

摄像机

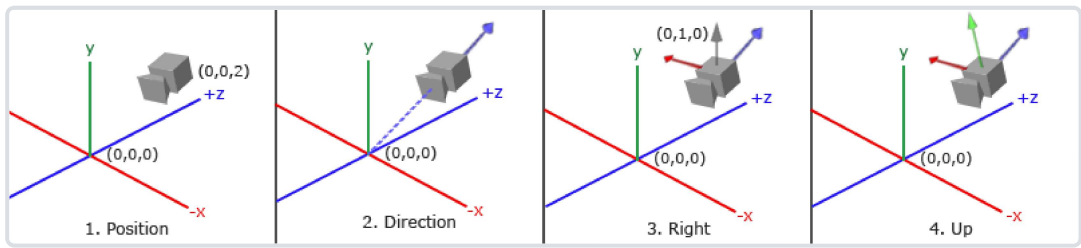
原文	Camera (http://learnopengl.com/#!Getting-started/Camera)
作者	JoeyDeVries
翻译	Django (http://bullteacher.com/), Meow J, Geequilm, BLumia (https://github.com/blumia/)
校对	暂未校对

前面的教程中我们讨论了观察矩阵以及如何使用观察矩阵移动场景（我们向后移动了一点）。OpenGL本身没有**摄像机(Camera)**的概念，但我们可以通过把场景中的所有物体往相反方向移动的方式来模拟出摄像机，产生一种**我们在移动**的感觉，而不是场景在移动。

本节我们将会讨论如何在OpenGL中配置一个摄像机，并且将会讨论FPS风格的摄像机，让你能够在3D场景中自由移动。我们也会讨论键盘和鼠标输入，最终完成一个自定义的摄像机类。

摄像机/观察空间

当我们讨论摄像机/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。要定义一个摄像机，我们需要它在世界空间中的位置、观察的方向、一个指向它右测的向量以及一个指向它上方的向量。细心的读者可能已经注意到我们实际上创建了一个三个单位轴相互垂直的、以摄像机的位置为原点的坐标系。



1. 摄像机位置

获取摄像机位置很简单。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。我们把摄像机位置设置为上一节中的那个相同的位置：

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```

不要忘记正z轴是从屏幕指向你的，如果我们希望摄像机向后移动，我们就沿着z轴的正方向移动。

2. 摄像机方向

下一个需要的向量是摄像机的方向，这里指的是摄像机指向哪个方向。现在我们让摄像机指向场景原点：(0, 0, 0)。还记得如果将两个矢量相减，我们就能得到这两个矢量的差吗？用场景原点向量减去摄像机位置向量的结果就是摄像机的指向向量。由于我们知道摄像机指向z轴负方向，但我们希望方向向量(Direction Vector)指向摄像机的z轴正方向。如果我们交换相减的顺序，我们就会获得一个指向摄像机正z轴方向的向量：

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```

方向向量(Direction Vector)并不是最好的名字，因为它实际上指向从它到目标向量的相反方向（译注：注意看前面的那个图，蓝色的方向向量大概指向z轴的正方向，与摄像机实际指向的方向是正好相反的）。

3. 右轴

Loading Mathjax/jax/output/HTML-CSS/jax.js

我们需要的另一个向量是一个**右向量**(Right Vector)，它代表摄像机空间的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**(Up Vector)。接下来把上向量和第二步得到的方向向量进行叉乘。两个向量叉乘的结果会同时垂直于两向量，因此我们会得到指向x轴正方向的那个向量（如果我们交换两个向量叉乘的顺序就会得到相反的指向x轴负方向的向量）：

```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```

4. 上轴

现在我们已经有了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：

