科目	计算机图形学
学号	16340294
姓名	张星
邮箱	dukestar@foxmail.com

Basic

题目一

实验要求

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

实验思路

- 颜色不一致：这个我在上次就已实现，正方体每个角有两个属性，位置和颜色，使用相应的shader渲染即可。
- 正交投影：model和view矩阵无需改变，只需将上次的透视投影改为正交即可，通过ImGui调整各项参数，观察变化。具体函数如下：

  ```
  projection = glm::ortho(left, right, bottom, top, nearPos, farPos);
  ```

  经过我的观察，前面两个参数标识了平头截体的左右坐标，后面两个表示了上下坐标，最后两个表示了近平面和远平面的坐标。当将left增大时，整个空间就向右移，正方体看起来就向左拉伸，变成扁的，其他也如此。当调整近坐标时，随着数字的增大，正方体开始从近处渐渐消失一部分，这是因为正方体有些坐标超出了平截头箱的范围，不再显示。
- 透视投影：与上次没有太大差别，它有四个参数：视野，宽高比，近平面，远平面。其中后面两个与正交投影作用相同，宽高比由屏幕的参数决定，视野越大，则物体越小，反之亦然。公式如下：

  ```
  projection = glm::perspective(glm::radians(perspective), (float)height / (float)width, nearPos, farPos);
  ```
- 主要代码详见函数：

  ```
  void Projection(GLFWwindow *window, const unsigned int shaderProgram);
  ```
- 除此以外，我还添加了ImGui，方便连续动态调整，不再赘述。

实验结果

见result1.gif。

题目二

实验要求

> 视角变换(View Changing):
>
> - 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

实验思路

主要代码详见：

```
void ViewChanging(GLFWwindow *window, const unsigned int shaderProgram);
```

为了方便观察，我在上题的基础上，取消了正方体的自转与平移功能，这样就能够更加直观地进行观察，效果也更加明显。

关键代码：

```
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f));
float cameraX = sin(glfwGetTime()) * radius;
float cameraZ = cos(glfwGetTime()) * radius;
view = glm::lookAt(glm::vec3(cameraX, 0.0f, cameraZ),
                   glm::vec3(0.0f, 0.0f, 0.0f),
                   glm::vec3(0.0f, 1.0f, 0.0f));
```

本题使用了 *lookAt* 方法，其中第一个参数为camera的位置，第二个为需要朝向的target，即正方体中心，第三个参数为一个up向量，我们指定其为y轴正方向。前两个参数相减，即可得到摄像机的方向向量，然后与第三个参数做外积，得到一个右向量，最后方向向量与右向量再做外积，就能得到上向量，并且由于叉乘的性质，这三个向量彼此正交。当然这一切都已经封装在 *lookAt* 里面了。

由于三角函数的性质，我们使用 *sin* 和 *cos* 函数结合 *glfwGetTime()* 即可得到x与z的坐标，因为这是在XoZ平面上，y是不用变化的，这样就可以得到一个持续旋转的摄像机了。

实验结果

详见result2.gif

题目三

实验要求

> 在GUI里添加菜单栏，可以选择各种功能。 *Hint:* 使摄像机一直处于一个圆的位置，可以参考以下公式：

实验思路

主要代码详见：

```
void ViewChanging(GLFWwindow *window, const unsigned int shaderProgram);
```

我的第二个函数是在第一个的基础上进行了改动，除了正方体本身之外的旋转、平移，一切功能都与第一题无异，不再赘述。

实验结果

详见result2.gif。

题目四

实验要求

> 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？在报告中写入。（Hints：你可能有不止一个摄像机）

答案

因为OpenGL中没有摄像机矩阵，我们通过将场景中所有的物体向相反方向移动来模拟摄像机。OpenGL将model和view矩阵合二为一，使它们保持为一个单独的矩阵堆栈，可以更加方便，节省空间。并且有助于转换相对于不同空间的位置或者方向等。

Bonus

题目一

实验要求

> 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("look around")，即类似FPS(First Person Shooting)的游戏场景。

实验思路

在之前的实验中，view矩阵使用了 *lookAt* 来获取，所以我们的核心任务就是根据输入来获取view矩阵，输入与 *lookAt* 相同，三个向量。这个函数并不难：这三个参数都是初始化时便已经传递给类了

```
glm::mat4 Camera::getViewMatrix() {
    return glm::lookAt(Position, Position + Front, Up);
}
```

除此以外，便是WSAD了，这个只需要将Position向量加以改变即可：

```
void Camera::moveForward(GLfloat const distance) {
    Position += Front * distance;
}
void Camera::moveBack(GLfloat const distance) {
    Position -= Front * distance;
}
void Camera::moveRight(GLfloat const distance) {
    Position += Right * distance;
}
void Camera::moveLeft(GLfloat const distance) {
    Position -= Right * distance;
}
```

其中用到了一个向量Right，这个向量需要用外积获得：

```
Right = glm::normalize(glm::cross(Front, Up));
```

至此，基本的功能就已经实现了。但是还有鼠标的操作，涉及到了欧拉角。欧拉角共有三种：俯仰角(Pitch)，偏航角(Yaw)以及滚转角(Roll)，本题中我们只涉及到前两种。其中俯仰角负责上下旋转，但是我们规定了不得超过+-89度，偏航角负责左右旋转，这个不做限制，代码如下：

```
void Camera::rotate(GLfloat const pitch, GLfloat const yaw) {
    this->pitch += pitch;
    this->yaw += yaw;
    if (this->pitch > 89.0f) {
        this->pitch = 89.0f;
    }
    if (this->pitch < -89.0f) {
        this->pitch = -89.0f;
    }
    updateVectors();
}
```

注意到这里有个 *updateVectors()* 函数，为何之前的WSAD中没有更新向量呢？因为只使用那四个方向键，不会改变Front和Right向量，而改变了欧拉角，则这两个向量都需要更新，所以有这个函数：

```
void Camera::updateVectors() {
    glm::vec3 vectors;
    vectors.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    vectors.y = sin(glm::radians(pitch));
    vectors.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    Front = glm::normalize(vectors);
    Right = glm::normalize(glm::cross(Front, Up));
    cameraUp = glm::normalize(glm::cross(Right, Front));
}
```

至此已经基本完成，不过还有个功能就是鼠标的滚轮，滑动可以使物体变大，这个我们结合透视投影也可做到：

```
void Camera::processZoom(GLfloat const zoom) {
    if (this->zoom >= 1.0f && this->zoom <= 45.0f)
        this->zoom -= zoom;
    if (this->zoom <= 1.0f)
        this->zoom = 1.0f;
    if (this->zoom >= 45.0f)
        this->zoom = 45.0f;
}

GLfloat Camera::getZoom() {
    return zoom;
}
```

主函数中如此调用：

```
Camera camera(glm::vec3(0.0f, 0.0f, 3.0f),
    glm::vec3(0.0f, 1.0f, 0.0f),
    glm::vec3(0.0f, 0.0f, -1.0f));
view = camera.getViewMatrix();

projection = glm::perspective(glm::radians(camera.getZoom()), (float)height / (float)width,
nearPos, farPos);
```

```
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.moveForward(deltaTime*SPEED);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.moveBack(deltaTime*SPEED);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.moveLeft(deltaTime*SPEED);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.moveRight(deltaTime*SPEED);
}
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.rotate(xOffset*MOUSESPEED, yOffset*MOUSESPEED);
}
```

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

其中deltaTime是记录了新的一帧与上一帧的时间，是变换更加流畅。

实验结果

详见result3.gif。