Homework 5 - Camera

Basic:

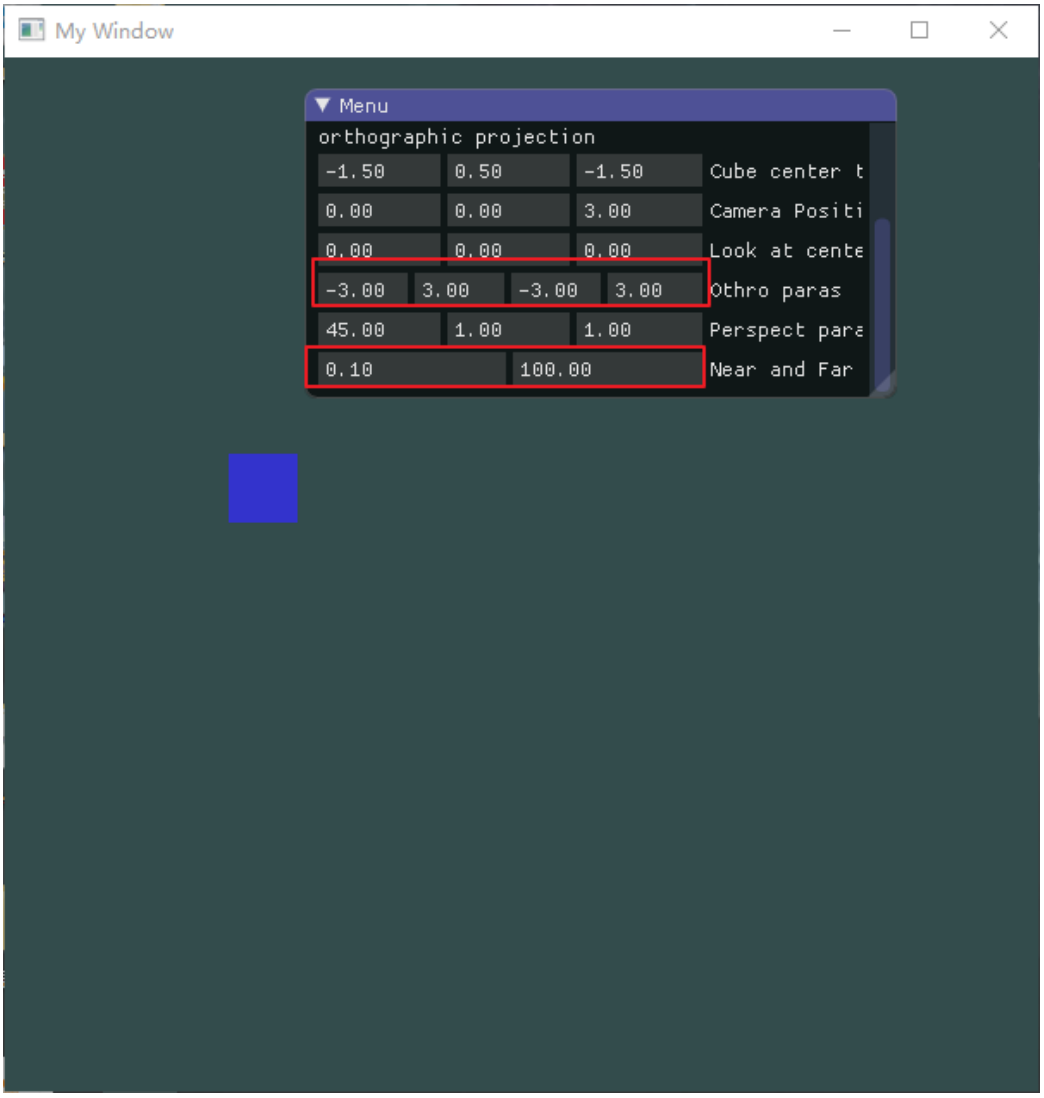
1. 正交投影

实验截图

参数1

left	right	bottom	top	near	far

-3	3	-3	3	0.1	100

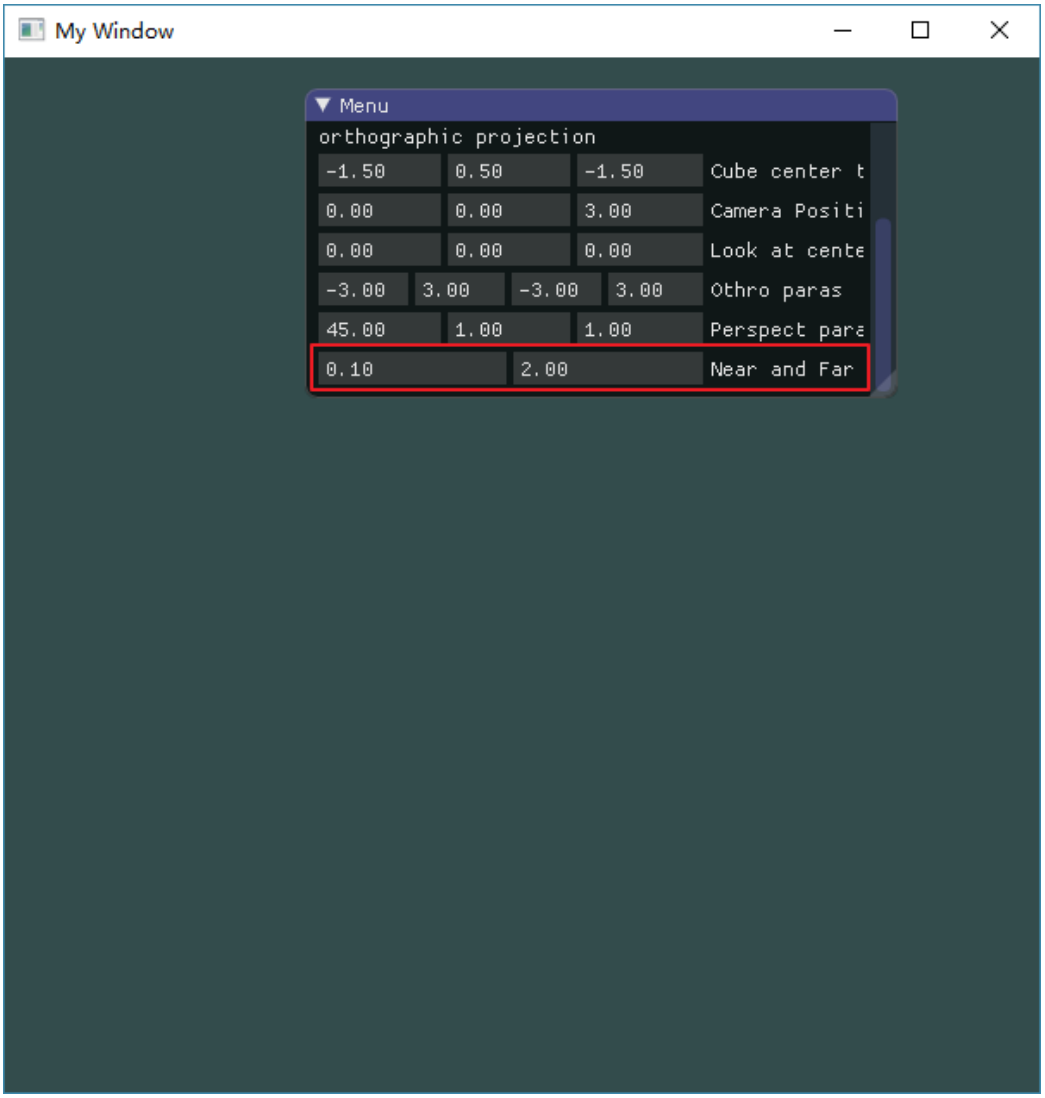


参数2

left	right	bottom	top	near	far
-5	3	-3	8	0.1	100

参数3

left	right	bottom	top	near	far
-3	3	-3	3	0.1	2



算法实现

使用 `glm::ortho` 函数创建投影立方体。

GUI 里面设置接口改变变量，然后将相应的变量传进 `model`，`view`，`projection` 的矩阵里面即可。没有太多的实现难点。

比较差异

根据这次作业我明白了正交投影函数在 OpenGL 中的映射关系，这是很重要的。

将 [left, right] 直接正则化到 NDC 中的 x 轴的 [-1, 1] 上，其他轴同理对应。那么这个时候， 顶点信息中顶点的位置信息就不再是只能限制在 [-1, 1] 上了，相反，它可以取任意值，但是只有当其处于创建的投影立方体内部，才能够显示出来。这个时候我们需要注意，**创建投影立方体所用的参数值的单位和顶点位置信息中所用的单位是相同的。**也就是说，如果我们想要沿用前一次作业的 Cube 的顶点位置信息的话（在 [-1, 1] 内），我们创建的投影立方体就不能太大，否则会看不见。