```
glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
```

在叉乘和一些小技巧的帮助下，我们创建了所有构成观察/摄像机空间的向量。对于想学到更多数学原理的读者，提示一下，在线性代数中这个处理叫做格拉姆—施密特正交化 (http://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process)(Gram-Schmidt Process)。使用这些摄像机向量我们就可以创建一个LookAt矩阵了，它在创建摄像机的时候非常有用。

Look At

使用矩阵的好处之一是如果你使用3个相互垂直（或非线性）的轴定义了一个坐标空间，你可以用这3个轴外加一个平移向量来创建一个矩阵，并且你可以用这个矩阵乘以任意向量来将其变换到那个坐标空间。这正是**LookAt**矩阵所做的，现在有了3个相互垂直的轴和一个定义摄像机空间的位置坐标，我们可以创建我们自己的LookAt矩阵了：

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

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。注意，位置向量是相反的，因为我们最终希望把世界平移到与我们自身移动的反方向。把这个LookAt矩阵作为观察矩阵可以很高效地把所有世界坐标变换到刚刚定义的观察空间。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。

幸运的是，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 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
    glm::vec3(0.0f, 0.0f, 0.0f),
    glm::vec3(0.0f, 1.0f, 0.0f));
```

`glm::LookAt`函数需要一个位置、目标和上向量。它会创建一个和在上一节使用的一样的观察矩阵。

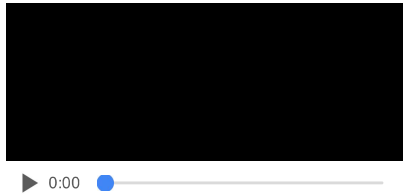
在讨论用户输入之前，我们先来做些有意思的事，把我们的摄像机在场景中旋转。我们会将摄像机的注视点保持在(0, 0, 0)。

我们需要用到一点三角学的知识来在每一帧创建一个x和z坐标，它会代表圆上的一点，我们将会使用它作为摄像机的位置。通过重新计算x和y坐标，我们会遍历圆上的所有点，这样摄像机就会绕着场景旋转了。我们预先定义这个圆的半径`radius`，在每次渲染迭代中使用GLFW的`glfwGetTime`函数重新创建观察矩阵，来扩大这个圆。

```
float radius = 10.0f;
float camX = sin(glfwGetTime()) * radius;
float camZ = cos(glfwGetTime()) * radius;
glm::mat4 view;
view = glm::lookAt(glm::vec3(camX, 0.0, camZ), glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0, 1.0, 0.0));
```

如果你运行代码，应该会得到下面的结果：

Loading [MathJax]/jax/output/HTML-CSS/jax.js



通过这一小段代码，摄像机现在会随着时间流逝围绕场景转动了。自己试试改变半径和位置/方向参数，看看**LookAt**矩阵是如何工作的。同时，如果你在哪卡住的话，这里有源码 (https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/7.1.camera_circle/camera_circle.cpp)。

自由移动

让摄像机绕着场景转的确很有趣，但是让我们自己移动摄像机会更有趣！首先我们必须设置一个摄像机系统，所以在我们的程序前面定义一些摄像机变量很有用：

```
glm::vec3 cameraPos    = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront  = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp     = glm::vec3(0.0f, 1.0f, 0.0f);
```

LookAt 函数现在成了：

```
view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

我们首先将摄像机位置设置为之前定义的**cameraPos**。方向是当前的位置加上我们刚刚定义的方向向量。这样能保证无论我们怎么移动，摄像机都会注视着目标方向。让我们摆弄一下这些向量，在按下某些按钮时更新**cameraPos**向量。

我们已经为GLFW的键盘输入定义过一个**processInput**函数了，我们来新添加几个需要检查的按键命令：

