练 习 题 报 告

课程名称 计算机图形学

项目名称 相机定位和投影

学　　院 计算机与软件学院

专　　业 软件工程（腾班）

指导教师 熊卫丹

报 告 人 洪子敬　学号 2022155033

一、 练习目的

1. 了解 OpenGL 中相机的模型视图变换的基本原理
2. 掌握 OpenGL 中相机观察变换矩阵的推导
3. 掌握在 OpenGL 中实现相机观察变换
4. 了解 OpenGL 中正交投影和透视投影变换
5. 了解在 OpenGL 中实现正交投影和透视投影变换。

二．练习完成过程及主要代码说明

练习要求：完善 Camera.cpp 和 main.cpp，实现正交投影和透视眼投影效果。

Task-1：在 Camera.cpp 中完善 lookAt、updateCamera 函数，main.cpp 中完善 display_2 函数。

lookAt 函数的补充：按照课上所讲内容我们知道此处是需要补充相机的观察矩阵，此矩阵主要由两部分组成，一是局部坐标系部分，二是平移部分；

对于前者，首先需要计算出三个方向的向量：前向向量 $n = eye - at$、右向向量 $u = cross(up, n)$（cross 是叉乘函数）和与平面平行的往下向量 $v = cross(n, u)$，不过要注意上述向量算完之后均要进行归一化（使用 openGL 中的 normalize 函数即可实现）。按照下面的矩阵格式进行排布即可：

$$viewMarix = \begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

对于后者，我们需要在前者基础上对该变换矩阵进行平移（上面的计算均不在原点，此操作目的是回到原点），操作上加上下面矩阵即可：

$$T = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

最终得到下面相机观察变换矩阵：

$$\begin{bmatrix} u_x & u_y & u_z & -xu_x - yu_y - zu_z \\ v_x & v_y & v_z & -xv_x - yv_y - zv_z \\ n_x & n_y & n_z & -xn_x - yn_y - zn_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

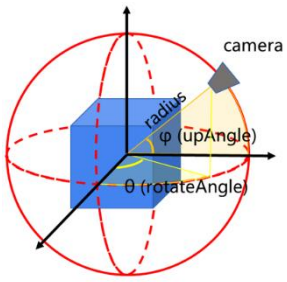
而在具体实现时就是进行向量的内积，将每个方向沿着图形中心坐标往回移动，即每个方向的向量与 eye 向量进行内积，再对结果取反（因为是反方向移动）：

$$\begin{bmatrix} u_x & u_y & u_z & -dot(u, eye) \\ v_x & v_y & v_z & -dot(v, eye) \\ n_x & n_y & n_z & -dot(n, eye) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

详细代码如下所示：（mat4(1.0f)是定义一个 4*4 的单位矩阵）

```
glm::mat4 Camera::lookAt(const glm::vec4& eye, const glm::vec4& at, const glm::vec4& up)
{
    // @TODO: Task1:请按照实验课内容补全相机观察矩阵的计算
    glm::vec3 n = glm::normalize(glm::vec3(eye - at));//前向向量的计算
    glm::vec3 u = glm::normalize(glm::cross(glm::vec3(up), n));//右向向量
    glm::vec3 v = glm::normalize(glm::cross(n, u));//平面
    //设置旋转和平移部分
    glm::mat4 c = glm::mat4(1.0f);
    c[0][0] = u.x;  c[0][1] = u.y;  c[0][2] = u.z;  c[0][3] = -glm::dot(u,glm::vec3(eye));
    c[1][0] = v.x;  c[1][1] = v.y;  c[1][2] = v.z;  c[1][3] = -glm::dot(v,glm::vec3(eye));
    c[2][0] = n.x;  c[2][1] = n.y;  c[2][2] = n.z;  c[2][3] = -glm::dot(n,glm::vec3(eye));
    c[3][0] = 0.0f; c[3][1] = 0.0f; c[3][2] = 0.0f; c[3][3] = 1.0f;
    c = glm::transpose(c);
    return c;
                                                                    洪子敬
}                                                                   2022155033
```

updateCamer 函数的补充：由实验指导书可知此处需要我们根据角度和距离半径更新计算相机 eye 的位置 lookAt 函数。如右图所示，相机的设置方式是球形轨迹，竖直向上的 y 轴，朝外的是 z 轴，横向的是 x 轴，rotateAngle 是绕 x 轴得到的夹角，upAngle 是绕 y 轴得到的夹角。根据几何关系我们可以得到相机位置的表示如下：



$$eye_x = radius * \sin(rotateAngle) * \cos(upAngle);$$

$$eye_y = radius * \sin(upAngle);$$

$$eye_x = radius * \cos(rotateAngle) * \cos(upAngle);$$

此外，由于 upAngle 大于 90 度时，相机坐标系 u 变量会变成反方向，所以要将 up 向量的 y 轴分量改为负方向，即将 up 向量的 y 分量改为相反数即可。

