Homework 5

这次的作业是在对物体做空间变换的基础上加入摄像机视角的变换，使观察的视角更灵活，最终实现一个camera类来管理摄像机的变换操作。

投影

首先将上次作业绘制的cube放置在（-1.5，0.5，-1.5）位置，这里没有对顶点数组中的位置数据进行改变，而是使用一个translate平移变换矩阵来完成：

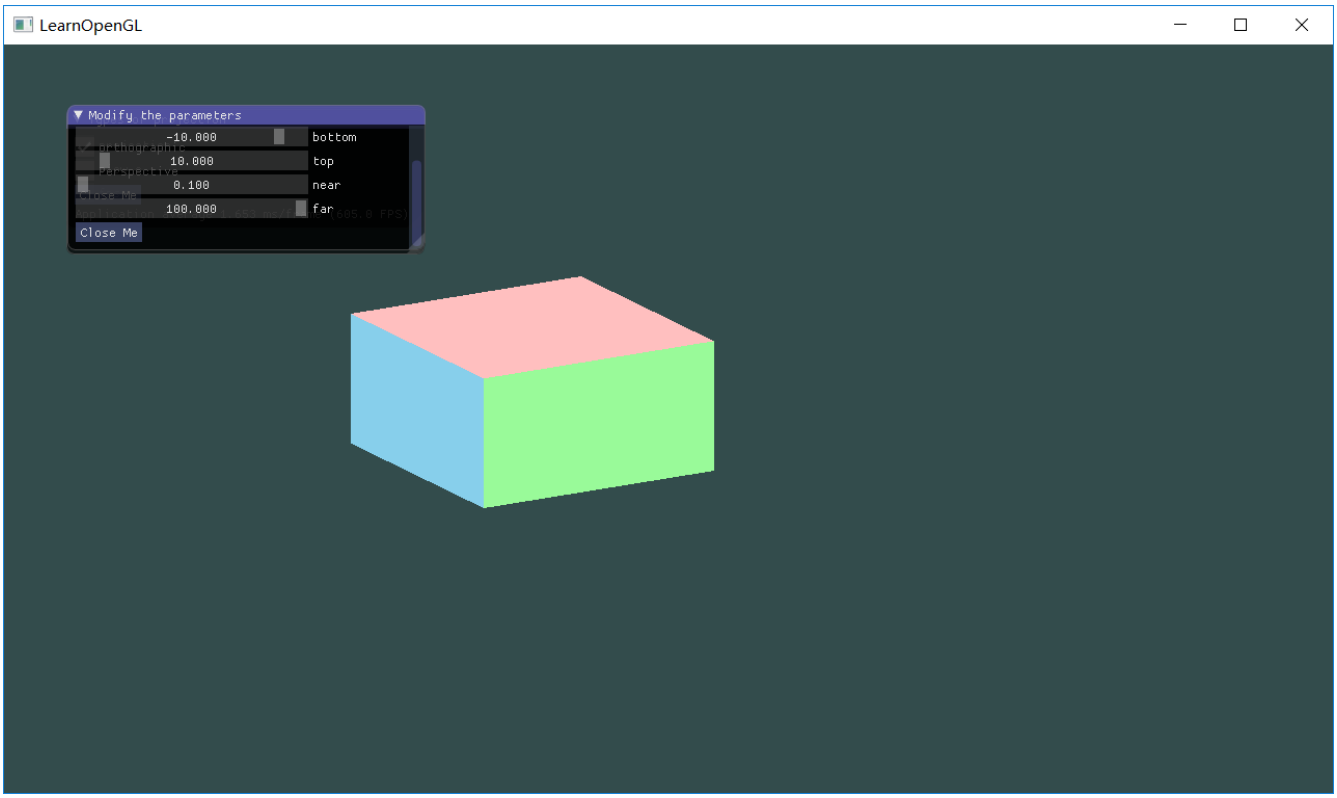
```
// 原本在（0，0，0）处
model = glm::translate(model, glm::vec3(-1.5f, 0.5f, -1.5f));
```

