计算机图形学实验报告

学号：16340054

姓名：戴馨乐

学院：数据科学与计算机学院

作业：第五次作业

Basic：

1. **投影(Projection):**
   a) **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**

i. 实现思路

这里只需要修改各个面的颜色，然后在 model 矩阵加入一个平移就好了

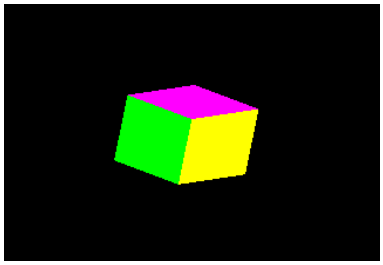
修改各个面的颜色

```
float vertices[] = {
// 后面
    -2.0f, -2.0f, -2.0f,  1.0f, 0.0f, 0.0f,
     2.0f, -2.0f, -2.0f,  1.0f, 0.0f, 0.0f,
     2.0f,  2.0f, -2.0f,  1.0f, 0.0f, 0.0f,
     2.0f,  2.0f, -2.0f,  1.0f, 0.0f, 0.0f,
    -2.0f,  2.0f, -2.0f,  1.0f, 0.0f, 0.0f,
    -2.0f, -2.0f, -2.0f,  1.0f, 0.0f, 0.0f,
// 前面
    -2.0f, -2.0f,  2.0f,  1.0f, 1.0f, 0.0f,
     2.0f, -2.0f,  2.0f,  1.0f, 1.0f, 0.0f,
     2.0f,  2.0f,  2.0f,  1.0f, 1.0f, 0.0f,
     2.0f,  2.0f,  2.0f,  1.0f, 1.0f, 0.0f,
    -2.0f,  2.0f,  2.0f,  1.0f, 1.0f, 0.0f,
    -2.0f, -2.0f,  2.0f,  1.0f, 1.0f, 0.0f,
// 左面
    -2.0f,  2.0f,  2.0f,  0.0f, 1.0f, 0.0f,
    -2.0f,  2.0f, -2.0f,  0.0f, 1.0f, 0.0f,
    -2.0f, -2.0f, -2.0f,  0.0f, 1.0f, 0.0f,
    -2.0f, -2.0f, -2.0f,  0.0f, 1.0f, 0.0f,
    -2.0f, -2.0f,  2.0f,  0.0f, 1.0f, 0.0f,
    -2.0f,  2.0f,  2.0f,  0.0f, 1.0f, 0.0f,
// 右面
     2.0f,  2.0f,  2.0f,  0.0f, 1.0f, 1.0f,
     2.0f,  2.0f, -2.0f,  0.0f, 1.0f, 1.0f,
     2.0f, -2.0f, -2.0f,  0.0f, 1.0f, 1.0f,
     2.0f, -2.0f, -2.0f,  0.0f, 1.0f, 1.0f,
     2.0f, -2.0f,  2.0f,  0.0f, 1.0f, 1.0f,
     2.0f,  2.0f,  2.0f,  0.0f, 1.0f, 1.0f,
```

移动正方体到(-1.5, 0.5, -1.5)位置

```
glm::mat4 modelview = glm::mat4(1.0f);
modelview = glm::rotate(modelview, glm::radians(50.0f), glm::vec3(0.5f, 1.0f, 0.0f));
modelview = glm::translate(modelview, glm::vec3(-0.5f, 0.5f, -1.5f));
```