```
void processInput(GLFWwindow *window)
{
    ...
    float cameraSpeed = 0.05f; // adjust accordingly
    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;
}
```

当我们按下**WASD**键的任意一个，摄像机的位置都会相应更新。如果我们希望向前或向后移动，我们就把位置向量加上或减去方向向量。如果我们希望向左右移动，我们使用叉乘来创建一个**右向量**(Right Vector)，并沿着它相应移动就可以了。这样就创建了使用摄像机时熟悉的横移(Strafe)效果。

注意，我们对**右向量**进行了标准化。如果我们没对这个向量进行标准化，最后的叉乘结果会根据**cameraFront**变量返回大小不同的向量。如果我们不对向量进行标准化，我们就得根据摄像机的朝向不同加速或减速移动了，但如果进行了标准化移动就是匀速的。

现在你就应该能够移动摄像机了，虽然移动速度和系统有关，你可能会需要调整一下**cameraSpeed**。

移动速度

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。结果就是，根据配置的不同，有些人可能移动很快，而有些人会移动很慢。当你发布你的程序的时候，你必须确保它在所有硬件上移动速度都一样。

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。

我们跟踪两个全局变量来计算出deltaTime值：

```
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
```

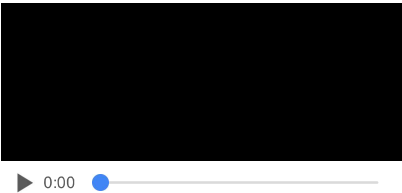
在每一帧中我们计算出新的deltaTime以备后用。

```
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

现在我们有deltaTime，在计算速度的时候可以将其考虑进去了：

```
void processInput(GLFWwindow *window)
{
    float cameraSpeed = 2.5f * deltaTime;
    ...
}
```

与前面的部分结合在一起，我们有了一个更流畅点的摄像机系统：



现在我们有了一个在任何系统上移动速度都一样的摄像机。同样，如果你卡住了，查看一下源码 (https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/7.2.camera_keyboard_dt/camera_keyboard_dt.cpp)。我们可以看到任何移动都会影响返回的deltaTime值。

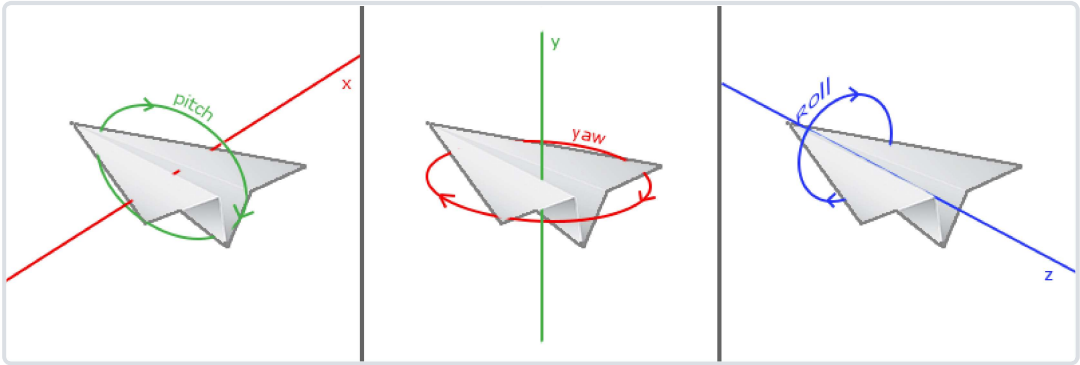
视角移动

只用键盘移动没什么意思。特别是我们还不能转向，移动很受限制。是时候加入鼠标了！

为了能够改变视角，我们需要根据鼠标的输入改变cameraFront向量。然而，根据鼠标移动改变方向向量有点复杂，需要一些三角学知识。如果你对三角学知之甚少，别担心，你可以跳过这一部分，直接复制粘贴我们的代码；当你想了解更多时候再回来看。

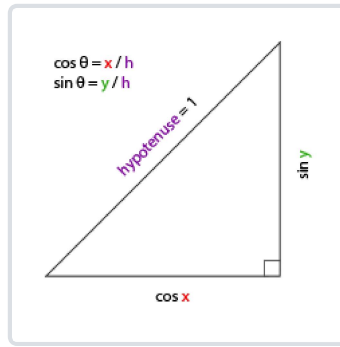
欧拉角

欧拉角(Euler Angle)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)，下面的图片展示了它们的含义：

