Homework 5 - Camera

黄树凯

16340085

作业要求

Basic

1. 投影(Projection):

- 把上次作业绘制的cube放置在(-1.5, 0.5, -1.5)位置，要求6个面颜色不一致
- 正交投影(orthographic projection)：实现正交投影，使用多组(left, right, bottom, top, near, far)参数，比较结果差异
- 透视投影(perspective projection)：实现透视投影，使用多组参数，比较结果差异

2. 视角变换(View Changing):

- 把cube放置在(0, 0, 0)处，做透视投影，使摄像机围绕cube旋转，并且时刻看着cube中心

3. 在GUI里添加菜单栏，可以选择各种功能。 Hint: 使摄像机一直处于一个圆的位置，可以参考以下公式：

```
camPosX=sin(clock()/1000.0)*Radius;
camPosZ=cos(clock()/1000.0)*Radius;
```

   原理很容易理解，由于圆的公式a^2 + b^2=1 ，以及有sin(x)^2 + cos(x)^2=1 ，所以能保证摄像机在XoZ平面的一个圆上。

4. 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，通过上面的作业启发，你认为是为什么呢？在报告中写入。（Hints：你可能有不止一个摄像机）

Bonus

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")，即类似FPS(First Person Shooting)的游戏场景 Hint: camera类的头文件可以参考如下（同样也可以自己定义，只要功能相符即可）：

```
class Camera{
public:
    ...
    void moveForward(GLfloat const distance);
    void moveBack(GLfloat const distance);
    void moveRight(GLfloat const distance);
    void moveLeft(GLfloat const distance);
    ...
    void rotate(GLfloat const pitch, GLfloat const yaw);
    ...
```

```
private:
    ...
    GLfloat pfov,pratio,pnear,pfar;
    GLfloat cameraPosX,cameraPosY,cameraPosZ;
    GLfloat cameraFrontX,cameraFrontY,cameraFrontZ;
    GLfloat cameraRightX,cameraRightY,cameraRightZ;
    GLfloat cameraUpX,cameraUpY,cameraUpZ;
    ...
};
```

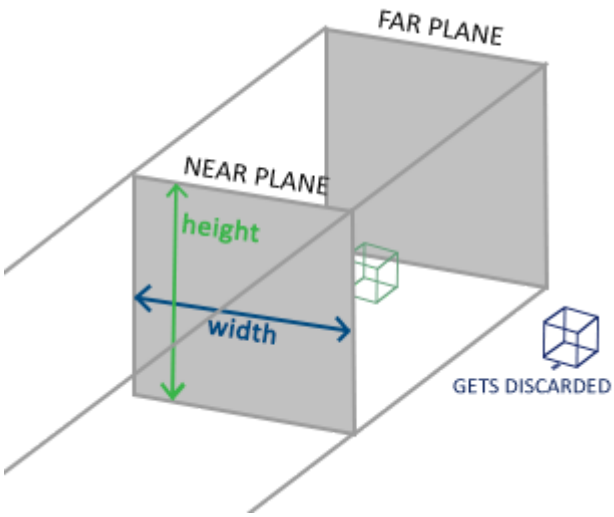
PS. void rotate(GLfloat const pitch, GLfloat const yaw) 里的pitch 、yaw 均为欧拉角（参考上方 References）

实现及结果

1. 投影

1.正交投影

正交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。创建一个正交投影矩阵需要指定可见平截头体的宽、高和长度。在使用正交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。正交平截头体直接将平截头体内部的所有坐标映射为标准化设备坐标，因为每个向量的w分量都没有进行改变；如果w分量等于1.0，透视除法则不会改变这个坐标。

要创建一个正交投影矩阵，可以使用GLM的内置函数`glm::ortho`：

```
glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);
```

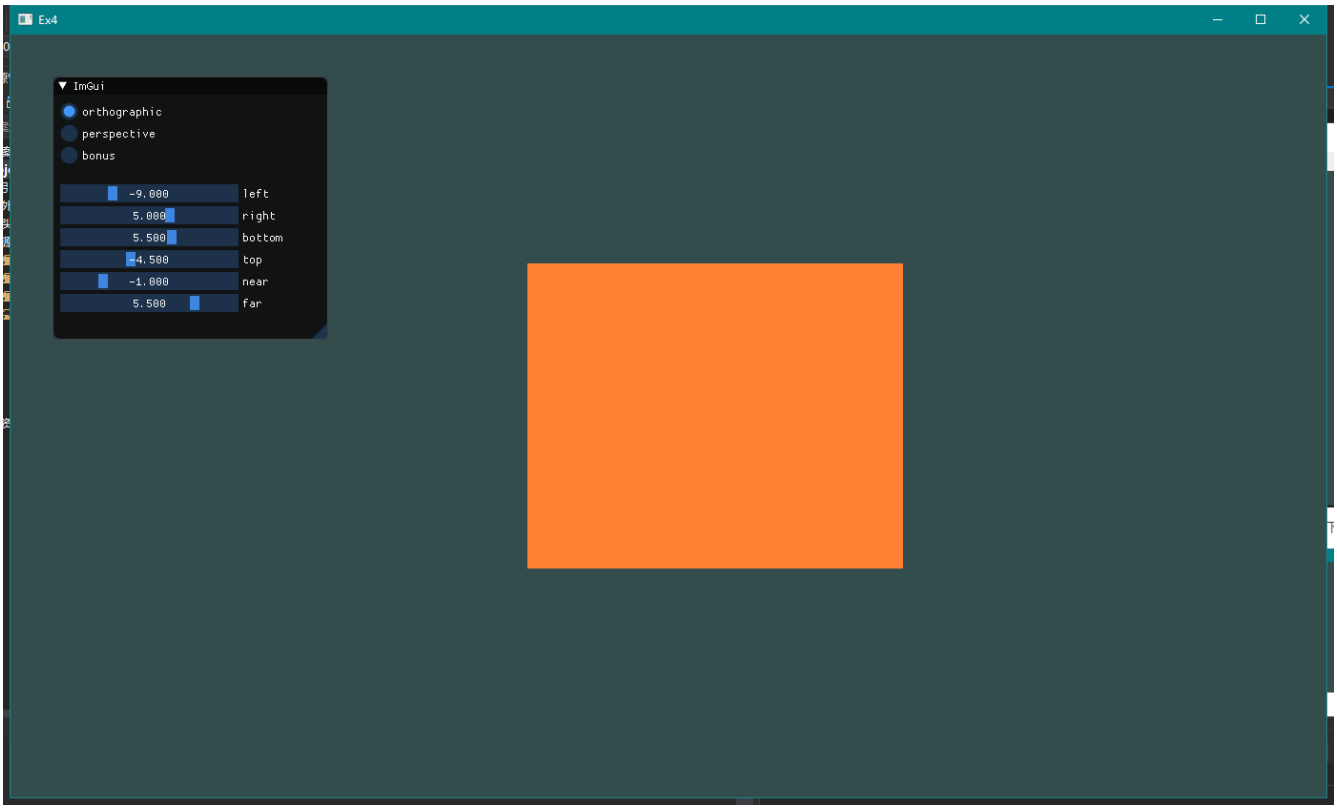
前两个参数指定了平截头体的左右坐标，第三和第四参数指定了平截头体的底部和顶部。通过这四个参数我们定义了近平面和远平面的大小，然后第五和第六个参数则定义了近平面和远平面的距离。这个投影矩阵会将处于这些x，y，z值范围内的坐标变换为标准化设备坐标。

正交投影矩阵直接将坐标映射到2D平面中，即屏幕，但实际上一个直接的投影矩阵会产生不真实的结果，当使用正交投影时，每一个顶点坐标都会直接映射到裁剪空间中而不经过任何精细的透视除法（它仍然会进行透视除法，只是w分量没有被改变（它保持为1），因此没有起作用）。因为正交投影没有使用透视，远处的物体不会显得更小，所以产生奇怪的视觉效果。