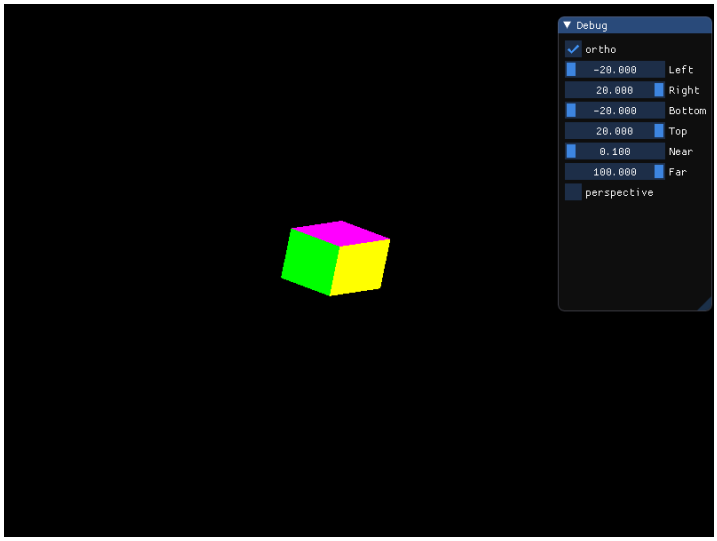
ii. 运行效果截图



b) 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

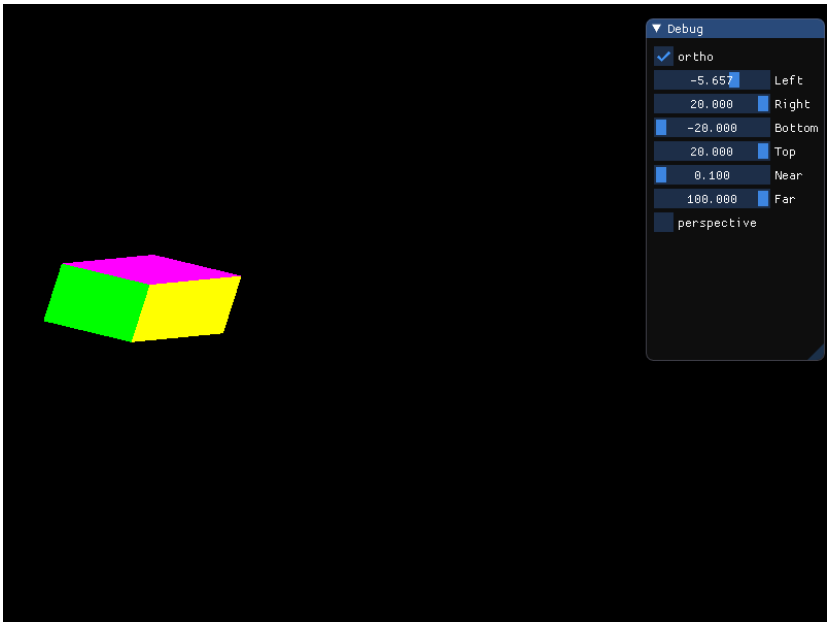
初始参数为：

$$\text{left} = -20.0\text{f}, \text{right} = 20.0\text{f}, \text{top} = -20.0\text{f}, \text{bottom} = 20.0\text{f}$$
$$\text{near} = 0.1\text{f}, \text{far} = 100.0\text{f}$$



left, right, top, bottom定义了投影平面，我们增大相应的参数，会有向相应方向拉伸的效果；反之，则有压缩的效果。

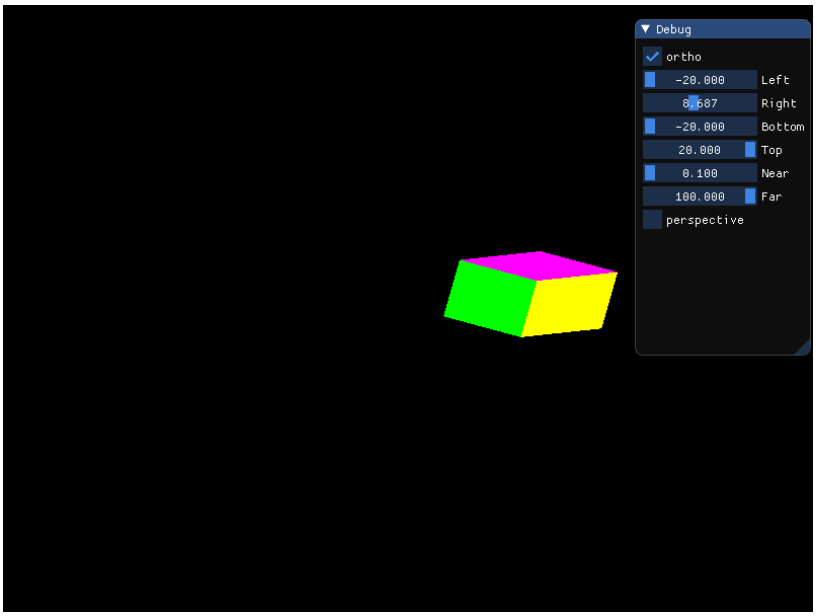
1. 变化left参数，这里增大left参数，将left变化为−5.675，如下：



这是因为我们增大了left参数，相当于说，定义的平截头体的左平面更加靠近我们的立方体。而且屏幕所展示的，就是平截头体之内的，所以，平截头体的左平面靠近立方体，而我们窗口的大小是不变的，看起来就是立方体更加靠近窗口的左边，就有了上面展示的，物体向左拉伸的效

