# 3D引擎学习课程

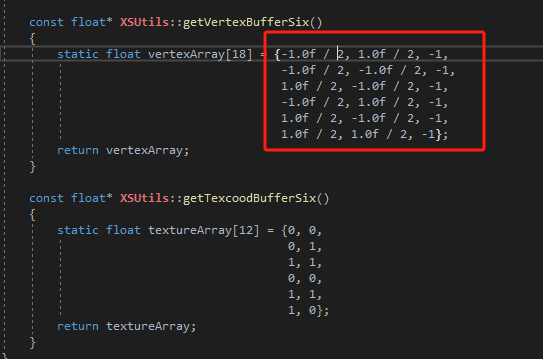
## 知识复习

我们已经知道3D渲染是以“点”为基础，顶点构建起3D世界的轮廓，是骨架。然后纹理是上色。上色过程中可以叠加各种复杂运算，以实现高级的渲染效果。

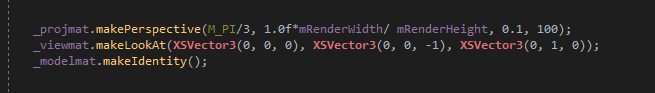
接下来我们干什么？接下来我们绘制最最简单的一个正方形，然后把它调整到屏幕上一个比较容易辨别的位置，比如左上角。虽然计算过程都是GPU内完成的，但是我们可以在CPU端完全模拟出这个计算过程，然后调试看这些计算的中间值，看看它到底是什么。这样我们就更明确的理解了3D渲染中矩阵的变换，以及每一步的结果所代表的物理含义了。