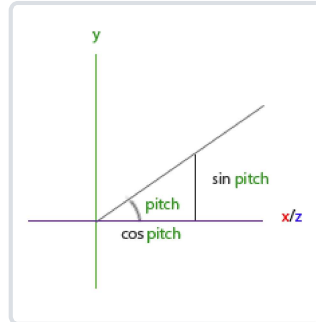


俯仰角是描述我们如何往上或往下看的角，可以在第一张图中看到。第二张图展示了偏航角，偏航角表示我们往左和往右看的程度。滚转角代表我们如何翻滚摄像机，通常在太空飞船的摄像机中使用。每个欧拉角都有一个值来表示，把三个角结合起来我们就能够计算3D空间中任何的旋转向量了。

对于我们的摄像机系统来说，我们只关心俯仰角和偏航角，所以我们不会讨论滚转角。给定一个俯仰角和偏航角，我们可以把它们转换为一个代表新的方向向量的3D向量。俯仰角和偏航角转换为方向向量的处理需要一些三角学知识，我们先从最基本的情况开始：



如果我们把斜边边长定义为1，我们就能知道邻边的长度是 $\cos x/h = \cos x/1 = \cos x$ ，它的对边是 $\sin y/h = \sin y/1 = \sin y$ 。这样我们获得了能够得到x和y方向长度的通用公式，它们取决于所给的角度。我们使用它来计算方向向量的分量：



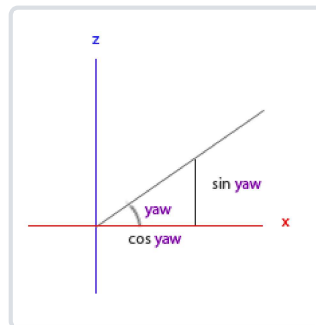
这个三角形看起来和前面的三角形很像，所以如果我们想象自己在xz平面上，看向y轴，我们可以基于第一个三角形计算来计算它的长度/y方向的强度(Strength)（我们往上或往下看多少）。从图中我们可以看到对于一个给定俯仰角的y值等于 $\sin \theta$ ：

```
direction.y = sin(glm::radians(pitch)); // 注意我们先把角度转为弧度
```

这里我们只更新了y值，仔细观察x和z分量也被影响了。从三角形中我们可以看到它们的值等于：

```
direction.x = cos(glm::radians(pitch));
direction.z = cos(glm::radians(pitch));
```

看看我们是否能够为偏航角找到需要的分量：



就像俯仰角的三角形一样，我们可以看到x分量取决于 $\cos(\text{yaw})$ 的值，z值同样取决于偏航角的正弦值。把这个加到前面的值中，会得到基于俯仰角和偏航角的方向向量：

```
direction.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw)); // 译注：direction代表摄像机的前轴(Front)，这个前轴是和本文第一幅图片的第二个摄像机的方向向量是相反的
direction.y = sin(glm::radians(pitch));
direction.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
```

这样我们就有了一个可以把俯仰角和偏航角转化为用来自由旋转视角的摄像机的3维方向向量了。你可能会奇怪：我们怎么得到俯仰角和偏航角？

鼠标输入

Loading [MathJax]/jax/output/HTML-CSS/jax.js

偏航角和俯仰角是通过鼠标（或手柄）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。它的原理就是，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。如果水平/竖直差别越大那么俯仰角或偏航角就改变越大，也就是摄像机需要移动更多的距离。

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（译注：即表示你正在操作这个程序，Windows中拥有焦点的程序标题栏通常是有颜色的那个，而失去焦点的程序标题栏则是灰色的），光标应该停留在窗口中（除非程序失去焦点或者退出）。我们可以用一个简单地配置调用来完成：

