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 计算机科学与技术

指导教师 周虹

报 告 人 吴嘉楷 学号 2022150168

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

1. 在 Camera.cpp 中完善 lookAt 函数

函数源码：

```
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
    // 计算相机的观察方向(从 eye 到 at 的方向)
    glm::vec3 forward = glm::normalize(glm::vec3(eye - at));
    // 计算相机的右向量（与 forward 和 up 向量垂直）
    glm::vec3 right = glm::normalize(glm::cross(glm::vec3(up), forward));
    // 计算相机的新上向量（与 forward 和 right 向量垂直）
    glm::vec3 newUp = glm::cross(forward, right);
    // 初始化观察矩阵
    glm::mat4 viewMatrix = glm::mat4(1.0f);
    // 设置观察矩阵的各个分量
    viewMatrix[0][0] = right.x;
    viewMatrix[1][0] = right.y;
    viewMatrix[2][0] = right.z;
    viewMatrix[0][1] = newUp.x;
    viewMatrix[1][1] = newUp.y;
    viewMatrix[2][1] = newUp.z;
    viewMatrix[0][2] = forward.x;
    viewMatrix[1][2] = forward.y;
    viewMatrix[2][2] = forward.z;
    // 更新观察矩阵的平移部分
    viewMatrix[3][0] = -glm::dot(right, glm::vec3(eye));
    viewMatrix[3][1] = -glm::dot(newUp, glm::vec3(eye));
    viewMatrix[3][2] = -glm::dot(forward, glm::vec3(eye));
    // 返回观察矩阵
    return viewMatrix;
}
```

函数说明：

这个 Camera::lookAt 函数的作用是生成一个相机的观察矩阵，用于将场景从世界空间变换到相机空间，也就是从相机的视角来看整个场景。它的参数分别是相机的位置 eye，相机所看的目标点 at，以及相机的上方向 up。

首先，函数通过计算 eye 和 at 之间的向量，得到了相机的观察方向 forward，这是从 eye 到 at 的归一化方向。然后，通过叉积计算了一个与 forward 和 up 垂直的向量 right，它代表了相机的右侧方向。接着，又使用叉积来计算与 forward 和 right 垂直的向量 newUp，从而得到一个与相机的视角方向相关的正交基（相互垂直的 right、newUp、forward 向量）。

接下来，函数初始化了一个单位矩阵 viewMatrix，并将这个正交基的三个方向向量分别填入矩阵的前三列，表示相机的旋转变换部分。之后，函数使用相机位置 eye 与这些方向向量的点积来计算平移部分，从而完成观察矩阵的平移变换，这样就能将世界坐标转换为相机坐标。

最终，函数返回这个观察矩阵，它将场景从世界空间映射到相机的视角下。

参考公式：

（1）观察平面法向量（VPN：View-plane Normal）

$$VPN = e - a$$

（2）归一化得到：

$$\boldsymbol{n} = \frac{VPN}{|VPN|}$$

（3）与 VUP 和 VPN 都垂直的方向向量

$$\boldsymbol{u} = \frac{VUP \times \boldsymbol{n}}{|VUP \times \boldsymbol{n}|}$$

（4）VUP 在照相机胶片平面上的投影

$$\boldsymbol{v} = \frac{\boldsymbol{n} \times \boldsymbol{u}}{|\boldsymbol{n} \times \boldsymbol{u}|}$$

（5）相机观察矩阵 viewMatrix

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

2. 完善 updateCamera 函数

函数源码：

```
void Camera::updateCamera()
{
    // 角度转弧度
    float radiansUp = glm::radians(upAngle);
    float radiansRotate = glm::radians(rotateAngle);
    //计算相机位置
    float eyex = radius * cos(radiansUp) * sin(radiansRotate);
    float eyey = radius * sin(radiansUp);
    float eyez = radius * cos(radiansUp) * cos(radiansRotate);
```

```
    // 更新相机位置
    eye = glm::vec4(eyex, eyey, eyez, 1.0);
    //设置相机的参考点（at）
    at = glm::vec4(0.0, 0.0, 0.0, 1.0);
    //设置相机的 VUP 方向，与世界坐标系的 y 方向相同（方向朝上）
    up = glm::vec4(0.0, 1.0, 0.0, 0.0);
    // 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 || upAngle < -90)
        up = glm::vec4(0.0, -1.0, 0.0, 0.0);
}
```

