# Homework 5

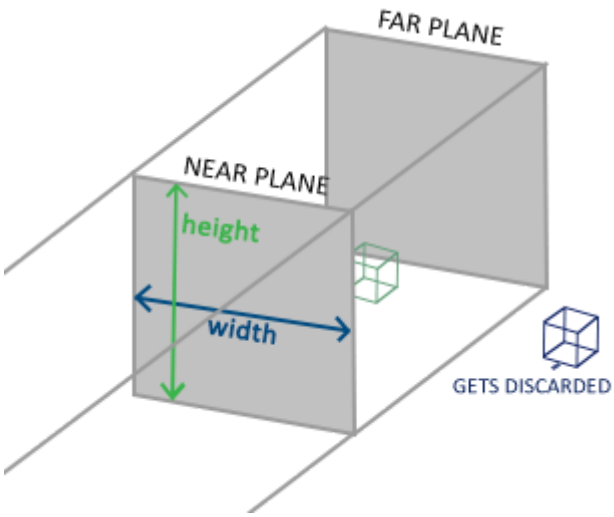
**16340282 袁之浩**

## Basic

### 1. 正交投影(orthographic projection)

#### 算法实现

正射投影矩阵定义了一个类似立方体的平截头箱，它定义了一个裁剪空间，在这空间之外的顶点都会被裁剪掉。创建一个正射投影矩阵需要指定可见平截头体的宽、高和长度。在使用正射投影矩阵变换至裁剪空间之后处于这个平截头体内的所有坐标将不会被裁剪掉。它的平截头体看起来像一个容器：




函数`glm::ortho`的前两个参数指定了平截头体的左右坐标，第三和第四参数指定了平截头体的底部和顶部。通过这四个参数我们定义了近平面和远平面的大小，然后第五和第六个参数则定义了近平面和远平面的距离。这个投影矩阵会将处于这些x，y，z值范围内的坐标变换为标准化设备坐标。

要创建一个正射投影矩阵，我们可以使用GLM的内置函数`glm::ortho`，定义两个数组保存函数的六个参数，使用ImGui::InputFloat4来实现参数的改变。

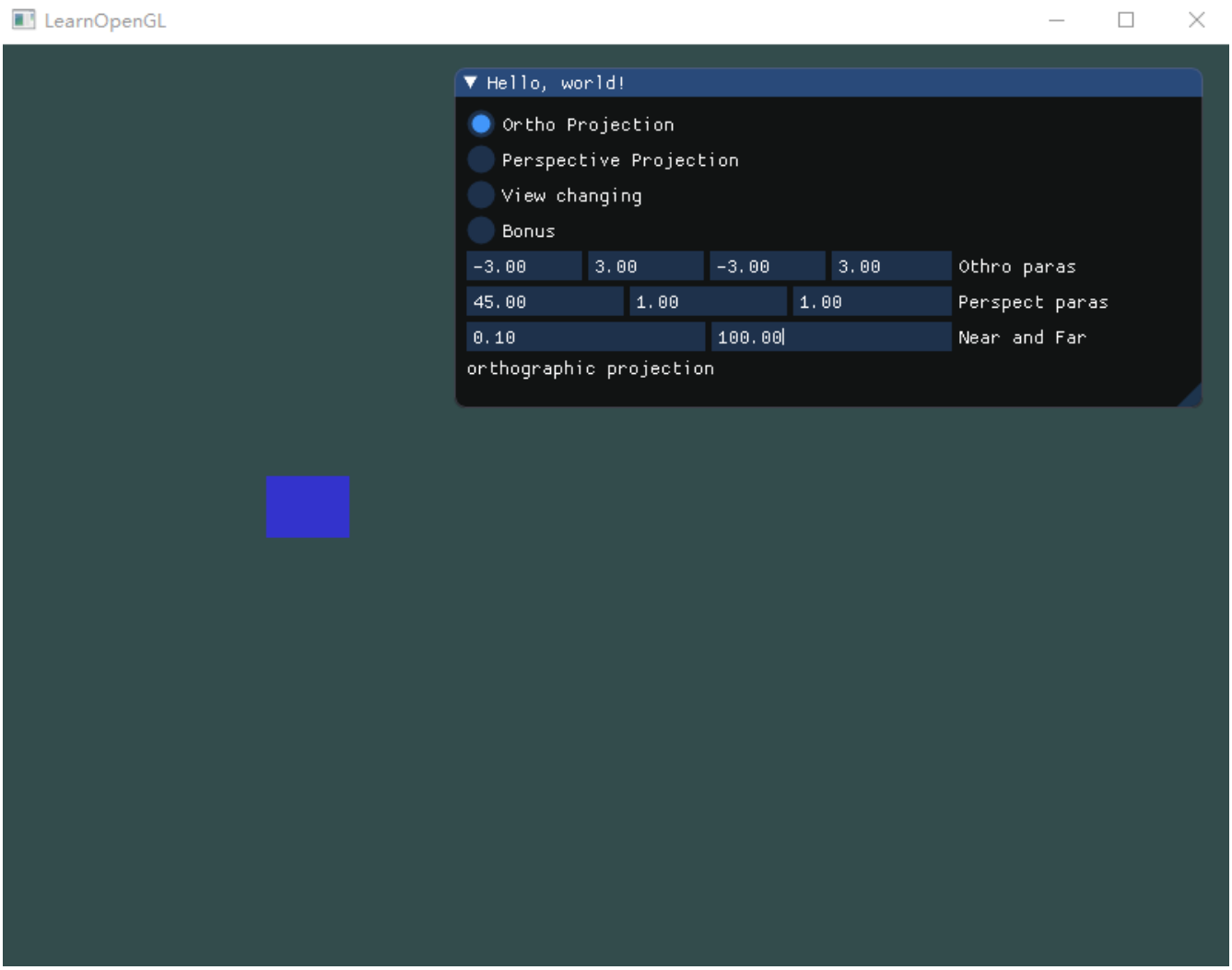
```
float ortho[4] = { -3.0f, 3.0f, -3.0f, 3.0f };
float near_far[2] = { 0.1f, 100.0f };

ImGui::InputFloat4("Othro paras", ortho, 2);
ImGui::InputFloat2("Near and Far", near_far, 2);

projection = glm::ortho(ortho[0], ortho[1], ortho[2], ortho[3], near_far[0],near_far[1]);
```