接着需要分别实现正交投影与透视投影。

首先是正交投影，当初始参数的设置为：

```
left = -10.0f
right = 10.0f
bottom = -10.0f
top = 10.0f
z_near = 0.1f
z_far = 30.0f
```

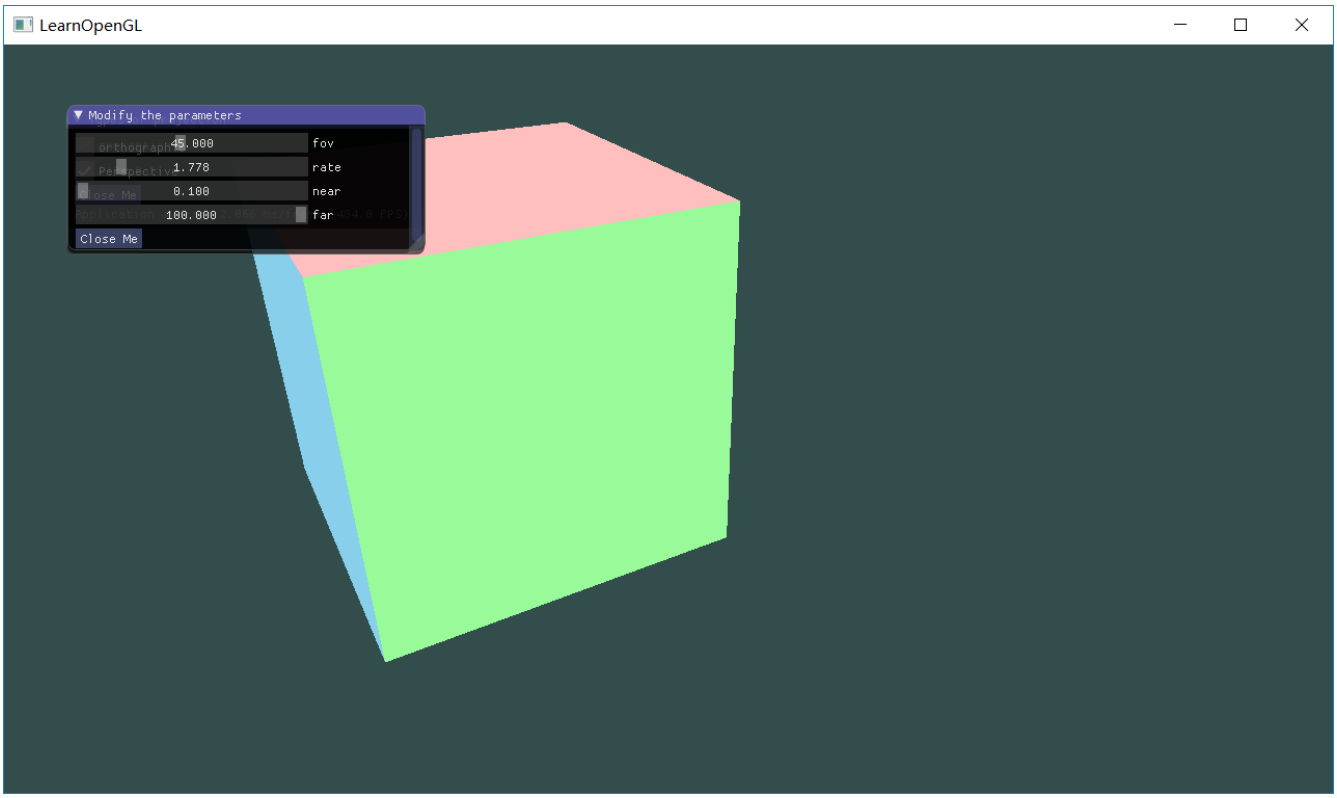
此时的渲染结果为：



对比透视投影，当初始参数为：