函数说明：

函数的主要功能是更新相机的位置和方向，使用极坐标的方式来确定相机的位置和视角。

首先，它将俯仰角度 upAngle 和旋转角度 rotateAngle 从角度转换为弧度，因为三角函数如 sin 和 cos 需要使用弧度来进行计算。接着，通过极坐标公式，计算了相机在三维空间中的位置，分别得到了相机的 X、Y 和 Z 坐标。eyex 表示相机在 X 轴的坐标，eyey 表示相机在 Y 轴的坐标，而 eyez 表示相机在 Z 轴的坐标，这些坐标是基于相机到目标点的距离 radius 以及旋转和俯仰角度计算的。

相机的位置 eye 被设置为一个四维向量，其坐标为 (eyex, eyey, eyez)，w 分量为 1.0，表示齐次坐标。参考点 at 被固定为原点 (0.0, 0.0, 0.0)，这意味着相机始终指向原点。相机的上方向 up 被初始设置为 (0.0, 1.0, 0.0)，表示在世界坐标系中，Y 轴是朝上的方向。

然后代码根据相机的俯仰角 upAngle 进行了一些调整。如果 upAngle 超过 90 度或者小于-90 度，意味着相机已经翻转到目标点的上方或者下方，这时需要将相机的 up 向量的 Y 分量反向设置为 -1.0，从而确保相机的上方向保持正确，避免视角混乱。这个调整的目的是为了避免在极端角度下相机的上方向与实际的上下方向相反，保持相机视角的一致性。整个函数实现了相机视角的灵活控制，允许通过调整角度来移动相机，并在必要时纠正相机的上方向，确保它始终保持正确的朝向。

原理示意：

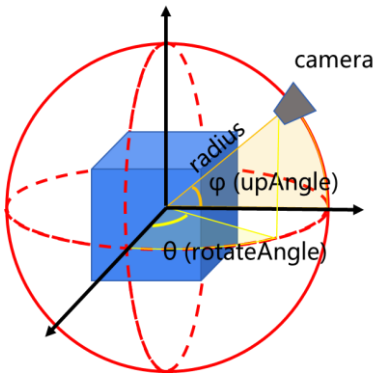

图 1 updateCamera 逻辑原理示意图

3. 在 Camera.cpp 中完善 ortho 函数，实现正交投影

函数源码：

```
glm::mat4 Camera::ortho(const GLfloat left, const GLfloat right,const GLfloat bottom, const GLfloat top,
    const GLfloat zNear, const GLfloat zFar){
    // 创建一个单位矩阵作为正交投影矩阵的初始值
    glm::mat4 orthoMatrix = glm::mat4(1.0f);
    // 设置正交投影矩阵的各个分量
    orthoMatrix[0][0] = 2.0f / (right - left);
    orthoMatrix[1][1] = 2.0f / (top - bottom);
    orthoMatrix[2][2] = -2.0f / (zFar - zNear);
    orthoMatrix[3][0] = -(right + left) / (right - left);
    orthoMatrix[3][1] = -(top + bottom) / (top - bottom);
    orthoMatrix[3][2] = -(zFar + zNear) / (zFar - zNear);
    return orthoMatrix;   // 返回正交投影矩阵
}
```

函数说明：

函数的主要功能是计算一个正交投影矩阵，用于在图形学中将三维物体投影到二维平面上。这个 Camera::ortho 函数接收六个参数，它们分别定义了投影的视锥体的左右边界（left 和 right）、上下边界（bottom 和 top）以及前后边界（zNear 和 zFar）。通过这些参数，函数能够确定投影的可视范围，并生成一个 4x4 的正交投影矩阵。

首先，函数创建了一个单位矩阵作为正交投影矩阵的基础矩阵，这意味着所有对角线元素初始为 1.0，其他元素为 0。

随后，矩阵的各个分量会根据投影范围被修改。具体来说，X 轴的缩放因子是通过 (right - left) 的差值计算的，用来把视椎体的左右边界映射到标准设备坐标的[-1, 1]范围内。同样，Y 轴的缩放因子是通过 (top - bottom) 来计算，作用是将上下边界映射到[-1, 1]。Z 轴的缩放因子则通过 (zFar - zNear) 来计算，确保 Z 轴的范围也映射到[-1, 1]，这里注意 Z 轴的缩放因子是负的，这是因为在 OpenGL 中，Z 轴是朝屏幕内的。

