Homework 5 - Camera

黄俊凯 16340082

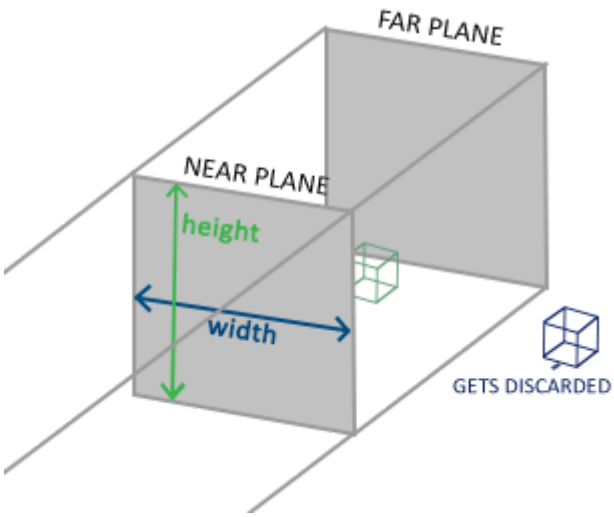
投影

将边长为4的Cube放置在 `(-1.5, 0.5, -1.5)`，只需要将 `Model` 矩阵右乘一个位移矩阵，使用 `glm::translate` 即可：

```
// move the cube from (0, 0, 0) to (-1.5, 0.5, -1.5)
model = glm::translate(model, glm::vec3(-1.5f, 0.5f, -1.5f));
```

正交投影 (Orthographic Projection)

