中国海洋大学
OCEAN UNIVERSITY OF CHINA

海纳百川 取则行远

计算机图形学

Lab1-3 实验报告

实验题目:	基于 CPU 的光栅化渲染器
姓　　名:	翟一航
年　　级:	2023 级
专　　业:	软件工程
实验时间:	2025 年 11 月 30 日

目录

1 实验 1：旋转与投影

1.1 实验要求

本次实验的任务是填写一个旋转矩阵和一个透视投影矩阵。给定三维下三个点 $v0(2.0, 0.0, -2.0)$, $v1(0.0, 2.0, -2.0)$, $v2(-2.0, 0.0, -2.0)$, 你需要将这三个点的坐标变换为屏幕坐标并在屏幕上绘制出对应的线框三角形。

提高项： 在 main.cpp 中构造一个函数，该函数的作用是得到绕任意过原点的轴的旋转变换矩阵。

1.2 遇到的问题与解决方法

问题一： 在进行透视投影时，发现生成的三角形图像倒置。

解决方案： 在检查后发现原来是透视投影的 top 参数未加负号。无负号时是 Y 轴向上为正的坐标系，有负号时是 Y 轴向下为正的坐标系。想要让三角形反转，需要加上负号。

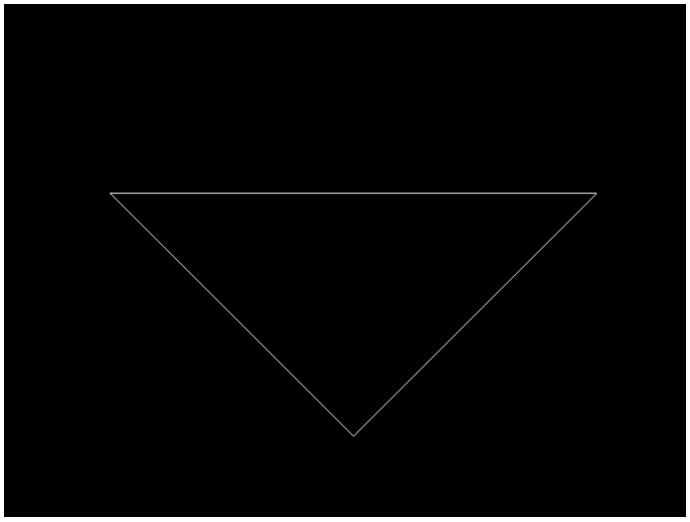

图 1: 倒置的三角形

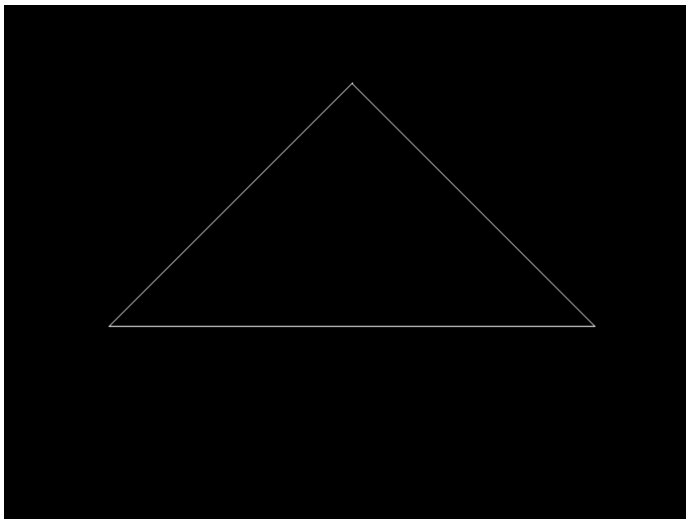

图 2: 加上负号后正置的三角形

问题二： 在编写提高项代码时，犹豫使用什么方法来编写旋转矩阵。

解决方案： 结合第一次作业，决定使用罗德里格斯旋转公式，直接对目标点进行坐标变换。

$$\mathbf{R}(n,\theta) = \begin{bmatrix} \cos\theta + n_x^2(1-\cos\theta) & n_x n_y(1-\cos\theta) - n_z\sin\theta & n_x n_z(1-\cos\theta) + n_y\sin\theta \\ n_y n_x(1-\cos\theta) + n_z\sin\theta & \cos\theta + n_y^2(1-\cos\theta) & n_y n_z(1-\cos\theta) - n_x\sin\theta \\ n_z n_x(1-\cos\theta) - n_y\sin\theta & n_z n_y(1-\cos\theta) + n_x\sin\theta & \cos\theta + n_z^2(1-\cos\theta) \end{bmatrix}$$

2 实验 2: Triangles and Z-buffering

2.1 实验要求

在屏幕上画出一个实心三角形，换言之，栅格化一个三角形。上一次实验中，在视口变化之后，我们调用了函数 rasterize_wireframe(const Triangle t)。但这一次，你需要自己填写并调用函数 rasterize_triangle(const Triangle t)。

提高项: 用 super-sampling 来解决走样问题。

2.2 遇到的问题与解决方法

问题一: 在使用超级采样法来解决走样问题后，发现前后差别不大，图像边缘均呈现锯齿状，但是反复检查超级采样的编写发现没有问题。

解决方案: 经检查后发现，原来是因为在 insideTriangle 函数中，对于位于边界上的点未能进行明确良好的处理，在返回值时设置了严格的大于、小于条件，从而忽略了恰好在三角形边界上的点，导致生成的三角形边缘出现明显的锯齿化。在修改判定条件后，发现问题解决。