当我们修改参数，left, right, bottom, top 的时候，可以发现:

1. 如果投影立方体的 height 和 width 调到不一样的话，Cube 的长宽压缩的程度会不一样，在视图中会变成长方形。
2. 随着 height 和 width 的增加，视野中的立方体会越来越小。

调整 near, far 的时候可以发现:

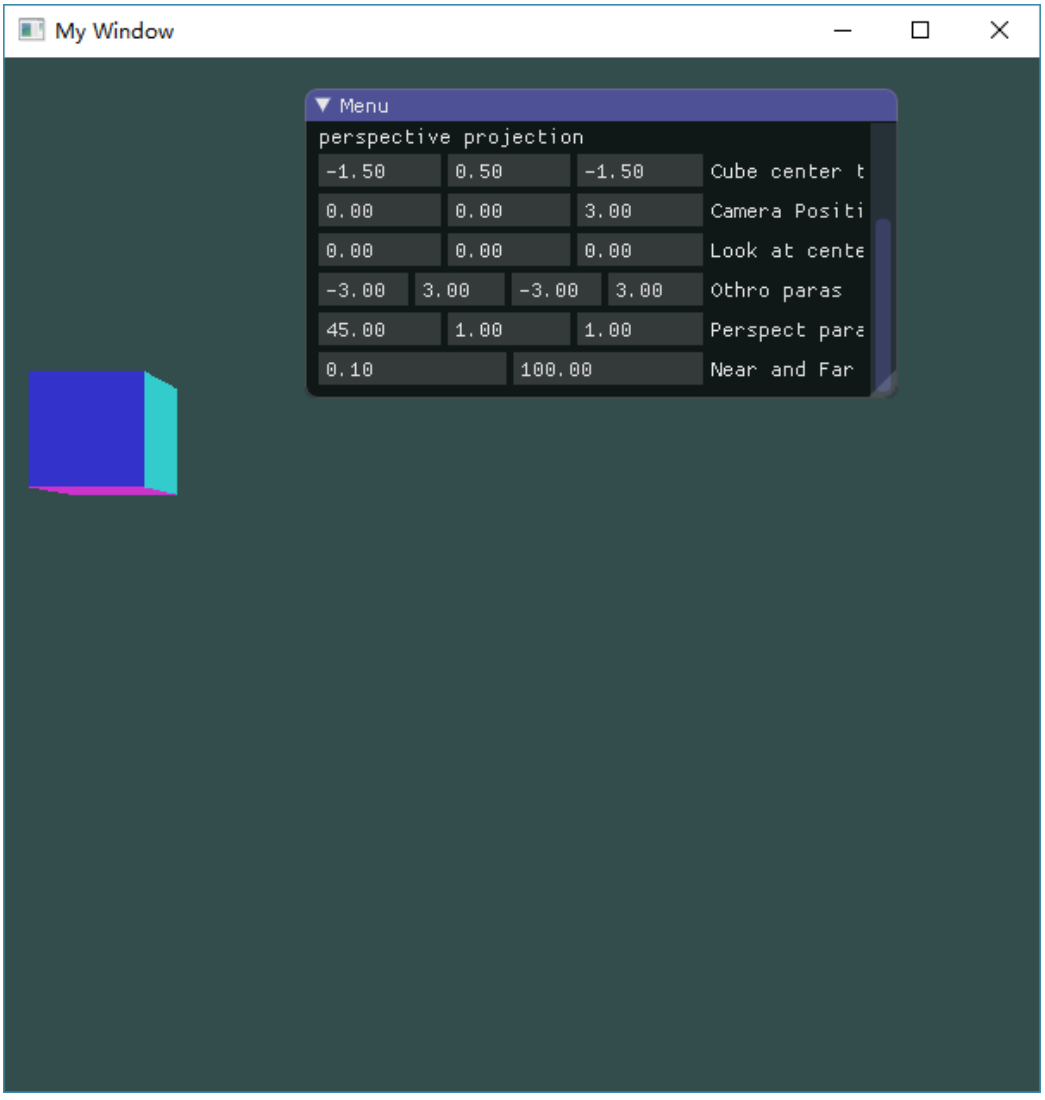
1. 当 far 太近或者 near 太远，使得投影立方体没能包裹住 Cube 的话，在视图中就看不见 Cube。