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：

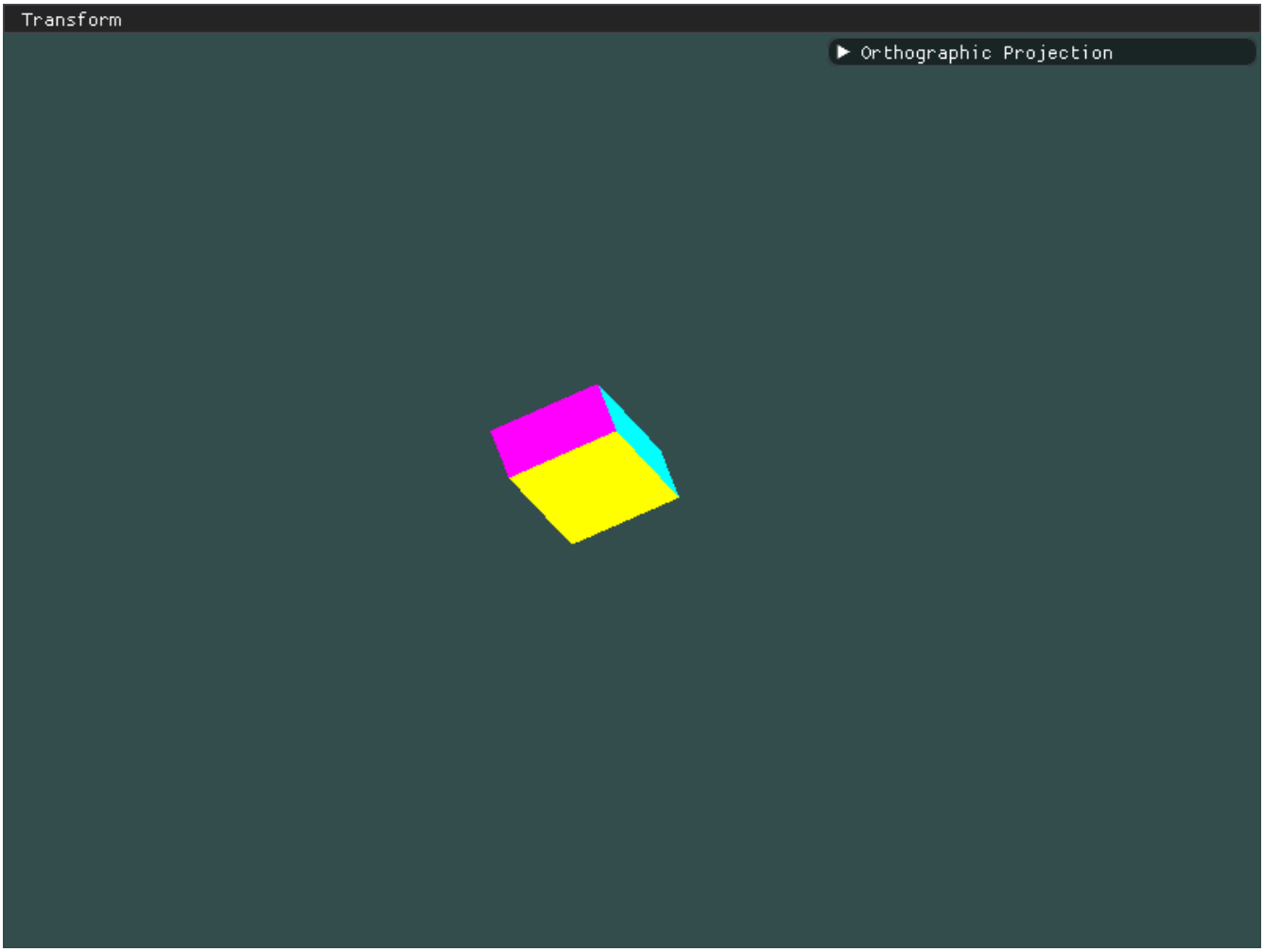


上面的平截头体定义了可见的坐标，它由由宽、高、近(Near)平面和远(Far)平面所指定。任何出现在近平面之前或远平面之后的坐标都会被裁剪掉。

使用 `glm::ortho` 即可帮我们生成一个正交投影矩阵：

```
proj = glm::ortho(left, right, bottom, top, nearP, farP);
```

