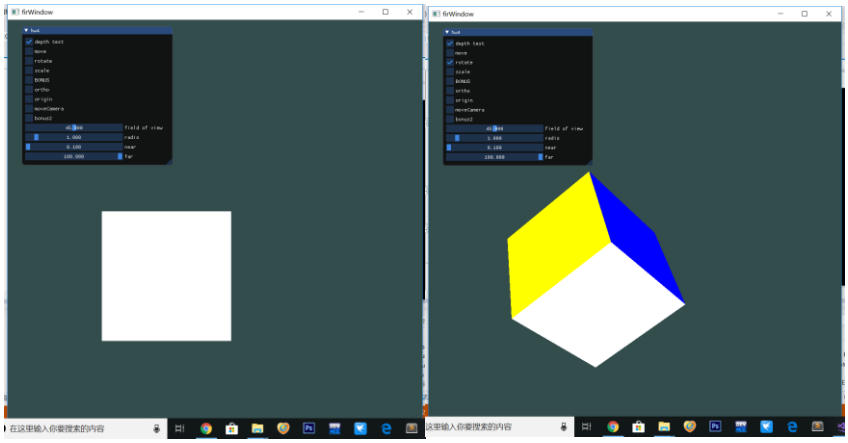
1. 投影(Projection):

（1） 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

这个做法比较简单，放在不同位置的话就直接通过 model 矩阵平移到那个位置就行了，顶点矩阵不用变。

然后要求 6 个面颜色不一样的话，就把顶点矩阵中每个面的所有点都设置成同一个颜色（一个面 6 个点），然后修改一下 shader 的代码，顶点着色器中把输入的 color 变成 vec4 直接输出到片段着色器，然后片段着色器直接用这个颜色输出，不用进行修改。



着色器代码:

```
1.  const char *vertexShaderSource = "#version 330 core\n"
2.              "layout (location = 0) in vec3 aPos;\n"
3.              "layout (location = 1) in vec3 color;\n"
4.
5.              "out vec4 outColor;\n"
6.
7.              "uniform mat4 model;\n"
8.              "uniform mat4 view;\n"
9.              "uniform mat4 projection;\n"
10.             "void main()\n"
11.             "{\n"
12.             "   gl_Position = projection * view * model * vec4(aPos, 1.0f);\
    n"
13.             "   outColor = vec4(color, 1.0f);\n"
14.             "}\0";
15.
16.         const char *fragmentShaderSource = "#version 330 core\n"
17.             "out vec4 FragColor;\n"
18.             "in vec4 outColor;\n"
19.
20.             "void main()\n"
21.             "{\n"
```

```
22.             "   FragColor = outColor;\n"
23.             "}\n\0";
```

顶点矩阵：位置和颜色

```
1.  //顶点矩阵
2.      float vertices[] = {
3.          -2.f, -2.f, -2.f,  0.0f, 0.0f, 0.0f,
4.          2.f, -2.f, -2.f,  0.0f, 0.0f, 0.0f,
5.          2.f,  2.f, -2.f,  0.0f, 0.0f, 0.0f,
6.          2.f,  2.f, -2.f,  0.0f, 0.0f, 0.0f,
7.          -2.f,  2.f, -2.f,  0.0f, 0.0f, 0.0f,
8.          -2.f, -2.f, -2.f,  0.0f, 0.0f, 0.0f,
```