2. 透视投影

实验截图

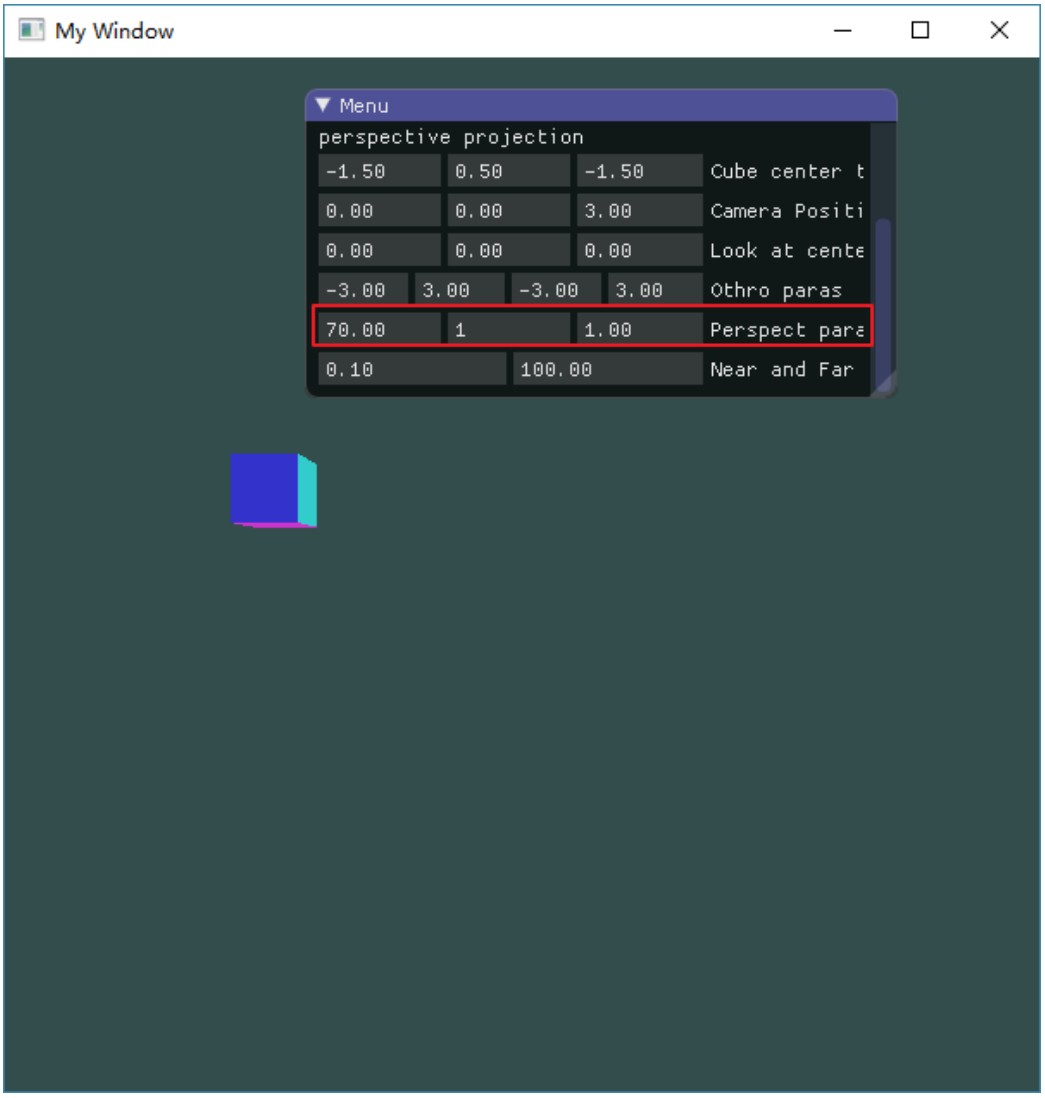
参数1

angle	width	height	near	far
45	1	1	0.1	100



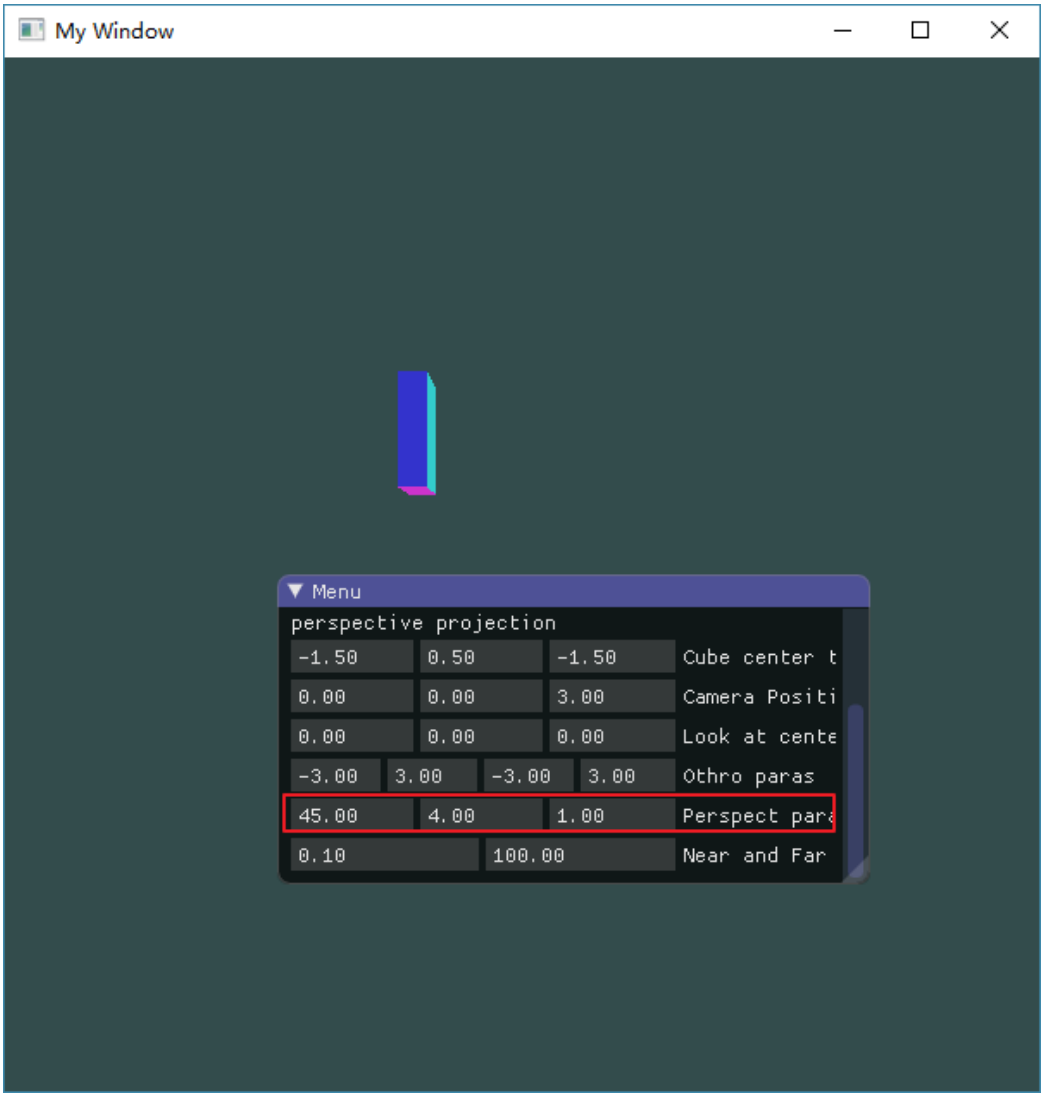
参数2

angle	width	height	near	far
70	1	1	0.1	100



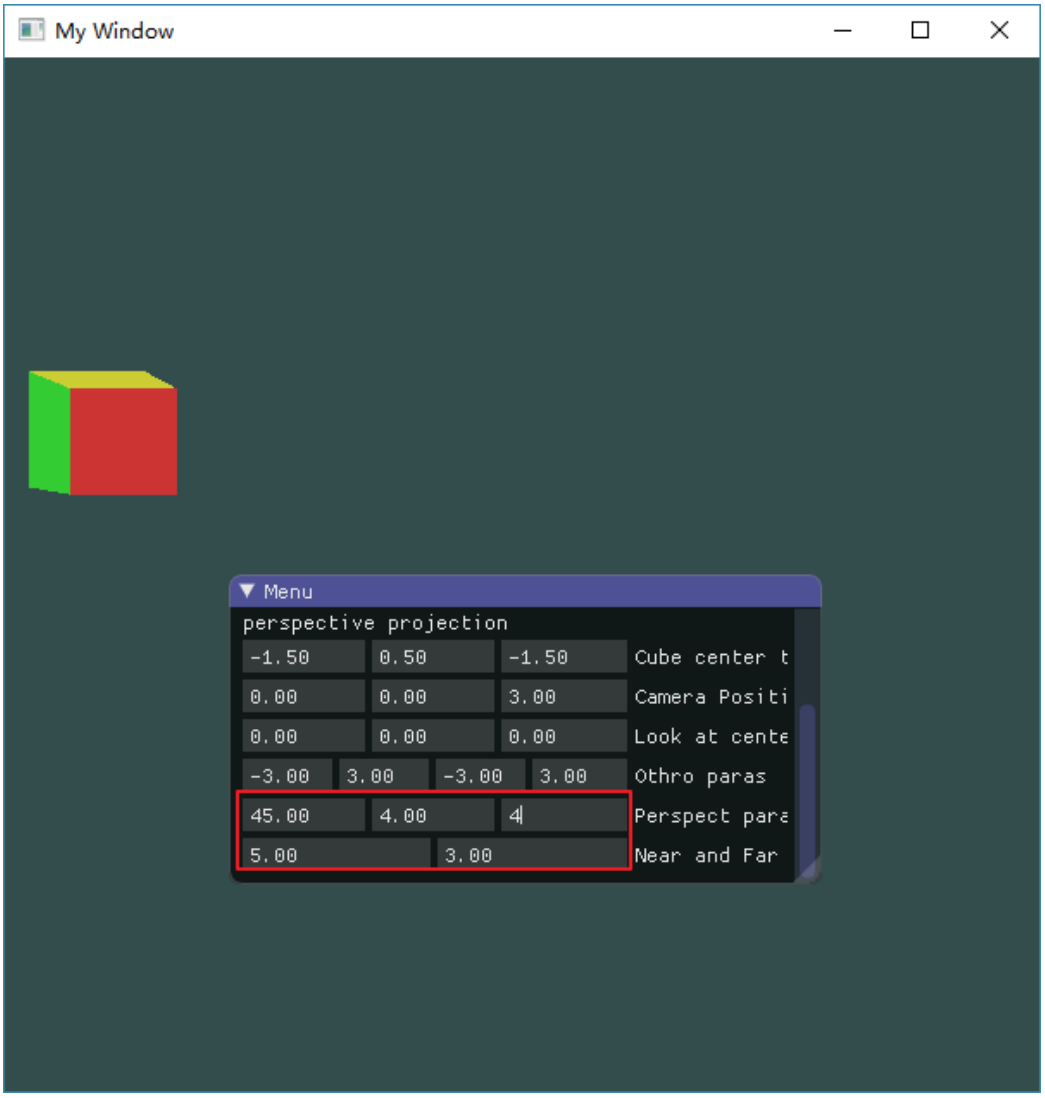
参数3

angle	width	height	near	far
70	4	1	0.1	100