当我们理解了这些中间值的数值含义之后，也不仅仅是加深了对3D渲染的矩阵变换的理解，它也有实际用处——屏幕拾取。下面我们就开始研究。

## 测试用例的数据和效果



以上这一组顶点，构建了一个四边形。它的XY坐标都是±0.5，Z坐标都是-1。这样构建的矩形就是一个边长为1的正方形。便于后面我们跟踪每一个顶点的中间计算结果。

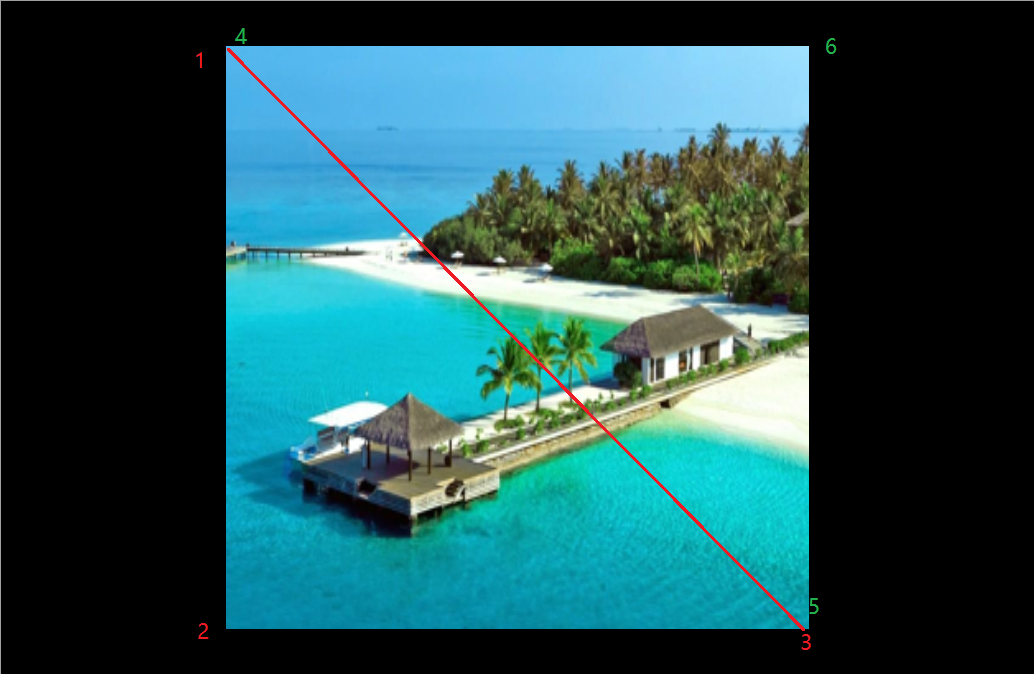


投影矩阵没什么说的，60度的FOV。

视角矩阵是站在原点，看向负Z轴，因为矩形在Z=-1，所以能看见。

模型矩阵是标准矩阵，无变化。

渲染出来的结果，注意图上顶点的连线顺序。我们就跟踪第一个顶点，来观察每一次矩阵变换后它的中间结果。



## 观察中间值

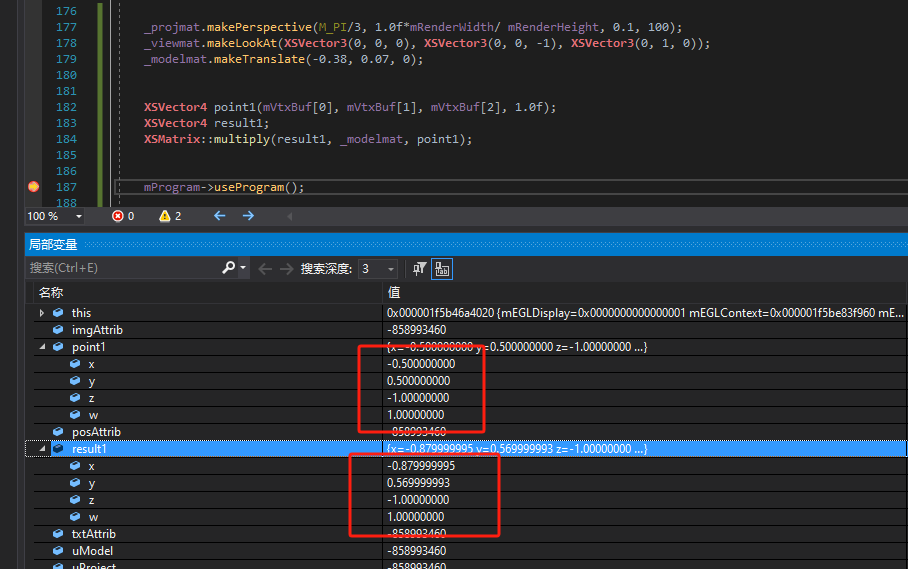
我们手动调节modelmat，让顶点1，接近屏幕左上角。因为屏幕边缘得到的像素坐标要么是0附近，要么是width或者height附近，比较直观。