还要修改这里

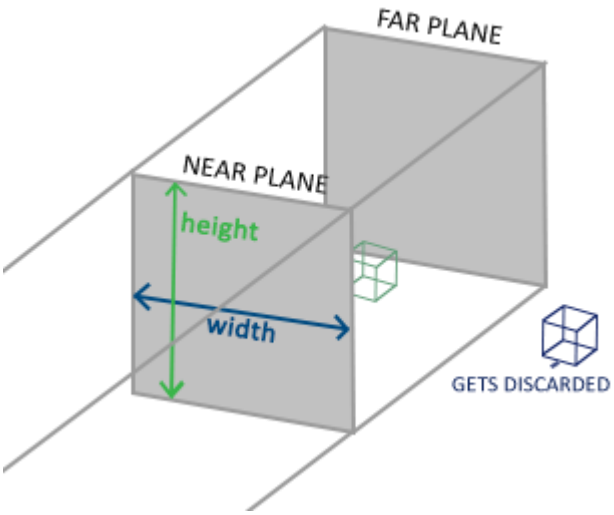
```
1.  //openGL 顶点数据要怎么看
2.      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void
    *)0);
3.      glEnableVertexAttribArray(0);
4.
5.      glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void
    *)(3 * sizeof(float)));
6.      glEnableVertexAttribArray(1);
```

model 矩阵

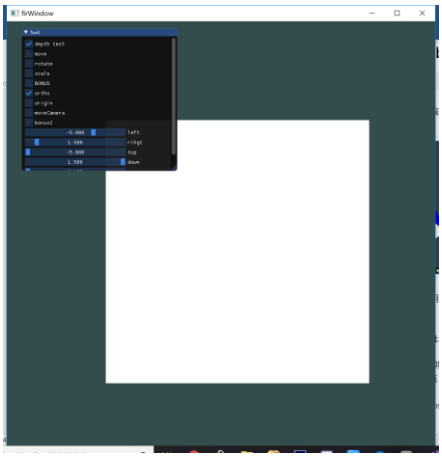
```
1.  model = glm::translate(model, glm::vec3((-1.5f, 0.5f, -1.5f)));
```

（2） 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

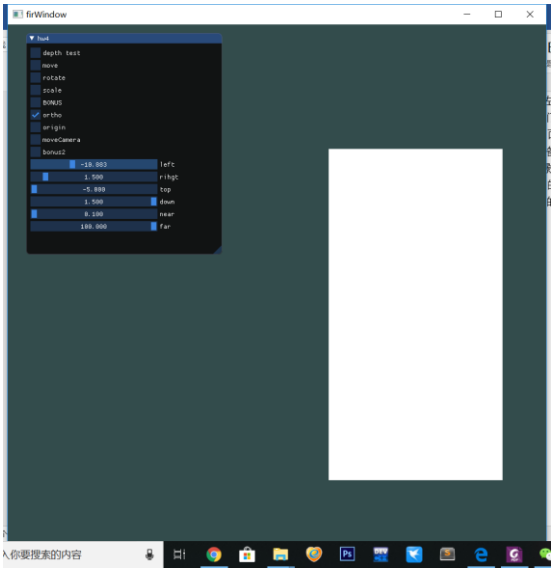
就直接用 projection = glm::ortho(orthoLeft, orthoRight, orthoDown, orthoTop, orthoNear, orthoFar);来修改 projection 矩阵就行了。