参数4

angle	width	height	near	far
45	4	4	5	3



算法实现

使用 `glm::perspective` 函数创建透视投影平截体。

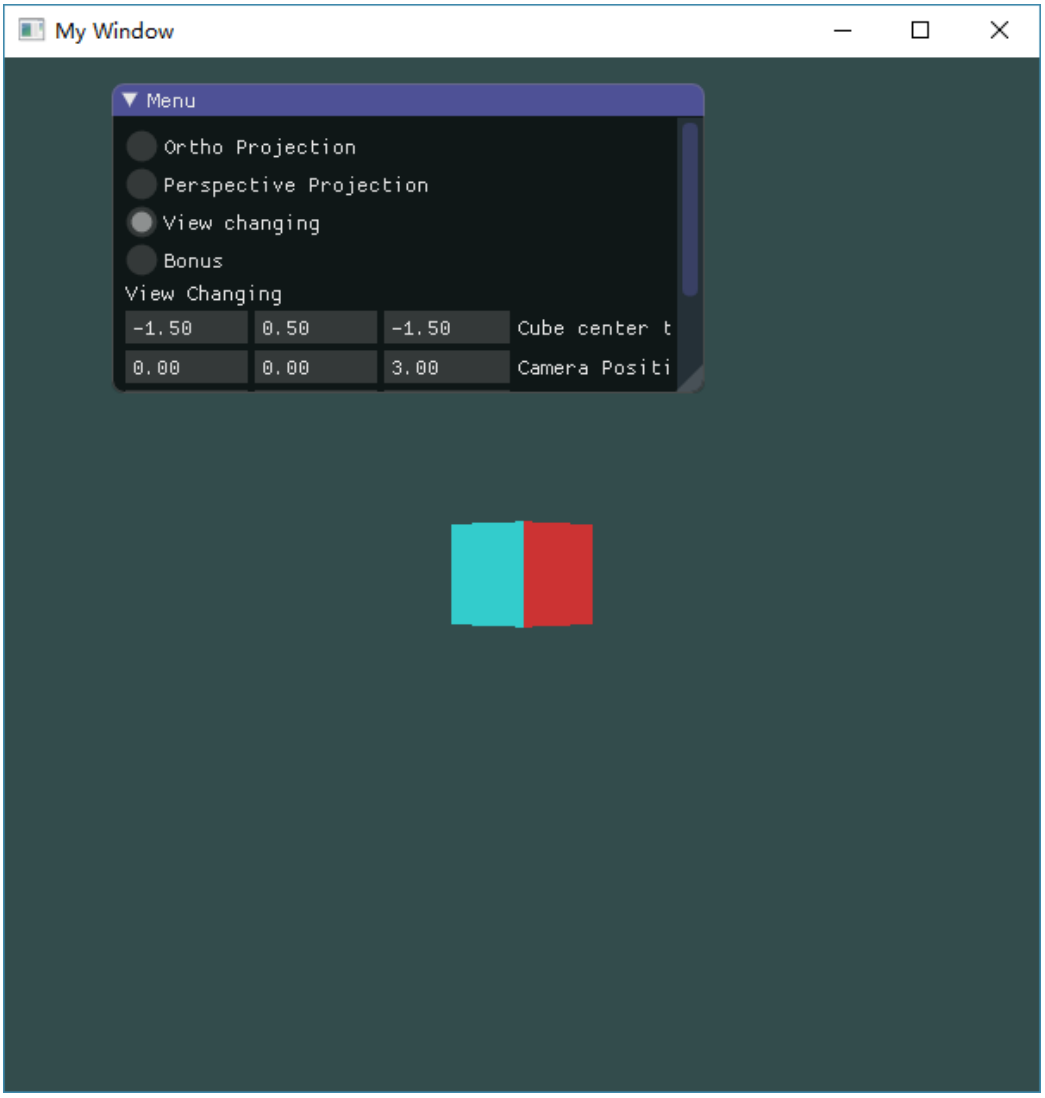
GUI 里面设置接口改变变量，然后将相应的变量传进 `model`，`view`，`projection` 的矩阵里面即可。没有太多的实现难点

参数比较

1. angle 增大，视图能看到的范围越大，Cube 在视野中越小
2. width 和 height 设置得不一样的话，Cube 的长宽压缩比例不等，会显示出一个长方体，同时位置也会因为映射范围的改变而改变
3. near 和 far 同正交投影，特别的，当 near > far 时，视图会变成反过来看。

3. 视角变换

实验截图



算法实现

通过时间函数修改 camera 的位置。
然后在每一帧的渲染中改变 camera 的 view 矩阵，即可实现。

```
float Radius = 5.0f;
float camPosX = sin(glfwGetTime()) * Radius;
float camPosZ = cos(glfwGetTime()) * Radius;

...

view = glm::lookAt(
    glm::vec3(camPosX, 0.0f, camPosZ),
    glm::vec3(0.0f, 0.0f, 0.0f),
    glm::vec3(0.0f, 1.0f, 0.0f)
);
```

4. 为什么 OpenGL 将 View 和 Model 两个矩阵合二为一?

对于每一个顶点，先做 model 变换后再作 view 变换，进行了两次的矩阵运算。那么 n 个点需要的运算是 2n 。 如果先将 View 和 model 合并后再进行变换，则需要的矩阵运算次数仅为 n + 1, 当 n 很大（一