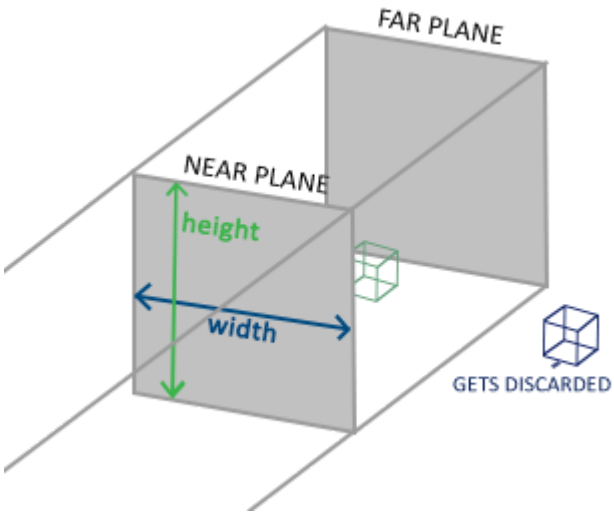
```
fov = 45.0f
ratio = (float)screenWidth / (float)screenHeight
z_near = 0.1f
z_far = 30.0f
```

其效果为：



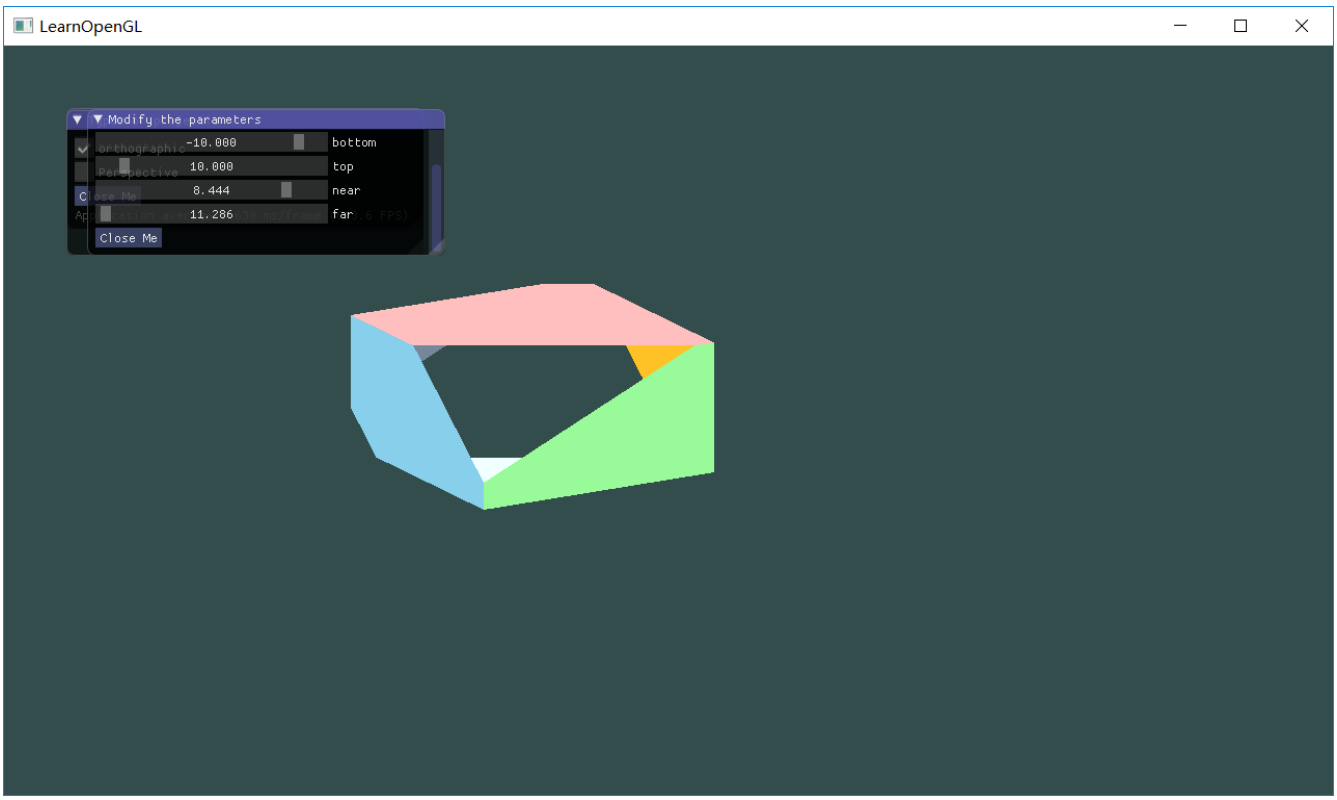
可见，正交投影时很明显不含有“透视”效果，不会有近大远小的感觉。相比之下透视投影就有着更加真实的观感。