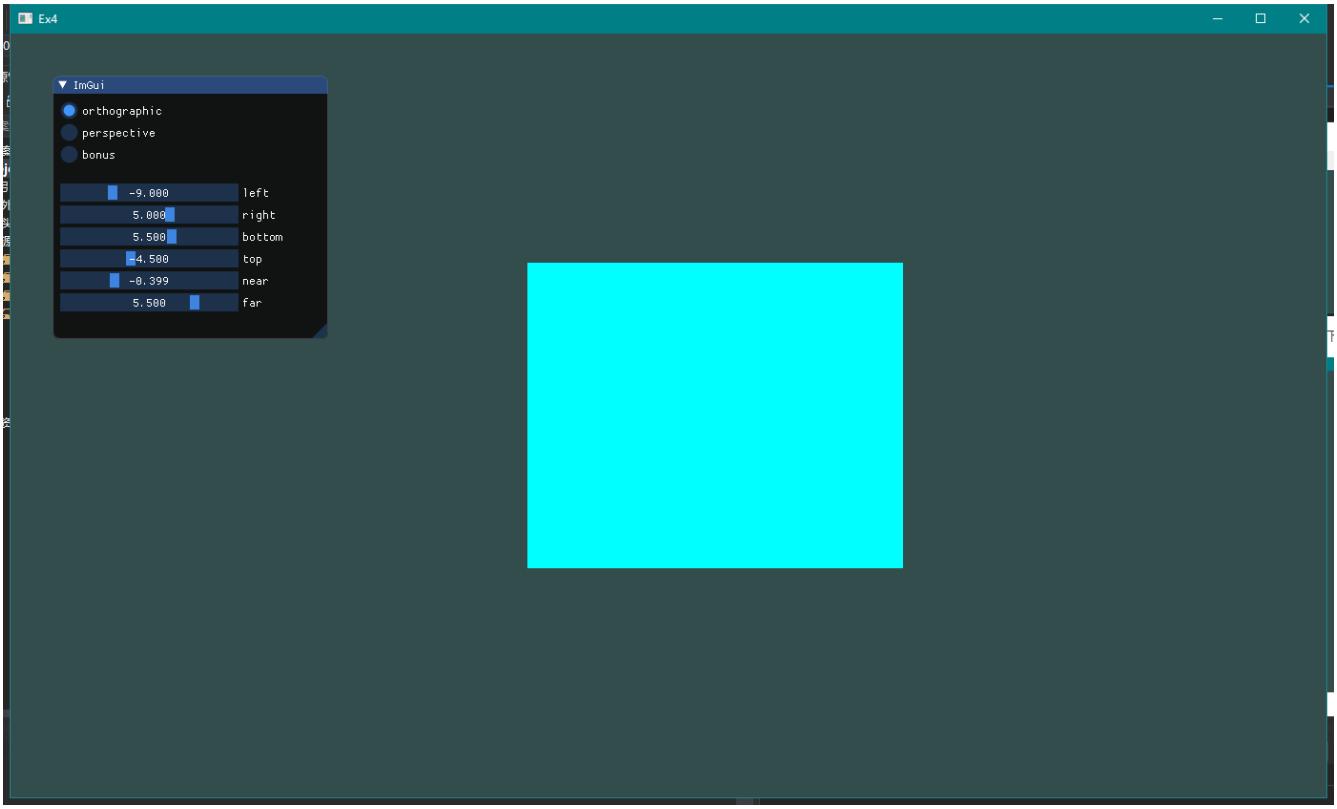
实现方法：

直接调用GLM的内置函数 `glm::ortho` 创建一个正交投影矩阵，将 cube 放置在 (-1.5f, 0.5f, -1.5f)的位置，将摄像机正对 cube 其中一个面，适当调整参数后，可以看到如下效果

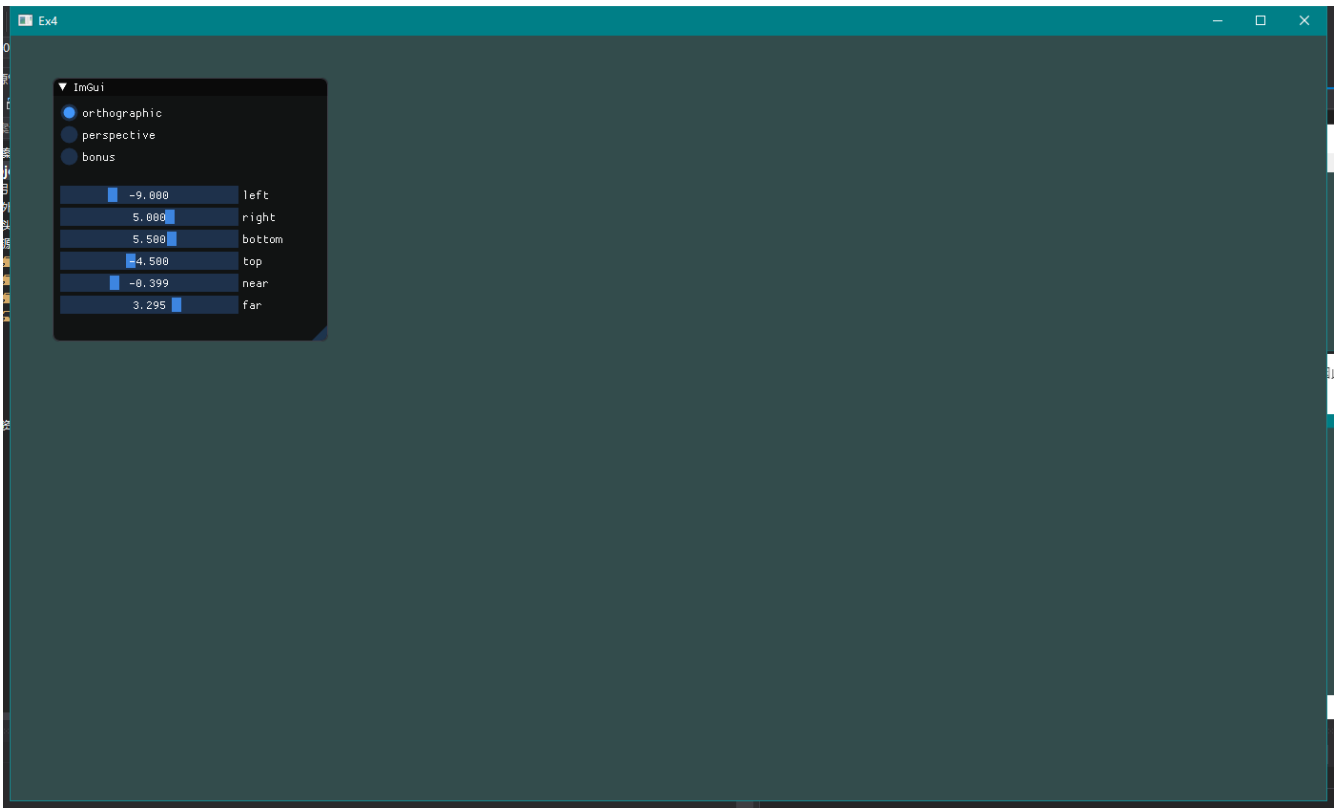
```
model = glm::translate(model, glm::vec3(-1.5f, 0.5f, -1.5f));  // 将 cube 放置在 (-1.5, 0.5, -1.5) 的位置
projection = glm::ortho(ortho_left, ortho_right, ortho_bottom, ortho_top, ortho_near, ortho_far);
```



