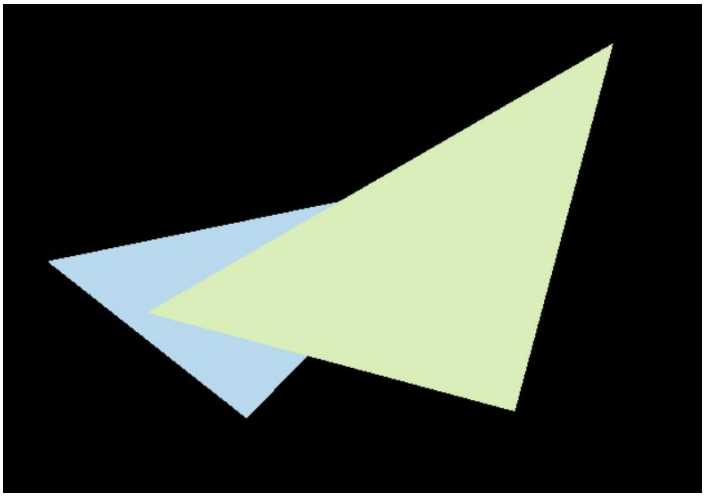
空间坐标来检查中心点是否在三角形内。

- 如果在内部，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。
- 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（depth buffer)。

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，我们需要用**插值**的方法得到其深度值。我们已经为你处理好了这一部分，因为有关这方面的内容尚未在课程中涉及。插值的深度值被储存在变量 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。

在此次作业中，我们提供了两个 hard-coded 三角形来测试你的实现，如果程序实现正确，你将看到如下所示的输出图像：



你可能会注意到，当我们放大图像时，图像边缘会有锯齿感。我们可以用 super-sampling 来解决这个问题，即对每个像素进行 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)。需要注意的点有，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 sample list。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。

有锯齿

抗锯齿

3 评分与提交

评分：

- 正确地提交所有必须的文件，且代码能够编译运行。
- 正确测试点是否在三角形内。
- 正确实现三角形栅格化算法。
- 正确实现 z-buffer 算法，将三角形按顺序画在屏幕上。
- 用 super-sampling 处理 Anti-aliasing。

提交：

当你完成作业后，请清理你的项目，在你的文件夹中需要包含所有的程序文件（无论是否修改）。同时，请提交实验结果的图片与添加一个 README 文件写下完成情况，并简要描述你在各个函数中实现的功能。最后，将上述内容打包，并用“学号_姓名_Homework1.zip”的命名方式提交。