问题二: 在进行提高项的编写时，最终输出的三角形出现了颜色混合的现象，三角形的边缘出现暗色晕圈，部分像素的颜色变淡。

解决方案: 这是因为在使用超级采样时，边缘的像素点不一定能够覆盖所有的采样点，但是在计算最终的颜色时，却无差别除以 4，这就会导致部分边缘像素点的颜色被暗化，应该修改 final_color 的计算方法，最后应该除以被覆盖的采样点点数目。

问题三: 生成的三角形在重叠部分表现错误，应该绿色在下蓝色在上。

解决方案: 但是实际情况是，绿色的深度小于蓝色的深度，应该绿色在上蓝色在下，这是因为在 rasterize_triangle 函数中进行深度测试时，误将条件写为:

z_interpolated < sample_depth_buf[sample_index]

应该修改为:

z_interpolated > sample_depth_buf[sample_index]

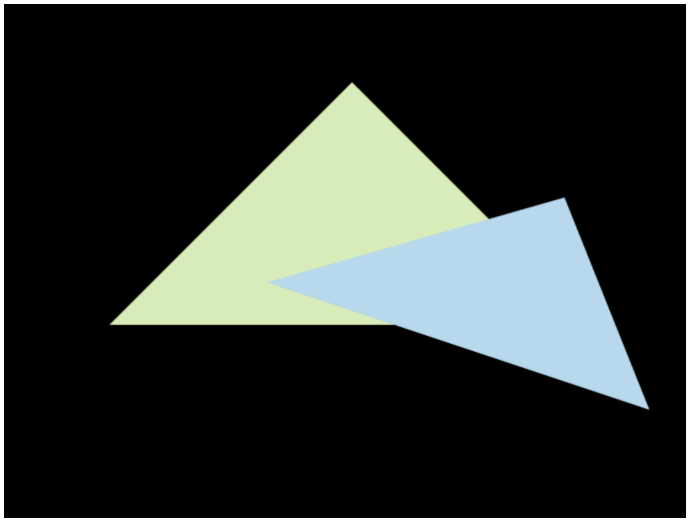

图 3: 深度测试错误图片

3 实验 3：Pipeline and Shading

3.1 实验要求

1. 修改函数 rasteriz_triangle(const Triangle t)in rasterizer.cpp: 在此处实现与实验 2 类似的插值算法，实现法向量、颜色、纹理颜色的插值。

2. 修改函数 get_projection_matrix() in main.cpp: 将你自己在之前的实验中实现的投影矩阵填到此处，此时你可以运行./Rasterizer output.png normal 来观察法向量实现结果。

3. 修改函数 phong_fragment_shader() in main.cpp: 实现 Blinn-Phong 模型计算 Fragment Color.

4. 修改函数 texture_fragment_shader() in main.cpp: 在实现 Blinn-Phong 的基础上，将纹理颜色视为公式中的 kd，实现 Texture Shading Fragment Shader.

5. 修改函数 bump_fragment_shader() in main.cpp: 在实现 Blinn-Phong 的基础上，仔细阅读该函数中的注释，实现 Bump mapping.

6. 修改函数 displacement_fragment_shader() in main.cpp: 在实现 Bump mapping 的基础上，实现 displacement mapping.

   提高项：使用双线性插值进行纹理采样, 在 Texture 类中实现一个新方法 Vector3f getColorBilinear(float u, float v) 并通过 fragment shader 调用它。

3.2 遇到的问题与解决方法

问题一： 最终的图像本来应该是被光线照到的地方明亮，没有被光线照到的地方暗，但是在生成的图像中却恰好相反。

解决方案：这是因为在计算光线方向时，应该使用光源的位置减去点位置（light.position - point），但是实际却写反了，从而导致了一系列的连锁反应，使得半程向量、漫反射分量、高光分量的计算也发生了错误，应该将源代码修改为：

Eigen::Vector3f light_dir = (light.position - point).normalized();

图 4: Blinn-Phong 光照光线方向计算错误

问题二： 使用 displacement 生成的图像中光照完全失效。

解决方案：这是由于对法线与切向量进行计算时没有进行归一化，而是使用了原始法线，并且还存在和上述问题一中相同的问题，导致光照效果完全丧失，应该将源代码修改为：

```
Eigen::Vector3f normal = payload.normal.normalized();
Eigen::Vector3f perturbed_normal = (TBN * ln).normalized();
t « x*y/sqrt(x*x+z*z), sqrt(x*x+z*z), z*y/sqrt(x*x+z*z);
t.normalize();
b = normal.cross(t);
b.normalize();
```

图 5: 错误图像

问题三： 进行双线性插值后发现图像质量并没有发生改变。

解决方案： 因为只在类定义中加入了双线性插值，但是却没有在 main.cpp 中进行应用，需要修改原来的对象引用。

4 心得体会

通过完成这三个计算机图形学实验，我深刻体会到了从基础变换到完整渲染的逐步构建过程：实验一让我亲手实现了模型变换和投影矩阵，理解了顶点从三维空间到屏幕坐标的完整变换流程；实验二通过三角形栅格化和深度缓冲实践了基本的光栅化原理，认识到插值和深度测试对渲染效果的关键作用；实验三则通过实现着色器管线、纹理映射和凹凸渲染技术，真正体会到现代图形学中光照模型与纹理采样如何协同工作以生成逼真视觉效果。这三个实验环环相扣，巩固了我对图形学基础理论的理解，更让我通过代码实践对于相关知识的掌握有了质的飞跃。