# 背景

一直对OpenGL的坐标变换感到模糊，偶然机会看到一篇不错的博客讲解得不错，跟大家分享下。

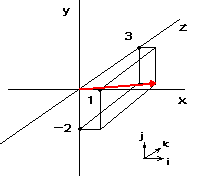
# 数学知识

## 矩阵

略

## 2.2空间坐标系的基和基矩阵

在3-D空间中，用空间坐标系来规范物体的位置，空间坐标系由3个相互垂直的坐标轴组成，就把它们作为观察3-D空间的基础，空间中物体的位置可以通过它们来衡量。当把这3个坐标轴上单位长度的向量记为3个相互正交的单位向量i,j,k，空间中每一个点的位置都可以被这3个向量线性表出，如P<1,-2,3>这个点可以表为i-2j+3k。



把这3个正交的单位向量称为空间坐标系的**基**，它们单位长度为1且正交，所以可以成为**标准正交基**。三个向量叫做**基向量**。现在用矩阵形式写出基向量和基。

i =  | 1 0 0 |           
j =  | 0 1 0 |   
k =  | 0 0 1 |

| i |    | 1 0 0 |     
B = | j | =  | 0 1 0 |  
  | k |    | 0 0 1 |

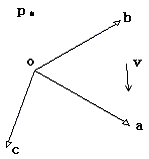
这样的矩阵叫它基矩阵。有了基矩阵，就可以把空间坐标系中的一个向量写成坐标乘上基矩阵的形式，比如上面的向量P可以写成：

P = C x B  
  
=>                   
 | 1 0 0 |  
| 1 -2 3 | = | 1 -2 3 | x | 0 1 0 |  
         | 0 0 1 |

这样的话，空间坐标系下的同一个向量在不同的基下的坐标是不同的。

## 2.3齐次坐标

从上面关于基的概念，对于一个向量*v*以及基*oabc*

**

可以找到一组坐标*(v1,v2,v3)*，使得

*v = v1 a + v2 b + v3 c*（*1*）

而对于一个点*p*，则可以找到一组坐标（*p1,p2,p3*），使得

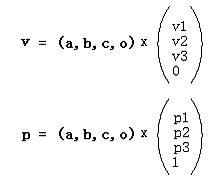
*p–o = p1 a + p2 b + p3 c*（*2*）

从上面对向量和点的表达，可以看出为了在坐标系中表示一个点（如*p*），把点的位置看作是对这个基的原点*o*所进行的一个位移，即一个向量——*p–o*（有的书中把这样的向量叫做位置向量——起始于坐标原点的特殊向量），在表达这个向量的同时用等价的方式表达出了点*p:*

*p = o + p1 a + p2 b + p3 c (3)*

1. *(3)*是坐标系下表达一个向量和点的不同表达方式。这里可以看出，虽然都是用代数分量的形式表达向量和点，但表达一个点比一个向量需要额外的信息。如果写出一个代数分量表达*(1, 4, 7)*，谁知道它是个向量还是个点！

我们现在把（*1*）（*3*）写成矩阵的形式：



这里*(a,b,c,o)*是坐标基矩阵，右边的列向量分别是向量*v*和点*p*在基下的坐标。这样，向量和点在同一个基下就有了不同的表达：*3D*向量的第*4*个代数分量是*0*，而*3D*点的第*4*个代数分量是*1*。像这种这种用*4*个代数分量表示*3D*几何概念的方式是一种齐次坐标表示。

“齐次坐标表示是计算机图形学的重要手段之一，它既能够用来明确区分向量和点，同时也更易用于进行仿射（线性）几何变换。”—— *F.S. Hill, JR*

这样，上面的*(1, 4, 7)*如果写成（*1,4,7,0*），它就是个向量；如果是*(1,4,7,1)*，它就是个点。

下面是如何在普通坐标*(Ordinary Coordinate)*和齐次坐标*(Homogeneous Coordinate)*之间进行转换：

从普通坐标转换成齐次坐标时，

如果(x,y,z)是个点，则变为(x,y,z,1);

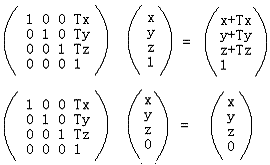
如果(x,y,z)是个向量，则变为(x,y,z,0)