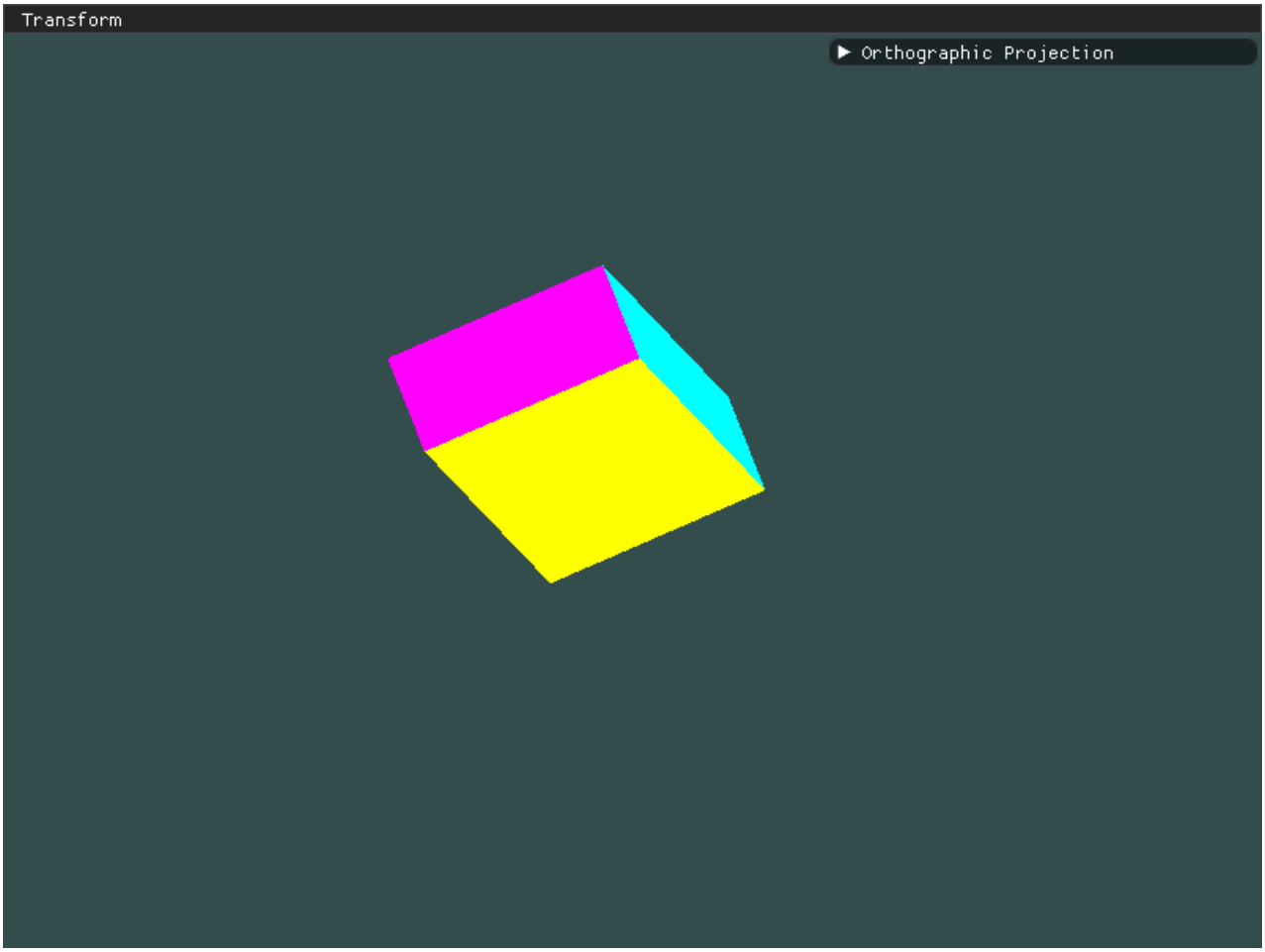
效果



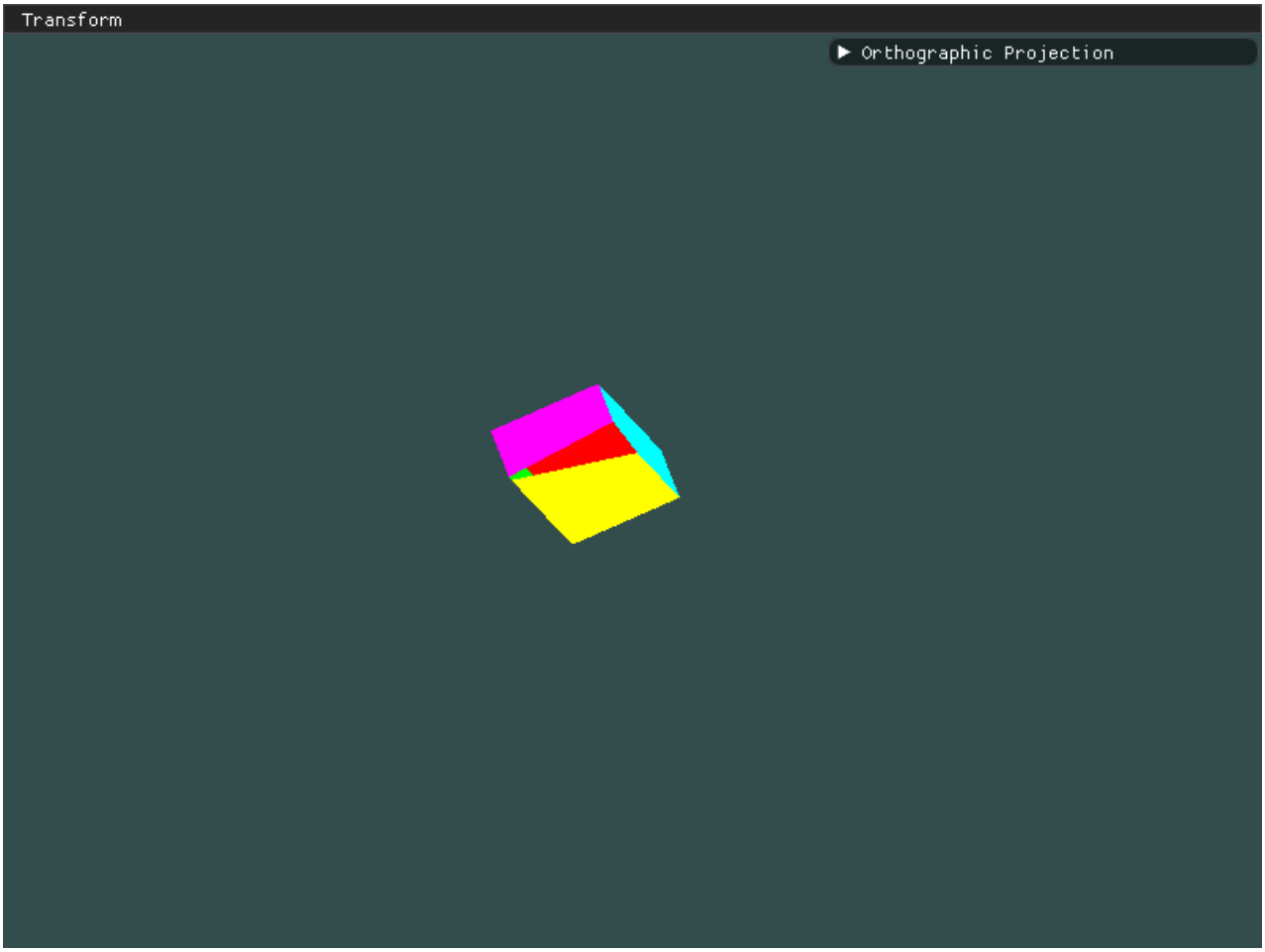
参数实验

在本次实验中，尝试调整了 `(left, right, bottom, top, nearP, farP)` ，其中：

1. `left, right, bottom, top` 