由于cube 的边长为4，当参数near大于 -0.5 时将剪裁掉near之前的部分，因此看到的会是另一个面，如下

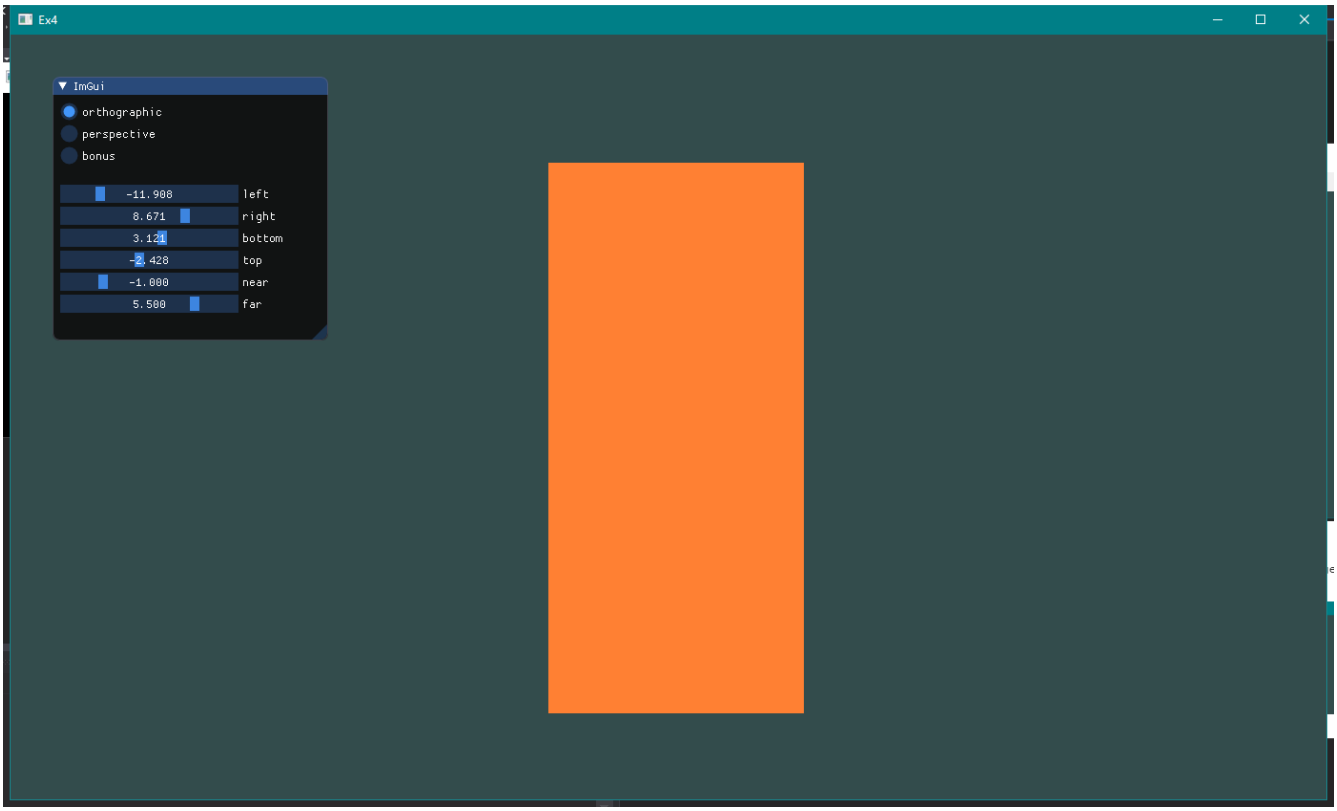


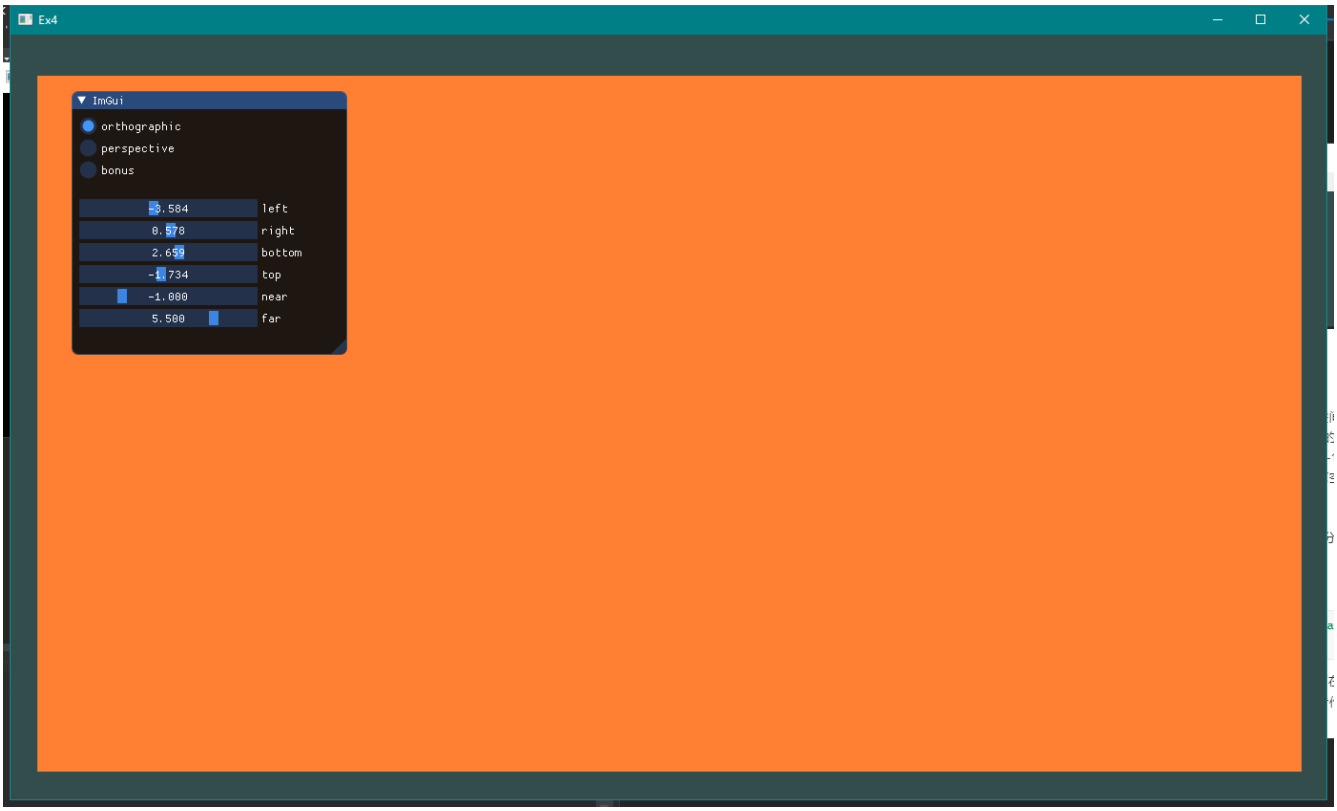
而当参数far 小于3.5 时，由于剪裁此时整个 cube 前后两个面都看不到，而其余四个面垂直于摄像机，因此也看不到，结果如下



改变 left, right, top, bottom 参数的效果如图


Ex4
ImGui
orthographic
perspective
bonus
-3.584
left
0.578
right
14.220
bottom
-12.832
top
-1.000
near
5.500
far


Ex4
ImGui
orthographic
perspective
bonus
-11.908
left
8.671
right
3.121
bottom
-2.428
top
-1.000
near
5.500
far



参数约大则该方向上距离摄像机约远，因此图像越小；参数越小，则该方向上距离摄像机越近，图像越大。

2.透视投影

透视投影是使用透视投影矩阵来完成的。这个投影矩阵将给定的平截头体范围映射到裁剪空间，除此之外还修改了每个顶点坐标的w值，从而使得离观察者越远的顶点坐标w分量越大。被变换到裁剪空间的坐标都会在-w到w的范围之间（任何大于这个范围的坐标都会被裁剪掉）。OpenGL要求所有可见的坐标都落在-1.0到1.0范围内，作为顶点着色器最后的输出，因此，一旦坐标在裁剪空间内之后，透视除法就会被应用到裁剪空间坐标上：

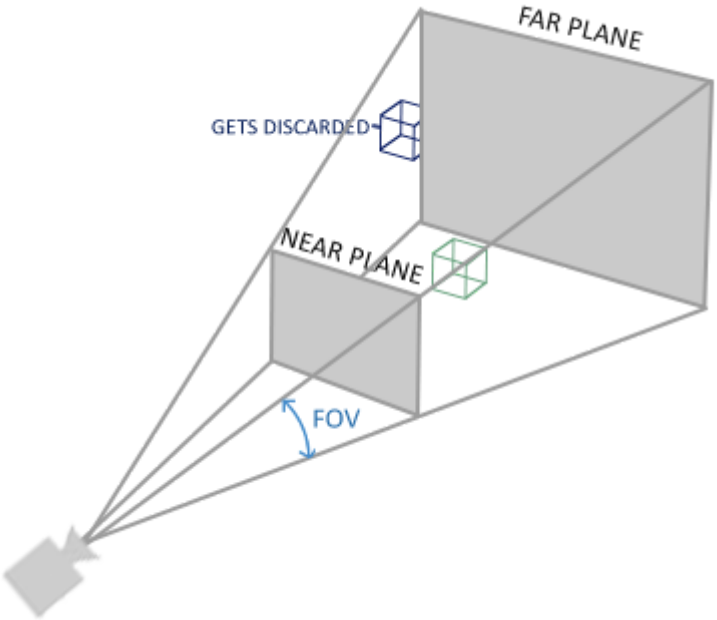
out = (x/w , y/w, z/w)

顶点坐标的每个分量都会除以它的w分量，距离观察者越远顶点坐标就会越小。这是也是w分量非常重要的另一个原因，它能够帮助我们进行透视投影。最后的结果坐标就是处于标准化设备空间中的。

在GLM中可以这样创建一个透视投影矩阵：

```
glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);
```

同样，`glm::perspective`所做的其实就是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：

