**glLookAt矩阵推导**

**http://blog.csdn.net/popy007/article/details/5120158 （很清楚的解释）**

<http://blog.csdn.net/augusdi/article/details/20450065>

<http://www.360doc.com/content/14/0822/11/13726687_403784935.shtml>

把物体从世界坐标系转换到视点坐标系的矩阵称为视图矩阵

gluLookAt通过指定一个视点、表面场景中的参考点以及up向量来构造一个视图变换矩阵，这个矩阵将代表场景中心的参考点映射到-Z轴，视点映射为原点。视图变换就是在相机坐标系下解释世界坐标系下的点，而这个变换矩阵，可以看做是将相机坐标系变换到与原来的世界坐标系

重点理解uvn坐标系统

Libgdx的矩阵推导

**public** Matrix4 setToLookAt (Vector3 position, Vector3 target, Vector3 up) {

*tmpVec*.set(target).sub(position);

**setToLookAt(*tmpVec*, up);**

**this.mul(*tmpMat*.setToTranslation(position.tmp().mul(-1)));**

**return** **this**;

}

**public** Matrix4 setToLookAt (Vector3 direction, Vector3 up) {

*l\_vez*.set(direction).nor();

*l\_vex*.set(direction).nor();

*l\_vex*.crs(up).nor();

*l\_vey*.set(*l\_vex*).crs(*l\_vez*).nor();

idt();

val[*M00*] = *l\_vex*.x;

val[*M01*] = *l\_vex*.y;

val[*M02*] = *l\_vex*.z;

val[*M10*] = *l\_vey*.x;

val[*M11*] = *l\_vey*.y;

val[*M12*] = *l\_vey*.z;

val[*M20*] = -*l\_vez*.x;

val[*M21*] = -*l\_vez*.y;

val[*M22*] = -*l\_vez*.z;

**return** **this**;

}

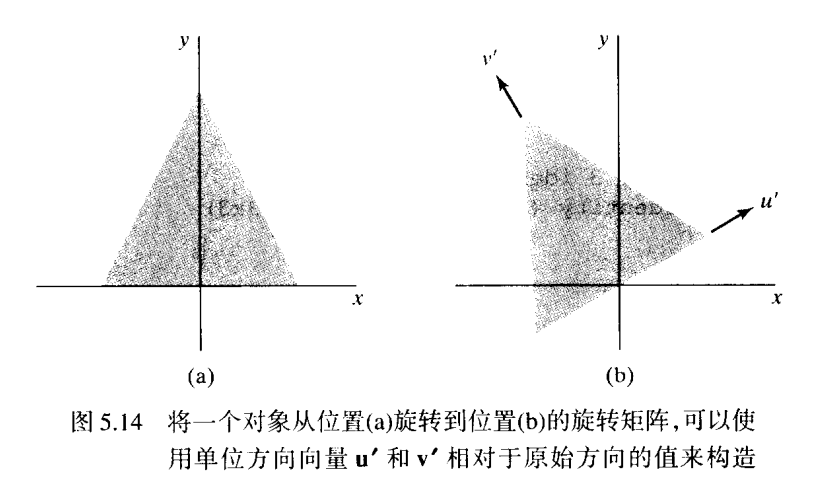
上面的公式在up={-(float)Math.sqrt(2.0f)/2, (float)Math.sqrt(2.0f)/2,0.0f}的情况下已经验证是正确的，**实际上是先求得了一个旋转矩阵，然后再与一个平移矩阵相乘的结果**，l\_vex,l\_vey,l\_vez实际上是观看位置的三维笛卡尔坐标系，注意坐标系的得到是先确定z轴（通过direction方向得到），在确定x轴（direction与up叉乘得到，右手法则），最后确定y轴（x轴与z轴叉乘得到，右手法则）