直观地决定了展示在屏幕上的内容多少，例如，将它们适当调小，那么可能屏幕就只能展示一个Cube；同时，为了让Cube完整地显示，需要注意调整这4个参数的范围。

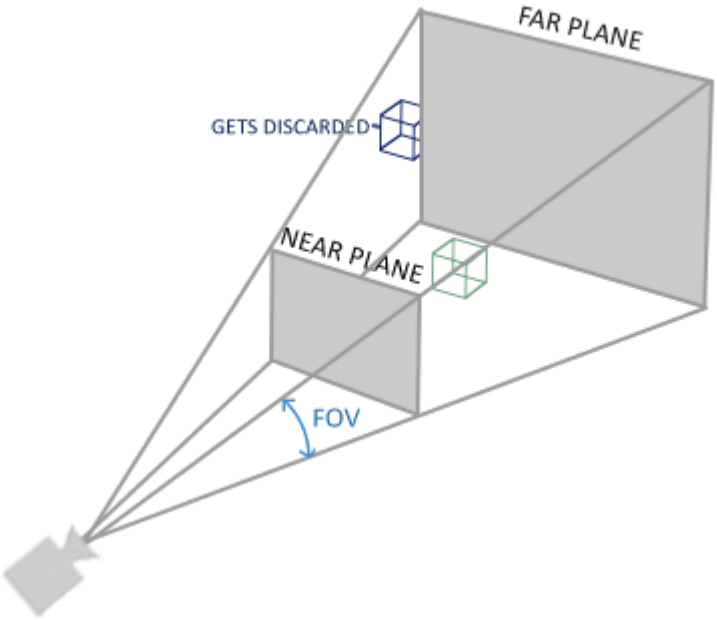


2. `nearP, farP` 决定了摄像机前方多长距离内的物体能被保留，例如将 `nearP` 调大直至超过Cube距离我们最近的z轴距离，如下图，Cube距离我们最近的部分被裁掉了。