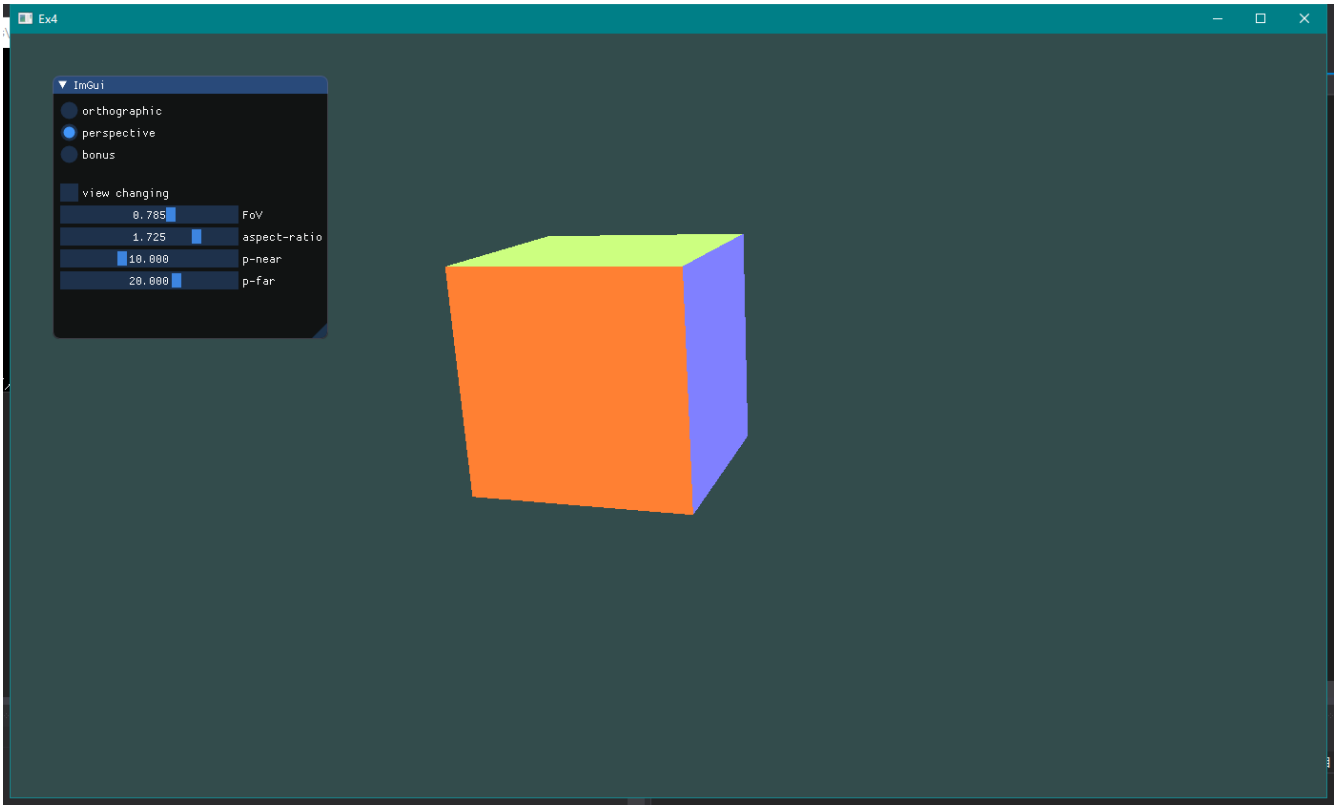


它的第一个参数定义了 fov 的值，它表示的是视野(Field of View)，并且设置了观察空间的大小。如果想要一个真实的观察效果，它的值通常设置为45.0f，。第二个参数设置了宽高比，由视口的宽除以高所得。第三和第四个参数设置了平截头体的**近**和**远**平面。通常设置近距离为0.1f，而远距离设为100.0f。所有在近平面和远平面内且处于平截头体内的顶点都会被渲染。

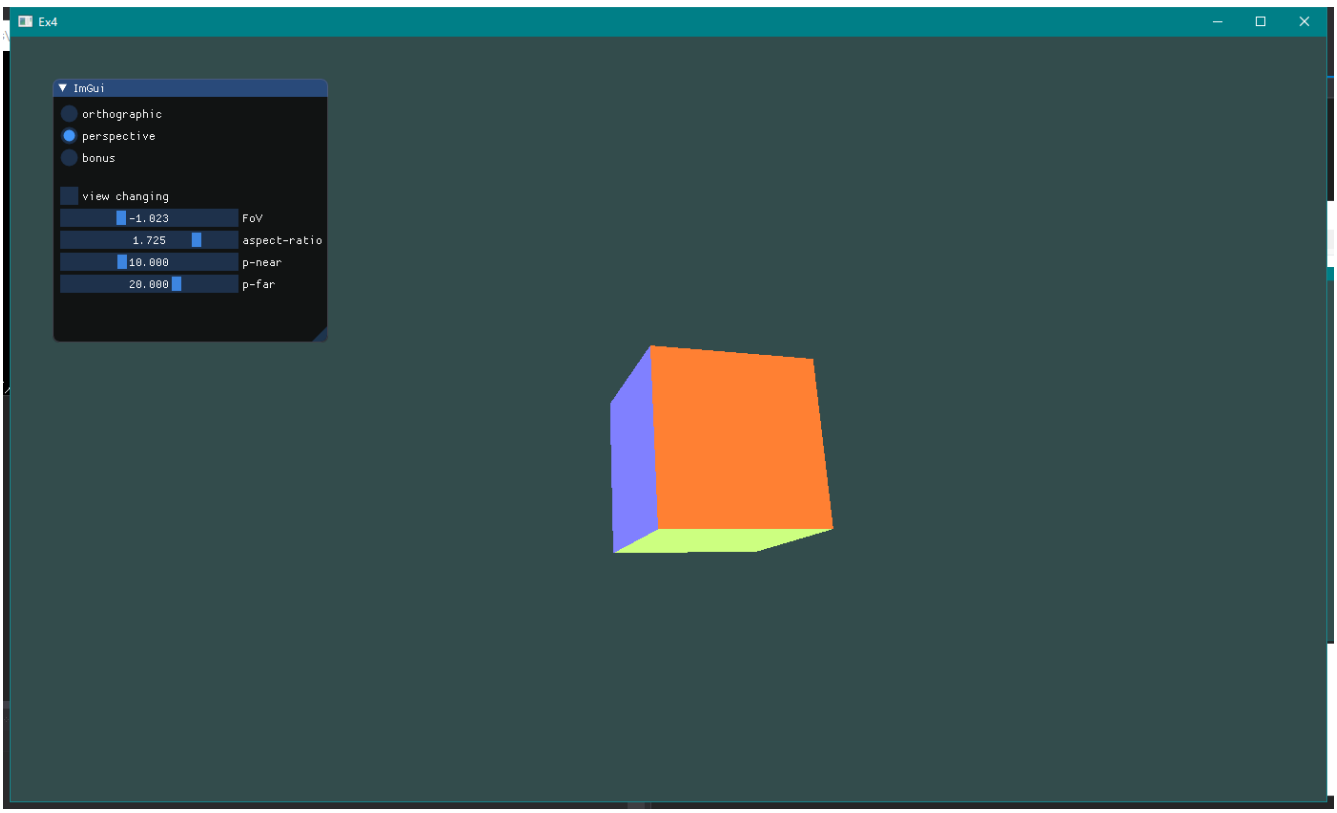
实现方法：

直接调用GLM的内置函数`glm::perspective`创建一个透视投影矩阵，将 cube 放置在 (-1.5f, 0.5f, -1.5f)的位置，适当调整view坐标使得 cube 更有立体效果

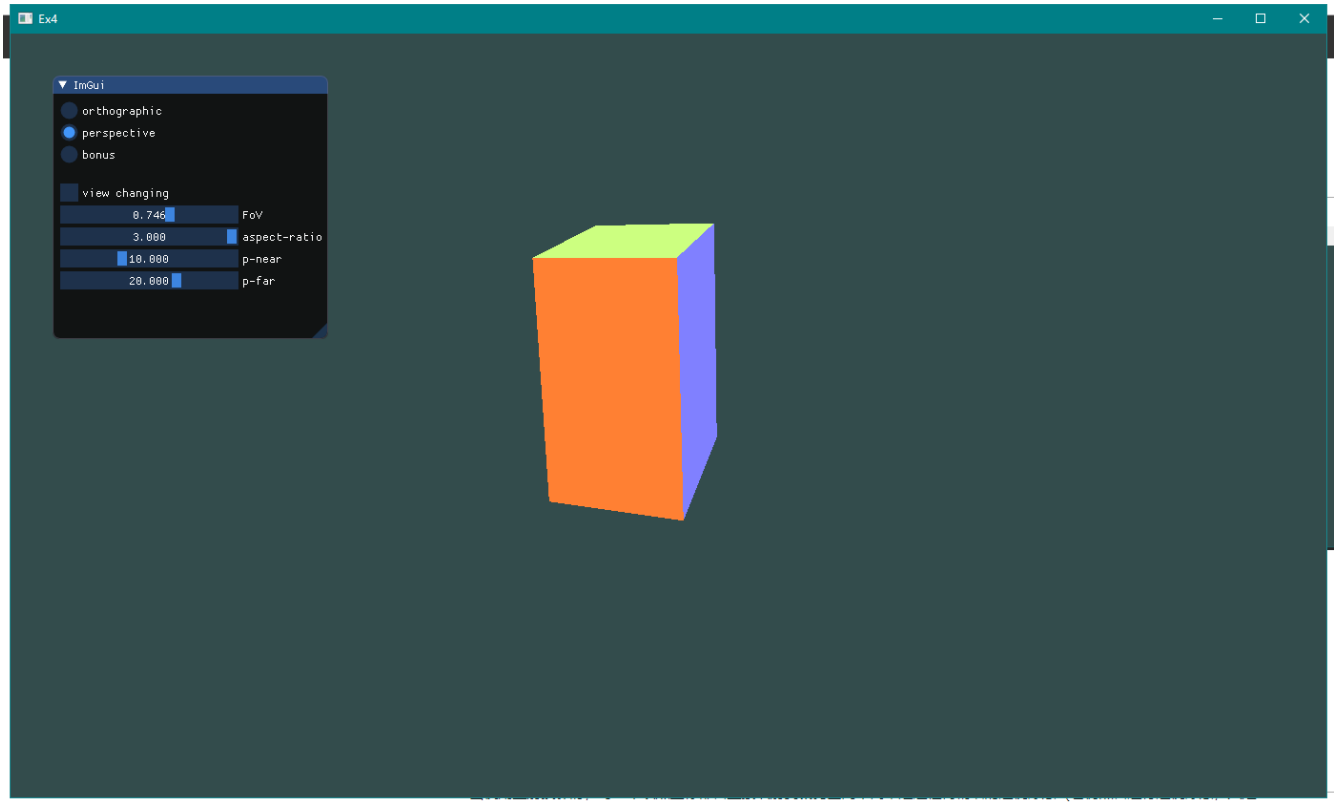
```
model = glm::translate(model, glm::vec3(-1.5f, 0.5f, -1.5f));  // 将 cube 放置在 (-1.5, 0.5, -1.5) 的位置
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -15.0f));
view = glm::rotate(view, glm::radians(25.0f), glm::vec3(1.0f, -1.0f, 0.0f));    // 调整观察视角
```



