Assignment 2: 变换与投影

Computer Graphics Teaching Stuff, Sun Yat-Sen University

- Due Date: 10月11号晚上12点之前，提交到zhangzk3@mail2.sysu.edu.cn邮箱。

> 到目前为止，我们已经学习了如何使用矩阵变换来排列二维或三维空间中的物体。所以现在是时候通过实现一些简单的变换矩阵来获得一些实践经验了。在接下来的三次作业中，我们将要求你去模拟一个基于CPU的光栅化渲染器的简化版本。大部分的代码框架都已经提供了，你的任务就是填补其中的一些空白，代码量不大，请放心食用。

1、总览

为了本次作业的任务是填写一个模型变换矩阵和一个透视投影矩阵。**给定三维空间中的三个点** $v_0(2.0, 0.0, -2.0)$、$v_1(0.0, 2.0, -2.0)$**和** $v_2(-2.0, 0.0, -2.0)$**，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（在代码框架中，我们已经提供了** `draw_triangle` **函数，所以你只需去构建变换矩阵即可）**。简而言之，回顾渲染管线的流程，我们需要进行模型变换、视图变换、投影变换，模型变换将物体从局部空间变换到世界空间，视图变换将物体从世界空间变换到摄像机空间，投影变换将物体从三维空间投影到二维平面上，然后光栅化绘制。在提供的代码框架中，我们留下了模型变换、投影变换的部分给你去完成。

如果你对上述有任何不清楚或疑问，请复习课堂笔记或参考 Learn OpenGL Coordinate-Systems

以下是你需要在 `main.cpp` 中修改的函数（请不要修改任何的函数名和其他已经填写好的函数，并保证提交的代码是已经完成并且能运行的），具体的作业要求在后面：

- `get_projection_matrix`：输入参数分别为视域角度 `eye_fov`（角度制）、屏幕宽高比 `aspect_ratio`、近平面z值 `zNear` 和远平面z值 `zFar`，该函数需要你根据这些参数构建透视投影矩阵并返回该投影矩阵；
- `get_model_matrix`：输入参数为旋转角度 `rotation_angle`（角度制），在此函数中，你需要实现三维中绕z轴旋转的变换矩阵，而不用处理平移和缩放；

当你在上述函数中正确地构建了模型与投影矩阵，光栅化器会创建一个窗口显示出一个线框三角形。由于光栅化器是逐帧渲染与绘制的，所以你可以使用**A**和**D**键去将该三角形绕z轴旋转。当你按下**Esc**键时，窗口会关闭且程序终止。

2、代码框架

在本次作业中，因为你并不需要去修改和使用 `Triangle` 类，所以你需要理解与修改的文件为：`rasterizer.hpp` 和 `main.cpp`，其中 `rasterizer.hpp` 作用是生成光栅化渲染器并绘制，而 `main.cpp` 负责启动和执行渲染循环。

光栅化器类 `rasterizer.hpp` 在该程序系统中起着重要的作用，其成员变量与函数如下。

成员变量：

- `Matrix4f model, view, projection`：三个变换矩阵，分别是模型矩阵、视图矩阵和投影矩阵；
- `vector frame_buf`：帧缓冲对象，用于存储需要在屏幕上绘制的颜色数据（就是一个像素矩阵）；

成员函数：

- `set_model(const Eigen::Matrix4f& m)`：将**模型矩阵**作为参数 `m` 传递给光栅化器；
- `set_view(const Eigen::Matrix4f& v)`：将**视图矩阵**作为参数 `v` 传递给光栅化器；
- `set_projection(const Eigen::Matrix4f& p)`：将**投影矩阵**作为参数 `p` 传递给光栅化器；
- `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。

在 `main.cpp` 中，我们模拟了OpenGL图形管线。我们首先定义了光栅化器类的实例，然后设置了其必要的变量。然后我们得到一个带有三个顶点的硬编码三角形（请不要修它）。在主函数上，我们定义了三个分别计算模型、视图和投影矩阵的函数，每一个函数都会返回相应的矩阵。接着，这三个函数的返回值会被 `set_model()`，`set_view()` 和 `set_projection()` 三个函数传入光栅化器中。最后，光栅化器在屏幕上显示出变换的结果。