从齐次坐标转换成普通坐标时，

如果是(x,y,z,1)，则知道它是个点，变成(x,y,z);

如果是(x,y,z,0)，则知道它是个向量，仍然变成(x,y,z)

以上是通过齐次坐标来区分向量和点的方式。从中可以思考得知，对于平移*T*、旋转*R*、缩放*S*这*3*个最常见的仿射变换，平移变换只对于点才有意义，因为普通向量没有位置概念，只有大小和方向，这可以通过下面的式子清楚地看出：



而旋转和缩放对于向量和点都有意义，可以用类似上面齐次表示来检测。从中可以看出，齐次坐标用于仿射变换非常方便。

此外，对于一个普通坐标的点P=(Px, Py, Pz)，有对应的一族齐次坐标(wPx, wPy, wPz, w)，其中w不等于零。比如，P(1, 4, 7)的齐次坐标有(1, 4, 7, 1)、（2, 8, 14, 2）、（-0.1, -0.4, -0.7, -0.1）等等。因此，如果把一个点从普通坐标变成齐次坐标，给x,y,z乘上同一个非零数w，然后增加第4个分量w；如果把一个齐次坐标转换成普通坐标，把前三个坐标同时除以第4个坐标，然后去掉第4个分量。

由于齐次坐标使用了*4*个分量来表达*3D*概念，使得平移变换可以使用矩阵进行，从而如*F.S. Hill, JR*所说，仿射（线性）变换的进行更加方便。由于图形硬件已经普遍地支持齐次坐标与矩阵乘法，因此更加促进了齐次坐标使用，使得它似乎成为图形学中的一个标准。

## 2.4 简单的线性插值

这是在图形学中普遍使用的基本技巧，我们在很多地方都会用到，比如2D位图的放大、缩小，Tweening变换，以及我们即将看到的透视投影变换等等。基本思想是：给一个x属于[a, b]，找到y属于[c, d]，使得x与a的距离比上ab长度所得到的比例，等于y与c的距离比上cd长度所得到的比例，用数学表达式描述很容易理解：

ty1

这样，从*a*到*b*的每一个点都与*c*到*d*上的唯一一个点对应。有一个*x*，就可以求得一个*y*。

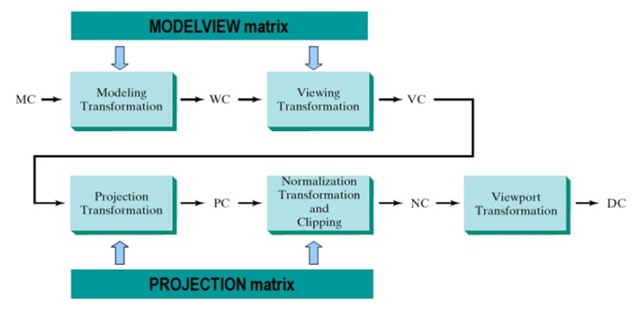
此外，如果*x*不在*[a, b]*内，比如*x < a*或者*x > b*，则得到的*y*也是符合*y < c*或者*y > d*，比例仍然不变，插值同样适用。

# OpenGL 渲染管线

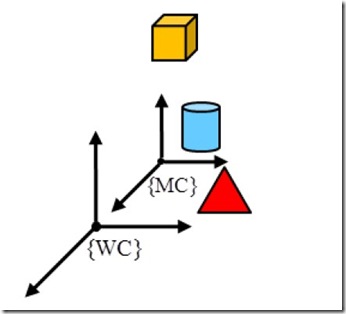
OpenGL渲染管线分为两大部分，**模型观测变换(ModelView Transformation)**和**投影变换(Projection Transformation)**。做个比喻，计算机图形开发就像我们照相一样，目的就是把真实的场景在一张照相纸上表现出来。那么**观测变换的过程就像是我们摆设相机的位置，选择好要照的物体，摆好物体的造型**。而**投影变换就像相机把真实的三维场景显示在相纸**上一样。下面就分别详细的讲一下这两个过程。