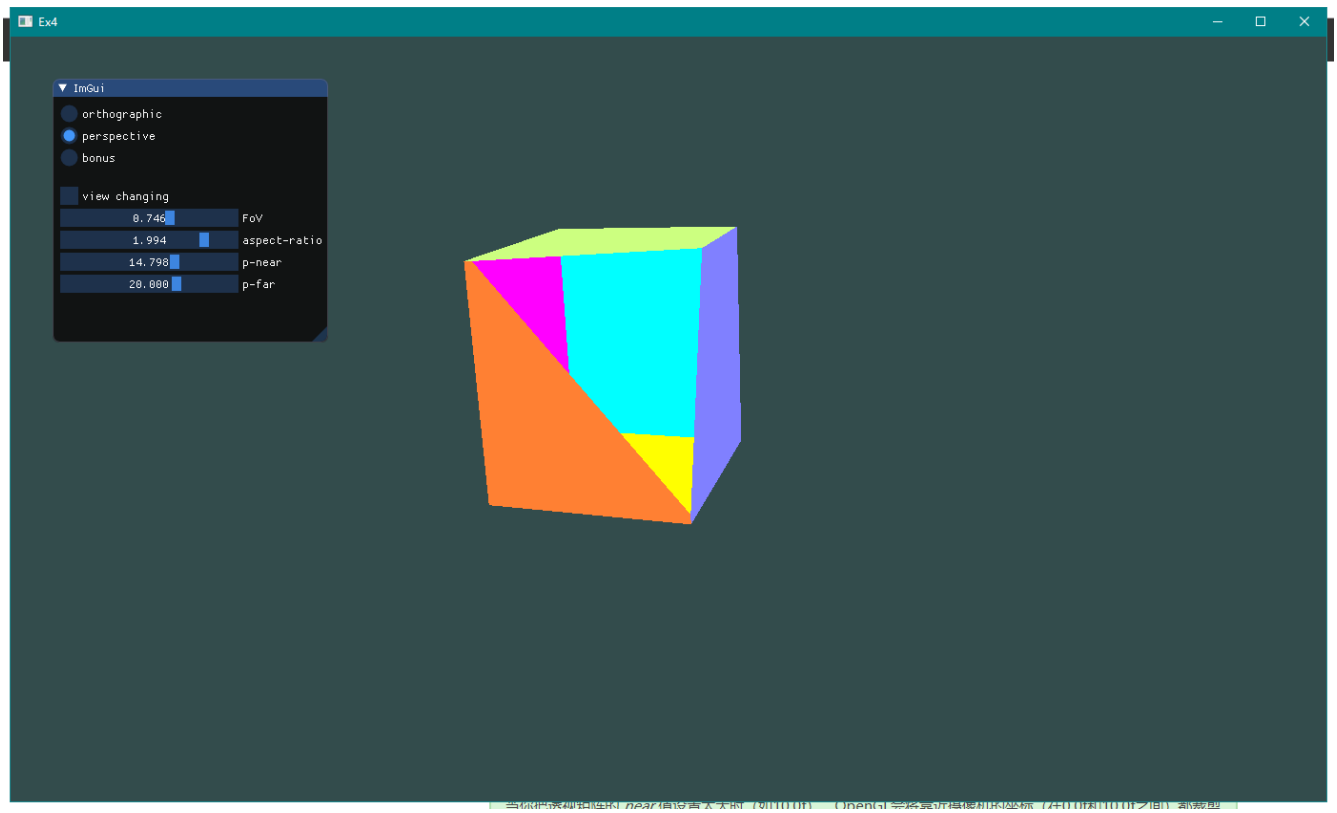
更改参数 FoV 改变视野，即观察空间的大小使得cube 倒转

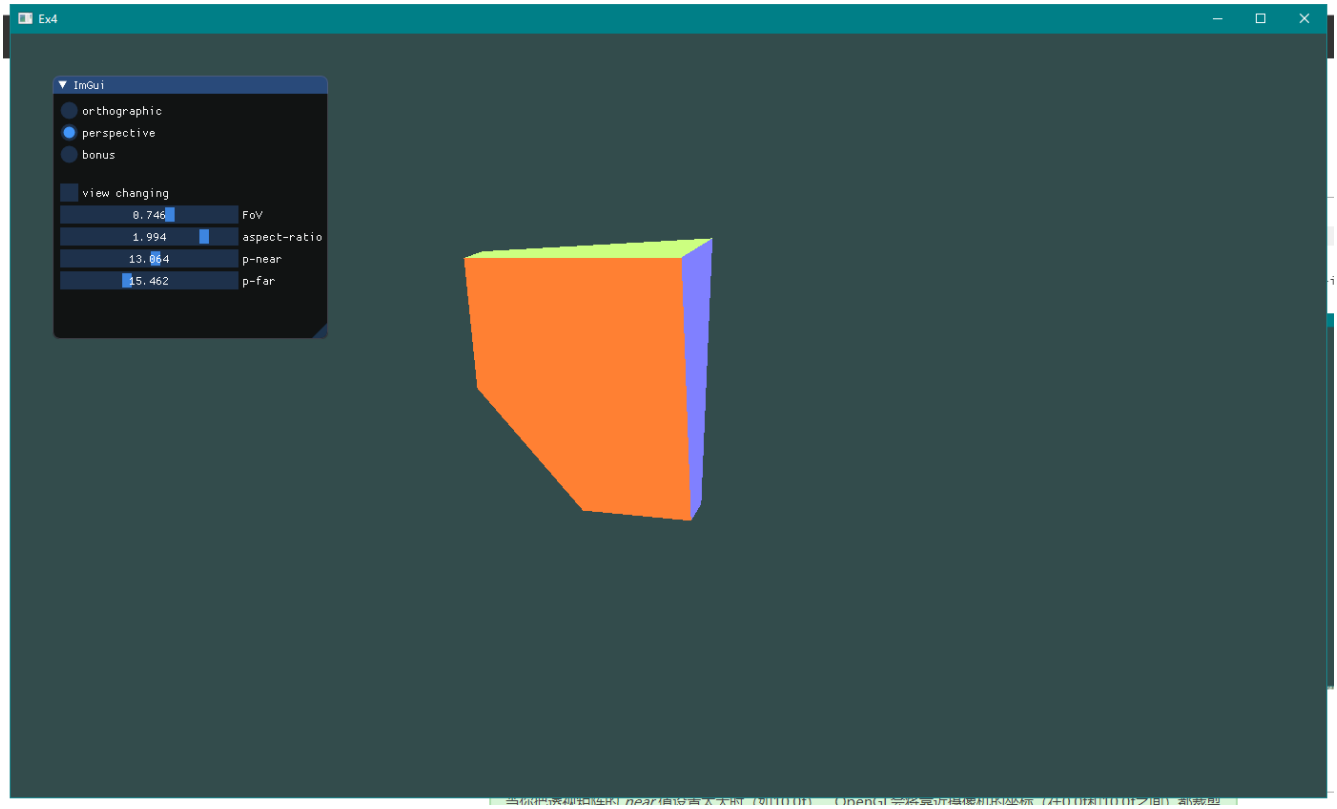


更改参数 aspect-ratio 使得 cube 宽高比改变



改变near 和 far，改变摄像机距离 cube 的距离以及最远能裁剪出来的范围，效果如下：





2.视角变换