模型矩添加一个移动，往左-X，往上+Y移动一点点，看下图，图像基本是和边缘重叠了。

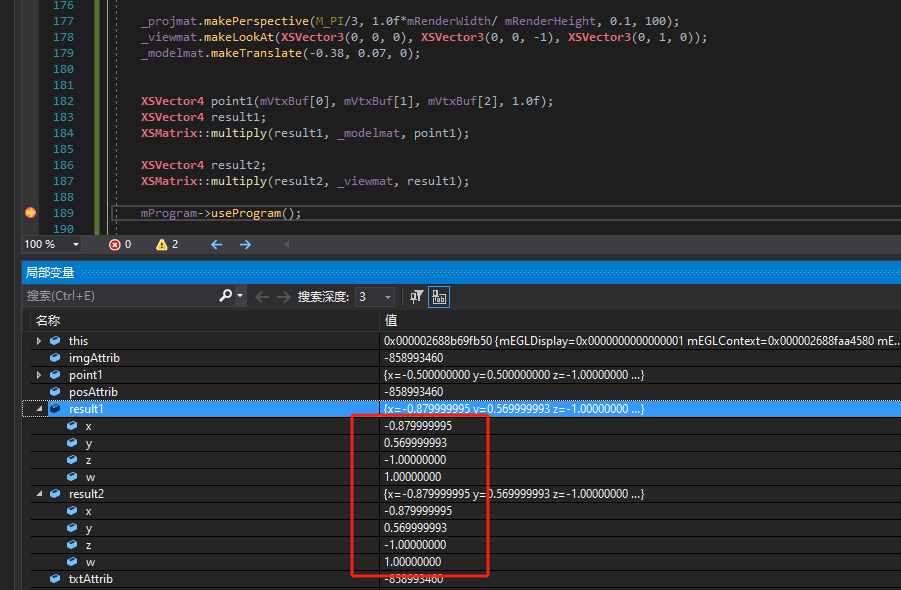


然后我们拿第一个顶点，直接和这些矩阵在CPU相乘，看结果。

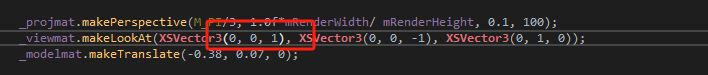


对于modelmat来说，只是添加了一个位移，所以它就是数值上相加即可。

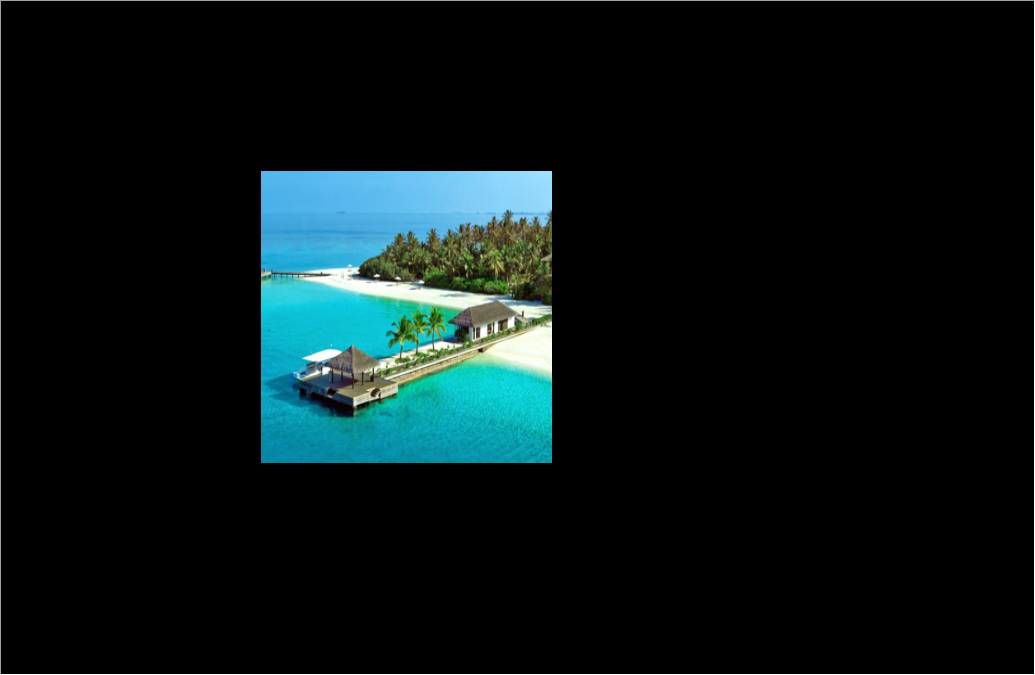
继续往下运算，应用了lookat矩阵之后，居然数值上没变化。这是因为OpenGL的视线矩阵，默认就是这个值。也就是上面的viewmat它刚好是一个标准矩阵。



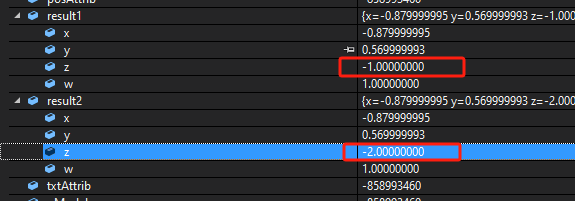
那我们把viewmat改成站在(0, 0, 1)，也就是离正方形远一点。看看效果