三维空间的旋转，绕z轴旋转很好求，接下来的绕x和y轴旋转可以通过轴的取代获取：

绕x轴，则之前的z轴由x轴取代，之前的x轴由-z取代

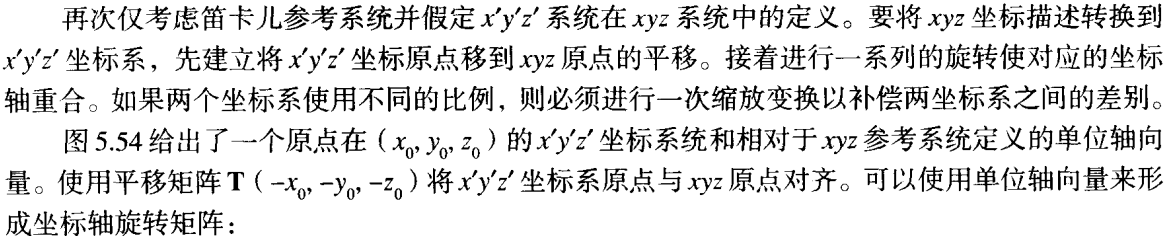
绕y轴，则之前的z轴由y轴取代，之前的y轴由x取代，之前的x轴由z轴取代

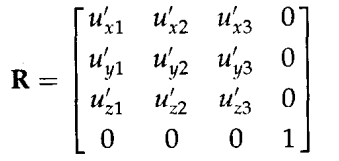
构造二维旋转矩阵（引用计算机图形学的原话）

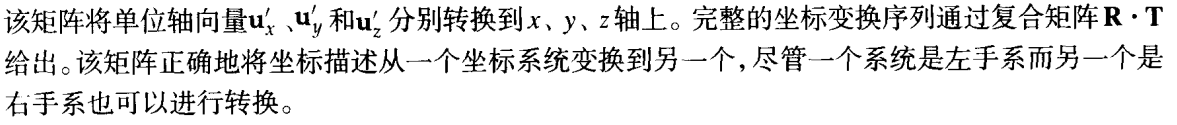


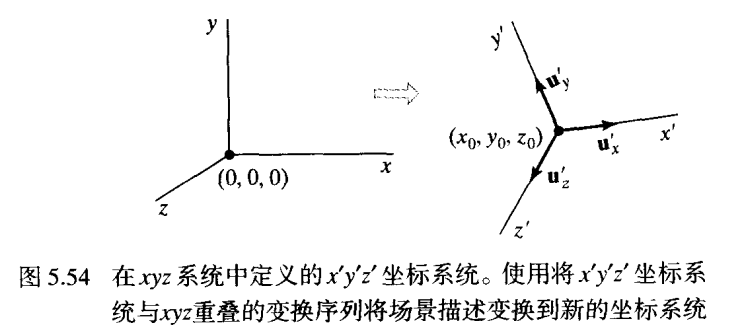
我们可以将u'的元素位置设置给旋转矩阵的第一行，将v'的元素设置给旋转矩阵的第二行，从而构造所期望的变换。**任何旋转矩阵的元素可以表示为一组正交单位向量的元素**

由于只有正弦函数受到旋转角符号变化的影响，因此逆矩阵也可以通过交换原矩阵中的行和列来得到，即可以利用旋转矩阵R的转置来取代逆矩阵计算（R^-1=R^T）









用过渡矩阵M乘以基向量(1,0,0)^T，发现得到的结果是过度矩阵M的第一列

用过渡矩阵M乘以基向量(0,1,0)^T，发现得到的结果是过度矩阵M的第二列

用过渡矩阵M乘以基向量(0,0,1)^T，发现得到的结果是过度矩阵M的第三列

这是一个关键的发现：矩阵的每一列都能解释为转换后的基向量

这个强有力的概念有两条重要性质：

1 有了一种简单的方法来形象化解释矩阵所代表的变换

2 有了反向建立矩阵的可能---给出一个期望的变换（如旋转、缩放等），能够构造一个矩阵代表此变换，我们所有做的一切就是计算基向量的变换，然后将变换后的基向量填入矩阵中

glMatrixMode(GL\_MODELVIEW)该语句指定了一个4\*4建模观察矩阵作为当前矩阵，使用glMatirxMode函数还可以设定另外另个模式：纹理模式和颜色模式。纹理模式用于映射表面的纹理图案，而颜色模式用于从一个颜色模型转换到另外一个颜色模型。glMatrixMode函数的默认变量是GL\_MODELVIEW

opengl有两个函数用来处理栈中的矩阵，这些栈处理函数比单独管理栈矩阵效率要高，尤其是在栈函数使用硬件实现时。例如，硬件实现可同时复制多个矩阵元素，而我们可以在栈中保留一个单位矩阵，从而在当前矩阵初始化时实现得比调用glLoadIdentity更快

这两个函数是glPushMatrix与glPopMatrix