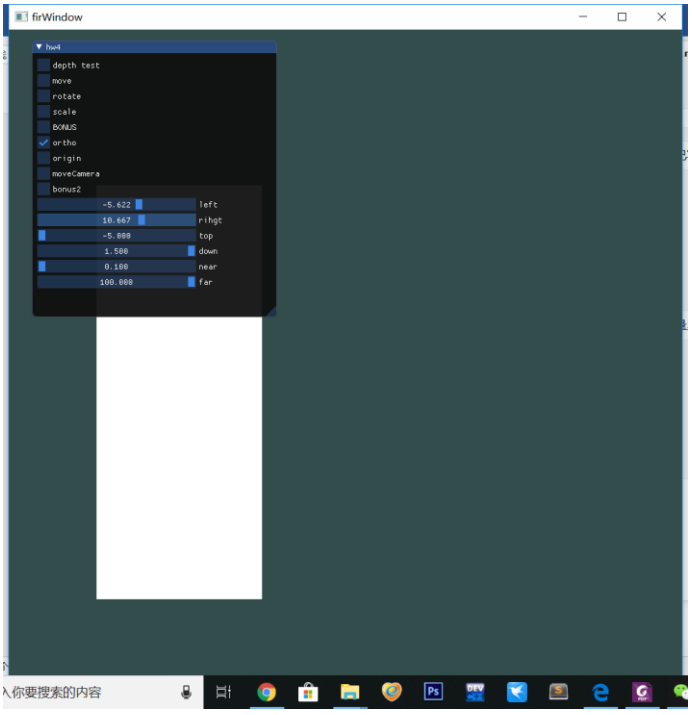
参数的含义就像我写的那样前两个参数指定了平截头体的左右坐标，第三和第四参数指定了平截头体的底部和顶部。通过这四个参数我们定义了近平面和远平面的大小，然后第五和第六个参数则定义了近平面和远平面的距离。这个投影矩阵会将处于这些 x，y，z 值范围内的坐标变换为标准化设备坐标。如果进行调整的话，看到的正方体可能会发生变形，即投影面的宽高会影响到看到的投影正方形的形状和大小，如果最近平面比正方体的 z 坐标还要远的话，就会看不到了，同理最远平面也是，所以说正交投影只能看到两个平面间的，其它都会被截掉。



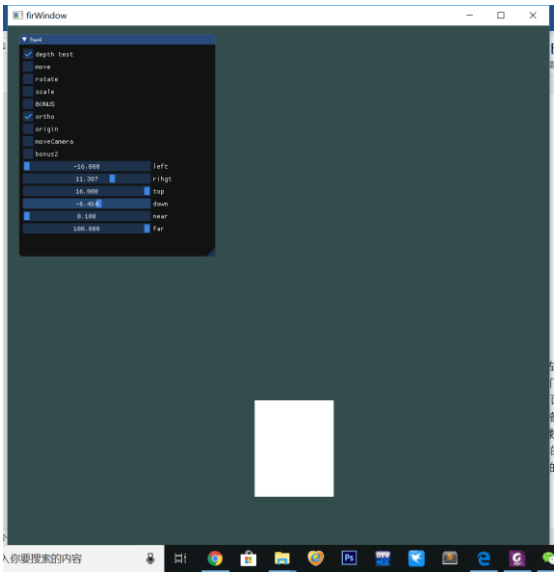
减小 left



增大 right



调整 top 和 down

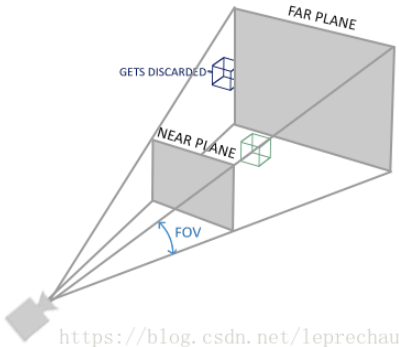


正射投影矩阵直接将坐标映射到 2D 平面中，就是屏幕中。

代码

```
1.  if (ortho)
2.              {
3.                  ImGui::SliderFloat("left", &orthoLeft, -16.0f, 0.0f);
4.                  ImGui::SliderFloat("rihgt", &orthoRight, 0.0f, 16.0f);
5.                  ImGui::SliderFloat("top", &orthoTop, 0.0f, 16.0f);
6.                  ImGui::SliderFloat("down", &orthoDown, -16.0f, 0.0f);
7.                  ImGui::SliderFloat("near", &orthoNear, 0.1f, 50.0f);
8.                  ImGui::SliderFloat("far", &orthoFar, 50.0f, 100.0f);
9.              }
10.   。。。。。。省略
11. if (ortho)
12.         {
13.             projection = glm::ortho(orthoLeft, orthoRight, orthoDown, orthoT
    op, orthoNear, orthoFar);
14.         }
```