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

在调用这个函数之后，无论我们怎么去移动鼠标，光标都不会显示了，它也不会离开窗口。对于FPS摄像机系统来说非常完美。

为了计算俯仰角和偏航角，我们需要让GLFW监听鼠标移动事件。（和键盘输入相似）我们会用一个回调函数来完成，函数的原型如下：

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
```

这里的`xpos`和`ypos`代表当前鼠标的位置。当我们用GLFW注册了回调函数之后，鼠标一移动`mouse_callback`函数就会被调用：

```
glfwSetCursorPosCallback(window, mouse_callback);
```

在处理FPS风格摄像机的鼠标输入的时候，我们必须在最终获取方向向量之前做下面这几步：

1. 计算鼠标距上一帧的偏移量。
2. 把偏移量添加到摄像机的俯仰角和偏航角中。
3. 对偏航角和俯仰角进行最大和最小值的限制。
4. 计算方向向量。

第一步是计算鼠标自上一帧的偏移量。我们必须先在程序中储存上一帧的鼠标位置，我们把它的初始值设置为屏幕的中心（屏幕的尺寸是800x600）：

```
float lastX = 400, lastY = 300;
```

然后在鼠标的回调函数中我们计算当前帧和上一帧鼠标位置的偏移量：

```
float xoffset = xpos - lastX;
float yoffset = lastY - ypos; // 注意这里是相反的，因为y坐标是从底部往顶部依次增大的
lastX = xpos;
lastY = ypos;

float sensitivity = 0.05f;
xoffset *= sensitivity;
yoffset *= sensitivity;
```

注意我们把偏移量乘以了`sensitivity`（灵敏度）值。如果我们忽略这个值，鼠标移动就会太大了；你可以自己实验一下，找到适合自己的灵敏度值。

接下来我们把偏移量加到全局变量`pitch`和`yaw`上：

```
yaw  += xoffset;
pitch += yoffset;
```

第三步，我们需要给摄像机添加一些限制，这样摄像机就不会发生奇怪的移动了（这样也会避免一些奇怪的问题）。对于俯仰角，要让用户不能看向高于89度的地方（在90度时视角会发生逆转，所以我们把89度作为极限），同样也不允许小于-89度。这样能够保证用户只能看到天空或脚下，但是不能超越这个限制。我们可以在值超过限制的时候将其改为极限值来实现：

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

注意我们没有给偏航角设置限制，这是因为我们不希望限制用户的水平旋转。当然，给偏航角设置限制也很容易，如果你愿意可以自己实现。

第四也是最后一步，就是通过俯仰角和偏航角来计算以得到真正的方向向量：

Loading [MathJax]/jax/output/HTML-CSS/jax.js

```
glm::vec3 front;
front.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw));
front.y = sin(glm::radians(pitch));
front.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
cameraFront = glm::normalize(front);
```

计算出来的方向向量就会包含根据鼠标移动计算出来的所有旋转了。由于`cameraFront`向量已经包含在GLM的`lookAt`函数中，我们这就没什么问题了。

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突然跳一下。这个问题产生的原因是，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，那么我们先吧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) // 这个bool变量初始时是设定为true的
{
    lastX = xpos;
    lastY = ypos;
    firstMouse = false;
}
```

最后的代码应该是这样的：

```
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;

    float sensitivity = 0.05;
    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw   += xoffset;
    pitch += yoffset;

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

    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);
}
```

现在我们可以自由地在3D场景中移动了！

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。与鼠标移动、键盘输入一样，我们需要一个鼠标滚轮的回调函数：

Loading [MathJax]/jax/output/HTML-CSS/jax.js

```
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;
}
```

当滚动鼠标滚轮的时候，`yoffset`值代表我们竖直滚动的大小。当`scroll_callback`函数被调用后，我们改变全局变量`fov`变量的内容。因为`45.0f`是默认的视野值，我们将会把缩放级别(Zoom Level)限制在`1.0f`到`45.0f`。