透视投影

透视投影即一个定义可视空间的大**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。



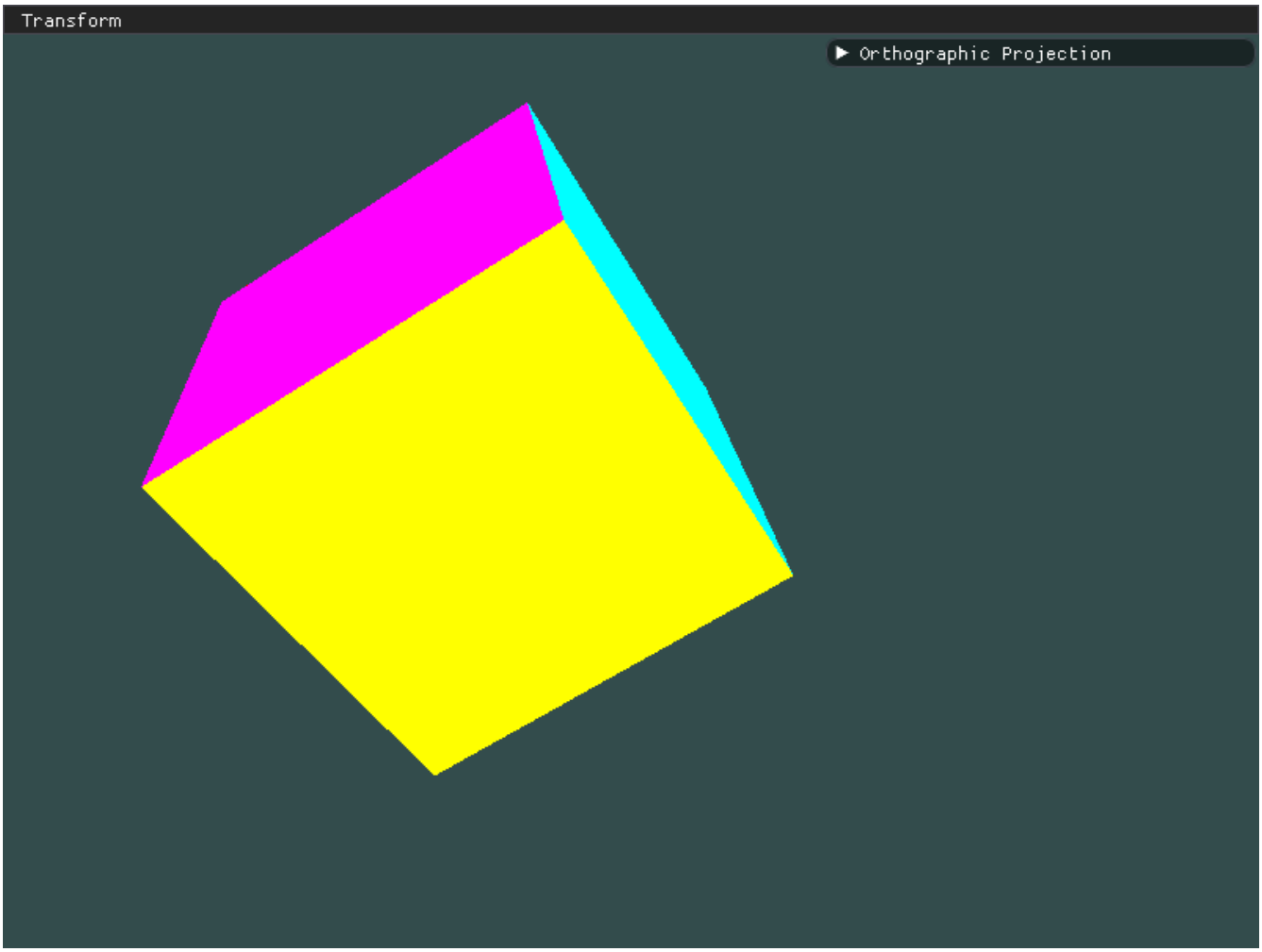
使用 `glm::perspective` 能帮我们快速创建一个透视投影矩阵：

```
proj = glm::perspective(glm::radians(fov), (float)WIDTH / (float)HEIGHT, nearP2, farP2);
```