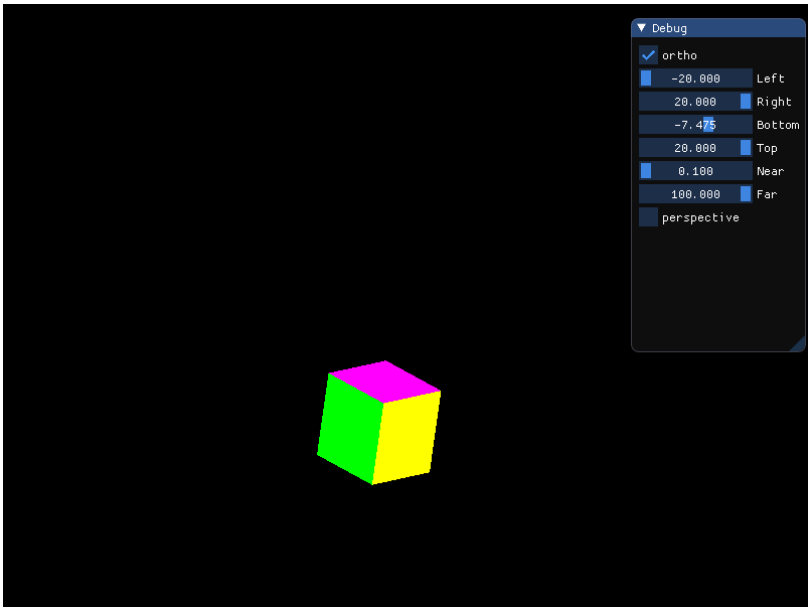
果了。

2. 变化right参数，这里缩小right参数，将right变化为8.687，如下：



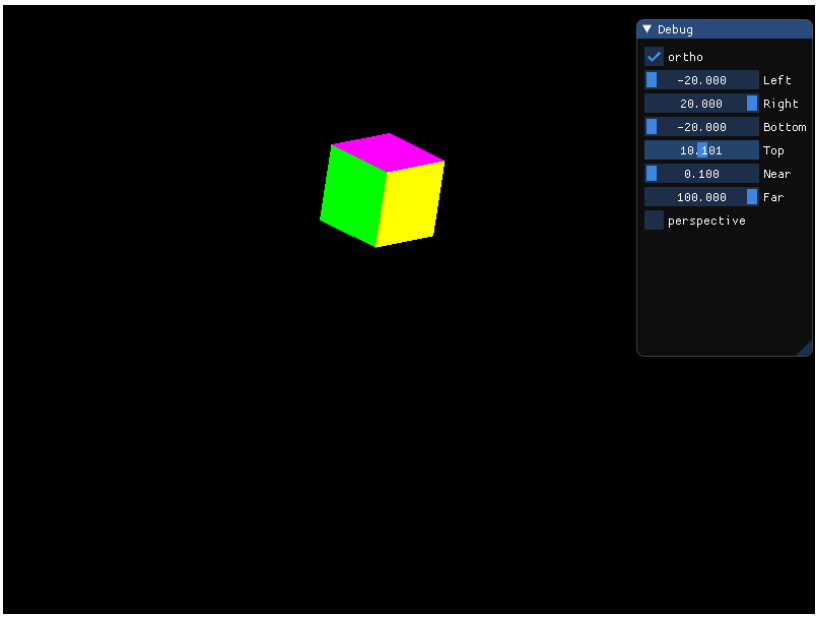
这个变化的原因和上述的left参数的变化的原因是一样的，right参数缩小，则右平面靠近立方体，所以形成了这种物体向右拉伸的效果。

3. 变化bottom参数，这里增大bottom参数为−7.475，如下：



可以看到立方体向下拉伸了，理由和上面是一样的。

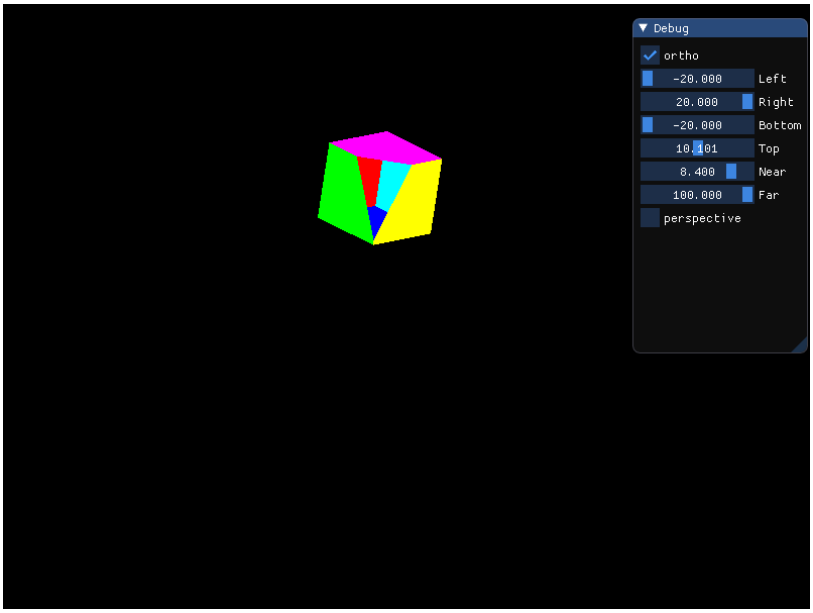
4. 变化top参数，将top参数缩小为10.101，如下：
可以看到立方体向上拉伸了，原因也和上面描述是类似的。