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

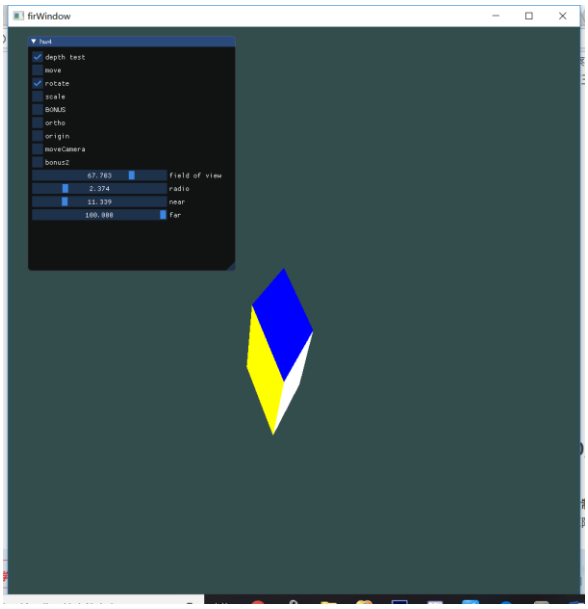


透视投影与我们平时所观察到的现象一致:，由于透视，两条平行线在很远的地方看起来会相交。如下图下面的本来的平行线会变得看上去不平行。

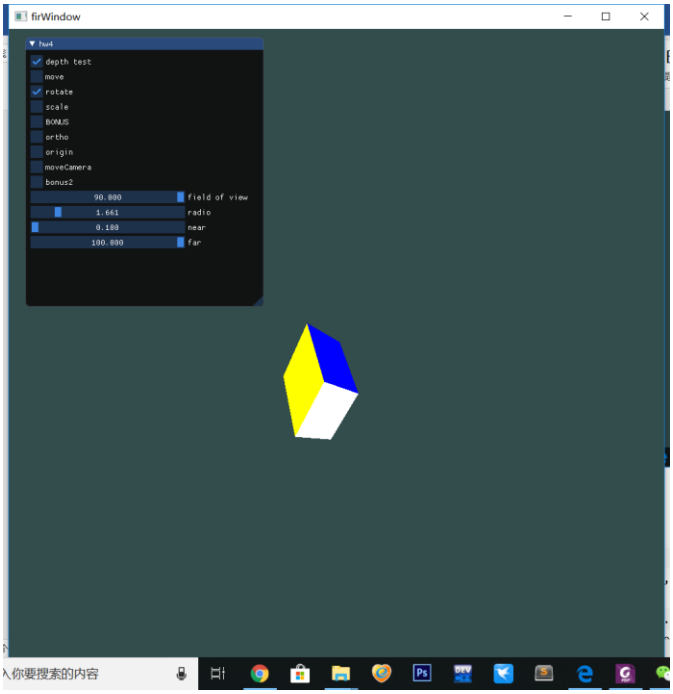
projection = glm::perspective(glm::radians(perView), radio, perNear, perFar);

使用的函数是这个，然后第一个参数是视角，可以进行调整视野观察空间大小，一般是 45 度，第二个是宽高比由视口的宽除以高所得，第三和第四个参数设置了近和远平面，如果超出这两个平面间的椎体范围同样是看不到的

