计算机学院实验报告

实验题目： GAMES101 Assignment 3		学号：202000130143
日期：12.19	班级：计科 20.1	姓名：郑凯饶
Email：1076802156@qq.com		

实验目的：

在这次编程任务中，我们会进一步模拟现代图形技术。我们在代码中添加了 Object Loader(用于加载三维模型), Vertex Shader 与 Fragment Shader，并且支持了纹理映射。

而在本次实验中，你需要完成的任务是:

1. 修改函数 `rasterize_triangle(const Triangle& t)` in rasterizer.cpp: 在此处实现与作业 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.

实验环境介绍：

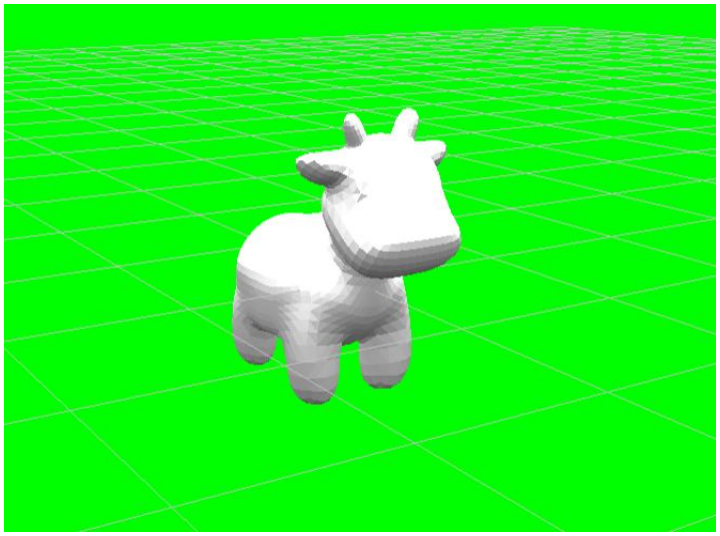
Dell Latitude 5411
Intel(R) Core(TM) i5-10400H CPU @ 2.60GHz(8GPUs),~2.6GHz
Windows 10 家庭中文版 64 位（10.0，版本 18363）
Visual Studio 2022

解决问题的主要思路：

这次实验不同之前，是在提供的代码框架上实现几种 shader，因此首先至少基本了解该框架：

1. 框架输入

```
bool loadout =
Loader.LoadFile("../models/spot/spot_triangulated_good.obj");
```



win 系统中的 vsc 配置了解析.obj 文件的插件，实际上该文件通过顶点描述了一个三角形平面集合（v 是顶点坐标，vt 是纹理坐标，vn 是顶点对应的法向量，f 是三角形对应的 3 个顶点的（v，vn，vt））。

OBJ_loader.h 进行模型加载：

1. 我们引入了一个第三方.obj 文件加载库来读取更加复杂的模型文件，这部分库文件在 OBJ_Loader.h file. 你无需详细理解它的工作原理，只需知道这个库将会传递给我们一个被命名被 TriangleList 的 Vector，其中每个三角形都有对应的点法向量与纹理坐标。此外，与模型相关的纹理也将被一同加载。**注意：如果你想尝试加载其他模型，你目前只能手动修改模型路径。**

2. Pipeline

4. 主渲染流水线开始于 rasterizer::draw(std::vector<Triangle> &TriangleList). 我们再次进行一系列变换，这些变换一般由 Vertex Shader 完成。在此之后，我们调用函数 rasterize_triangle.

初始化光栅化渲染器，设置纹理、shader、mvp 变换矩阵，进入 draw 函数进行渲染：

```
    Eigen::Matrix4f mvp = projection * view * model;    // mvp 变换
    for (const auto& t:TriangleList)    // 逐个进行光栅化
    {
        Triangle newtri = *t;
```

① 对每个三角形进行光栅化；

② 对视点进行 mv 变换，p（投影）变换不影响视点，mark 一下 std::transform 函数的用法；
③ 对三角形顶点进行 mvp 变换，齐次除法后 w=1，（x, y, z）表示真实坐标；
计算 mvp 变换后的法线 normal，推导如下：