5. 变化near和far参数

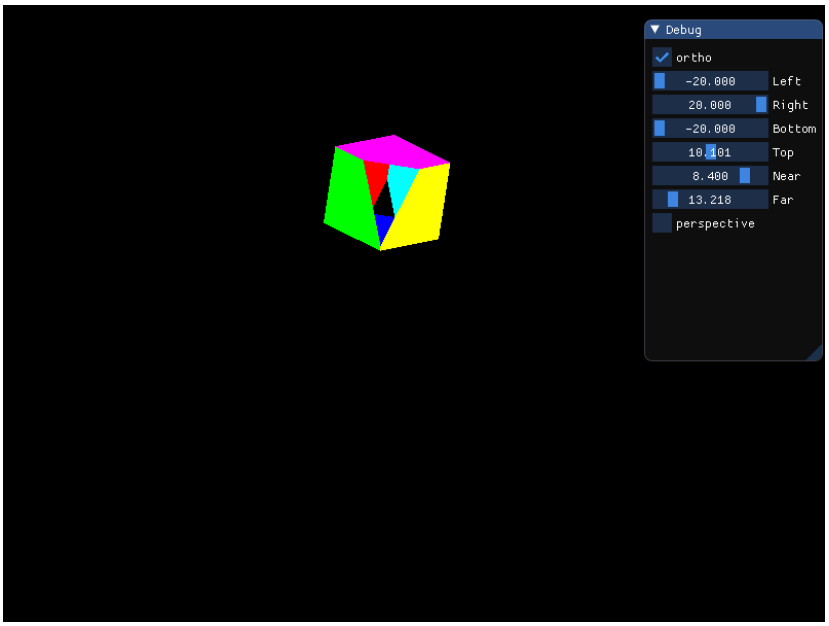
near参数代表了平截头体的近平面，far参数代表了平截头体的原平面

1) 这里我们先增大near参数，将near参数增大到8.400，如下：



可以看到，仿佛就是切开了前部的一个角，可以看得到内部后面的 3 个平面。这是因为增大了near参数，相当于将近平面后移。假如近平面后移过了立方体，那么立方体超出近平面的部分就会被丢弃掉，这是因为 OpenGL 仅仅只保留平截头体之内的部分，所以看起来就是被切掉了一个角这种效果。

2) 然后我们缩小far参数，将far参数缩小到了13.218，如下：



可以看到，在立方体的后部也被切掉了一个角。这是因为缩小far参数，相当于远平面向前移，当移动过了立方体，那么在远平面之外的部分就会被丢弃，就有了这种切掉了一个角的效果。

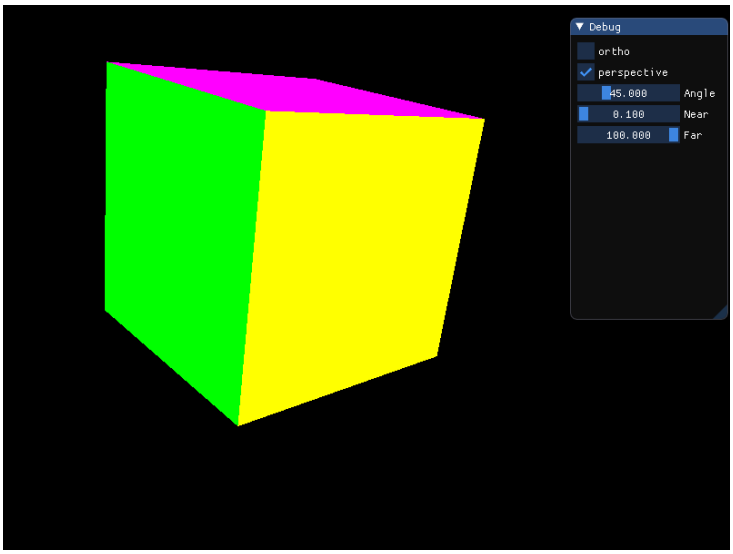
可以看到，正射投影生成的立方体，每个面看起来都是一样近的，和我们平时看到的立方体不一样，它没有透视，点不分远近，就有了上面这种效果。

c) 透视投影(perspective projection)：实现透视投影，使用多组参数，比较结果差异

初始参数为：

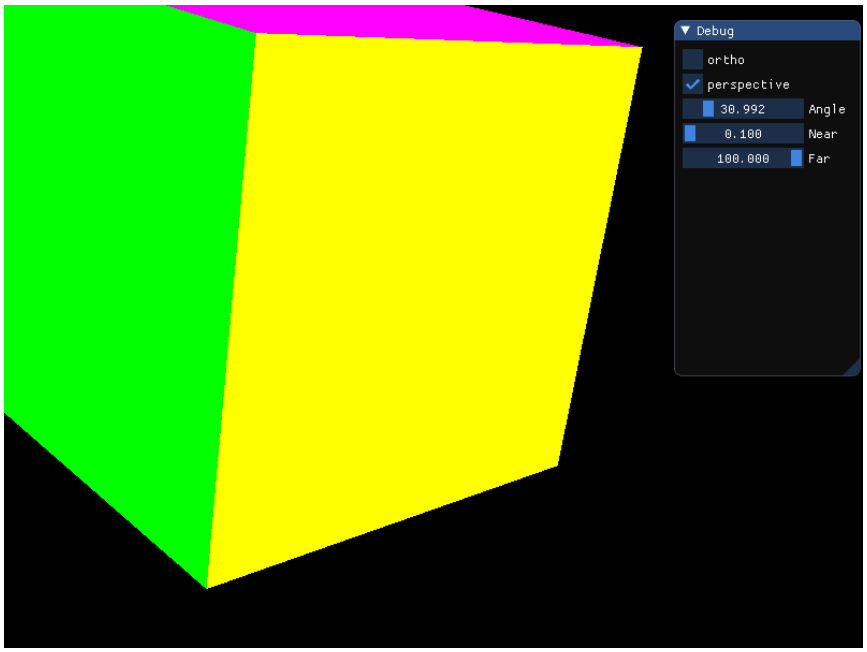
$$\text{fov} = 45.0\text{f}, \text{ratio} = \frac{\text{width}}{\text{height}}, \text{near} = 0.1\text{f}, \text{far} = 100.0\text{f}$$