除了缩放因子外，矩阵还需要进行平移操作，以确保视锥体的中心对齐到原点。X 轴的平移值是通过 -(right + left) / (right - left) 计算的，这会将 X 轴的中心移动到坐标原点。Y 轴的平移也是类似的，通过 -(top + bottom) / (top - bottom) 来计算，确保 Y 轴的中心在原点。Z 轴的平移则通过 -(zFar + zNear) / (zFar - zNear) 来设置，确保 Z 轴的中心同样对齐。

最终，函数返回这个经过计算后的正交投影矩阵，它能将物体的三维坐标缩放和平移到标准设备坐标系中，通常用于二维渲染或没有透视效果的 3D 场景。

投影矩阵：

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

4. 在 Camera.cpp 中完善 perspective 函数，实现透视投影

函数源码：

```
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
    //  创建一个单位矩阵作为透视投影矩阵的初始值
    glm::mat4 perspectiveMatrix = glm::mat4(1.0f);
    //  计算透视投影矩阵的元素值
    float f = 1.0f / tan(glm::radians(fov / 2.0f)); //  计算焦距
    perspectiveMatrix[0][0] = f / aspect;   // X  缩放因子
    perspectiveMatrix[1][1] = f;                  // Y  缩放因子
    perspectiveMatrix[2][2] = (zFar + zNear) / (zNear - zFar); // Z  缩放因子
    perspectiveMatrix[2][3] = -1.0f;           // Z  平移因子（注意为负值）
    perspectiveMatrix[3][2] = (2.0f * zFar * zNear) / (zNear - zFar); // Z  平移因子
    perspectiveMatrix[3][3] = 0.0f;             // W  缩放因子
    //  返回透视投影矩阵
    return perspectiveMatrix;
}
```

函数说明：

透视投影矩阵用于在三维图形学中模拟人类眼睛的透视效果，物体离相机越远，它们看起来越小。Camera::perspective 函数接收四个参数来定义透视投影的属性：视角（fov，即视野范围的角度）、宽高比（aspect，即宽度与高度的比值）、近剪切平面（zNear，即相机能看到的最近的距离）以及远剪切平面（zFar，即相机能看到的最远的距离）。函数返回一个 4x4 的透视投影矩阵。

首先，函数创建了一个单位矩阵 perspectiveMatrix 作为透视投影矩阵的初始值，这意味着矩阵的对角线元素为 1.0，其他位置的元素为 0。接下来，函数需要根据传入的参数修改矩阵的各个元素以形成透视投影。

透视投影的核心是根据给定的视角（fov）和宽高比（aspect）计算出投影平面的缩放因子。首先，函数通过 tan(glm::radians(fov / 2.0f)) 计算了视角的一半的正切值，并取它的倒数，这相当于计算了焦距。这个值 f 用来表示 Y 轴的缩放因子，同时为了确保图像宽高比的正确性，X 轴的缩放因子需要除以宽高比 aspect，因此 perspectiveMatrix[0][0] = f / aspect，而 perspectiveMatrix[1][1] = f 表示 Y 轴的缩放因子。

Z 轴的缩放和平移与透视效果直接相关。perspectiveMatrix[2][2] 和 perspectiveMatrix[3][2] 通过 (zFar + zNear) / (zNear - zFar) 和 (2.0f * zFar * zNear) / (zNear - zFar) 来设置，它们用来压缩 Z 轴的范围，将物体的深度值映射到标准化设备坐标系的[-1, 1]范围内。Z 轴的缩放值为负，因为 OpenGL 的深度坐标通常是负的。perspectiveMatrix[2][3] = -1.0f 是个特殊的设置，它将透视投影中的 W 分量变为负数，从而实现透视效果，也就是物体越远，投影越小。

最后，perspectiveMatrix[3][3] 被设置为 0.0f，这是透视投影矩阵的一个重要特征。它确保了 W 分量在投影过程中按比例缩放，从而实现视角变换和深度感。