## 3.1模型观测变换

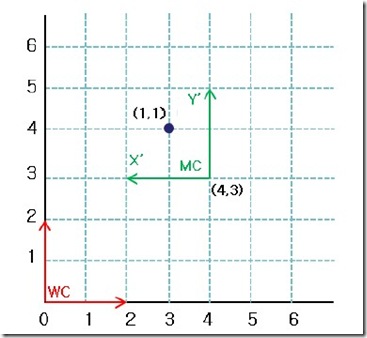
先来弄清楚OpenGL中的****渲染管线****。管线是一个抽象的概念，之所以称之为****管线是因为显卡在处理数据的时候是按照一个固定的顺序来的，而且严格按照这个顺序****。就像水从一根管子的一端流到另一端，这个顺序是不能打破的。先来看看下面的图：



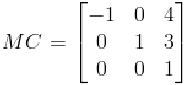
图中显示了OpenGL图形管线的主要部分，也是在进行图形编程的时候常常要用到的部分。一个顶点数据从图的**左上角(MC)进入管线**，最后从图的**右下角(DC)输出**。**MC是Model Coordinate的简写，表示模型坐标**。**DC**是**Device Coordinate**的简写，表示**设备坐标**。当然DC有很多了，什么显示器，打印机等等。这里DC我们就理解成常说的屏幕坐标好了。MC当然就是3D坐标了(注意：我说的3D坐标，而不是世界坐标)，这个3D坐标就是模型坐标，也说成本地坐标(相对于世界坐标)。MC要经过模型变换(Modeling Transformation)才变换到世界坐标，如下图。



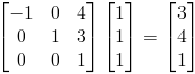
变换到世界坐标WC(World Coordinate)说简单点就是如何用世界坐标系来表示本地坐标系中的坐标。为了讲得更清楚一些，这里举个2D的例子。如图：



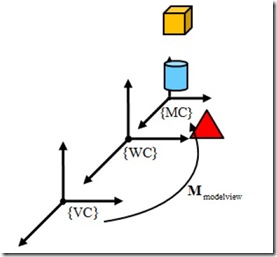
图中红色坐标系是世界**坐标系WC**，绿色的是**模型坐标系MC**。现在有一个顶点，在模型坐标系中的坐标为(1,1)，现在要把这个模型坐标转换到世界坐标中来表示。从图中可以看出，点(1,1)在世界坐标系中的坐标为(3,4)，现在我们来通过计算得到我们希望的结果。首先我们要把模型坐标系MC在世界坐标系中表示出来，使用**齐次坐标(Homogeneous Coordinate )**可以表示为矩阵(注意，本教程中使用的矩阵都是以列向量组成)：



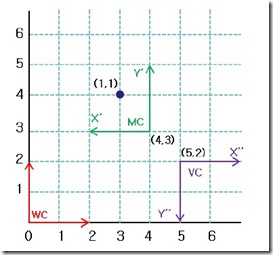
其中，矩阵的**第一列为MC中x轴在WC中的向量表示**，**第二列为MC中y轴WC中的向量表示**，**第三列为MC中的原点在WC中的坐标**。对齐次坐标系不了解的同学，请先学习前面第二章节的内容。有了这个模型变换矩阵后，用这个矩阵乘以在MC中表示的坐标就可以得到该坐标在世界坐标系中的坐标。所以该矩阵和MC中的坐标(1,1)相乘有:



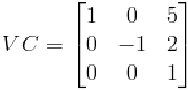
这也正是我们需要的结果。现在让我们把相机坐标也加进去，相机坐标也称为观测坐标(View Coordinate)，如下图。



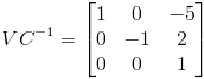
来看看MC坐标中的点(1,1)如何在相机坐标中表示。如下图：



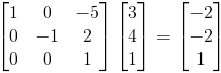
从图中可以直接看出MC中的点(1,1)在相机坐标系VC中为(-2,-2)。和上面同样的道理，我们可以写出相机坐标系VC在世界标系WC中可以表示为：