先确认渲染效果，你看物体变小(变远)了。



但是在顶点的变换结果上，实际上是顶点，往Z轴负方向退了1个距离。



**这里需要总结一下知识：**

**1、相机的默认位置是pos(0, 0, 0) to(0, 0, -1)， up(0, 1, 0)。**

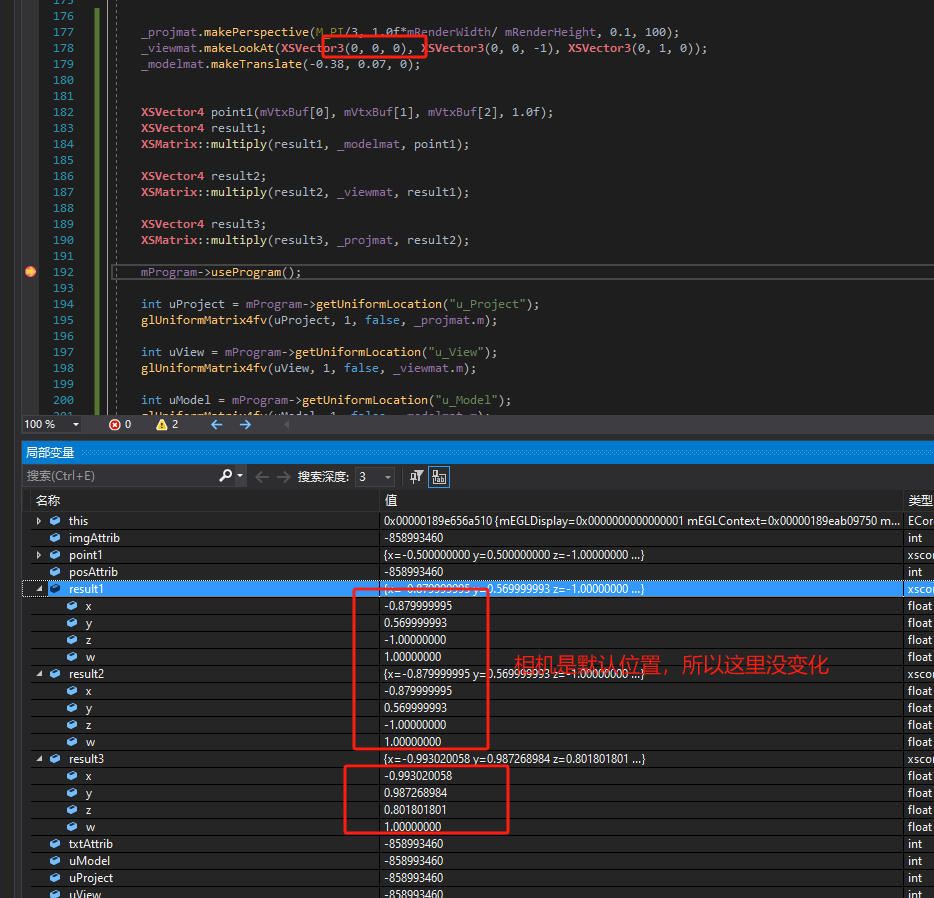
**2、相机矩阵的变换实际上是保存的和这个默认位置的差值。**

**2、物体顶点乘以相机矩阵，实际上是上述差值的逆变换，直接应用到顶点上。也就是最终GPU处理的时候，相机矩阵还是默认值，但是顶点经过了反向变换。**

所以上面的数据，相机位置是(0, 0，1)，顶点位置是（?, ?, -1）。然后变换之后，得到顶点（?, ?, --2）。物理含义就是相对OpenGL的默认相机位置(0, 0, 0)的等效位置是(?, ?, -2)，渲染出来效果是一样的。