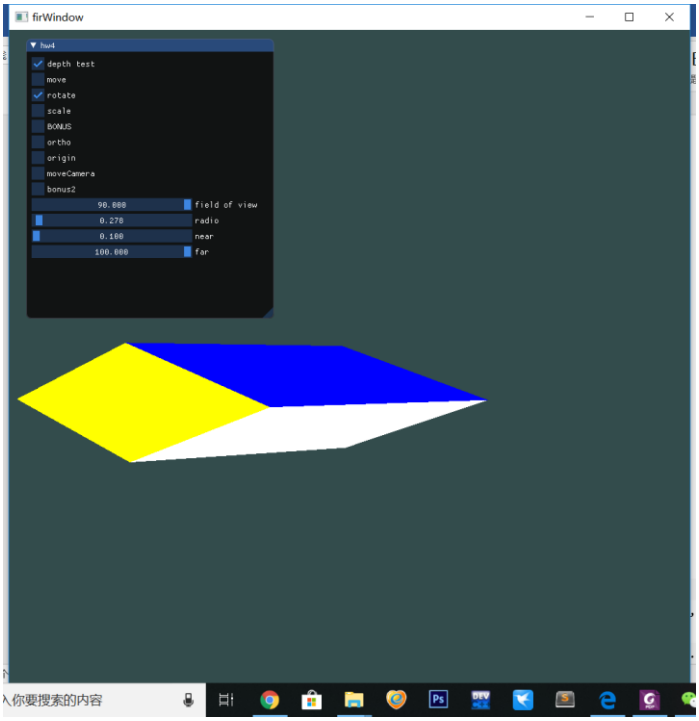
出现不平行的情况:



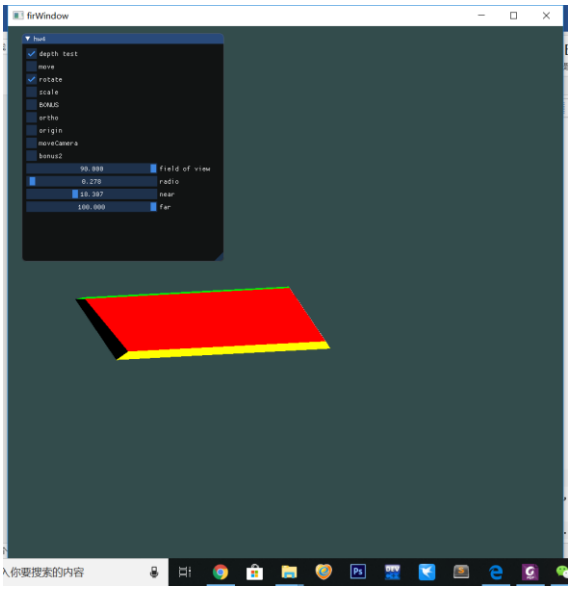
增大视角会看的比较小



改变长宽比会发生形变



当近平面比第一个面要近就看不到他了



代码:

```
else
            {
                ImGui::SliderFloat("field of view", &perView, 0.0f, 90.0f);

                ImGui::SliderFloat("radio", &radio, 0.1f, 10.0f);
                ImGui::SliderFloat("near", &perNear, 0.1f, 50.0f);
                ImGui::SliderFloat("far", &perFar, 50.0f, 100.0f);
            }

else
        {
            projection = glm::perspective(glm::radians(perView), radio, perNear, perFar);
        }
```

2. 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

   这个的话用到了 lookat 函数,

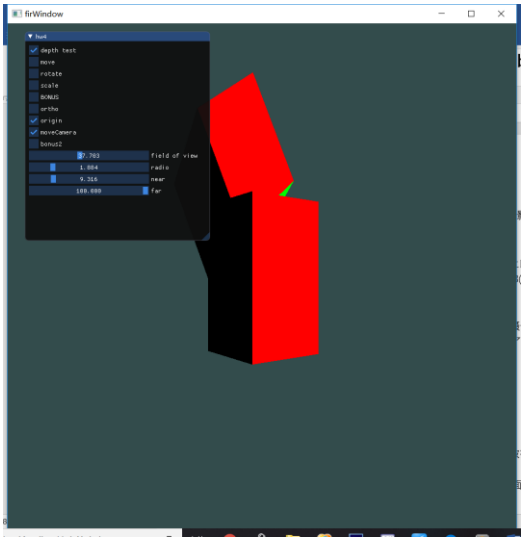
   //第一个是摄像机位置，第二个是目标位置，第三个是向上向量

```
 view = glm::lookAt(glm::vec3(camX, 0.0f, camZ), glm::vec3(0, 0, 0), glm::vec3(0.0f, 1.0f, 0.0f));
```