效果如下：



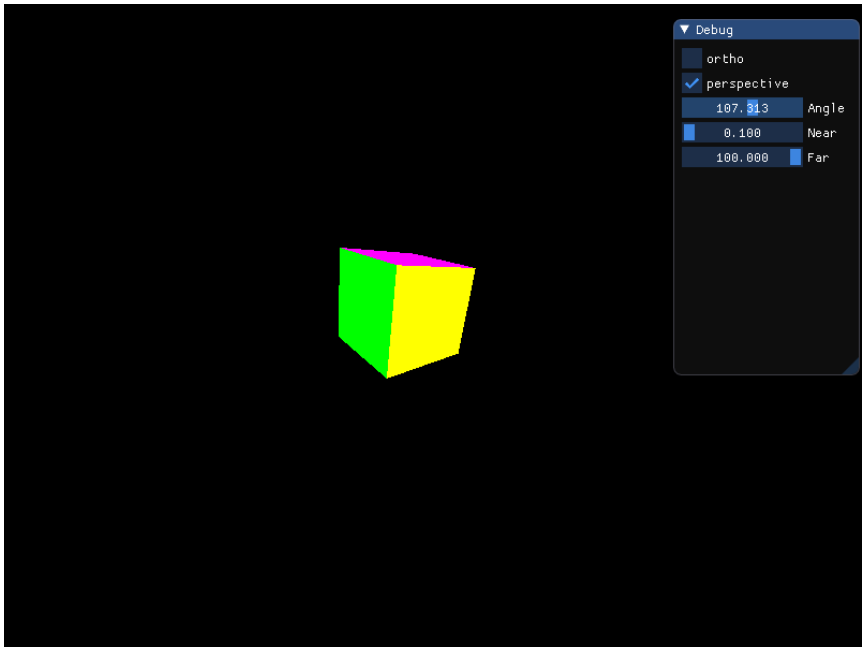
fov，field of view，表示视野，fov越大，则表示视野越大，那么物体看起来就会变小，反之则会变大。ratio则是宽高比，由窗口的高和宽的比组成。near和far代表近平面和远平面，和正射投影是一样的。