#### 对比分析

见demo1.gif

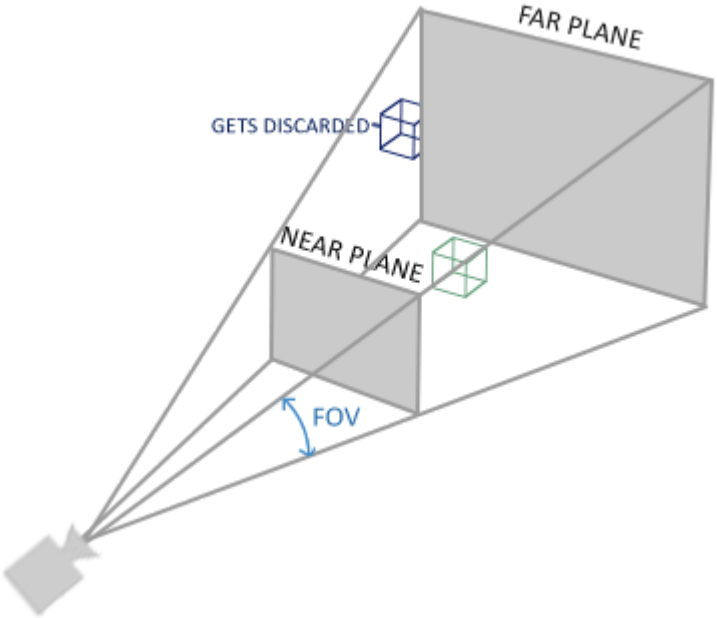



改变前两个参数会导致立方体左右移动，同时发生水平方向的缩放，改变第三、四个参数会导致立方体上下移动，同时发生垂直方向的缩放，最后两个参数设太大或太小将导致看不到立方体。

## 2. 透视投影(perspective projection)

**算法实现**

要定义一个透视投影矩阵，可以使用函数`glm::perspective`，它创建了一个定义了可视空间的大**平截头体**，任何在这个平截头体以外的东西最后都不会出现在裁剪空间体积内，并且将会受到裁剪。一个透视平截头体可以被看作一个不均匀形状的箱子，在这个箱子内部的每个坐标都会被映射到裁剪空间上的一个点。下面是一张透视平截头体的图片：