那么世界坐标系中的点转换为相机坐标系中的点我们就需求VC的逆矩阵：



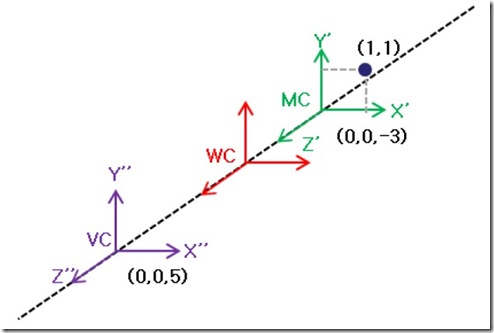
那么世界坐标系WC中的点(3,4)在相机坐标系VC中坐标为：



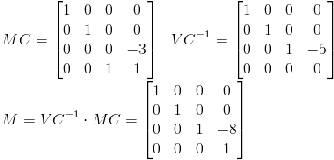
上面的变换过程，就是把**模型坐标变换为相机坐标**。在OpenGL中，当申明顶点的时候，有时候说的是世界坐标**，这是因为初始化的时候世界坐标系、模型坐标系和相机坐标系是一样的，重合在一起的**。所以OpenGL中提供了模型观测变换，它是把模型坐标系直接转换为相机坐标系。现在已经计算得到了VC-1和MC，如果把VC-1和MC相乘，就可以得到模型坐标在相机坐标中的表示。为了得到模型坐标系中的坐标在相机坐标系中的表示，这就是OpenGL中的ModelView变换矩阵。这也是ModelView变换的名字的由来，它是通过了上面两个步骤得到的。那么这里，ModelView变换矩阵M为：

ty11

现在只要用上面的****模型观测矩阵M乘以模型坐标系MC中的坐标就可以得到相机坐标系中的坐标****了。模型观测变换的关键就是要得到相机坐标系中的坐标，因为光照等计算都是在这个这个坐标系中完成的。下面我们实际OpenGL程序中检查一下。在程序中，为了计算方便，使用下图中的模型。



根据图中的数据，我们分别可以写出对应****MC和VC-1****，从而求得****观测变换矩阵M****。



0

1

现在程序中用****glGetFloatv()****这个函数来获得当前矩阵数据来检查一下。具体测试代码如下：

· · float m[16] = {0}; //用来保存当前矩阵数据

· · glMatrixMode(GL\_MODELVIEW);

· · glLoadIdentity();

· · glGetFloatv(GL\_MODELVIEW\_MATRIX, m);

· · //相机设置，View 变换

· · gluLookAt(0.0, 0.0, 5.0,

· · 0.0, 0.0, 0.0,

· · 0.0, 1.0, 0.0);

· · glGetFloatv(GL\_MODELVIEW\_MATRIX, m);

· · //投影设置

· · glMatrixMode(GL\_PROJECTION);

· · glLoadIdentity();

· · glOrtho(-10,10,-10,10,-10,10);

· · glMatrixMode(GL\_MODELVIEW);

· · //Modeling变换

· · glTranslatef(0, 0, -3);

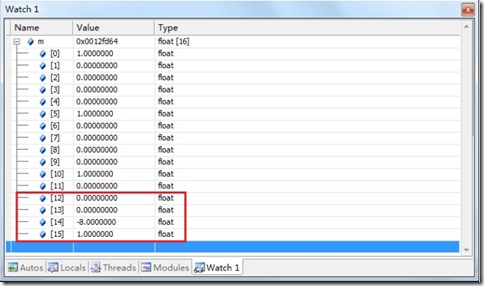
· · glGetFloatv(GL\_MODELVIEW\_MATRIX, m);

· · glBegin(GL\_POINTS);

· · glVertex3f(1,1,0);

· · glEnd();

如果在上面程序段中最后一个****glGetFloatv(GL\_MODELVIEW\_MATRIX, m)****处设定断点的话，就可以看到下图所显示的数据。



个人理解：ModelView 变换矩阵，就是完成从模型坐标到View坐标的转换，是坐标系之间的大变换。注意，ModelView 既有Model，也有View。不只是一个Model的矩阵。

只是对Model的平移或旋转的函数为  glTranslatef等函数，称作模型变换！它的坐标是基于模型本身的，即位于模型坐标系类，比如glTranslatef(0, 0, -3)的3个坐标值。

