计算机图形学 Homework5

15331416 赵寒旭

1. 运行结果

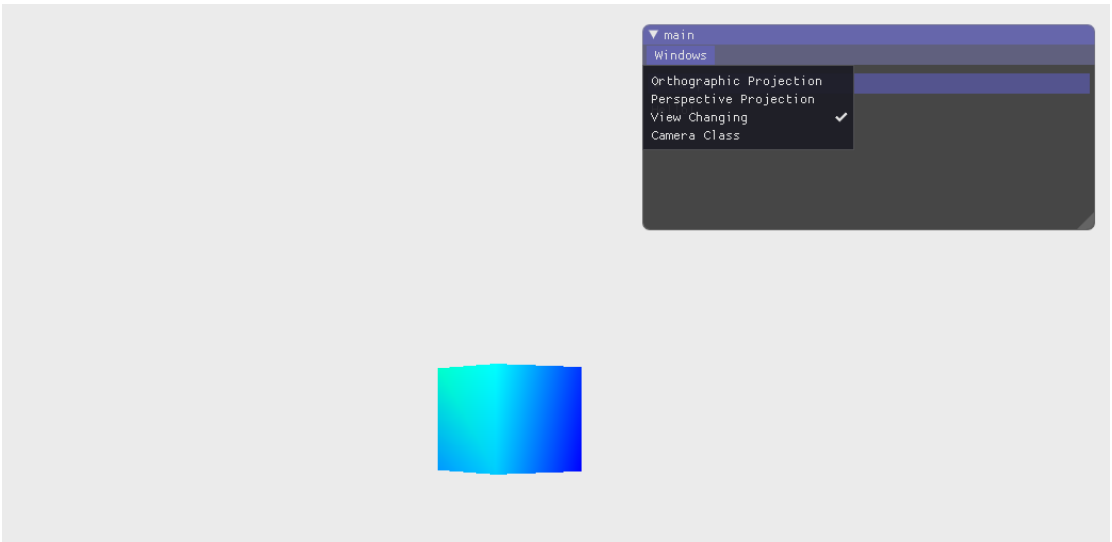
1）正交投影

参数选择通过移动滑块自行调整。

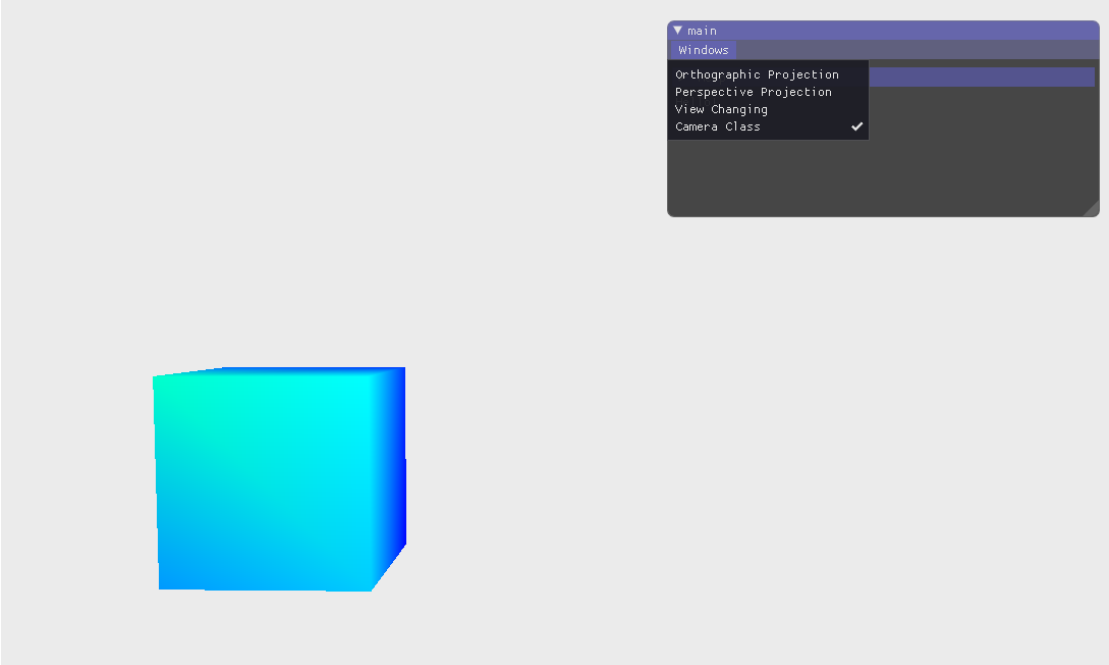
2）透视投影

参数选择通过移动滑块自行调整。

3）视角变换



4）Camera 类



另有视频演示在/doc 文件夹下。

2. 问题回答

问：在显示生活中，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？（你可能有不止一个摄像机）

答：我们获得物体最终的屏幕坐标是靠几次坐标变换得到的，从物体的局部坐标开始，先通过 Model matrix 将其转换到世界坐标，再由 View matrix 转换到观察空间坐标，考虑有多个摄像机的情况，如果我们将这两次操作合并为一个变换矩阵 ModelView matrix，可以减少计算量，同时获得和分步操作完全相同的结果。

3. 实现思路

3.1 投影

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。

1）正交投影

实现正交投影，使用多组（left, right, bottom, top, near, far）参数，比较结果差异。

（1）cube 放置在(-1.5,0.5,-1.5)位置

```
model = glm::translate(model, glm::vec3(-1.5f, 0.5f, -1.5f));
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

（2）用 ImGui 中的滑块修改参数值，比较不同参数组合下的结果差异

```
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, 0.0f, 1.0f);
ImGui::SliderFloat("far", &ortho_far, 0.0f, 20.0f);
```

（3）正交投影

```
projection = glm::ortho(ortho_left, ortho_right, ortho_bottom, ortho_top, ortho_near, ortho_far);
```

2）透视投影

实现透视投影，使用多组参数，比较结果差异。

（1）cube 放置在(-1.5,0.5,-1.5)位置

```
model = glm::translate(model, glm::vec3(-1.5f, 0.5f, -1.5f));
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

（2）用 ImGui 中的滑块修改参数值，比较不同参数组合下的结果差异

```
ImGui::SliderFloat("fov", &pres_fov, 0.0f, 90.0f);
ImGui::SliderFloat("ratio", &pres_ratio, 0.0f, 3.0f);
ImGui::SliderFloat("near", &pres_near, 0.0f, 3.0f);
ImGui::SliderFloat("far", &pres_far, 0.0f, 100.0f);
```

（3）透视投影

```
projection = glm::perspective(pres_fov, pres_ratio, pres_near, pres_far);
```