目标位置始终是正方体的中心，然后向上向量也不变，然后摄像机位置就可以随时间进行改变，获取时间然后在一个 xoz 平面的圆上进行移动就行了。

```
1.  float radius = 20.0f;
2.  float camX = sin(glfwGetTime())*radius;
3.  float camZ = cos(glfwGetTime())*radius;
```

我还顺手画多了一个正方体，但是看起来还是和正方体自己旋转差不多的效果。



3. 在 GUI 里添加菜单栏，可以选择各种功能。
加了几个 checkbox 来执行不同的操作，在正交投影和透视投影中都能够对函数调用的参数进行调整（slidebar），来看不同的参数是什么效果。

4. 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）

首先根据老师上课所说，如果将两个矩阵分开，那么在计算时如果先将顶点矩阵和 model 相乘计算，再和 view 相乘计算，这个计算量是要远远比先通过 view 和 model 进行计算再和顶点矩阵计算的。

如果有两个摄像机的话，就可以快速的通过两个 modelView 矩阵转换成不同摄像坐标系的坐标，而如果是分开的话，就需要两个 model 和两个 view 矩阵来（或者是 1 个 model 两个 view）这样的话就要存储多一个矩阵，如果是有很多个摄像机的话，就要存储多很多矩阵，所以合起来的话不仅比较方便易读，而且能节省空间。

Bonus：

1. 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"),
   简而言之，这个 camera 类是用来返回 view 矩阵的，通过不同的按键或鼠标的事件，在类内部计算出不同的 view 矩阵输出，用于坐标转换。

   键盘输入的话，可以用 glfw 的 getKey 函数来判断事件

```
if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    {
        cam->moveForward(0.1);
    }
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    {
        cam->moveBack(0.1);
    }
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    {
        cam->moveLeft(0.1);
    }
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    {
        cam->moveRight(0.1);
    }
```

   然后鼠标移动事件的话，我是用 glfw 的 getCusorPos 函数，没有用那个回调函数

```
double xpos = 0, ypos = 0;
    glfwGetCursorPos(window, &xpos, &ypos);
    cam->rotate(xpos, ypos);
```

   类内部的实现的话，根据那个教程来

   在实现键盘移动的时候，要有几个关键的向量

```
glm::vec3 Position;
    glm::vec3 Front;
    glm::vec3 Up;
```

然后 lookat 函数就变成了

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

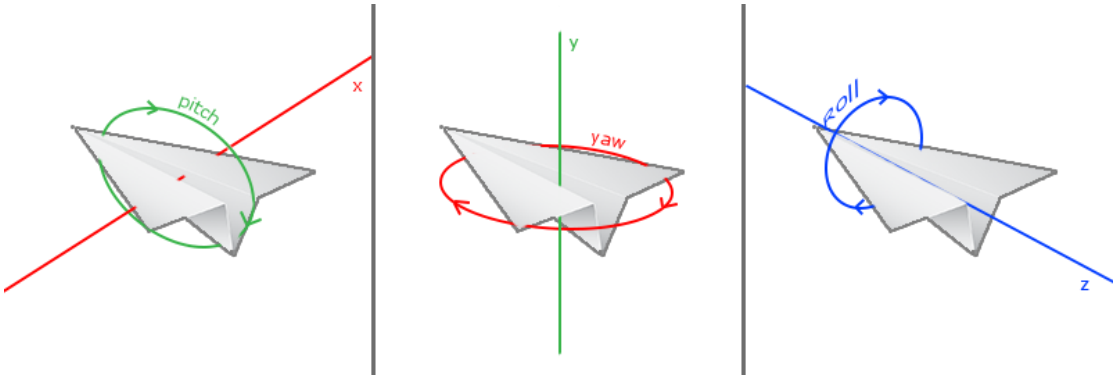
将摄像机位置设置为之前定义的 position。方向是当前的位置加上我们刚刚定义的方向向量。这样能保证无论我们怎么移动，摄像机都会注视着目标方向

