练 习 题 报 告

课程名称 计算机图形学

项目名称 投影和硬阴影

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. 熟悉在 OpenGL 中实现正交投影变换。
2. 了解使用投影变换实现场景的硬阴影效果。

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. 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：

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

3. 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：

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

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

4. 在 main.cpp 中完善 display()函数

4.1 计算阴影投影矩阵并更新模型视图矩阵

关键源码：

```
glm::mat4 shadowModelMatrix = glm::mat4(0.0f); // 初始化阴影的模型变换矩阵。
shadowModelMatrix[0][0] = -light_position[1];
shadowModelMatrix[1][0] = light_position[0];
shadowModelMatrix[1][2] = light_position[2];
shadowModelMatrix[2][2] = -light_position[1];
shadowModelMatrix[1][3] = 1;
shadowModelMatrix[3][3] = -light_position[1];

modelMatrix = shadowModelMatrix * modelMatrix; // 更新阴影的模型变换矩阵。
```

程序说明：

这段代码构建了一个用于生成阴影的模型变换矩阵 shadowModelMatrix，实现的是将物体在一个平面上生成阴影的效果。首先，代码初始化了 shadowModelMatrix 为一个全零矩阵，以便后续只设置特定的矩阵元素。接着，通过一系列的元素赋值，根据光源的 light_position 来设置矩阵的关键位置，从而影响阴影的投影效果。

shadowModelMatrix[0][0] = -light_position[1]; 是基于光源的 y 坐标对阴影的 x 轴进行缩放。shadowModelMatrix[1][0] = light_position[0]; 则根据光源的 x 坐标调节了 y 轴上的偏移，使得阴影在 y 方向上具有朝向光源方向的效果。shadowModelMatrix[1][2] = light_position[2]; 是为了根据光源的 z 坐标来改变阴影在 z 方向上的位置。

另外，shadowModelMatrix[1][3] = 1; 和 shadowModelMatrix[3][3] = -light_position[1]; 分别为偏移和缩放因子，用于使阴影的投影距离和位置受光源高度的影响。

最终，将 shadowModelMatrix 与物体的 modelMatrix 相乘后，可以得到该物体的阴影变换矩阵，实现将物体的阴影投射到指定平面上的效果。

投影矩阵公式参考：

$$\begin{bmatrix} x_k' \\ y_k' \\ z_k' \\ w_k \end{bmatrix} = \begin{bmatrix} x_l y - x y_l \\ 0 \\ z_l y - y_l z \\ y - y_l \end{bmatrix} = \begin{bmatrix} -y_l & x_l & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & z_l & -y_l & 0 \\ 0 & 1 & 0 & -y_l \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4.2 投影三角形的绘制

关键源码：

```
// 传递 isShadow 变量。
glUniform1i(tri_object.shadowLocation, 0);
// 传递 unifrom 关键字的矩阵数据。
glUniformMatrix4fv(tri_object.modelLocation, 1, GL_FALSE, &modelMatrix[0][0]);
glUniformMatrix4fv(tri_object.viewLocation, 1, GL_FALSE, &camera->viewMatrix[0][0]);
glUniformMatrix4fv(tri_object.projectionLocation, 1, GL_FALSE, &camera->projMatrix[0][0]);
// 绘制
glDrawArrays(GL_TRIANGLES, 0, triangle->getPoints().size());
```

代码说明：

在绘制过程中，我们可以简单地将三角形绘制两次即可。第一次是按照常规方式绘制，第二次是使用了阴影变换矩阵之后对新的阴影三角形进行绘制。也就是说，第一次绘制时，将阴影投影矩阵设置为单位矩阵，而第二次绘制是计算出矩阵值之后变换三角形得到投影三角形。由于第一次绘制代码已给出，我们只需要补充第二次绘制的代码，从而绘制出新的阴影三角形。

（1）设置阴影开关

这里使用 glUniform1i 函数将 isShadow 的值传递给着色器中的 shadowLocation 变量。shadowLocation 是一个整型的 uniform 变量，用来控制是否启用阴影效果。此处设置为 0，表示不启用阴影。

