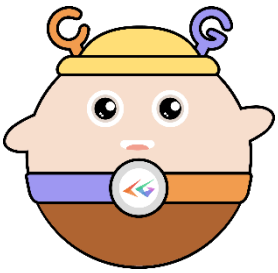
華中科技大學



计算机图形学课程

实验：完成摄像机类的创建

目录

完成摄像机类的创建

1.摄像机/观察空间：

摄像机/观察空间(Camera/View Space)的时候，是在讨论以摄像机的视角作为场景原点时场景中所有的顶点坐标：观察矩阵把所有的世界坐标变换为相对于摄像机位置与方向的观察坐标。

我们在构造摄像机类的时候。着重强调 4 个重要的向量：

- 摄像机位置：摄像机位置简单来说就是世界空间中一个指向摄像机位置的向量；
- 摄像机方向：指的是摄像机指向哪个方向。一般我们让摄像机指向场景原点：(0, 0, 0)；
- 右轴：代表摄像机空间的 x 轴的正方向。为获取右向量我们需要先使用一个小技巧：先定义一个上向量(比如（0，1，0）)。接下来把上向量和第二步得到的方向向量进行叉乘。得到的结果即为右向量；
- 上轴：有了 x 轴向量和 z 轴向量，获取一个指向摄像机的正 y 轴向量就相对简单了：我们把右向量和方向向量进行叉乘得到上轴。

2.Look At 矩阵

LookAt 矩阵就像它的名字表达的那样：它会创建一个看着(Look at)给定目标的观察矩阵。具体形式如下：

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

其中 R 是右向量，U 是上向量，D 是方向向量 P 是摄像机位置向量。GLM 已经提供了这些支持。我们要做的只是定义一个摄像机位置，一个目标位置和一个表示世界空间

中的上向量的向量（我们计算右向量使用的那个上向量）。接着 GLM 就会创建一个 LookAt 矩阵，我们可以把它当作我们的观察矩阵。

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

3.自由移动

现在我们为 GLFW 的键盘输入定义一个 processInput 函数，我们来新添加几个需要检查的按键命令。使用 GLFW 的 glfwGetKey 函数，它需要一个窗口以及一个按键作为输入。这个函数将会返回这个按键是否正在被按下。我们使用 WSAD 来模拟摄像机的前后左右运动。

```
    void processInput(GLFWwindow *window)
    {
        ...
        float cameraSpeed = 0.05f;
        if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
            cameraPos += cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
            cameraPos -= cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
            cameraPos -= glm::normalize(glm::cross(cameraFront,
cameraUp)) * cameraSpeed;
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
            cameraPos += glm::normalize(glm::cross(cameraFront,
cameraUp)) * cameraSpeed;
    }
```

【这里的 cameraSpeed 为预先定义的移动速度，方便控制移动快慢】

目前我们的移动速度是个常量。理论上没什么问题，但是实际情况下根据处理器的能力不同，有些人可能会比其他人每秒绘制更多帧，也就是以更高的频率调用 processInput 函数。

图形程序和游戏通常会跟踪一个时间差(Deltatime)变量，它储存了渲染上一帧所用的时间。我们把所有速度都去乘以 deltaTime 值。结果就是，如果我们的 deltaTime 很大，就意味着上一帧的渲染花费了更多时间，所以这一帧的速度需要变得更高来平衡渲染所花去的时间。使用这种方法时，无论你的电脑快还是慢，摄像机的速度都会相应平衡，这样每个用户的体验就都一样了。

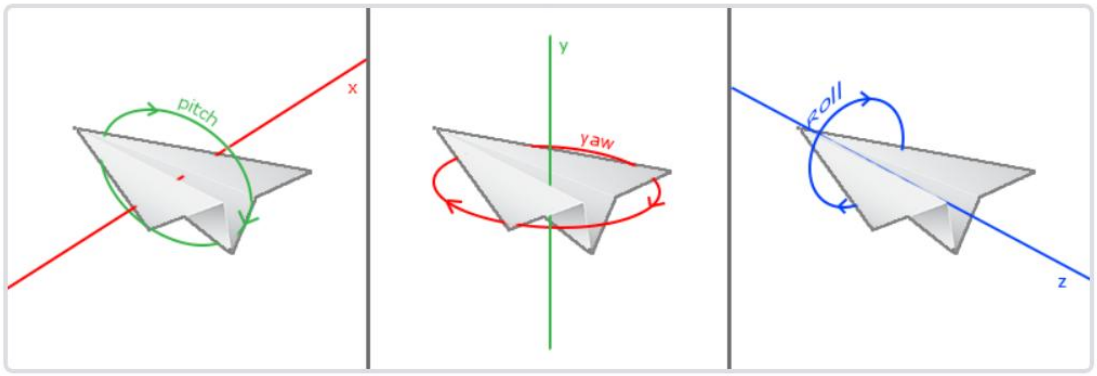
```
float deltaTime = 0.0f; // 当前帧与上一帧的时间差
float lastFrame = 0.0f; // 上一帧的时间
```

```
//每一帧计算出新的deltaTime
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
//从而控制移动速度
float cameraSpeed = 2.5f * deltaTime;
```