正交投影的棱台视见体相关公式：

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

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

5. 在 main.cpp 中完善 display_3 函数

```
void display_3()
{
    glViewport(WIDTH / 2, HEIGHT / 2, WIDTH / 2, HEIGHT / 2);
    camera_3->updateCamera();

    glm::mat4 modelMatrix = glm::mat4(1.0); // 物体模型的变换矩阵
    camera_3->viewMatrix = camera_3->lookAt(camera_3->eye, camera_3->at, camera_3->up); // 调用 Camera::lookAt 函数计算视图变换矩阵

    // @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);
    // 传递投影变换矩阵
    glUniformMatrix4fv(cube_object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
    glUniformMatrix4fv(cube_object.viewLocation, 1, GL_FALSE, &camera_3->viewMatrix[0][0]);
    glUniformMatrix4fv(cube_object.projectionLocation, 1, GL_FALSE, &camera_3->projMatrix[0][0]);
    glUseProgram(cube_object.program);
    glDrawArrays(GL_TRIANGLES, 0, cube->getPoints().size());
}
```

图 2 display_3 函数

函数说明：

display_3 函数调用 camera_3->ortho 计算正交投影矩阵 projMatrix，使用参数中的 scale 值来定义正交投影的左右、上下边界，并设置 zNear 和 zFar 来确定近远剪切平面的距离。这一步确保了物体被以正交投影的方式投影到屏幕上，没有透视缩放效果。

6. 在 main.cpp 中完善 display_4 函数

```
void display_4()
{
    glViewport(WIDTH / 2, 0, WIDTH / 2, HEIGHT / 2);
    camera_4->updateCamera();

    glm::mat4 modelMatrix = glm::mat4(1.0); // 物体模型的变换矩阵
    camera_4->viewMatrix = camera_4->lookAt(camera_4->eye, camera_4->at, camera_4->up); // 调用 Camera::lookAt

    // @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
    );

    // 传递投影变换矩阵
    glUniformMatrix4fv(cube_object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
    glUniformMatrix4fv(cube_object.viewLocation, 1, GL_FALSE, &camera_4->viewMatrix[0][0]);
    glUniformMatrix4fv(cube_object.projectionLocation, 1, GL_FALSE, &camera_4->projMatrix[0][0]);

    glUseProgram(cube_object.program);
    glDrawArrays(GL_TRIANGLES, 0, cube->getPoints().size());
}
```

图 3 display_4 函数

函数说明：

display_4 调用了 camera_4->perspective 函数计算透视投影矩阵 projMatrix，这个函数基于视野角（fov）、宽高比（aspect）、近剪切平面（zNear）和远剪切平面（zFar）来计算透视投影矩阵。透视投影的特点是远处的物体会显得更小，从而模拟人眼的真实视觉效果。

7. 修改窗口标题

```
GLFWwindow* mainwindow = glfwCreateWindow(WIDTH, HEIGHT, "2022150168_吴嘉楷_实验3.1", NULL, NULL);
```