（2）传递矩阵数据

glUniformMatrix4fv 是 OpenGL 中的一个函数，它用来将一个 4x4 的浮点矩阵传递给着色器中的 uniform 变量。在 3D 图形中，矩阵用于描述对象的变换，比如旋转、缩放和平移。tri_object.modelLocati、tri_object.viewLocation 和 tri_object.projectionLocation 是着色器程序中用来存储模型矩阵、视图矩阵和投影矩阵的 uniform 变量的位置。

GL_FALSE 在这里表示传递给 glUniformMatrix4fv 的矩阵数据不需要转置，即保持 OpenGL 默认的行优先存储方式。

（3）绘制三角形

glDrawArrays 函数告诉 OpenGL 按照指定的模式（这里是 GL_TRIANGLES，意味着每个三个顶点构成一个三角形）来渲染一系列顶点。

函数的第一个参数 GL_TRIANGLES 指定了绘制模式，第二个参数 0 表示从顶点数组的起始位置（索引 0）开始绘制，第三个参数 triangle->getPoints().size()指定了要绘制的顶点数量，这个数量是通过调用 triangle 对象的 getPoints 方法获得的顶点集合的大小。

5. 修改运行窗口标题、尺寸

```
GLFWwindow* mainwindow = glfwCreateWindow(1000, 1000, "2022150168_吴嘉楷_实验3.2", NULL, NULL);
```

图 1 运行窗口配置

6. 运行结果

图 2 初始效果

图 3 沿着 X 轴移动光源后的结果

图 4 沿着 Y 轴移动光源后的结果

图 5 沿着 Z 轴移动光源后的结果

图 6 改变旋转角度后的结果

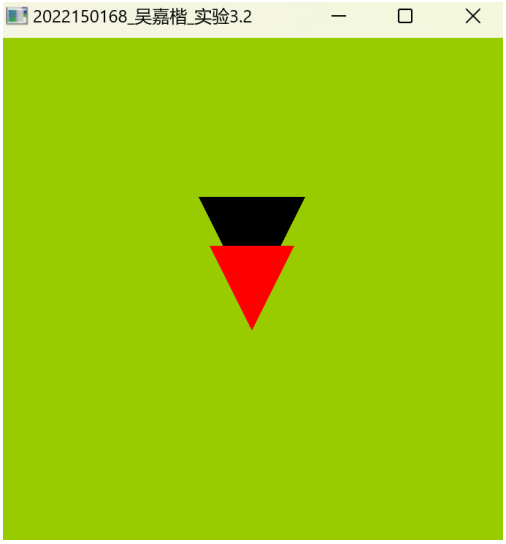

图 7 改变 up 角度后的结果

图 8 缩小大小后的结果

图 9 沿着 X、Y、Z 轴移动光源后的结果

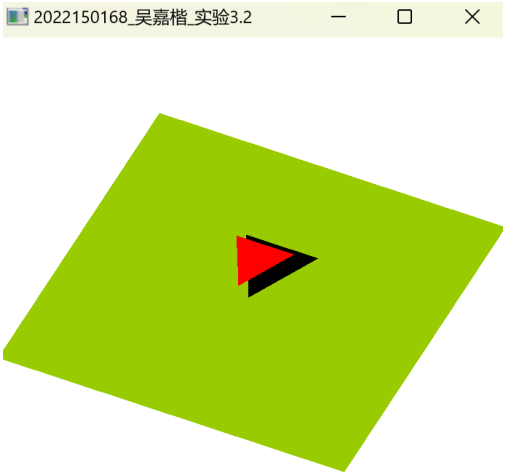

图 10 改变旋转、up、scale 后的结果

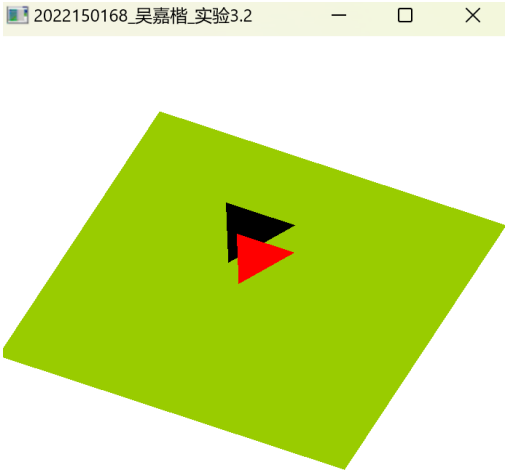

图 11 综合各种改动后的结果