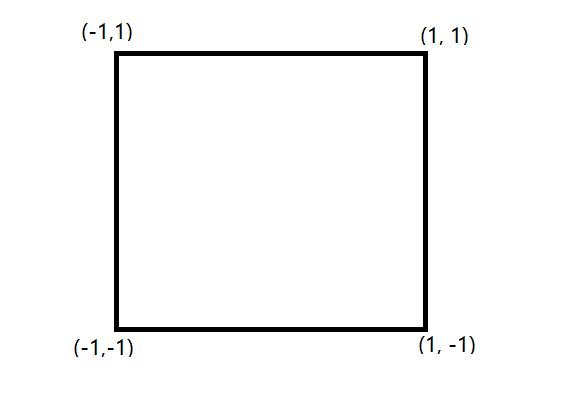
继续继续，我们继续乘以投影矩阵，看看是什么？但在那之前，我们把相机参数还原到之前，模型靠近窗口边缘的值。

这里我们看下result3，它的x和y的值是不是很有意思了。他们都靠近±1。



**那是因为，因为。。。**

顶点乘完所有变换矩阵，最后乘以投影矩阵后的数值，会落在±1的范围内。也就是屏幕坐标是



左上角坐标是(-1, 1)，所以上面顶点的变换结果是(-0.98…，0.98…)。

再然后，OpenGL内部，会根据你定义的viewport，创建一个width\*height的RGBA buffer，然后把上面的顶点的±1的坐标，光栅化填写到buffer里。而超过这个±1的顶点就认为超过了屏幕就不显示了。

那我们验证下这个结论。因为我们顶点定义的刚好是±0.5。如果我们不传任何矩阵，或者传的投影矩阵都是标准矩阵，那图像应该是长宽各占50%，居中。



看结果，是符合预期的。宽和高都占50%，居中显示。因为现在屏幕的坐标实际上是±1，图像坐标是±0.5。



以上就是OpenGL，矩阵变换的底层逻辑。

## 有什么用？

我们已经找到屏幕坐标和OpenGL viewport坐标的对应关系了。那么我们可以反向计算。当鼠标点击窗口的某个点之后，先等比缩放换算到±1的区间。然后乘以逆向的投影矩阵，逆向的model矩阵，逆向的view矩阵，就可以换算出这个点在模型上的位置。哈哈哈，其实是不行的，因为深度在换算过程中被丢弃了。实际使用的时候是屏幕上选取的一个点，加上相机的位置，形成一条射线。我们可以判断这条射线，是否穿过了屏幕中的某个物体。就是通过上述的运算。

实际应用中，一般是先讲模型的顶点变换到世界坐标系。

将屏幕拾取点击的坐标B，加上相机的pos坐标A，得到一个AB向量。然后换算AB在世界坐标系下的方向和起点，然后挨个根据物体的轮廓，判断拾取射线是否穿越了这个物体。就可以达到屏幕拾取的功能了。

## 顶点运算总结

1. 模型建模，得到一个顶点坐标V1。
2. 顶点乘以模型变换矩阵，缩放，旋转，位移。得到世界坐标系下的坐标V2。
3. V2乘以相机矩阵的变换，这里OpenGL或者DX的坐标系可能略有区别。但基本原理差不多，就是将模型V2换算到相机坐标系。得到V3。
4. 相机坐标系V3乘以投影矩阵，得到屏幕坐标系V4，V4还没有像素宽高参与运算，它的范围一般是±1。
5. 光栅化，将±1的图像，换算到width\*height的framebuffer中，然后再根据swapbuffer，和窗口UI进行图像交换。显示到界面上。

## 纹理贴图

纹理贴图其实很简单。顶点已经计算了在FBO中的位置了。而他们对应的纹理坐标也是知道的。再最后光栅化的阶段，根据纹理坐标去图像采样，把图像填到FBO里去就可以了。这个阶段已经是2D的功能了。