1. 变化fov参数
   a) 将fov变大，将fov变化为30.992，如下：



将fov缩小后，视野变小了，物体离观察则近了，物体看起来更大了

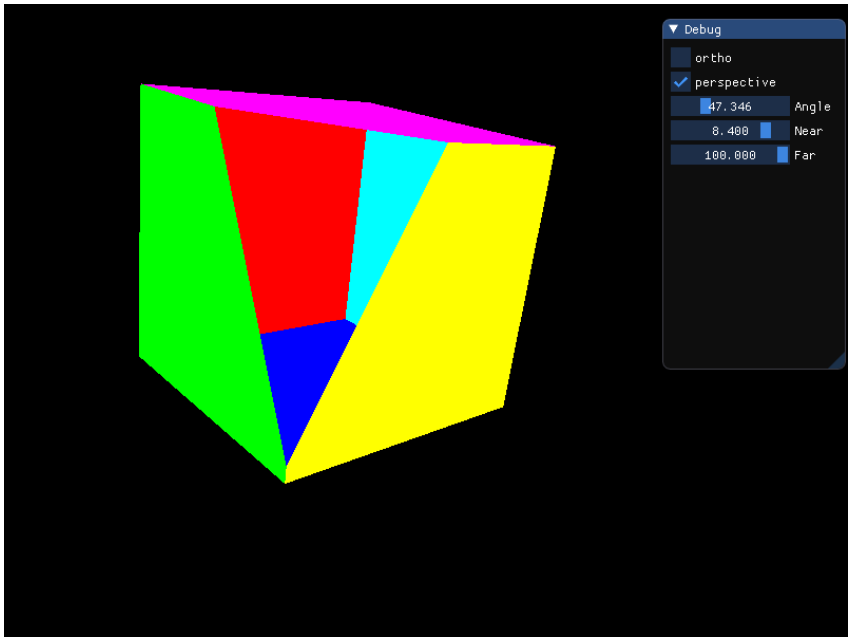
   b) 将fov变小，将fov增大为 187.313，如下：



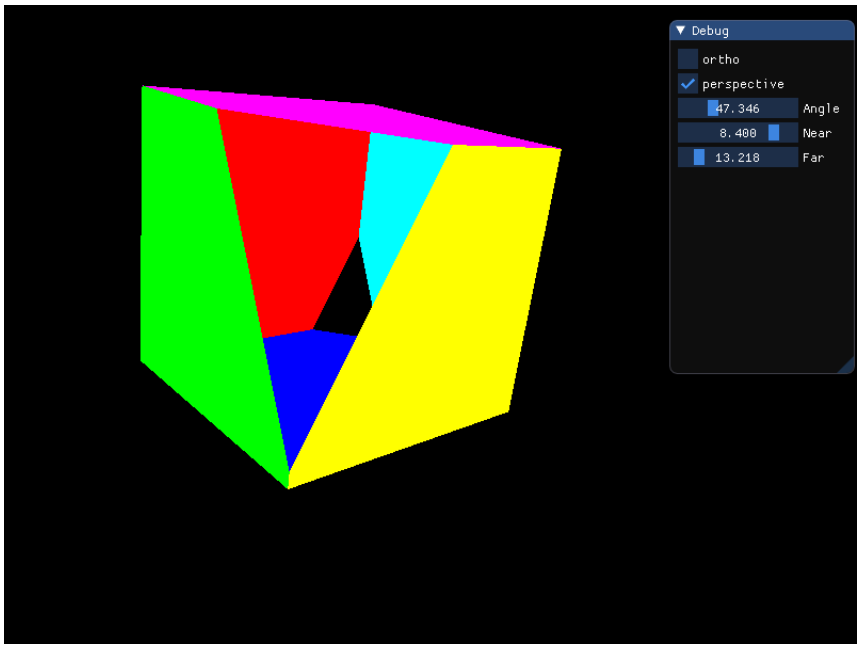
将fov增大之后，视野变大了，物体看起来变小了

2. 变化near和far参数

a) 增大near参数，将near增大到8.400，如下：



b) 缩小far参数，将far缩小到13.218，如下：



可以看到，这个效果和上面的正射投影是一样的，原因也是一样的。这里看得到，应用了透视的透视投影，立方体看起来和我们看到的一样了，因为各个点有了远近之分，远的点看起来小，近的点看起来大，更加符合我们视觉看到的。

2. **视角变换(View Changing):**

   a) **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**

摄像机，其实就是我们的观察者角度。为了定义摄像机，首先需要使用一个变化矩阵，可以用这个矩阵乘以任意的坐标向量就可以转换到观察空间。首先我们可以先定义摄像机的位置cameraPos和摄像机观察的目标cameraTarget，然后由cameraPos − cameraTarget标准化得到的向量，就可以得到了摄像机坐标的z轴；为了得到x轴，先定义一个上向量Up(0.0, 1.0,0.0)，这个和z平面一起形成的平面属于摄像机坐标的oyz平面，然后得到正交于这两个向量的标准化向量，就得到了摄像机坐标的x坐标；最后，正交于z轴和x轴的向量，就是x轴。最后加上平移向量，就得到了需要的LookAt矩阵。可以看到生成LookAt矩阵，需要 3 个向量:

1. cameraPos向量，摄像机的位置
2. cameraTarget向量，摄像机的目标
3. Up上向量

glm库提供了相关的函数:

```
view = glm::lookAt(glm::vec3(camX, 0.0f, camZ), glm::vec3(0.0f, 0.0f, -10.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

然后，我们需要摄像机沿着一个以立方体的中心$(0.0f, 0.0f, -10.0f)$，在xOz平面，以一定半径地移动。我们记立方体地中心位置为(C_x, C_y, C_z)。

对于摄像机的移动位置，我们用以下公式:

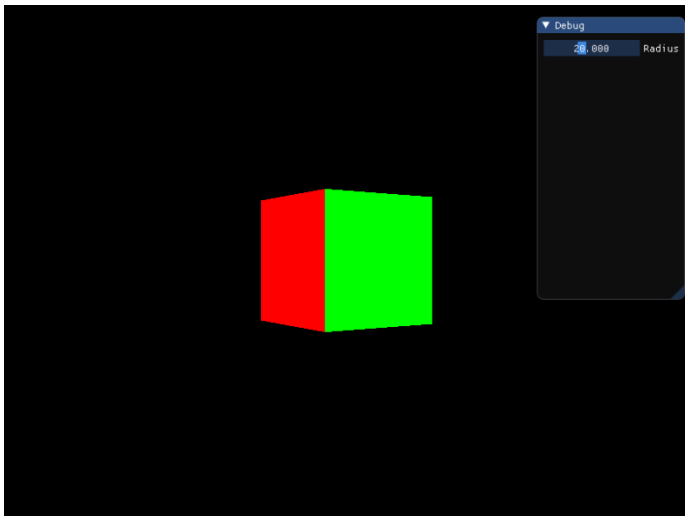
$$camX = \sin(\theta) * radius$$

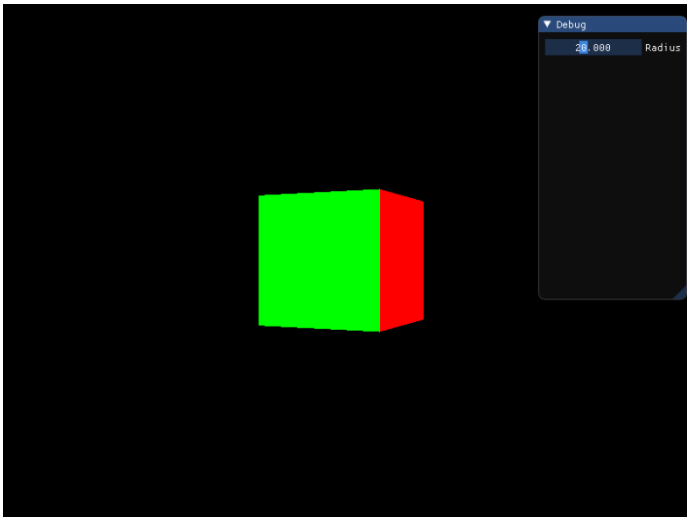
$$camZ = \cos(\theta) * radius$$

然后需要对θ进行改变，随着时间进行变化，所以θ的值，是应该和电脑的时钟得到的，在GLFW中，为glfwGetTime()。