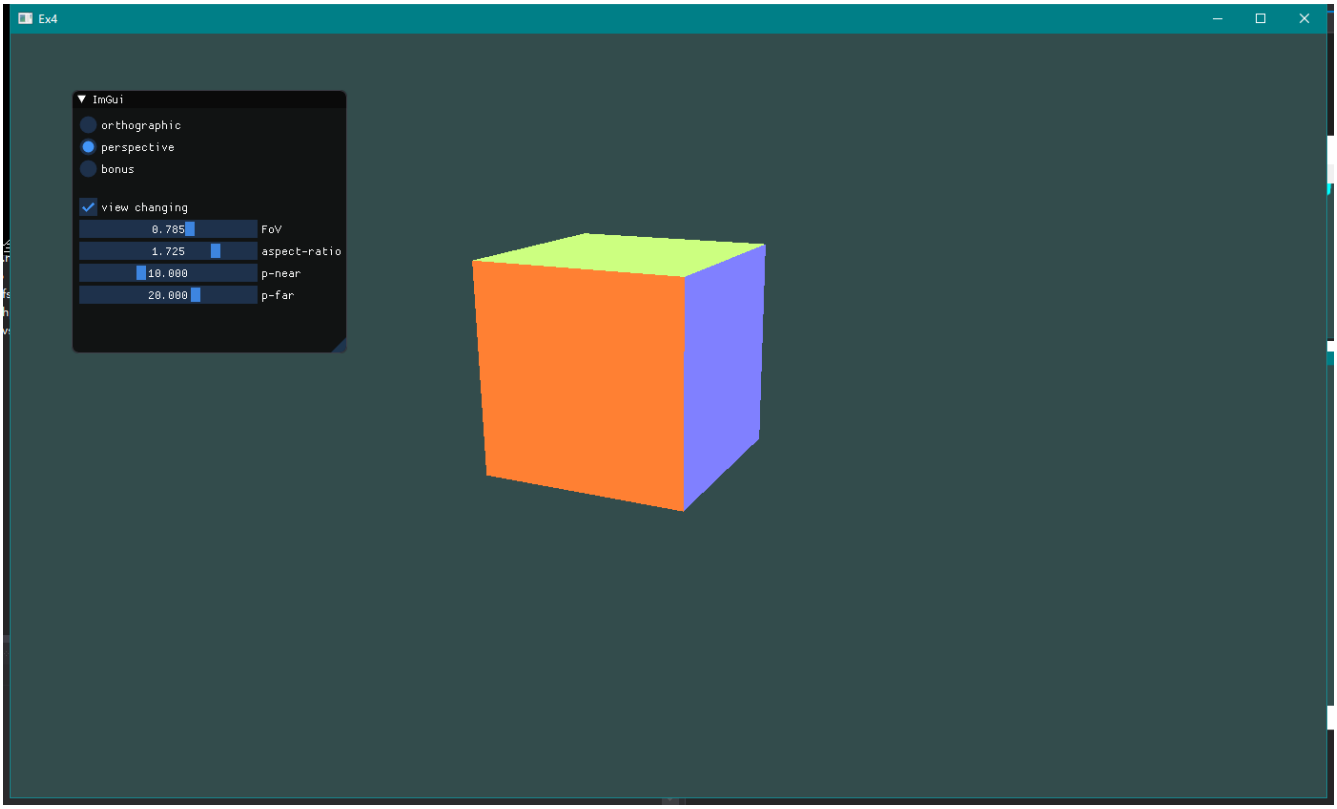
把cube放置在(0, 0, 0)处，做透视投影，使摄像机围绕cube旋转，并且时刻看着cube中心

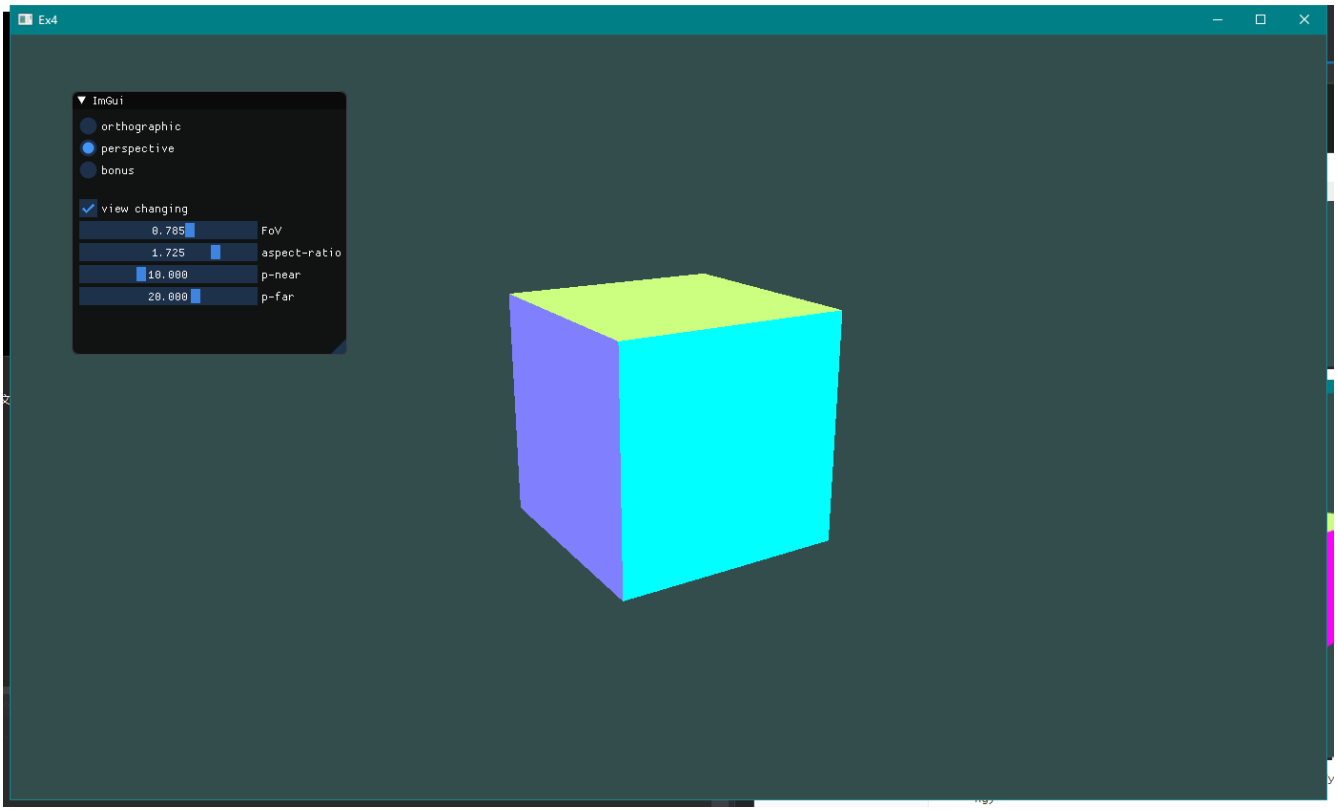
利用GLM的内置函数`lookAt`实现摄像机对准cube中心，根据时间计算摄像机所在圆上的位置，使得摄像机保持自动旋转，从而实现视角的变换