详细代码如下所示：

```
void Camera::updateCamera()
{
    // @TODO: Task1 设置相机位置和方向
    // 根据rotateAngle和upAngle设置x
    float eyex = radius * sin(glm::radians(rotateAngle)) * cos(glm::radians(upAngle));
    // 根据upAngle设置y
    float eyey = radius * sin(glm::radians(upAngle));
    // 根据rotateAngle和upAngle设置z
    float eyez = radius * cos(glm::radians(rotateAngle)) * cos(glm::radians(upAngle));;

    eye = glm::vec4(eyex, eyey, eyez, 1.0);
    up = glm::vec4(0.0, 1.0, 0.0, 0.0);
    at = glm::vec4(0.0, 0.0, 0.0, 1.0);

    // @TODO: Task1:设置相机参数
    // 使用相对于at的角度控制相机的时候，注意在upAngle大于90的时候，相机坐标系的u向量会变成相反的方向，
    // 要将up的y轴改为负方向才不会发生这种问题
    if (upAngle > 90.0f || upAngle < -90.0f) {
        up = glm::vec4(0.0, -1.0, 0.0, 0.0);                        洪子敬
    }                                                               2022155033
```

display_2 函数的补充：函数中需要我们补充的就是 lookAt 函数的调用，如下所示，传入相机的几个方向向量 eye、at 和 up 即可：

```
// 调用 Camera::lookAt 函数计算视图变换矩阵
camera_2->viewMatrix = camera_2->lookAt(camera_2->eye, camera_2->at, camera_2->up);
```

Task-2：在 Camera.cpp 中完善 ortho 函数，main.cpp 中完善 display_3 函数。

ortho 函数的补充：根据实验指导书我们知道此处是需要补充正交投影矩阵，定义视景体由

6 个面：前面 near、后面 far、上面 top、下面 bottom、左面 left 和右面 right 组成，使用的投影矩阵如下：

$$N=\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

详细代码如下所示：

```
glm::mat4 Camera::ortho(const GLfloat left, const GLfloat right,
    const GLfloat bottom, const GLfloat top,
    const GLfloat zNear, const GLfloat zFar)
{
    // @TODO: Task2:请按照实验课内容补全正交投影矩阵的计算
    //创建4*4的单位矩阵
    glm::mat4 c = glm::mat4(1.0f);
    c[0][0] = 2.0f / (right - left);
    c[0][3] = -(right + left) / (right - left);
    c[1][1] = 2.0f / (top - bottom);
    c[1][3] = -(top + bottom) / (top - bottom);
    c[2][2] = -2.0f / (zFar - zNear);
    c[2][3] = -(zFar + zNear) / (zFar - zNear);
    c = glm::transpose(c);
    return c;
}
```

洪子敬
2022155033

display_3 函数的补充：使用 Camera 类中定义投影参数 zNear 和 zFar 以及正交投影参数 scale 出传入 ortho 函数即可，代码如下所示：

```
// @TODO: Task2: 调用 Camera::ortho 函数计算正交投影矩阵
camera_3->projMatrix = camera_3->ortho(-camera_3->scale, camera_3->scale, -camera_3->scale, camera_3->scale, camera_3->zNear, camera_3->zFar);
```

Task-3：在 Camera.cpp 中完善 perspective 函数，main.cpp 中完善 display_4 函数。

perspective 函数的补充：根据实验指导书可知此处需要我们补充透视投影矩阵，与正交投影相似，也是多个面的绘制，但其视景体已经变成了棱台，使用的投影矩阵如下：

$$N=\begin{bmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$top = near * \tan\left(\frac{fov}{2}\right)$$

$$right = top * aspect$$

详细代码如下所示：

```cpp
glm::mat4 Camera::perspective(const GLfloat fov, const GLfloat aspect,
    const GLfloat zNear, const GLfloat zFar)
{
    // @TODO: Task3:请按照实验课内容补全透视投影矩阵的计算
    glm::mat4 c = glm::mat4(1.0f);
    GLfloat top = zNear * tan(fov * M_PI / 180 / 2);
    GLfloat right = top * aspect;
    c[0][0] = zNear / right;
    c[1][1] = zNear / top;
    c[2][2] = -(zFar + zNear) / (zFar - zNear);
    c[2][3] = -(2 * zFar * zNear) / (zFar - zNear);
    c[3][2] = -1.0f;
    c[3][3] = 0.0f;
    c = glm::transpose(c);
    return c;
}
```

洪子敬
2022155033

display_4 函数的补充：使用 Camera 类中定义投影参数 zNear 和 zFar 以及透视投影参数 fov、aspect 传入 perspective 函数即可，代码如下所示：

```cpp
// @TODO: Task3: 调用 Camera::perspective 函数计算透视投影矩阵
camera_4->projMatrix = camera_4->perspective(camera_4->fov, camera_4->aspect, camera_4->zNear, camera_4->zFar);
```

结果展示：运行程序可以得到下面的效果，练习成功完成。