```
float camX = sin(glfwGetTime()) * radius;
float camZ = cos(glfwGetTime()) * radius - 10.0f;
```

效果截图:





3. **在 GUI 里添加菜单栏，可以选择各种功能**
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？**

   Model 矩阵表示物体的摆放位置，View 矩阵表示摄像机的观察位置，在固定管线的 OpenGL 中，将两个矩阵合起来变成一个 ModelView 矩阵，因为视图其实可以看作物体的摆放改变，视图的改变是物体变换的逆方向变换的，所以将其合二为一。另外，假如存在另外一个摄像机，那么 View 矩阵可以用于新的摄像机，实现多视角观察。

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**

   首先，对于摄像机的移动，可以封装起来。摄像机主要有以下几种移动：
   1. 上下左右移动
   2. 视角上下左右变化
   3. 对目标物体使用滚轮缩放

   然后，摄像机的移动，其实就是改变上面说的 3 个向量cameraPos，cameraTarget和Up。其中，Up向量是固定的，就是$(0.0,1.0,0.0)$，然后摄像机的位置cameraPos之后是可以改变的，所以初始化的值不重要，放在要看的立方体前方比较人性化，这里初始化为$(0.0,0.0,3.0)$。这里摄像机的目标，并不是固定的了，是随着摄像机的移动而变化的，但是以人为例，摄像机看向的目标，就是人的正前方，相对于人的正前方，我们用这样一个移动向量cameraFront来表示，初始化为$(0.0,0.0,-1.0)$。

   然后可以大致定义出下面这样一个Camera类：

```
class Camera {
public:
    void moveForward();
    void moveBack();
    void moveLeft();
    void moveRight();
    void scroll(double xoffset, double yoffset);
    void moveMouse(double xpos, double ypos);
    void updateSpeed();
    glm::vec3 getCameraPos() { return cameraPos; }
    glm::vec3 getCameraTarget() { return cameraPos + cameraFront; }
    glm::vec3 getCameraUp() { return cameraUp; }
    float getfov() { return fov; }
    Camera(float width, float height)
    {
        // move
        cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
        cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
        cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
        // speed
        lastFrame = 0.0f;
        // mouse
        yaw = -90.0f;
        pitch = 0.0f;
        lastX = width / 2;
        lastY = height / 2;
        firstMouse = true;
        // scroll
        fov = 45.0f;
    }
private:
    glm::vec3 cameraPos, cameraFront, cameraUp;
    float lastX, lastY;
    bool firstMouse;
    float lastFrame;
    float cameraSpeed;
    float yaw, pitch;
    float fov;
};
```

另外，其余变量是后续几个移动函数用到的。

1. 上下左右移动

   我们先定义一个摄像机的速度CameraSpeed。为了使得摄像机在各个系统下，移动速度都一样，我们通过上下帧的时间差来求出移动速度。记上一帧为lastFrame，当前帧为currentFrame，那么可以得到Δtime = currentFrame − lastFrame，最后让Δtime乘上一个常量C，就得到了移动的速度了。这里我定义这个常量 C 为2.5f。

```
void Camera::updateSpeed()
{
    float currentFrame = glfwGetTime();
    float deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;
    cameraSpeed = 4.0f * deltaTime;
}
```

   a) 向上移动摄像机

   向上移动，就是让摄像机的位置cameraPos加上移动的距离，记移动速度CameraSpeed乘上移动向量CameraFront

```
void Camera::moveForward()
{
    cameraPos += cameraSpeed * cameraFront;
}
```

   b) 向下移动计算机

   向下移动，和向上移动类似，只不过是减去移动的距离表示后退

```
void Camera::moveBack()
{
    cameraPos -= cameraSpeed * cameraFront;
}
```

   c) 向左移动计算机

   向左移动的移动向量，可以通过求向前的移动向量CameraFront和Up向量的正交向量来求得。同样，也是求出移动的距离之后，摄像机的位置就减去移动的距离。

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

   d) 向右移动计算机

   向右移动和向左移动一样，只不过是加上移动的距离。

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

   效果如下：

2. 移动摄像机的视角

可以将视角看作一个飞机，可以上下抬头，这就是**俯仰角**；可以左右摆动，这就是**偏航角**；可以翻滚，这就是**滚转角**。目前只需要考虑的是俯仰角pitch和偏航角yaw。

通过三角函数推导，计算方向向量可以通过下面的公式得到：

$$direction.x = \cos(pitch) * \cos(yaw)$$

$$direction.y = \sin(pitch)$$

$$direction.z = \cos(pitch) * \sin(yaw)$$

为了计算俯仰角和偏航角，我们需要记录：

1. 当前鼠标的位置xpos, ypos
2. 上一次鼠标的位置lastX, lastY

然后计算x, y方向的偏移：

$$xoffset = xpos - lastX$$

$$yoffset = lastY - ypos$$

为了使得移动不要过于敏感，这个偏移量常常需要乘以一个常数sensity

然后，俯仰角yaw是在xOz平面左右移动，变化的是x，所以yaw+= xoffset，偏航角pitch在yOz平面上下移动，变化的是y，所以pitch+= yoffset

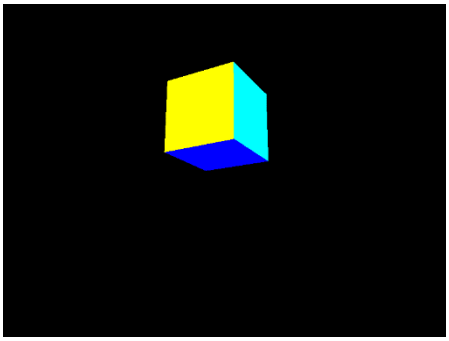
最后使用计算得到的yaw和pitch来计算方向向量就可以了，计算完成后，将这个新的方向向量赋值给原先的移动向量cameraFront，这样就改变了视角的方向。