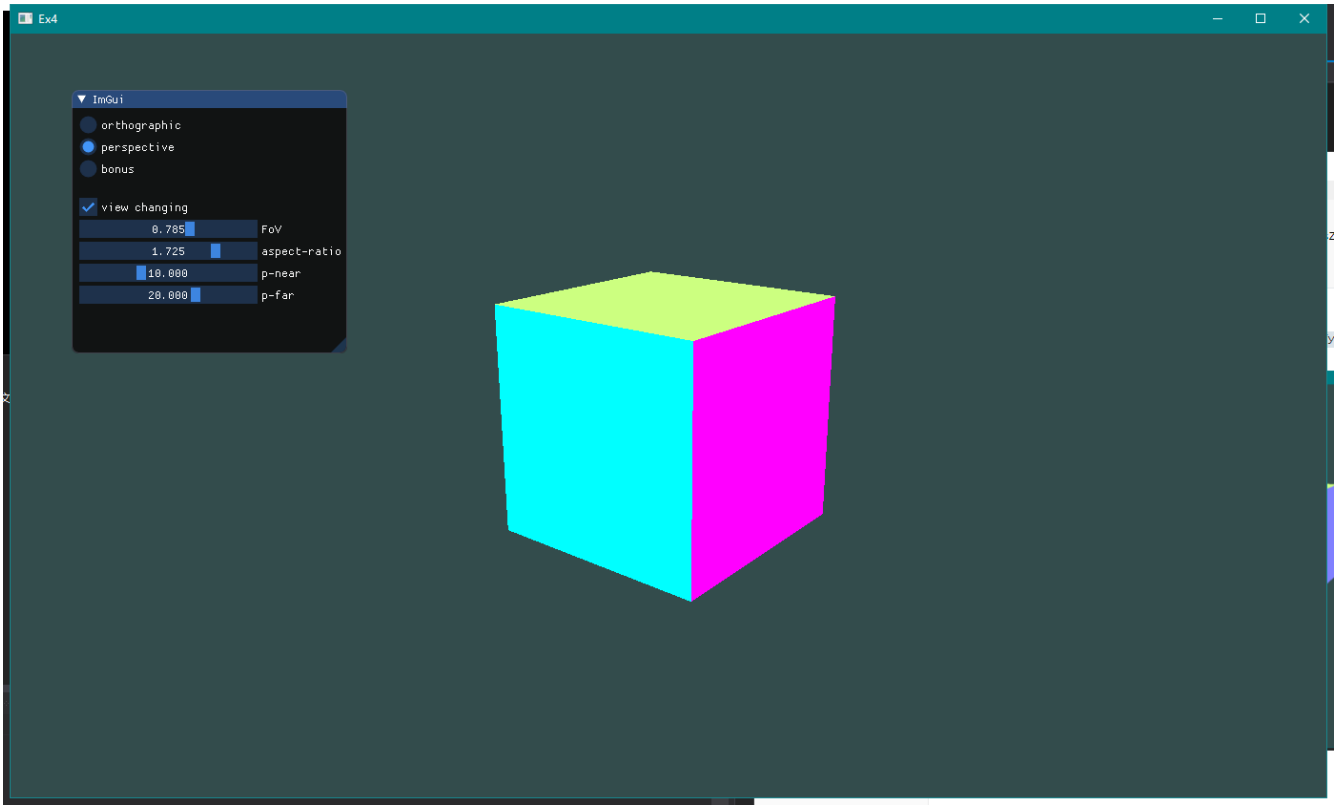
只需计算在 x 轴和 z轴上的坐标，使得视角围绕着 y 轴旋转

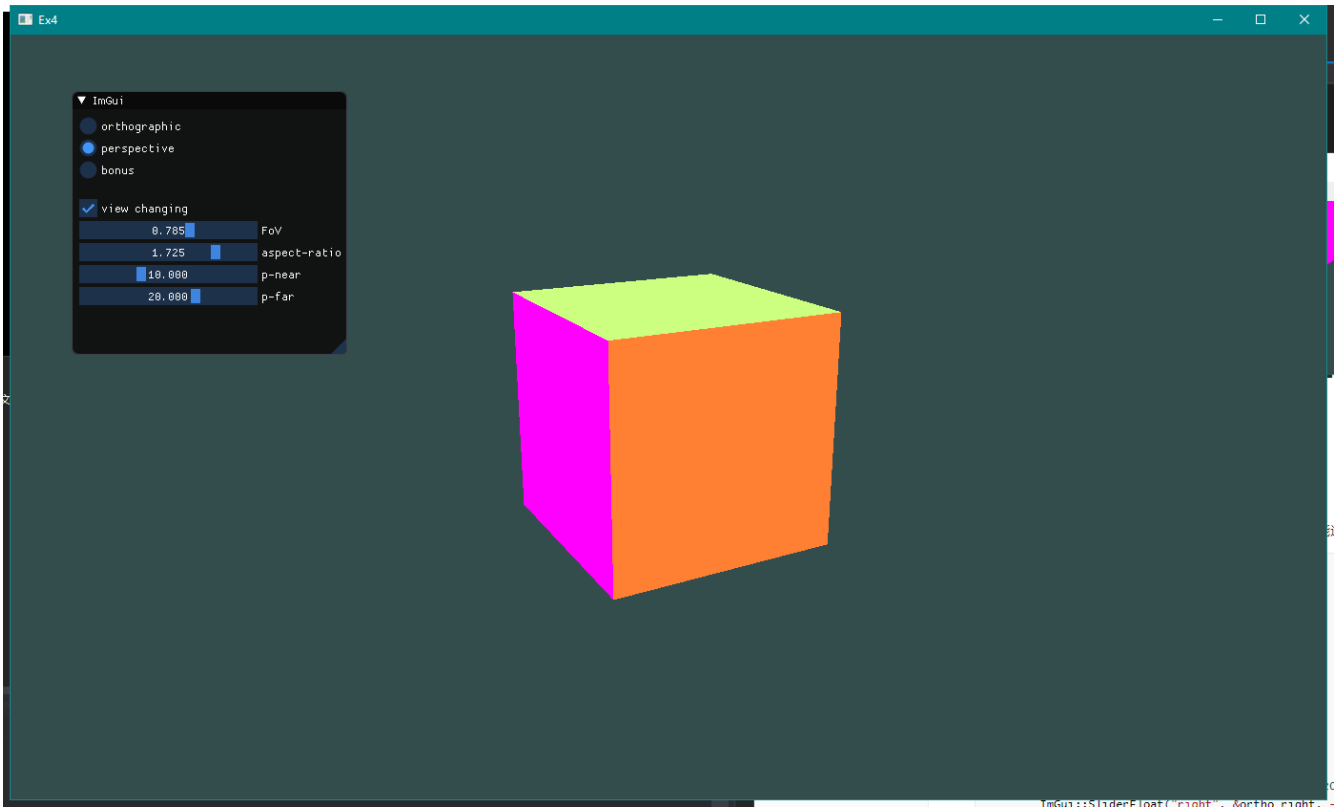
```
// 将 cube 放置在 (0.0, 0.0, 0.0) 的位置
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f));
// 初始化摄像机位置
float radius = 15.0f;
// 根据时间计算摄像机所在圆上的位置，使得摄像机保持自动旋转，从而实现视角的变换
float camPosX = sin(glfwGetTime()) * radius;
float camPosZ = cos(glfwGetTime()) * radius;
// 使摄像机围绕 cube 旋转，并且时刻看着 cube 中心
view = glm::lookAt(glm::vec3(camPosX, 5.0f, camPosZ), glm::vec3(0.0f, 0.0f, 0.0f),
glm::vec3(0.0f, 1.0f, 0.0f));
myShader.setMat4("view", view);
```

【动画效果见附件 gif 图】


Ex4
ImGui
orthographic
perspective
bonus
view changing
0.785
FoV
1.725
aspect-ratio
10.000
p-near
20.000
p-far


Ex4
ImGui
orthographic
perspective
bonus
view changing
0.785
FoV
1.725
aspect-ratio
10.000
p-near
20.000
p-far