同样使用ImGui::InputFloat4来实现参数的改变。

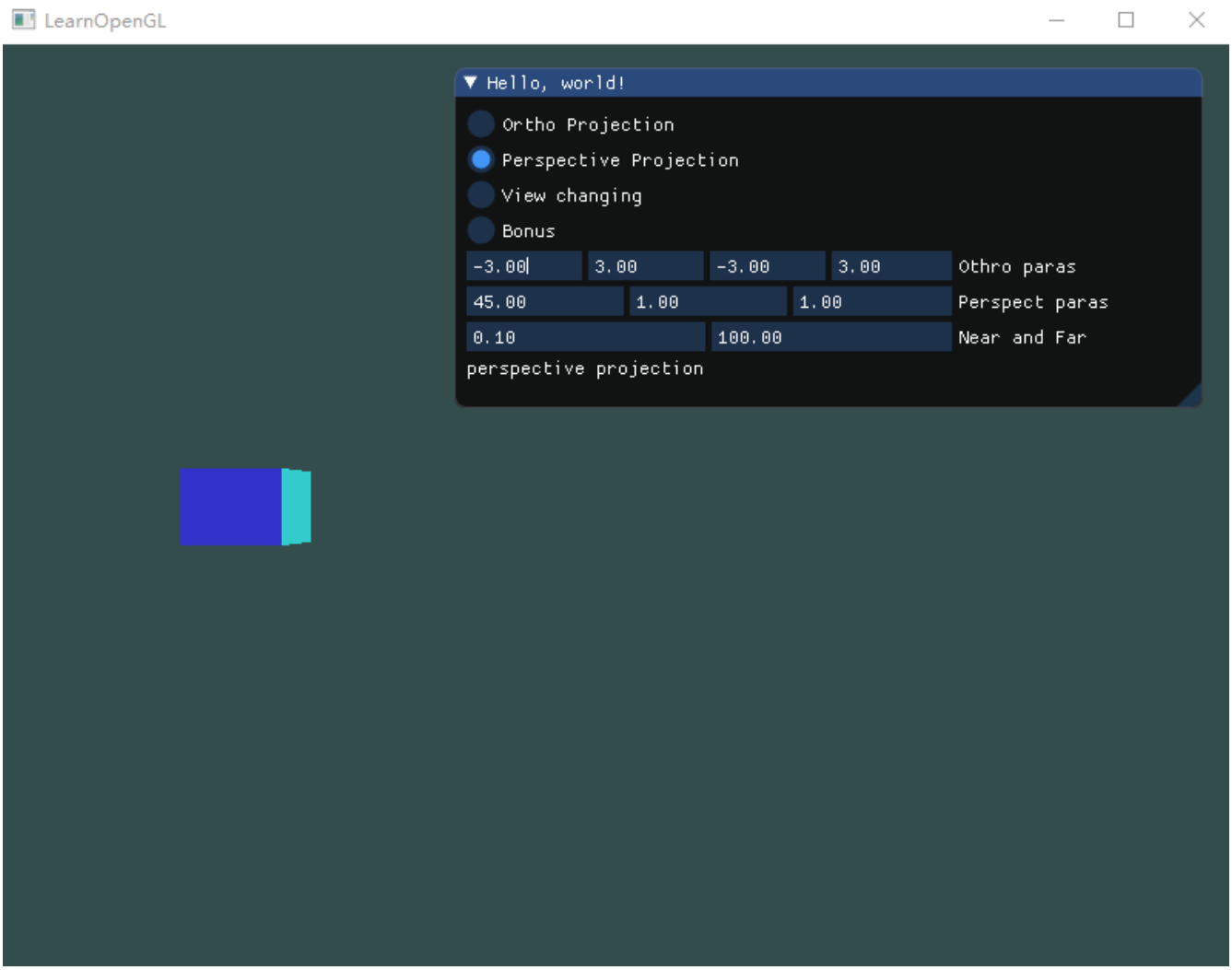
```
float perspect[3] = { 45.0f, 1.0f, 1.0f };
float near_far[2] = { 0.1f, 100.0f };

ImGui::InputFloat3("Perspect paras", perspect, 2);
ImGui::InputFloat2("Near and Far", near_far, 2);

projection = glm::perspective(perspect[0], perspect[1] / perspect[2], near_far[0],
near_far[1]);
```

**对比分析**

见demo2.gif




第一个参数定义了fov的值，它表示的是视野(Field of View)，并且设置了观察空间的大小。第二和第三个值分别表示视口的宽和高。函数的第二个参数就是由视口的宽除以高所得。第三和第四个参数设置了平截头体的**近**和**远**平面。和透视投影一样，所有在近平面和远平面内且处于平截头体内的顶点都会被渲染。