这一步是计算进行了 `model, view` 变换之后的每个点上的法线向量，这一步对于后续正确计算三种光线折射十分重要，因为法线在变换的过程中可能无法点的切线向量（如果存在切线数据的话）保持垂直。我们假设某一点的法线向量为 n，切线向量为 t，简化 `model, view` 变换矩阵为 M。在原有的模型中

$$n^T t = 0$$

经过变换之后，切线向量变成了

$$Mt$$

按照法线的定义，变换之后的法线向量 n'有

$$n'^T Mt = 0$$

由于 $n^T t = n^T M^{-1} Mt = 0$，所以

$$n'^T = n^T M^{-1} = ((M^{-1})^T n)^T \Rightarrow n' = (M^{-1})^T n$$

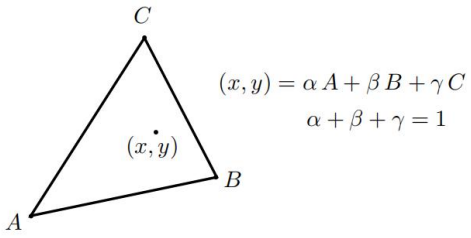
所以经过了 `model, view` 变换的点的法线向量并不是 $\text{View} \cdot \text{Model} \cdot n$，而是

$$((\text{View} \cdot \text{Model})^{-1})^T \cdot n$$

④ 视窗口变换；
⑤ 设置变换顶点、法线后的三角形 newtri，调用 `rasterize_triangle(newtri, viewspace_pos)`进行渲染。

rasterize_triangle：
① 构建矩阵包围盒，方便表达；
② Z-buffer，对包围盒内每个像素点作判断，是否在三角形片元内部，这样等效于对片元进行处理；
③ 计算重心坐标 Barycentric，用作插值系数；

Barycentric Coordinates: Formulas



$$(x, y) = \alpha A + \beta B + \gamma C$$
$$\alpha + \beta + \gamma = 1$$

$$\alpha = \frac{-(x - x_B)(y_C - y_B) + (y - y_B)(x_C - x_B)}{-(x_A - x_B)(y_C - y_B) + (y_A - y_B)(x_C - x_B)}$$
$$\beta = \frac{-(x - x_C)(y_A - y_C) + (y - y_C)(x_A - x_C)}{-(x_B - x_C)(y_A - y_C) + (y_B - y_C)(x_A - x_C)}$$
$$\gamma = 1 - \alpha - \beta$$

④ 对颜色、法线、纹理坐标、视点进行插值；

⑤ 用 shader 计算该像素的颜色。

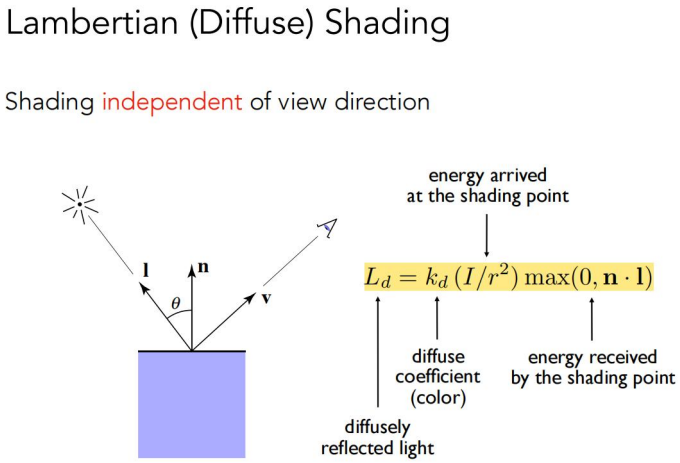
3. Shader

① phong_fragment_shader

这个书和 ppt 都讲得比较清楚了。

Diffuse light，有一个漫反射系数，也可以理解为颜色，模型中主要由 diffuse light 表征颜色，而颜色实际为没有吸收的光线，对光线的吸收自然是由漫反射系数 kd 决定的。后面 texture 方法映射到纹理上不同位置，本质上是映射到不同的系数 kd，注意 kd 是一个 3d 向量，表征对 rgd 三个通道的反射能力。

然后光强会随着距离以平方关系衰减，以及和法线和入射光线夹角余弦存在比例关系。这些都是模型的内容，至于模型为什么这样近似描述整个照明系统，有更多物理原理的考究，我也没有深入了解。