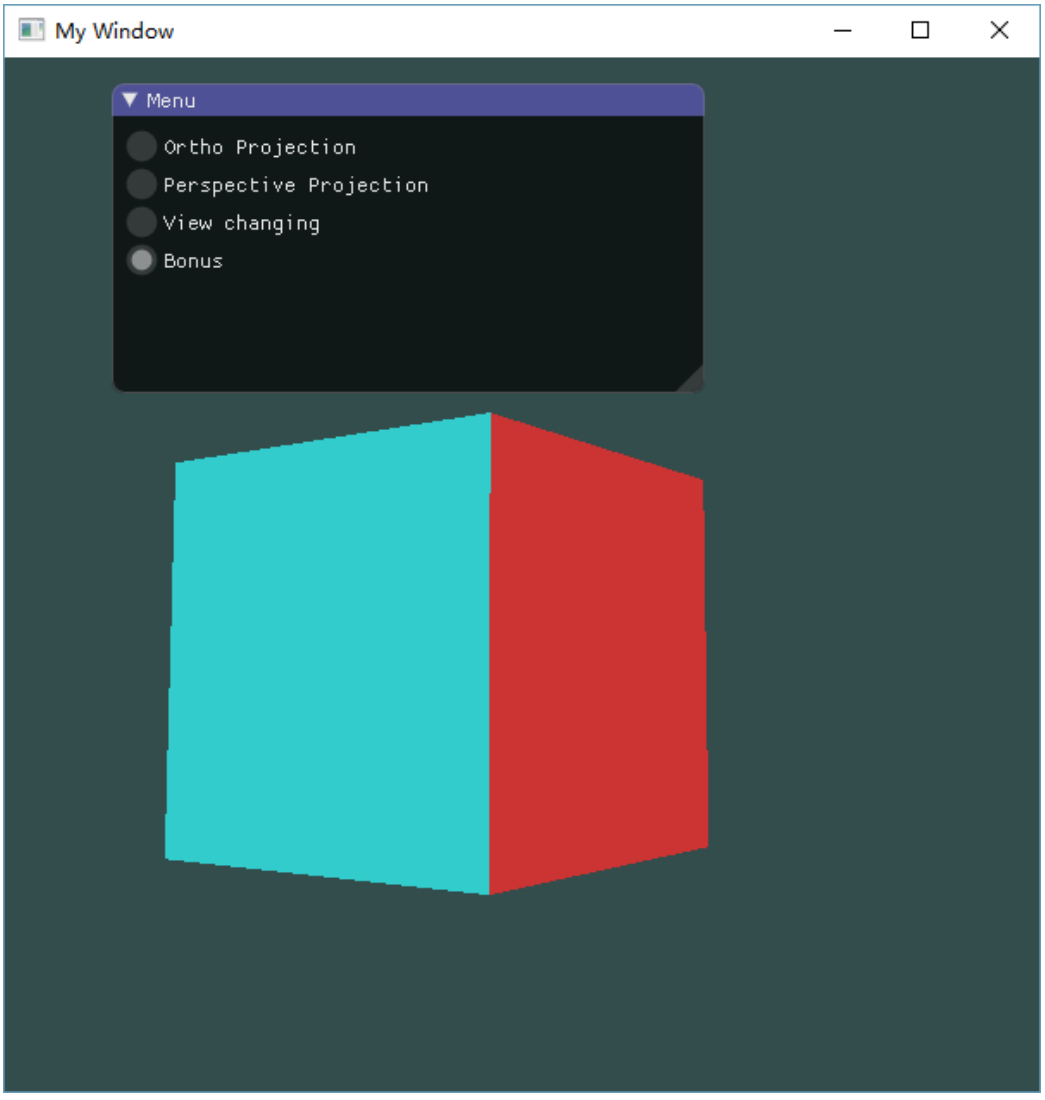
般情况下也很大）的时候，运算开销的减少是非常明显的。

至于为什么不把 projection 也包括进去，那是因为， 在 view 和 projection 之间，有时候需要插入 lighting 的操作。具体可以参考这这个 stackoverflow 问题.

Bonus

实现一个 Camera 类， 有 FPS 的效果

实验截图



算法实现

1. Camera 类中包含必要的摄像机成员变量。 通过各种成员函数改变相应的成员变量。 每次调用 `GetViewMatrix` 函数的时候就生成一个当前摄像机位置信息的 view 矩阵返回出来。
2. 通过回调函数来监听键盘输入事件。每次监听到事件之后就改变摄像机的位置。

```
1. if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
2.     camera.moveForward(deltaTime);
3. if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
```

```
    camera.moveBack(deltaTime);
if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    camera.moveLeft(deltaTime);
if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    camera.moveRight(deltaTime);
```

3. 通过回调函数来监听鼠标输入事件。每次监听到事件之后就改变摄像机的位置。通过改变 `xoffset` 和 `yoffset` 来改变俯仰角(Pitch)和偏航角(Yaw)

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to top

    lastX = xpos;
    lastY = ypos;

    camera.ProcessMouseMovement(xoffset, yoffset);
}
```

4. 通过一个全局变量 `deltaTime` 来计算两帧之间的时间，用来平衡在不同机器上的移动速率。