```
void Camera::moveMouse(double xpos, double ypos)
{
    if (firstMouse) {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }
    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos;
    lastX = xpos;
    lastY = ypos;

    float sensitivity = 0.05f;
    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw += xoffset;
    pitch += yoffset;
    if (pitch > 180.0f)
        pitch = 180.0f;
    if (pitch < -180.0f)
        pitch = -180.0f;
    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));
    cameraFront = glm::normalize(front);
}
```

效果如下：

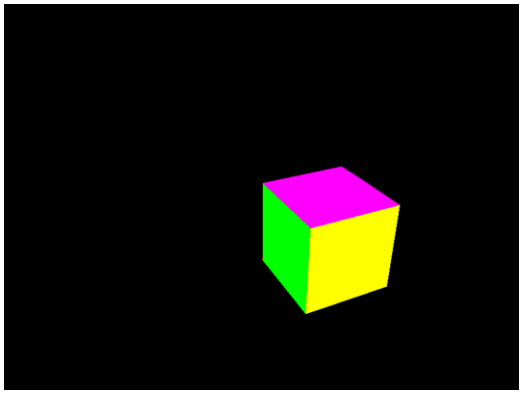
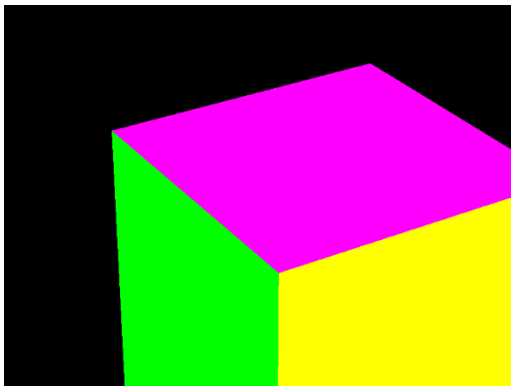


3. 缩放

   缩放调整的是投影矩阵的视野fov，缩小即fov增大，视野变大；放大即fov减小，视野变小。

```
void Camera::scroll(double xoffset, double yoffset)
{
    if (fov >= 1.0f && fov <= 45.0f) fov -= yoffset;
    if (fov <= 1.0f) fov = 1.0f;
    if (fov >= 45.0f) fov = 45.0f;
}
```

   效果如下：

当然，上面仅仅只是实现了Camera类，还需要应用在程序中才可以有上面的效果

1. 首先，新建一个Camera类

```
Camera camera(width, height);
```

   然后，对于上下左右移动，是通过输入 wsad 来进行移动的，那么，需要 OpenGL 来监听来自键盘的输入，然后根据这 4 个键是否按下，来决定相应的移动。

   监听函数：

```
void process_input(GLFWwindow* window) {
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
        glfwSetWindowShouldClose(window, true);
    }
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
        camera.moveForward();
}
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
        camera.moveBack();
    }
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) {
        camera.moveLeft();
    }
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) {
        camera.moveRight();
    }
}
```

在渲染循环中，不断调用该函数实现监听

```
while (!glfwWindowShouldClose(window)) {
    process_input(window);
    glfwSetCursorPosCallback(window, mouse_callb
```

2. 对于鼠标移动的监听函数以及该函数的注册
   对于鼠标的监听函数：

```
void mouse_callback(GLFWwindow * window, double xpos, double ypos)
{
    camera.moveMouse(xpos, ypos);
}
```

对该函数的注册：

```
process_input(window);
glfwSetCursorPosCallback(window, mouse_callback);
glfwSetScrollCallback(window, scroll_callback);
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DI
```

为了实现更加真实的 FPS 效果，可以将窗口模式设置为鼠标隐藏模式。

```
glfwSetScrollCallback(window, scroll_callback);
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
```

3. 滚轮来控制缩放的监听函数和注册函数也是一样的

```
void scroll_callback(GLFWwindow * window, double xoffset, double yoffset)
{
    camera.scroll(xoffset, yoffset);
}
```

```
process_input(window);
glfwSetCursorPosCallback(window, mouse_callback);
glfwSetScrollCallback(window, scroll_callback);
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
```

4. 这里的确是修改了摄像机的相关向量，但是还需要让观察矩阵和投影矩阵来使用，才可以达到修改了摄像机和物体的效果。

```
// view矩阵(camera)
glm::mat4 view = glm::mat4(1.0f);
view = glm::lookAt(camera.getCameraPos(), camera.getCameraTarget(), camera.getCameraUp());
// 投影矩阵
glm::mat4 projection = glm::mat4(1.0f);
projection = glm::perspective(glm::radians(camera.getfov()), float(width) / height, 0.1f, 100.0f);
```