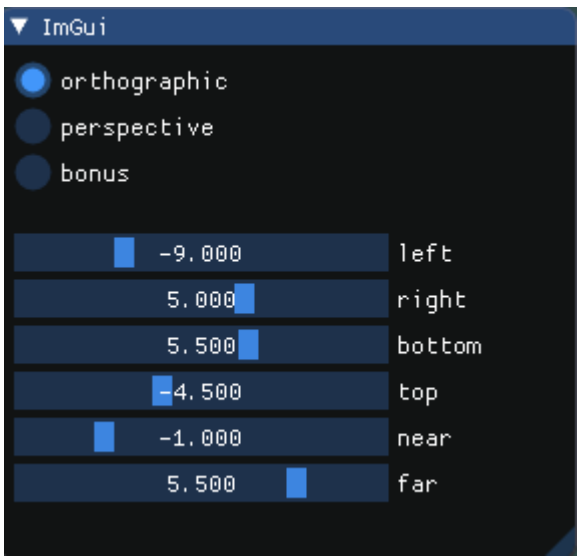


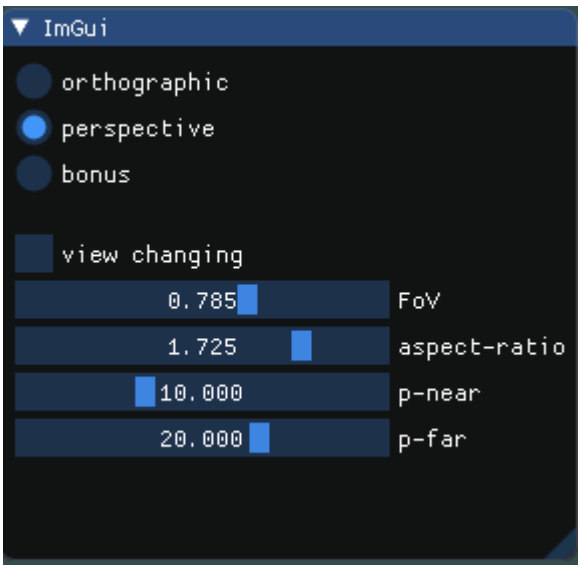
3.添加 ImGui

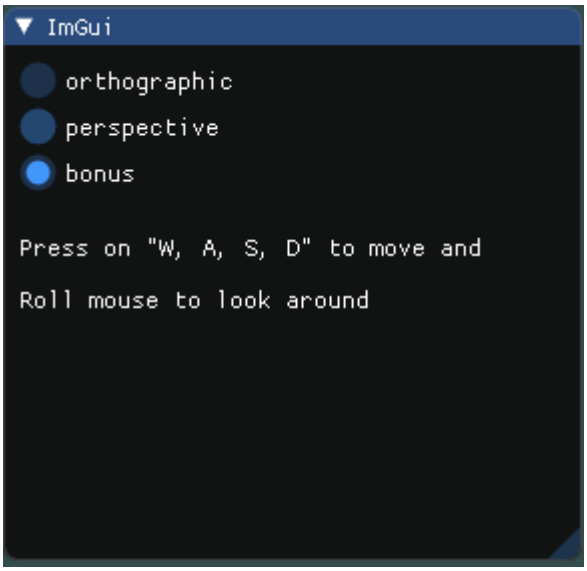
根据投影需要的参数定义相关变量并绑定到滑动条上即可，功能选择用一个变量标识实现单选 UI

```
// ImGui 菜单
{
    ImGui::Begin("ImGui");
    ImGui::StyleColorsDark();
    // 功能选择
    ImGui::RadioButton("orthographic", &mode, 0);
    ImGui::RadioButton("perspective", &mode, 1);
    ImGui::RadioButton("bonus", &mode, 2);
    ImGui::NewLine();
    // 正交投影选项
    if (mode == orthographic_projection) {
        // 各项参数
        ImGui::SliderFloat("left", &ortho_left, -20.0f, 20.0f);
        ImGui::SliderFloat("right", &ortho_right, -20.0f, 20.0f);
        ImGui::SliderFloat("bottom", &ortho_bottom, -20.0f, 20.0f);
        ImGui::SliderFloat("top", &ortho_top, -20.0f, 20.0f);
        ImGui::SliderFloat("near", &ortho_near, -3.0f, 6.0f);
        ImGui::SliderFloat("far", &ortho_far, -10.0f, 10.0f);
    }
    // 透视投影选项
    else if (mode == perspective_projection) {
        ImGui::Checkbox("view changing", &view_changing);   //  视角自动变换
        // 参数
        ImGui::SliderFloat("FoV", &FoV, -3.0f, 3.0f);
        ImGui::SliderFloat("aspect-ratio", &aspect_ratio, -3.0f, 3.0f);
        ImGui::SliderFloat("p-near", &p_near, 5.0f, 20.0f);
        ImGui::SliderFloat("p-far", &p_far, 10.0f, 25.0f);
    }
    // 加分项
    else if (mode == bonus) {
        ImGui::Text("Press on \"W, A, S, D\" to move and\n\nRoll mouse to look around ");
    }
    ImGui::End();
}
```

具体效果







4.View matrix, Model matrix 和 ModelView matrix

坐标变换矩阵栈（ModelView)是用来存储一系列的变换矩阵，栈顶就是当前坐标的变换矩阵，进入OpenGL管道的每个坐标(齐次坐标)都会先乘上这个矩阵，结果才是对应点在场景中的世界坐标。OpenGL中的坐标变换都是通过矩阵运算完成的。

OpenGL中的 modelview 矩阵变换是一个马尔科夫过程：上一次的变换结果对本次变换有影响，上次 modelview 变换后物体在世界坐标系下的位置是本次 modelview 变换的起点。默认时本次变换和上次变换不独立。OpenGL物体建模实际上是分两步走的。第一步，在世界坐标系的原点位置绘制出该物体；第二步，通过 modelview 变换矩阵对世界坐标系原点处的物体进行仿射变换，将该物体移动到世界坐标系的目标位置处。

5.视角自由移动【bonus】

实现一个摄像机类，需要的变量包括摄像机的位置坐标，往左右、前后、上下的距离变化变量，以及欧拉角

移动的距离通过速度来计算，用两个全局变量来计算按下按键持续的时间，再根据预设好的速度可以算出将要移动的距离

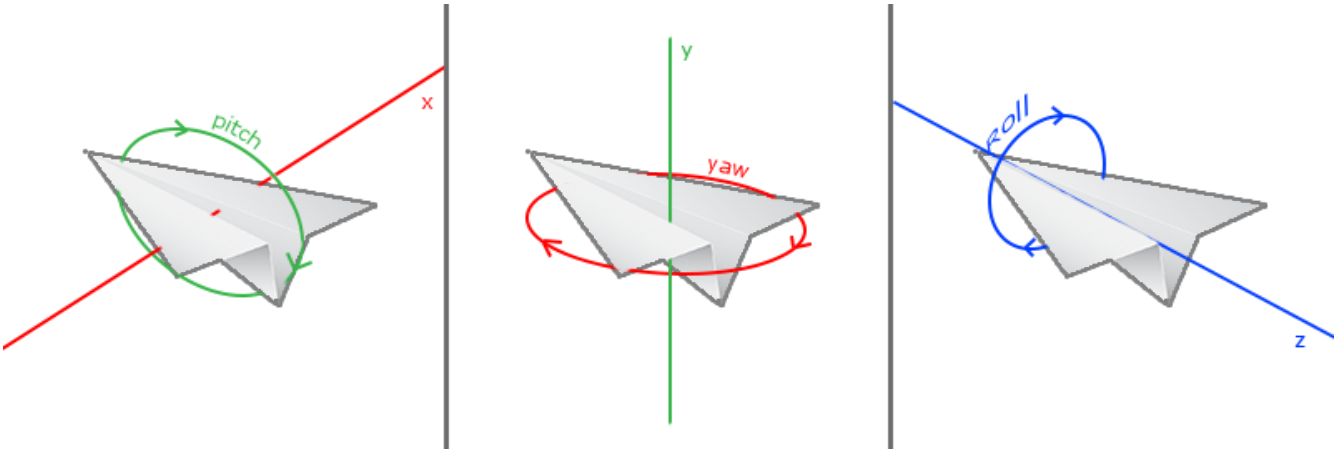
```
float cameraSpeed = 2.5f * deltaTime;
```

根据按下不同的键盘按键，在对应的方向上进行移动

```
void processInput(GLFWwindow *window) {
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.moveForward(deltaTime * SPEED);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.moveBack(deltaTime * SPEED);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.moveLeft(deltaTime * SPEED);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.moveRight(deltaTime * SPEED);
    if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS)
        camera.moveUp(deltaTime * SPEED);
    if (glfwGetKey(window, GLFW_KEY_LEFT_SHIFT) == GLFW_PRESS)
        camera.moveDown(deltaTime * SPEED);
}

void moveForward(GLfloat const distance) {
    Position += Front * distance;
}
void moveBack(GLfloat const distance) {
    Position -= Front * distance;
}
void moveRight(GLfloat const distance) {
    Position += Right * distance;
}
void moveLeft(GLfloat const distance) {
    Position -= Right * distance;
}
void moveUp(GLfloat const distance) {
    Position += Up * distance;
}
void moveDown(GLfloat const distance) {
    Position -= Up * distance;
}
```

欧拉角(Euler Angle)是可以表示3D空间中任何旋转的3个值，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：



俯仰角是描述如何往上或往下看的角，可以在第一张图中看到。第二张图展示了偏航角，偏航角表示往左和往右看的程度。滚转角代表如何翻滚摄像机。每个欧拉角都有一个值来表示，把三个角结合起来我们就能够计算3D空间中任何的旋转向量了。

对于我们的摄像机系统来说，只关心俯仰角和偏航角。给定一个俯仰角和偏航角，可以把它们转换为一个代表新的方向向量的3D向量。

```
direction.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw)); // 译注：direction代表摄像机的前轴(Front)，这个前轴是和本文第一幅图片的第二个摄像机的方向向量是相反的
direction.y = sin(glm::radians(pitch));
direction.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
```

可以把俯仰角和偏航角转化为用来自由旋转视角的摄像机的3维方向向量。

```
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;
    lastX = xpos;
    lastY = ypos;

    camera.rotate(xoffset * 0.25f, yoffset * 0.25f);
}

void rotate(GLfloat const yaw, GLfloat const pitch) {
        Yaw += yaw;
        Pitch += pitch;
        float radian_yaw = glm::radians(Yaw), radian_pitch = glm::radians(Pitch);
        // 计算摄像机欧拉角度的前向量
        glm::vec3 front = glm::vec3(cos(radian_yaw) * cos(radian_pitch), sin(radian_pitch),
sin(radian_yaw) * cos(radian_pitch));
        Front = glm::normalize(front);
        Right = glm::normalize(glm::cross(Front, WorldUp));
        Up = glm::normalize(glm::cross(Right, Front));
    }
```

【实现效果具体见附件 gif 图】