在ImGui界面可以更改投影相关的参数，对正交投影来说，其定义了一个类似立方体的平截头箱，它相当于一个裁剪空间，看起来像一个容器：

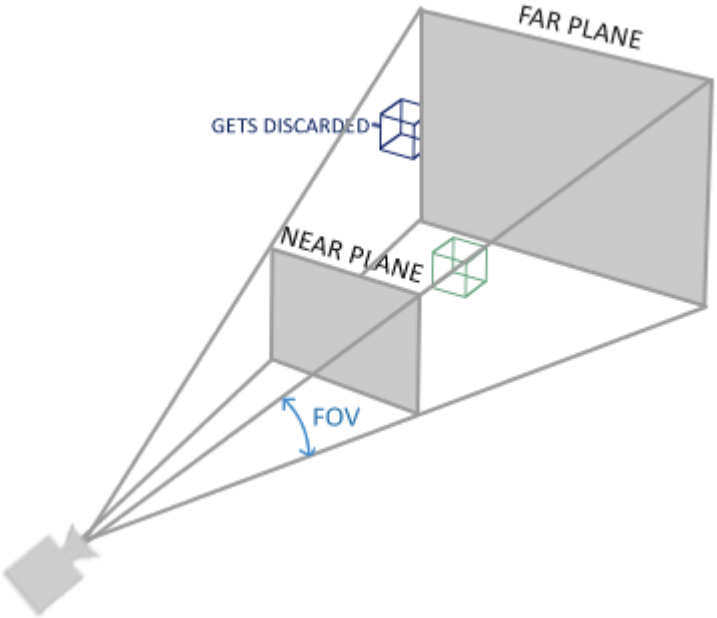


其中，left和right参数指定了平截头体的左右坐标，bottom和top则指定了平截头体的底部和顶部，改变他们会造成立方体呈现效果的拉伸或是压缩，如：

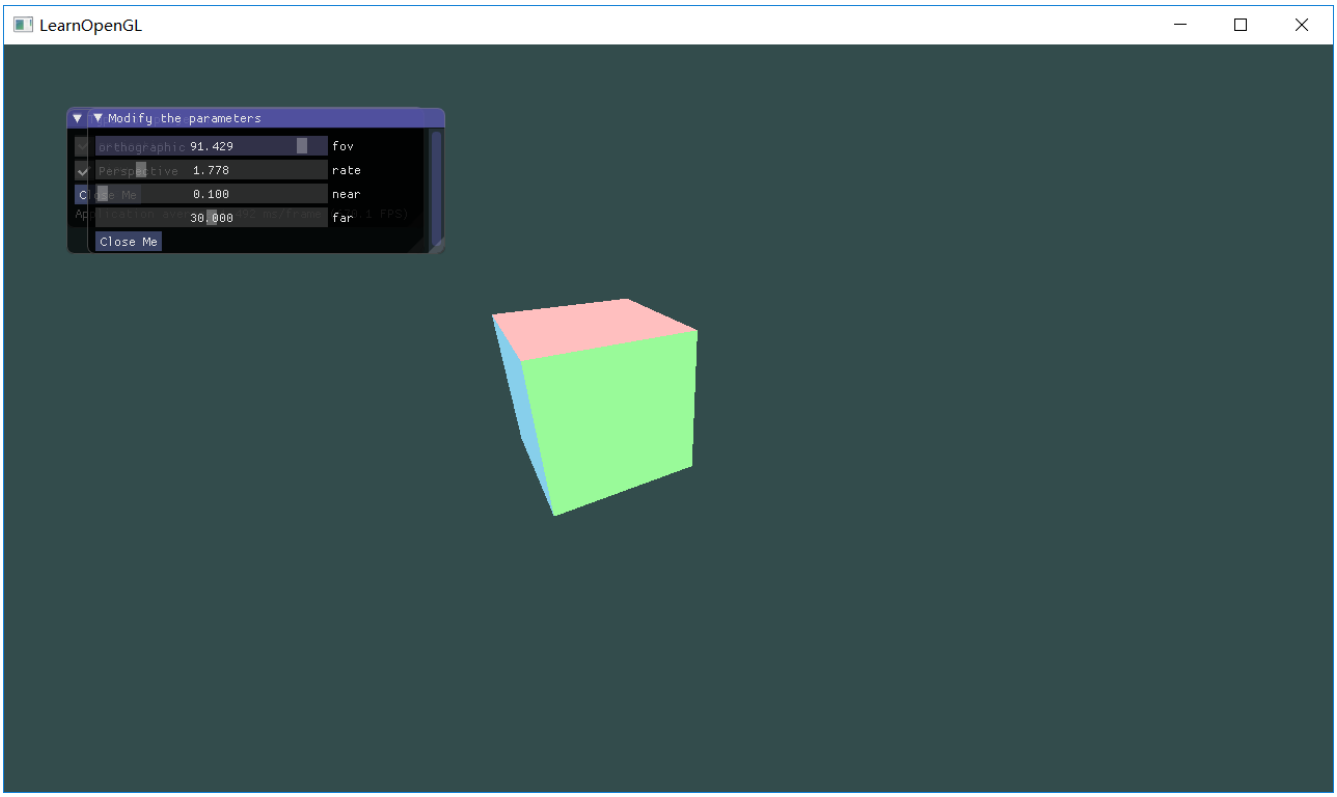
near和far参数则对应平截头体的近平面和远平面的距离，改变他们可能造成立方体有一部分在截头体外而不被渲染，如：



对透视投影来说，它对应的可视空间的平截头体与正交投影的不同，它像是一个不均匀形状的箱子，如图：



其near与far参数的作用与在正交投影中是一样的，而它的fov参数表示的是视野，对应观察空间的大小，真实观察效果下这个参数通常为45.0f（初始参数），如果将它调大一些，原本的立方体看上去好像缩小了：



透视投影的第二个参数设置了宽高比，初始参数设置的是视口的宽除高，改变它同样会产生拉伸或压缩的效果。

视角变换

首先将立方体放置在（0，0，0）即原本的位置上并做透视投影，此时修改摄像机的view矩阵使得摄像机时刻围绕立方体中心旋转，并对准立方体中心。

为了完成此视角变换，需要引入时间函数，并随时间变化周期性改变摄像机的位置、方向（指向摄像机的z轴正方向，这里为(0, 0, 0)）、右轴与上轴（右轴与z轴正方向做叉乘即可得到上轴）。这里设置计算机围绕y轴进行旋转，所以摄像机z轴正向为（0，0，0），右轴为（0，1，0）。最终使用lookAt矩阵来设置观察矩阵。

摄像机的位置计算与lookAt矩阵的设置：

```
float radius = 10.0f, camX = sin(curr_time - ini_time) * radius,
                      camZ = cos(curr_time - ini_time) * radius;
view = glm::lookAt(glm::vec3(camX, 0.0f, camZ),
                      glm::vec3(0.0f, 0.0f, 0.0f),
                      glm::vec3(0.0f, 1.0f, 0.0f));
```