在用模型、视图、投影矩阵对给定几何体进行变换后，我们得到三个顶点的正则化空间坐标(canonical space coordinate)。正则化空间坐标是由三个取值范围在之间的 `x` 、 `y` 、 `z` 坐标构成。我们下一步需要做的就是视口变换，将坐标映射到我们的屏幕中 `(window_width * window_height)`，**这些在光栅化器中都已经完成，所以不需要担心。**但是，你需要去理解这步操作是如何运作的，目前不是很理解也没关系，后面可以再回来看。

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
```

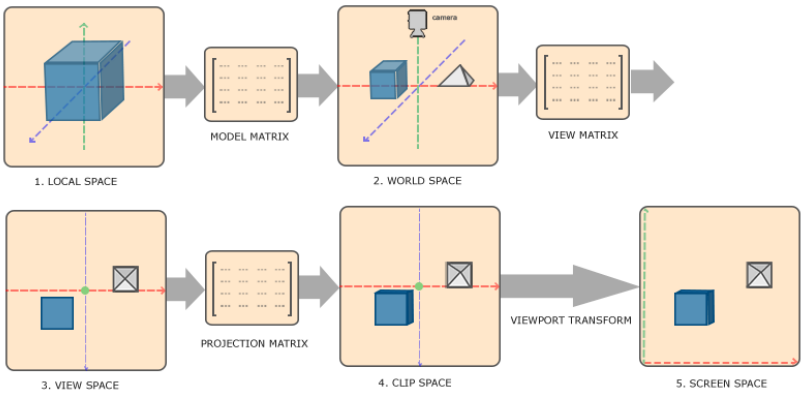
cmake 之后，每次修改编译代码只需执行 make 命令即可。

4、从3D到2D

同学们可能刚上完课对整个变换管线的过程有些模糊，这里再梳理一下。变换管线从物体的局部空间开始，最后变换到屏幕空间。整个变换过程主要涉及到5个不同的坐标系统：

- 局部空间(Local Space，或者称为物体空间(Object Space))
- 世界空间(World Space)
- 观察空间(View Space，或者称为视觉空间(Eye Space))
- 裁剪空间(Clip Space)
- 屏幕空间(Screen Space)

你现在可能会对什么是坐标空间，什么是坐标系统感到非常困惑，所以我们将用一种更加通俗的方式来解释它们。为了将坐标从一个坐标系变换到另一个坐标系，我们需要用到几个变换矩阵，最重要的几个分别是**模型**(Model)、**观察**(View)、**投影****(**Projection)三个矩阵。我们的顶点坐标起始于局部空间(Local Space)，在这里它称为局部坐标(Local Coordinate)，它在之后会变为世界坐标(World Coordinate)，观察坐标(View Coordinate)，裁剪坐标(Clip Coordinate)，并最后以屏幕坐标(Screen Coordinate)的形式结束。下面的这张图展示了整个流程以及各个变换过程做了什么：



1. 局部坐标是对象相对于局部原点的坐标，也是物体起始的坐标。
2. 下一步是将局部坐标变换为世界空间坐标，世界空间坐标是处于一个更大的空间范围的。这些坐标相对于世界的全局原点，它们会和其它物体一起相对于世界的原点进行摆放。
3. 接下来我们将世界坐标变换为观察空间坐标，使得每个坐标都是从摄像机或者说观察者的角度进行观察的。
4. 坐标到达观察空间之后，我们需要将其投影到裁剪坐标。裁剪坐标会被处理至 `-1.0` 到 `1.0` 的范围内，并判断哪些顶点将会出现在屏幕上。
5. 最后，我们将裁剪坐标变换为屏幕坐标，我们将使用一个叫做视口变换(Viewport Transform)的过程。视口变换将位于 -1.0 到 1.0 范围的坐标变换到屏幕空间，在OpenGL中由 glViewport 函数 实现。最后变换出来的坐标将会送到光栅器，将其转化为片段。

我们之所以将顶点变换到各个不同的空间的原因是有些操作在特定的坐标系统中才有意义且更方便。例如，当需要对物体进行修改的时候，在局部空间中来操作会更说得通；如果要对一个物体做出一个相对于其它物体位置的操作时，在世界坐标系中来做这个才更说得通，等等。如果我们愿意，我们也可以定义一个直接从局部空间变换到裁剪空间的变换矩阵，但那样会失去很多灵活性。

接下来我们将要更仔细地讨论各个坐标系统。

（1）、局部空间

局部空间是指物体所在的坐标空间，即对象最开始所在的地方。想象你在一个建模软件（比如说Blender）中创建了一个立方体。你创建的立方体的原点有可能位于 `(0, 0, 0)` ，即便它有可能最后在程序中处于完全不同的位置。甚至有可能你创建的所有模型都以 `(0, 0, 0)` 为初始位置，然而它们会最终出现在世界的不同位置。所以，你的模型的所有顶点都是在**局部**空间中：它们相对于你的物体来说都是局部的。