3.2 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。

（1）把 cube 放置在(0,0,0)处

```
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f));
```

（2）使摄像机围绕 cube 旋转，并且时刻看着 cube 中心

```
GLfloat radius = 4.0f;
GLfloat camX = sin(glfwGetTime()) * radius;
GLfloat camZ = cos(glfwGetTime()) * 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));
```

lookAt 三个参数分别表示：摄像机位置，目标位置，表示上向量的世界空间中的向量（我们使用上向量计算右向量）。

为每一帧创建摄像机的 x 和 z 坐标确保摄像机在一个圆周上运动，通过重复计算 camX 和 camZ 来遍历所有圆圈上的点，这样摄像机就会绕着场景旋转。

预定义圆的半径 radius 为 4.0f。

我们将目标位置设为(0,0,0)保证摄像机始终看着 cube 中心。

此时 GLM 会创建一个 LookAt 矩阵，我们可以把它当作我们的观察矩阵，每次迭代都会创建一个新的观察矩阵。

（3）做透视投影

```
projection = glm::perspective(45.0f, 1.5f, 0.1f, 100.0f);
```

3.3 Camera 类的实现

要求实现一个 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，即类似 FPS 的游戏场景。

1）Camera 类定义

```
class Camera {
public:
    glm::vec3 cameraPos; // 摄像机位置
    glm::vec3 cameraFront; // 摄像机指向
    glm::vec3 cameraUp; // 右轴
    glm::vec3 cameraRight; // 上轴
    glm::vec3 worldUp;
    GLfloat cameraYaw;
    GLfloat cameraPitch;
    GLfloat cameraSpeed; // WASD
    GLfloat cameraSensitivity; // 鼠标
    GLfloat fov;
```

```
    // constructor
    Camera();
    Camera(glm::vec3 position, glm::vec3 worldup, GLfloat yaw, GLfloat pitch);
    // WASD移动
    void moveForward(GLfloat deltaTime);
    void moveBack(GLfloat deltaTime);
    void moveLeft(GLfloat deltaTime);
    void moveRight(GLfloat deltaTime);
    // 鼠标移动调整视角
    void ProcessMouseMovement(GLfloat xoffset, GLfloat yoffset);
    // 生成观察矩阵
    glm::mat4 GetViewMatrix();
private:
    // 更新数据成员
    void updateCameraVectors();
};
```