接下来的任务就是怎么更新这些向量
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 Front。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（方向向量 front 和向上的向量叉乘，还要标准化，否则可能不同方向的速度不一样,这里向上的向量是指永恒指向 y 正方向的，不随 view 坐标系变化），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效果

```
void moveForward(float const distance)
    {

        Position += Front * distance;
    }
    void moveBack(float const distance)
    {
        Position -= Front *  distance;
    }
    void moveRight(float const distance)
    {
        Position += Right * distance;
    }
    void moveLeft(float const distance)
    {
        Position -= Right * distance;
    }
```

然后在用鼠标旋转
欧拉角(Euler Angle)是可以表示 3D 空间中任何旋转的 3 个值，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：就是沿不同轴旋转

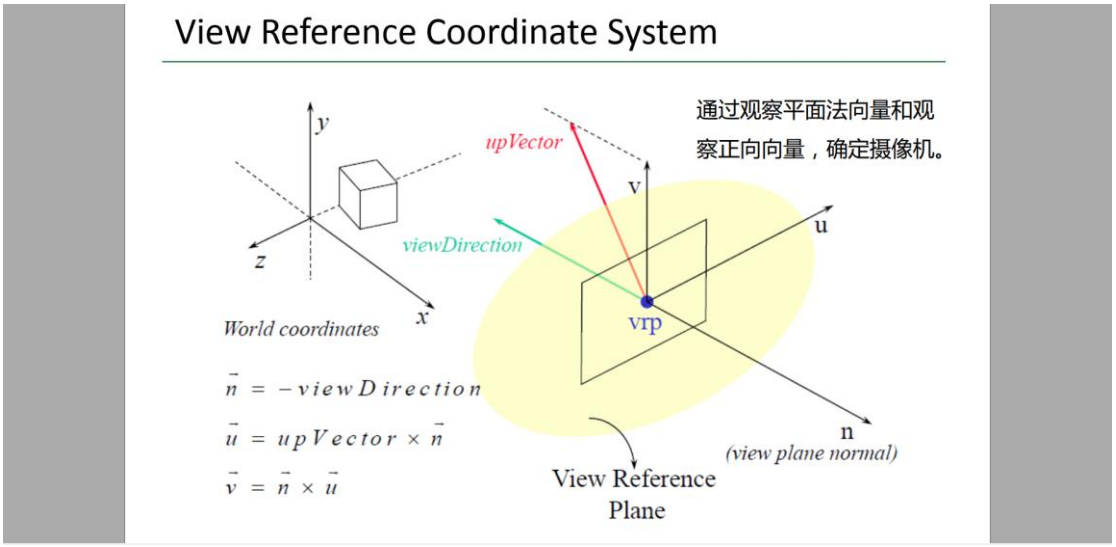


俯仰角是描述我们如何往上或往下看的角，可以在第一张图中看到。第二张图展示了偏航角，偏航角表示我们往左和往右看的程度。滚转角代表我们如何翻滚摄像机，通常在太空飞船的摄像机中使用。每个欧拉角都有一个值来表示，把三个角结合起来我们就能够计算 3D 空间中任何的旋转向量了。

而在这次的作业中，主要要用到俯仰角和偏航角转换为方向向量
经过一系列复杂的三角变换推导就可以根据这两个角得到新的方向向量

```
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));
        Front = glm::normalize(front);
```

然后还要更新这个 view 坐标系下的另外两个基（就是用老师上课讲的叉乘的方法）




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

然后这两个角的计算就是利用 x 和 y 坐标的变化来计算出来的

```
float xoffset = xpos - prexPos;
```

```
        float yoffset = preyPos - 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
        xoffset *= SENSITIVITY;
        yoffset *= SENSITIVITY;

        prexPos = xpos;
        preyPos = ypos;

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

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;
```

接下来就用这两个角来计算出方向向量，紧接着返回 view 矩阵就好了