（2）、世界空间

如果我们将我们所有的物体导入到程序当中，它们有可能会全挤在世界的原点 `(0, 0, 0)` 上，这并不是我们想要的结果。我们想为每一个物体定义一个位置，从而能在更大的世界当中放置它们。世界空间中的坐标正如其名：是指顶点相对于（游戏）世界的坐标。如果你希望将物体分散在世界上摆放（特别是非常真实的那样），这就是你希望物体变换到的空间。物体的坐标将会从局部变换到世界空间；该变换是由模型矩阵(Model Matrix)实现的。

模型矩阵是一种变换矩阵，它能通过对物体进行位移、缩放、旋转来将它置于它本应该在的位置或朝向。你可以将它想像为变换一个房子，你需要先将它缩小（它在局部空间中太大了），并将其位移至郊区的一个小镇，然后在 y 轴上往左旋转一点以搭配附近的房子。

（3）、观察空间

观察空间经常被人们称之为摄像机(Camera)（所以有时也称为摄像机空间(Camera Space)或视觉空间(Eye Space)）。观察空间是将世界空间坐标转化为用户视野前方的坐标而产生的结果。因此观察空间就是从摄像机的视角所观察到的空间。而这通常是由一系列的位移和旋转的组合来完成，平移/旋转场景从而使得特定的对象被变换到摄像机的前方。这些组合在一起的变换通常存储在一个**观察矩阵**

(View Matrix)里，它被用来将世界坐标变换到观察空间。

（4）、裁剪空间

经过上述的变换之后，我们希望所有的坐标都能落在一个特定的范围内，且任何在这个范围之外的点都应该被裁剪掉(Clipped)。被裁剪掉的坐标就会被忽略，所以剩下的坐标就将变为屏幕上可见的片段。这也就是裁剪空间(Clip Space)名字的由来。

因为将所有可见的坐标都指定在 -1.0 到 1.0 的范围内不是很直观，所以我们会指定自己的坐标集(Coordinate Set)并将它变换回标准化设备坐标系。为了将顶点坐标从观察变换到裁剪空间，我们需要定义一个**投影矩阵(Projection Matrix)**，它指定了一个范围的坐标，比如在每个维度上的 `-1000` 到 `1000` 。投影矩阵接着会将在这个指定的范围内的坐标变换为标准化设备坐标的范围 `(-1.0, 1.0)` 。所有在范围外的坐标不会被映射到在 `-1.0` 到 `1.0` 的范围之间，所以会被裁剪掉。

（5）、屏幕空间

x和y变换到$[-1,1]$ 之后，紧接着需要变换到屏幕空间，因为紧接着的图元光栅化阶段是在屏幕上进行的。设屏幕的宽为w，高为h （屏幕分辨率为$w \times h$），则屏幕上的顶点的x取范围为$[0, w-1]$，y的取值范围为$[0, h-1]$。所以需要将$[-1,1]$的x和y映射到屏幕范围。

关于变换管线的过程就梳理到这， 有兴趣的同学可以看看OpenGL的变换介绍，链接在这里。

5、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，邮件和文件名格式为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：

Job1、编译我们提供的代码并运行，将运行的结果截图。（5分）

（注：由于一开始并没有构建投影矩阵，没有实现从3D到2D的投影，所以你在屏幕上看到的并不是一个三角形）

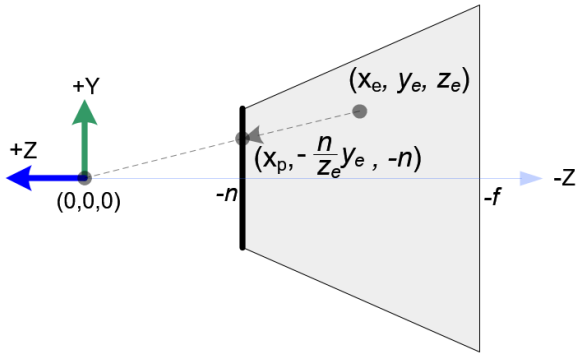
Job2、构建透视投影矩阵，将编译运行结果截图，并简述一下矩阵是如何构建。（50分）

如下所示， `get_projection_matrix` 函数在 `main.cpp` 文件中， `projection` 初始化为单位矩阵，你需要填充这个矩阵内容：

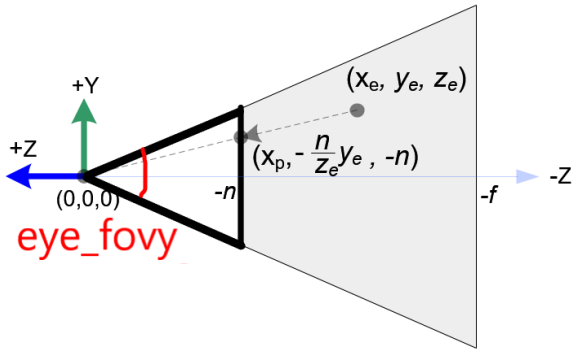
```
Eigen::Matrix4f get_projection_matrix(float eye_fov, float aspect_ratio,