默认构造函数，初始化各个数据成员的值：

```
Camera::Camera() {
    this->cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
    this->cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
    this->cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
    this->cameraRight = glm::normalize(glm::cross(cameraUp, cameraPos));
    this->worldUp = glm::vec3(0.0f, 1.0f, 0.0f);

    this->cameraYaw = -90.0f;
    this->cameraPitch = 0.0f;
    this->cameraSpeed = 2.0f; // WASD
    this->cameraSensitivity = 0.02f; // 鼠标
    this->fov = 45.0f;
}
```

2）键盘控制移动

以向前移动函数为例，WASD 其他操作同理：

```
void Camera::moveForward(GLfloat deltaTime) {
    GLfloat v = this->cameraSpeed * deltaTime;
    this->cameraPos += v * this->cameraFront;
}
```

此处参数 deltaTime 存储上一帧所用的时间，把所有速度都去乘以 deltaTime 值，使摄像机的速度一直保持一致。

初始化：

```
float deltaTime = 0.0f;
float lastFrame = 0.0f;
```

每次渲染时更新：

```
GLfloat currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

具体按键及响应由输入控制函数 processInput 函数控制：

main 函数中调用

```
processInput(window);
```

函数具体定义：

```
// 输入控制
void processInput(GLFWwindow *window)
{
    // glfwGetKey
    // 输入：一个窗口以及一个按键(这里检查用户是否按下了返回键Esc)
    // 返回：这个按键是否正在被按下
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
    {
        glfwSetWindowShouldClose(window, true);
    }
    GLfloat cameraSpeed = 0.05f;
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.moveForward(deltaTime);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.moveBack(deltaTime);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.moveLeft(deltaTime);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.moveRight(deltaTime);
}
```

响应修改对象 camera 中摄像机位置的值，下一次渲染时获取新的 view 矩阵：

```
view = camera.GetViewMatrix();
```

此函数返回一个 LookAt 矩阵作为观察矩阵：

```
glm::mat4 Camera::GetViewMatrix() {
    return glm::lookAt(this->cameraPos, this->cameraPos + this->cameraFront, this->cameraUp);
}
```

3）鼠标控制移动视角

响应鼠标移动的回调函数：

```
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.ProcessMouseMovement(xoffset, yoffset);
}
```

在回调函数中计算当前帧和上一帧鼠标位置的偏移量，传入 Camera 类的成员函数 ProcessMouseMovement 控制摄像机的视角变化。

把偏移量乘上预先定义的 cameraSensitivity 控制视角的变化程度，同时为了防止视角的过分变换和超越，限制俯仰角不能超过 89 度也不能低于-89 度。

```
void Camera::ProcessMouseMovement(GLfloat xoffset, GLfloat yoffset) {
    xoffset *= this->cameraSensitivity;
    yoffset *= this->cameraSensitivity;
    this->cameraYaw += xoffset;
    this->cameraPitch += yoffset;
    if (this->cameraPitch > 89.0f) {
        this->cameraPitch = 89.0f;
    }
    if (this->cameraPitch < -89.0f) {
        this->cameraPitch = -89.0f;
    }
    this->updateCameraVectors();
}
```

此时我们获得了新的俯仰角和偏航角，可以据此得到新的由相机位置到观察目标的 front 向量 cameraFont 并更新相关数据：

```
void Camera::updateCameraVectors() {
    glm::vec3 front;
    front.x = cos(glm::radians(this->cameraYaw)) * cos(glm::radians(this->cameraPitch));
    front.y = sin(glm::radians(this->cameraPitch));
    front.z = sin(glm::radians(this->cameraYaw)) * cos(glm::radians(this->cameraPitch));
    this->cameraFront = glm::normalize(front);
    this->cameraRight = glm::normalize(glm::cross(this->cameraFront, this->worldUp));
    this->cameraUp = glm::normalize(glm::cross(this->cameraRight, this->cameraFront));
}
```

在下一次渲染时，重新获取 LookAt 矩阵作为 view 矩阵，就可以更新观察矩阵直接感受到视角的变化。