其中，

1. 第一个参数为视角FOV，越大的话，意味着这个平截头体可以包含更多的内容
2. 第二个参数为宽高比
3. 第三、四个参数分别为近平面和原平面距离摄像机的距离

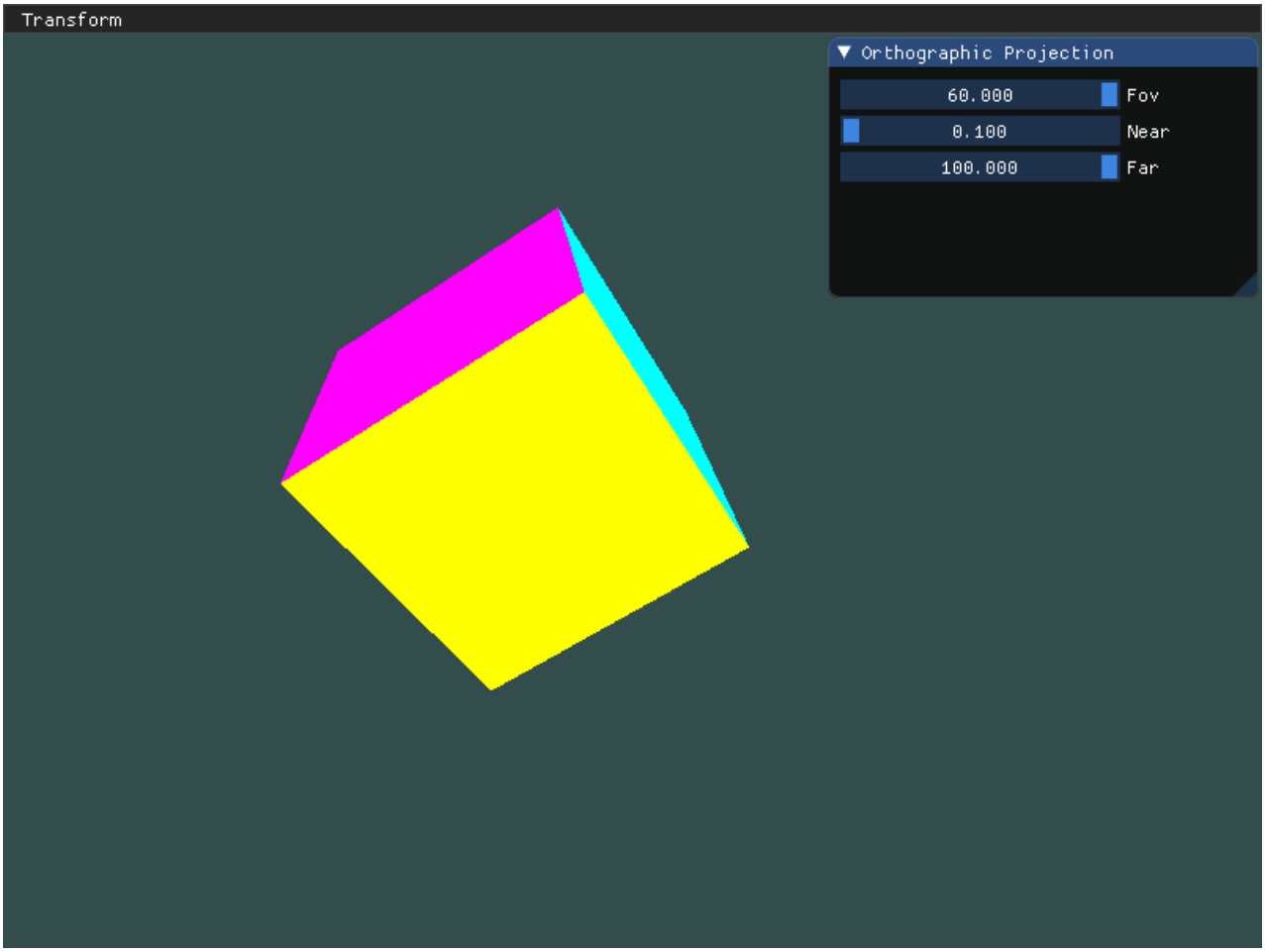
效果



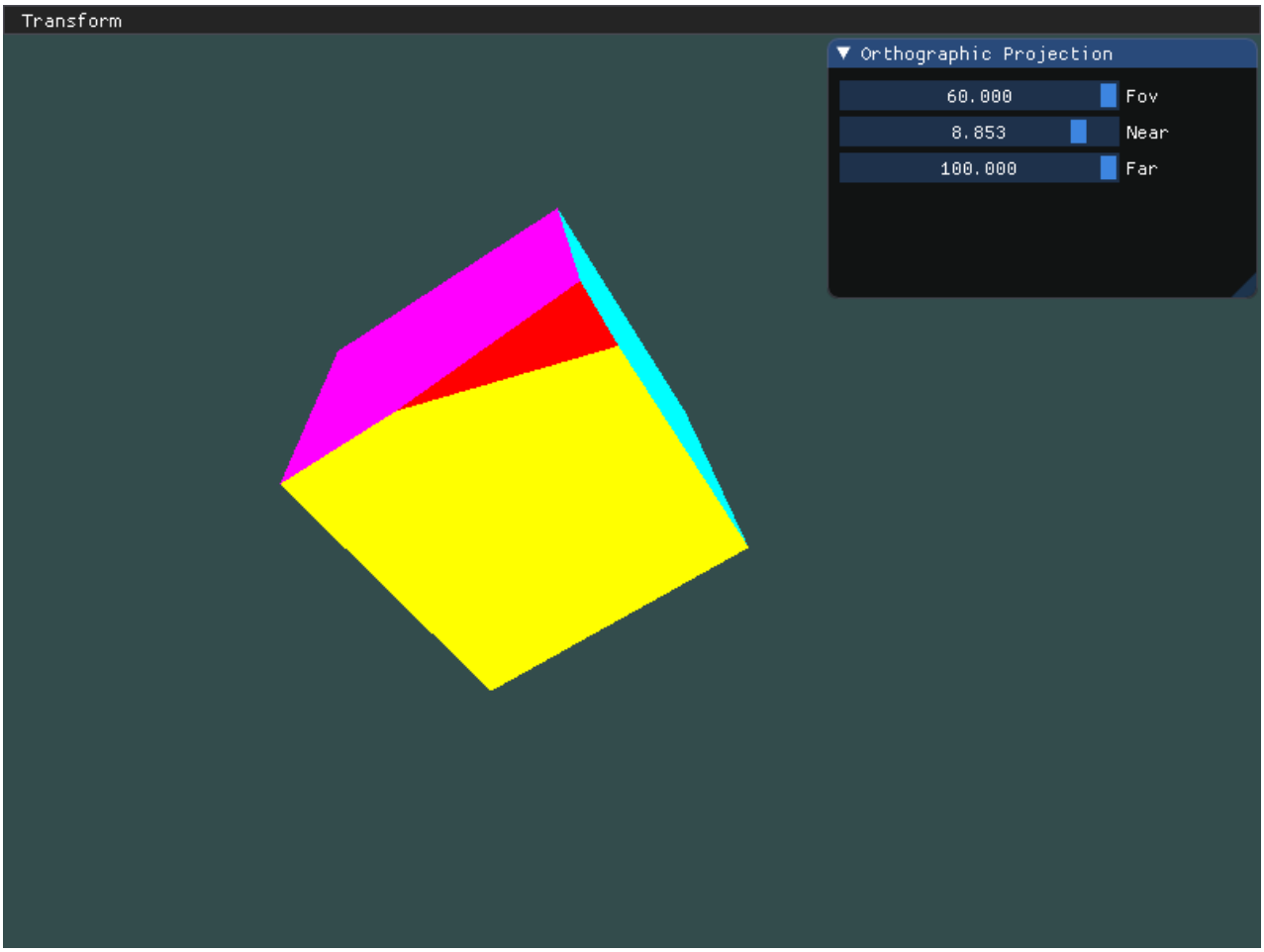
相比正交投影，我们可以发现透视投影的效果更贴近显示中的场景，得益于透视投影矩阵，离得远的点坐标更小。