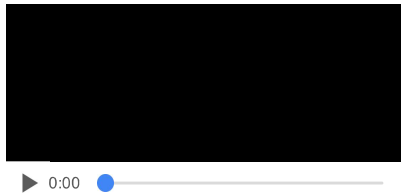
我们现在在每一帧都必须把透视投影矩阵上传到GPU，但现在使用`fov`变量作为它的视野：

```
projection = glm::perspective(glm::radians(fov), 800.0f / 600.0f, 0.1f, 100.0f);
```

最后不要忘记注册鼠标滚轮的回调函数：

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

现在，我们就实现了一个简单的摄像机系统了，它能够让我们在3D环境中自由移动。



你可以去自由地实验，如果遇到困难，可以对比源代码 ([https://learnopengl.com/code_viewer_gh.php?](https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/7.3.camera_mouse_zoom/camera_mouse_zoom.cpp)

`code=src/1.getting_started/7.3.camera_mouse_zoom/camera_mouse_zoom.cpp`)。

注意，使用欧拉角的摄像机系统并不完美。根据你的视角限制或者是配置，你仍然可能引入万向节死锁 (http://en.wikipedia.org/wiki/Gimbal_lock)问题。最好的摄像机系统是使用四元数(Quaternions)的，但我们将会把这个留到后面讨论。（译注：这里 (https://github.com/cybercser/OpenGL_3_3_Tutorial_Translation/blob/master/Tutorial%2017%20Rotations.md)可以查看四元数摄像机的实现)

摄像机类

接下来的教程中，我们将会一直使用一个摄像机来浏览场景，从各个角度观察结果。然而，由于一个摄像机会占用每篇教程很大的篇幅，我们将会从细节抽象出来，创建我们自己的摄像机对象，它会完成大多数的工作，而且还会提供一些附加的功能。与着色器教程不同，我们不会带你一步一步创建摄像机类，我们只会提供你一份（有完整注释的）代码，如果你想知道它的内部构造的话可以自己去阅读。

和着色器对象一样，我们把摄像机类写在一个单独的头文件中。你可以在这里 ([https://learnopengl.com/code_viewer_gh.php?](https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/camera.h)

`code=includes/learnopengl/camera.h`)找到它，你现在应该能够理解所有的代码了。我们建议您至少看一看这个类，看看如何创建一个自己的摄像机类。

我们介绍的摄像机系统是一个FPS风格的摄像机，它能够满足大多数情况需要，而且与欧拉角兼容，但是在创建不同的摄像机系统，比如飞行模拟摄像机，时就要当心。每个摄像机系统都有自己的优点和不足，所以确保对它们进行了详细研究。比如，这个FPS摄像机不允许俯仰角大于90度，而且我们使用了一个固定的上向量(0, 1, 0)，这在需要考虑滚转角的时候就不能用了。

使用新摄像机对象，更新后版本的源码可以在这里 ([https://learnopengl.com/code_viewer_gh.php?](https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/7.4.camera_class/camera_class.cpp)

`code=src/1.getting_started/7.4.camera_class/camera_class.cpp`)找到。

练习

Loading [MathJax]/jax/output/HTML-CSS/jax.js

- 看看你是否能够修改摄像机类，使得其能够变成一个**真正的FPS**摄像机（也就是说不能够随意飞行）；你只能呆在xz平面上：参考解答 (https://learnopengl.com/code_viewer.php?code=getting-started/camera-exercise1)
- 试着创建你自己的LookAt函数，其中你需要手动创建一个我们在一开始讨论的观察矩阵。用你的函数实现来替换GLM的LookAt函数，看看它是否还能一样地工作：参考解答 (https://learnopengl.com/code_viewer.php?code=getting-started/camera-exercise2)

Powered by MkDocs (<http://www.mkdocs.org/>) and Yeti (<http://bootswatch.com/yeti/>)