## 3. 视角变换

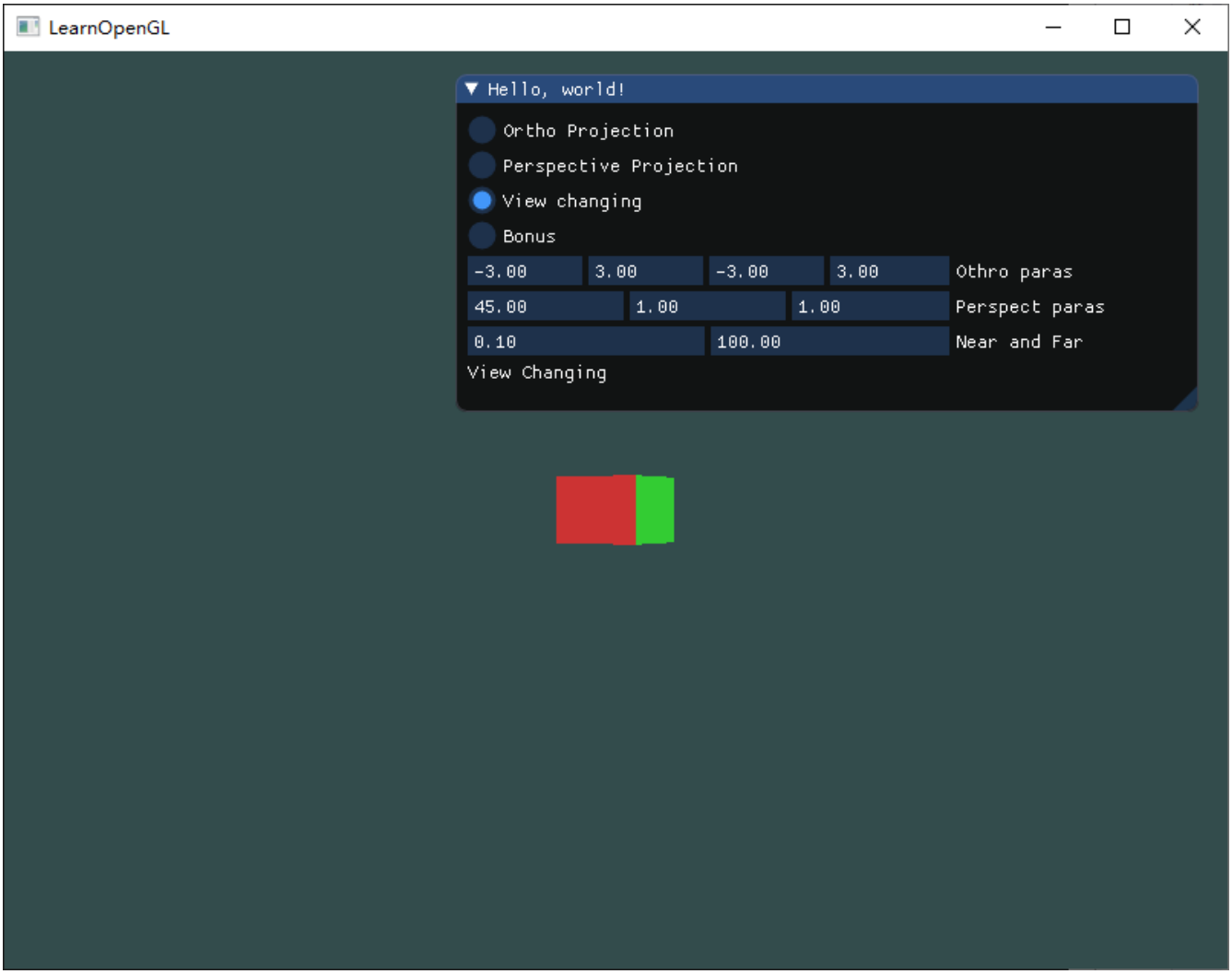
**算法实现**

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。

```
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));
```

**实验截图**




### 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。

## Bounus

### 1. 实现一个camera类，当键盘输入w,a,s,d ，能够前后左右移动；当移动鼠标，能够视角移动("look around")

Camera 类中实现摄像机移动和旋转，每次渲染是通过 `camera.GetViewMatrix()` 获得当前摄像机位置信息的view矩阵。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。

实现按键移动

```
    void ProcessKeyboard(Camera_Movement direction, float deltaTime)
    {
        float velocity = MovementSpeed * deltaTime;
        if (direction == FORWARD)
            Position += Front * velocity;
        if (direction == BACKWARD)
            Position -= Front * velocity;
        if (direction == LEFT)
            Position -= Right * velocity;
        if (direction == RIGHT)
            Position += Right * velocity;
    }
```

实现鼠标改变视角

```
    void ProcessMouseMovement(float xoffset, float yoffset, GLboolean constrainPitch =
true)
    {
        xoffset *= MouseSensitivity;
        yoffset *= MouseSensitivity;

        Yaw += xoffset;
        Pitch += yoffset;

        // Make sure that when pitch is out of bounds, screen doesn't get flipped
        if (constrainPitch)
        {
            if (Pitch > 89.0f)
                Pitch = 89.0f;
            if (Pitch < -89.0f)
                Pitch = -89.0f;
        }

        // Update Front, Right and Up Vectors using the updated Euler angles
        updateCameraVectors();
    }
```

在main函数中监听鼠标和键盘事件，并调用相应函数。