参数实验

1. `Fov` ：视角的大小，前面说到，值越大，能展示的内容越多，意味着相同大小的物体在Fov更大的情况下，展示在屏幕上越小，如Fov为60：


Transform
Orthographic Projection
60.000
Fov
0.100
Near
100.000
Far

2. `Near, Far` 与正交投影一样，Near如果超过了Cube距离我们最近的点，就会被裁掉，如下图：



视角变换

要让摄像机围绕Cube在XoZ平面上旋转，我们只需要让摄像机的x轴和z轴的坐标满足以下条件：

```
float radius = 15.0f;
float camX = sin(glfwGetTime()) * radius;
float camZ = cos(glfwGetTime()) * radius;
```

然后使用 `glm::lookAt` 来生成 `View` 矩阵：

```
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));
```

其中，

1. 第一个参数为摄像机的位置
2. 第二个参数为目标位置
3. 第三个参数为 `Up` 向量

在上一次作业中，我们为了让摄像机往后移，是让 `view` 矩阵乘以一个位移矩阵，让世界空间内的点往前移。在这里其实原理是一样的，看起来具有魔力的 `LookAt` 矩阵，实际上是根据摄像机坐标系和摄像机位置生成的：

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

效果参见附带的GIF图

OpenGL 的 ModelView Matrix

实现 Camera 类

代码参照 LearnOpenGL，其将输入的操作进行了抽象，使用起来更泛化。

使用WASD移动摄像机

修改之前写的 `processInput` 这个键盘输入回调函数：

```
// keyboard input event callback
// -----------------------------------------------------------------------------
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.processKeyboard(FORWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.processKeyboard(BACKWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.processKeyboard(LEFT, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.processKeyboard(RIGHT, deltaTime);
}
```

这里就可以看到对`Camera`类的移动接口进行抽象的好处，在类的外面进行输入处理，而`Camera`只需要专心处理摄像机移动即可。

为了让摄像机动起来，我们只需要修改它的位置，这里有一个需要注意的点，只有摄像机位置移动的时候，其指向应该一直是正前方，所以我们需要一个 `Front` 单位向量：

```
    Front(glm::vec3(0.0f, 0.0f, -1.0f)),
```