② texture_fragment_shader

在 Phong shader 的基础上增加纹理映射，注意添加 uv 范围约束[0, 1]。

③ bump_fragment_shader

根据代码提示完成。

实验步骤：

1. 阅读代码框架，深入理解重要部分原理；

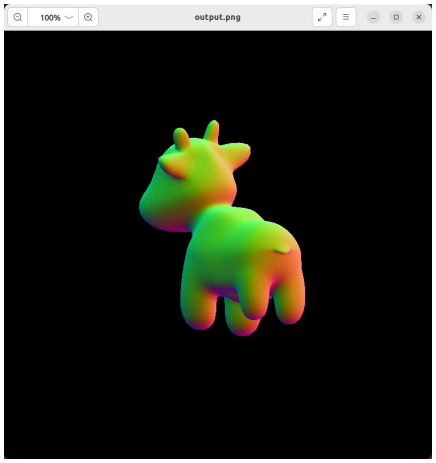
2. 根据提示完成各个 shader；

实验结果展示及分析：




实验中存在的问题及解决：

get_projection_matrix 存在问题，一开始的效果是下面这样的：



Angle 应该都是缺省的 140°，但和其他人的不同，仔细看了投影变换后，**修改 squish 矩阵以及边界 top 的定义、计算**：

透视变换分为两步：①squish(将梯台压缩称为长方体)，②正交投影

Perspective Projection

- How to do perspective projection
  - First "squish" the frustum into a cuboid (n -> n, f -> f) ($M_{persp->ortho}$)
  - Do orthographic projection (M_{ortho}, already known!)

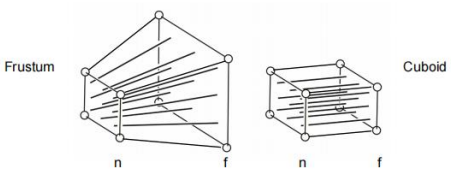
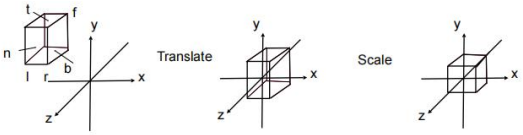


Fig. 7.13 from *Fundamentals of Computer Graphics, 4th Edition*

Orthographic Projection

- Transformation matrix?
  - Translate (**center** to origin) first, then scale (length/width/height to **2**)

$$M_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



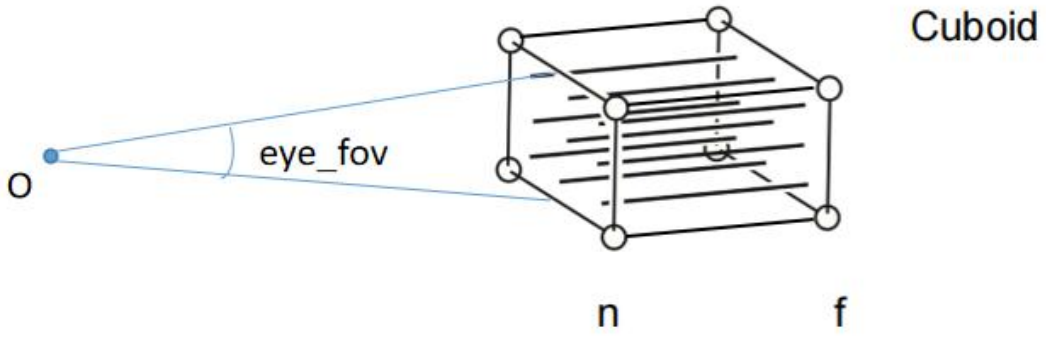
前者对应变换矩阵为

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

矩阵定义很明确，比较麻烦的是 n, f 这些量在 code 中要从一些物理量中求得。

```
Eigen::Matrix4f get_projection_matrix(float eye_fov, float aspect_ratio,
float zNear, float zFar)
{   // eye_fov: 垂直可视角
```

参数	含义
eye_fov	垂直可视角
aspect_ratio	宽高比
zNear	距离 Camera 较近的边界
zFar	距离 Camera 较远的边界



其中，eye_fov 是垂直可视角，也叫视场角（field of view），它取一半，用弧度表示得到 halfEyeRadian，其正切值乘 zNear 得到 top.