float zNear, float zFar)

{
    // Students will implement this function

    Eigen::Matrix4f projection = Eigen::Matrix4f::Identity();

    // TODO: Implement this function
```

```
    // Create the projection matrix for the given parameters.

    // Then return it.

    return projection;

}
```

注意，透视投影矩阵用于观察空间到裁剪空间的转换，其不包含透视除法。这里解释一下各个参数的含义。要投影的三维点在摄像机坐标中，回顾一下，在摄像机坐标下，视角正前方朝向z轴的**负朝向**，所以视锥体的远平面和近平面的z值是负值，如下图所示，记近平面和远平面的z值分别为$-n$和$-f$，传入的参数`zNear`和`zFar`就是n和f。



投影的目的就是将摄像机空间的三维点投影到近平面上。通过相似三角形的原理我们得到了投影的x'和y'，紧接着需要将x'和y'分别映射到$[-1,1]$，这个就是正交投影的过程。传入的`aspect_ratio`参数是屏幕的宽度和高度的比值，即w/h，而`eye_fov`是视域范围角度，如下图红色部分所示。根据三角函数的关系，我们可以得出屏幕的高度的一半为$zNear * tan(eye_fovy/2)$，然后根据屏幕的宽高比`aspect_ratio`我们就可以得到屏幕宽度的一半。有了屏幕的宽度和高度，就可以将x'和y'映射到$[-1,1]$了。



此部分可能略难，这里提供一篇文章供大家参考去实现。**注意stl中的三角函数例如`cos`、`sin`之类的函数要求输入参数为弧度制，但我们提供的eye_fov是角度制，因此需要进行转换。**

Job3、构建旋转变换矩阵，截图三张旋转结果，并简述一下矩阵是如何构建的。（30分）

这里要求你构建一个绕z轴旋转的矩阵的，`model`初始化为单位矩阵，然后你需要填充该矩阵从而实现绕z轴旋转的矩阵。**注意，`rotation_angle`需要转换成弧度制。**

```
Eigen::Matrix4f get_model_matrix(float rotation_angle)

{

    Eigen::Matrix4f model = Eigen::Matrix4f::Identity();

    // TODO: Implement this function

    // Create the model matrix for rotating the triangle around the Z axis.

    // Then return it.

    return model;

}
```

编译运行之后，你可以通过按A键和D键控制旋转。

Job4、谈谈你对四维齐次坐标的理解。（15分）