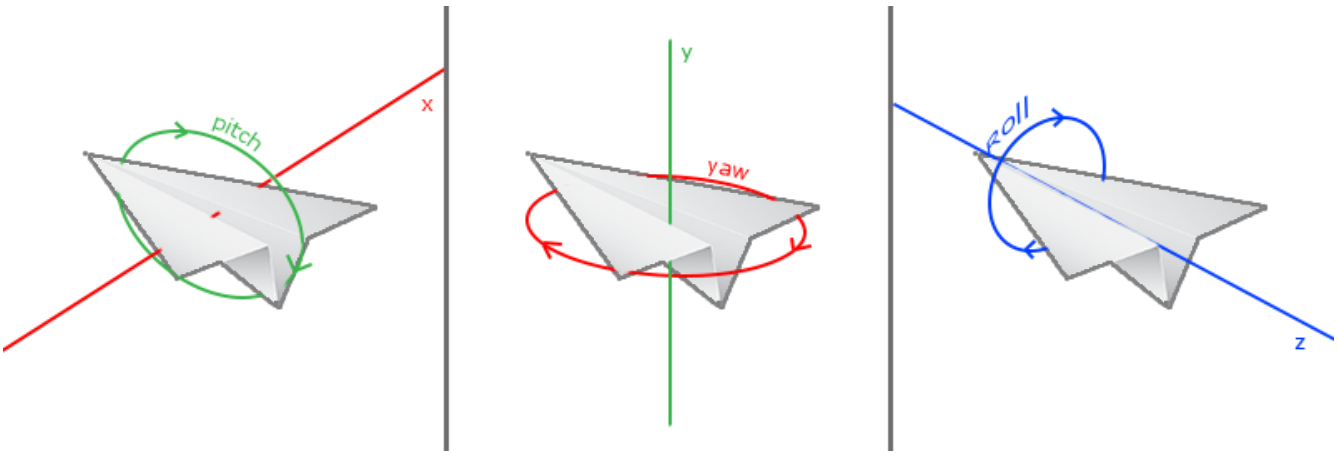
然后使用 `glm::lookAt` 来生成 `View` 矩阵：

```
// Returns the LookAt Matrix
glm::mat4 getViewMatrix() {
    return glm::lookAt(Position, Position + Front, Up);
}
```

注意到第二个参数是摄像机的目标点，所以将摄像机的位置减去 `Front` 向量，即可以让摄像机一直指向我们指定的方向。

移动鼠标让视角移动

本实验使用欧拉角来完成目标，欧拉角(Euler Angle)是可以表示3D空间中任何旋转的3个值，由莱昂哈德·欧拉(Leonhard Euler)在18世纪提出。一共有3种欧拉角：俯仰角(Pitch)、偏航角(Yaw)和滚转角(Roll)，下面的图片展示了它们的含义：



我们只需要用到 Pitch 和 Yaw，当鼠标移动时，我们记录它的偏移量，然后对前面的两个角进行修改：

```
// Processes mouse input
void processMouseMovement(float xoffset, float yoffset, GLboolean constrainPitch = true) {
    xoffset *= MouseSensitivity;
    yoffset *= MouseSensitivity;

    Pitch += yoffset;
    Yaw   += xoffset;

    // 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
    updateCameraVectors();
}
```

生成新的欧拉角后，我们就需要更新 `Front, Right, Up` 三个摄像机坐标系向量了：

```
// Calculates the front vector from the Camera's Euler Angles
void updateCameraVectors() {
    glm::vec3 front;
    front.x = cos(glm::radians(Pitch)) * cos(glm::radians(Yaw));
    front.y = sin(glm::radians(Pitch));
    front.z = cos(glm::radians(Pitch)) * sin(glm::radians(Yaw));
    Front = glm::normalize(front);
    // Re-calculate the Right and Up vector
    Right = glm::normalize(glm::cross(Front, WorldUp));
    Up    = glm::normalize(glm::cross(Right, Front));
}
```

鼠标滚轮实现缩放

如果还记得上面的透视投影中 `fov` 这个参数，我们就知道如何实现缩放了，通过鼠标滚轮来改变 `fov` 的值，值变小表示画面放大，同时对缩放进行了限制：

```
// Process mouse scroll-wheel event
void processMouseScroll(float yoffset) {
    if (Zoom >= 1.0f && Zoom <= 60.0f) Zoom -= yoffset;
    if (Zoom < 1.0f)  Zoom = 1.0f;
    if (Zoom > 60.0f) Zoom = 60.0f;
}
```

效果参见附带GIF图