4.视角移动

为了能够改变视角，我们需要根据鼠标的输入改变 cameraFront 向量，这里我们将用到数学里的欧拉角的相关知识。

欧拉角是可以表示 3D 空间中任何旋转的 3 个值，由莱昂哈德·欧拉(Leonhard Euler)在 18 世纪提出。一共有 3 种欧拉角：俯仰角(Pitch)、偏航角(Yaw)和滚转角(Roll)，如下图所示：



俯仰角（Pitch）是描述我们如何往上或往下看的角，可以在第一张图中看到。第二张图展示了偏航角(Yaw)，偏航角表示我们往左和往右看的程度。滚转角(Roll)代表

我们如何翻滚摄像机，欧拉角都有一个值来表示，把三个角结合起来我们就能够计算3D空间中任何的旋转向量.

【对于摄像机系统来说，我们只关心俯仰角和偏航角，所以我们不会在代码中加入滚转角】

5.鼠标输入

偏航角和俯仰角是通过鼠标移动获得的，水平的移动影响偏航角，竖直的移动影响俯仰角。通过储存上一帧鼠标的位置，在当前帧中我们当前计算鼠标位置与上一帧的位置相差多少。

首先我们要告诉GLFW，它应该隐藏光标，并捕捉(Capture)它。捕捉光标表示的是，如果焦点在你的程序上，光标应该停留在窗口中（除非程序失去焦点或者退出）。我们可以用GLFW为我们提供的方法来配置：

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

让GLFW监听鼠标移动事件。（和键盘输入相似）使用的函数如下：

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos;
    lastX = xpos;
    lastY = ypos;

    yaw += xoffset;
    pitch += yoffset;
// pitch 和 yaw 的变化
    glm::vec3 front;
    front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    front.y = sin(glm::radians(pitch));
    front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    cameraFront = glm::normalize(front);
}
```

这里我们还引入了一个量 sensitivity（灵敏度）来控制鼠标移动对视角影响的幅度。

```
    float sensitivity = 0.1f;
        xoffset *= sensitivity;
        yoffset *= sensitivity;
```

其次我们需要添加一些限制，防止摄像机发生奇怪的移动（对于俯仰角，要让用户不能看向高于 89 度的地方，因为在 90 度时视角会发生逆转，所以我们把 89 度作为极限）

```
    if (pitch > 89.0f)
            pitch = 89.0f;
        if (pitch < -89.0f)
            pitch = -89.0f;
```

如果你现在运行代码，你会发现在窗口第一次获取焦点的时候摄像机会突然跳一下。想一想原因是什么？

（在你的鼠标移动进窗口的那一刻，鼠标回调函数就会被调用，这时候的 xpos 和 ypos 会等于鼠标刚刚进入屏幕的那个位置。这通常是一个距离屏幕中心很远的地方，因而产生一个很大的偏移量，所以就会跳了。我们可以简单的使用一个 bool 变量检验我们是否是第一次获取鼠标输入，如果是，那么我们先把鼠标的初始位置更新为 xpos 和 ypos 值，这样就能解决这个问题；接下来的鼠标移动就会使用刚进入的鼠标位置坐标来计算偏移量了）

```
    if (firstMouse)
        {
            lastX = xpos;
            lastY = ypos;
            firstMouse = false;
        }
```

6.缩放

作为我们摄像机系统的一个附加内容，我们还会来实现一个缩放(Zoom)接口。在之前的教程中我们说视野(Field of View)或 fov 定义了我们可以看到场景中多大的范围。当视野变小时，场景投影出来的空间就会减小，产生放大(Zoom In)了的感觉。我们会使用鼠标的滚轮来放大，和鼠标移动视角一样，我们规定鼠标滚轮事件实现视野的方法和缩小：

```
    void scroll_callback(GLFWwindow* window, double xoffset, double
yoffset)
    {
      if(fov >= 1.0f && fov <= 45.0f)
        fov -= yoffset;
      if(fov <= 1.0f)
        fov = 1.0f;
      if(fov >= 45.0f)
        fov = 45.0f;
    }
```

其中，45.0f 是默认设置的视野值。

同样之前记得用 GLFW 相应函数注册监控滚轮事件

```
    glfwSetScrollCallback(window, scroll_callback);
```

这样我们的摄像机类就完成了，运行程序后你可以通过 WASD 和鼠标控制你的视角。