效果：

在现实生活中，我们一般将摄像机摆放的空间View matrix和被拍摄的物体摆设的空间Model matrix分开，但是在OpenGL中却将两个合二为一设为ModelView matrix，我认为这可能是因为在现实中可能有不止一个摄像机进行拍摄。而在opengl中只存在一个摄像机，实际上这个摄像机也只是模拟出来的概念，本质上移动摄像机与移动物体是一样的。Model和View都是在做空间变换的操作（虽然定义上一个关乎物体一个关乎摄像机），所以可以合二为一。相比之下projection矩阵就对应了裁剪的功能，所以一般不合并。

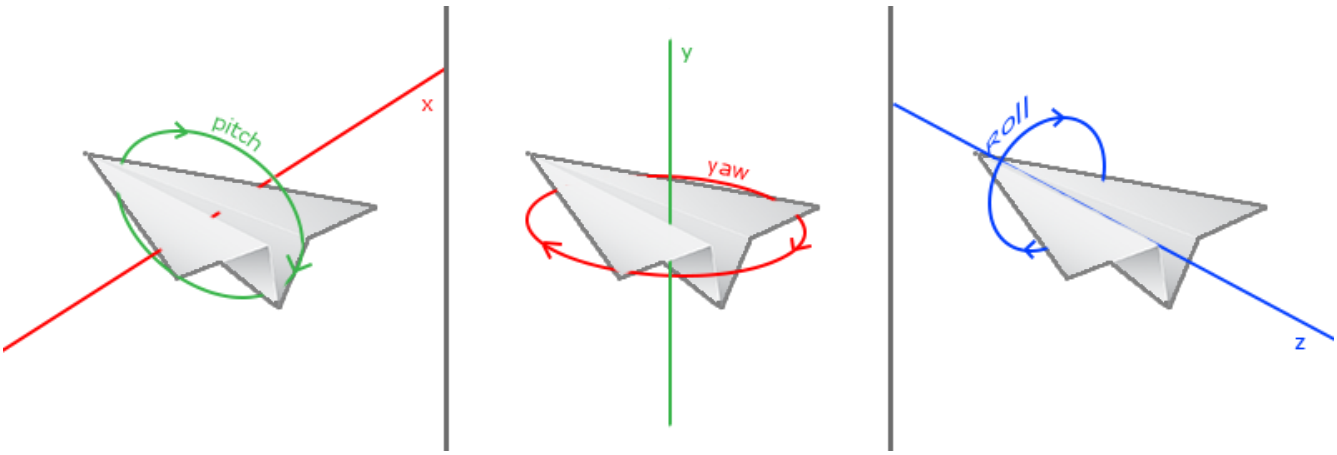
FPS摄像头

作业的最后，我实现了一个camera类来模拟FPS的游戏场景。

首先将键盘输入w，a，s，d对应为摄像头的前后左右移动，这只需捕捉键盘输入并相应改变摄像头的位置，如：

```
void Camera::moveForward() {
    cameraPos += cameraSpeed * cameraFront;
}
void Camera::moveBack() {
    cameraPos -= cameraSpeed * cameraFront;
}
void Camera::moveRight() {
    cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
void Camera::moveLeft() {
    cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```