图 4 窗口配置

8. 实现效果

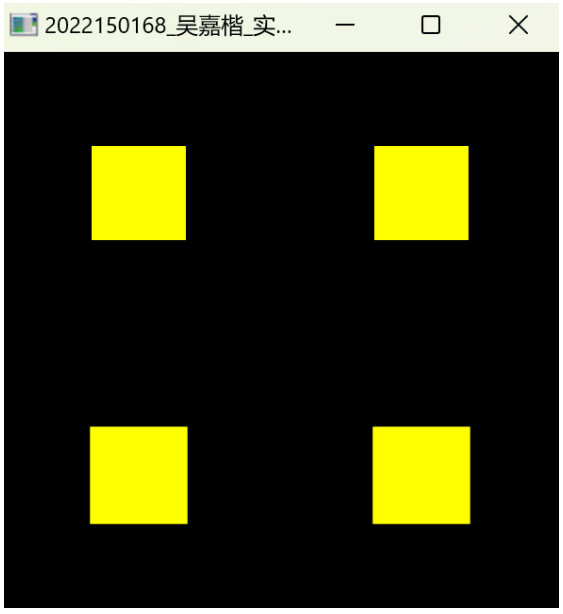

图 5 初始时的四个正方体视图

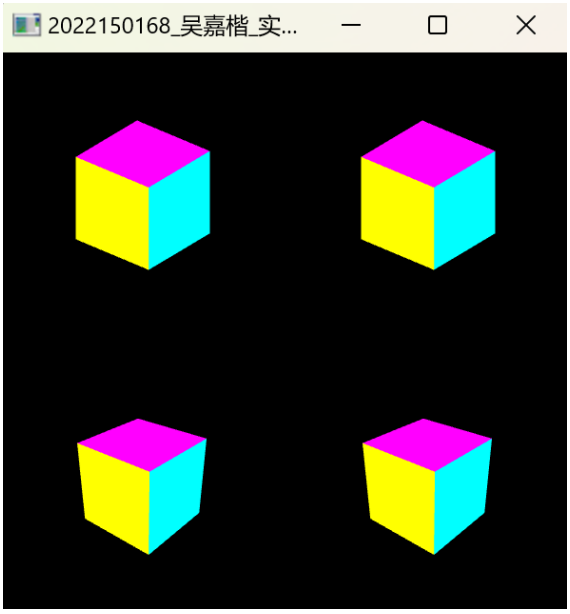

图 6 旋转相机角度之后的视图

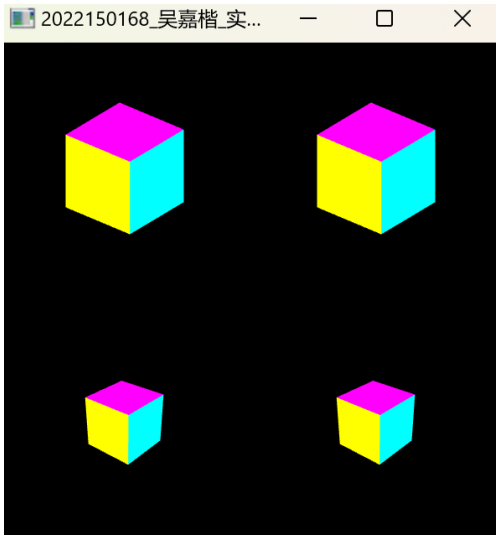

图 7 增加透视投影的 FOV。

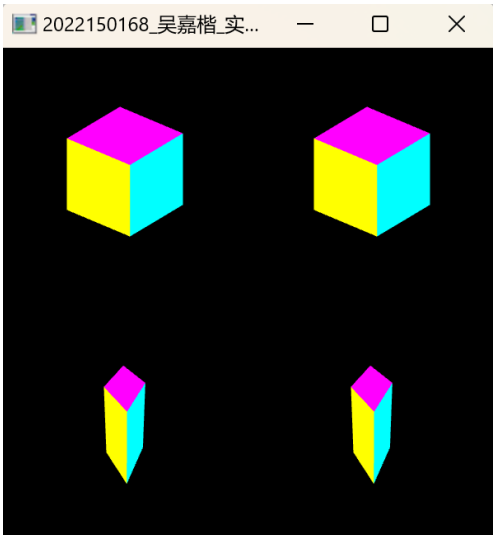

图 8 增加透视投影的宽度

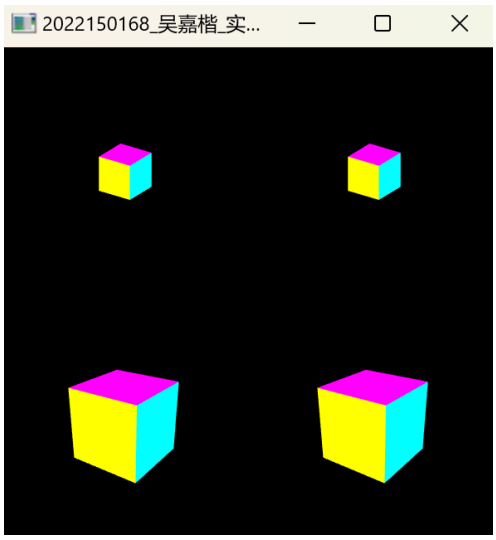

图 9 增加正交投影的范围