接着实现鼠标对应的视角移动，这里使用的是欧拉角的方法。欧拉角是可表示3D空间中任何旋转的三个值，有三种欧拉角：俯仰角（pitch）、偏航角（Yaw）与滚转角（Roll），如图所示：



俯仰角是描述往上或下看的角，偏航角表示往左或右看的程度，滚转角则代表如何翻滚摄像机。对FPS摄像头来说，只关心俯仰角和偏航角。

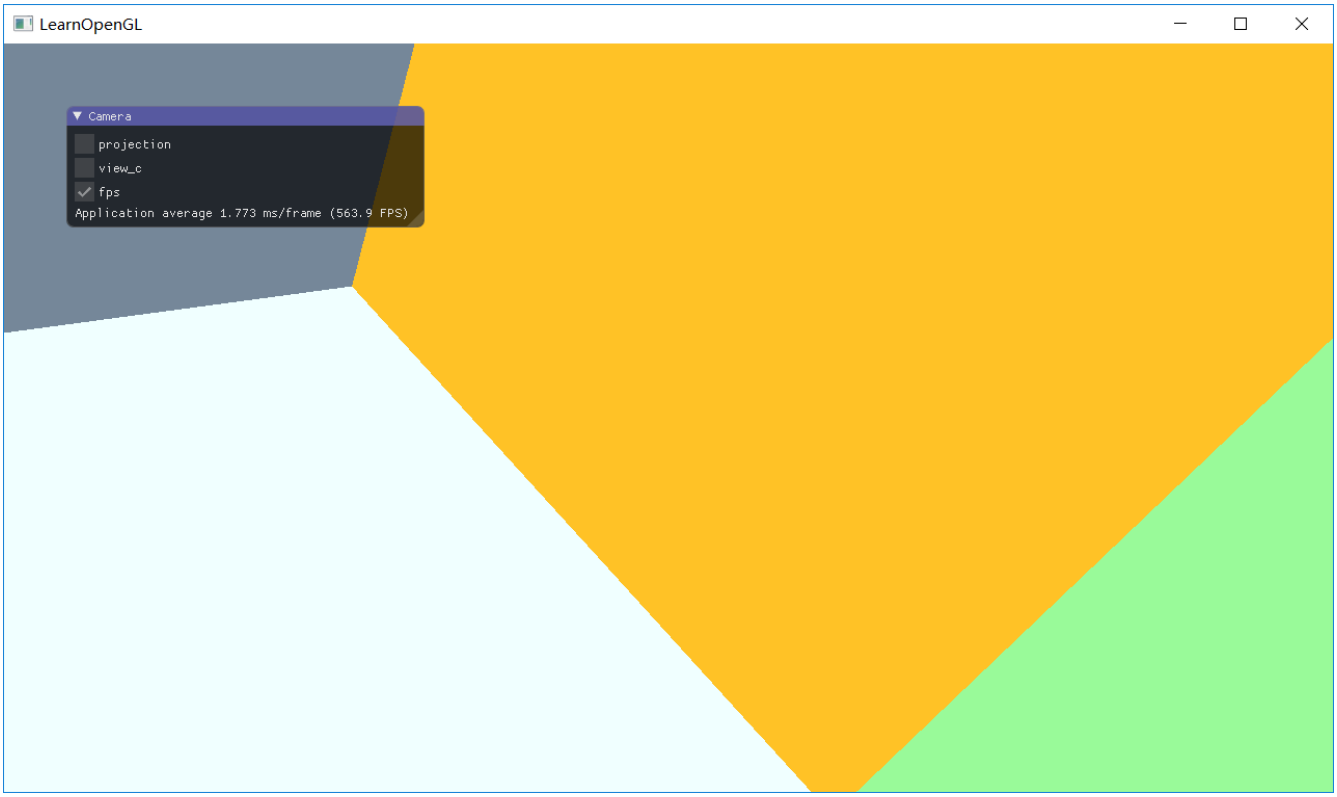
俯仰角对应在xoz平面看向y轴的情况，偏航角则对应xoz平面上的偏转情况，组合起来可以得到基于俯仰角和偏航角的方向向量（其中direction代表摄像机的前轴即z轴正向）：

```
direction.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw));
direction.y = sin(glm::radians(pitch));
direction.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
```

对于俯仰角还存在一些限制，这里限制它无法超过89度且无法小于-89度，即最多只能看到天空或脚下，防止摄像机产生一些奇怪的移动：

```
if(pitch > 89.0f)
  pitch =  89.0f;
if(pitch < -89.0f)
  pitch = -89.0f;
```

最终效果（移动到立方体内部）：


LearnOpenGL
Camera
projection
view_c
fps
Application average 1.773 ms/frame (563.9 FPS)