只是针对View进行设置的函数为  gluLookAt，它的坐标系是View坐标系，比如

gluLookAt(0.0, 0.0, 5.0,0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

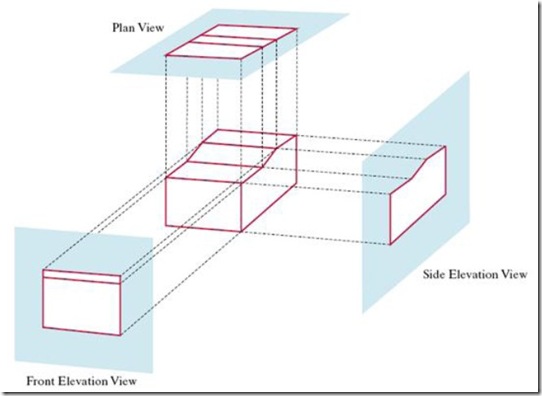
它里面的坐标的原点位于相机坐标系的原点。

## 3.2投影变换

先还是复习一下OpenGL的渲染管线。在**投影变换(Projection Transformation)**中也分为两个部分，第一个部分是将上个阶段得到的**坐标转换为平面坐标**，第二个部分是将转换后的**平面坐标在进行归一化并进行剪裁**。一般地，将三维坐标转换为平面坐标有两种投影方式：**正交投影(Orthogonal Projection)和透视投影(Perspective Projection)**。

### 3.2.1 正交投影

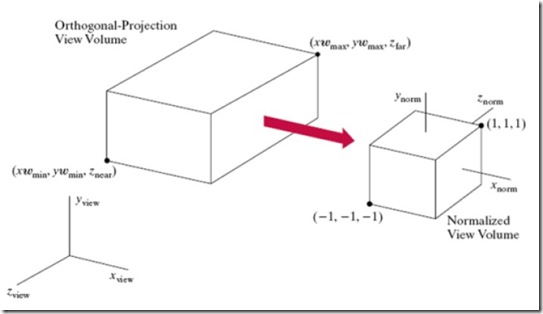
正交投影很简单，如下图，对于三维空间中的坐标点和一个二维平面，要在对应的平面上投影，只需将非该平面上的点的坐标分量改为该平面上的坐标值，其余坐标不变。



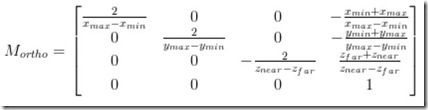
比如将点(1,1,5)正交投影到z=0的平面上，那么投影后的坐标为(1,1,0)。在openGL中，设置正交投影可以使用函数:

glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar)

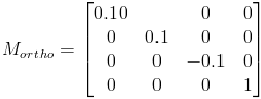
该函数可以设置正交投影的投影空间，在该空间以外的坐标点就不会被投影到投影平面上。函数中的六个参数分是投影空间六个平面，如下图：



在上图中，大的投影空间是根据这六个参数设置的投影空间，OpenGL会自动将该空间归一化，也就是将该空间或立方体转化为变长为1的正六面体投影空间，并且该正六面体的中心在**相机坐标系的原点**。一旦设置使用glortho函数设置投影空间，OpenGL会生成投影矩阵。这个矩阵的作用就是将坐标进行正交投影并且将投影后的坐标正规化(转换到-1到1之间)。要注意的是，生成该矩阵的时候，OpenGL会把右手坐标系转换为左手坐标系。原因很简单，右手坐标系的Z轴向平面外的，这样不符合我们的习惯。该矩阵的矩阵后面章节分析，这里只给出正交投影矩阵。



这个矩阵看来很复杂，其实计算很简单。举个例子，现在设置了这样的正交投影空间glOrtho(-10,10,-10,10,-10,10)，这是个正六面体空间，变长为10。把这些参数带入上面的矩阵可以得到



现在还是在OpenGL程序中来检查一下。在OpenGL程序中添加下面代码段:

· //投影设置

· glMatrixMode(GL\_PROJECTION);

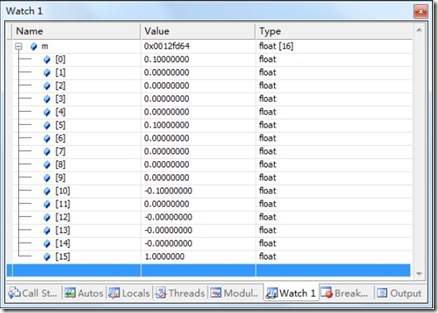
· glLoadIdentity();

· glOrtho(-10,10,-10,10,-10,10);

· glMatrixMode(GL\_MODELVIEW);

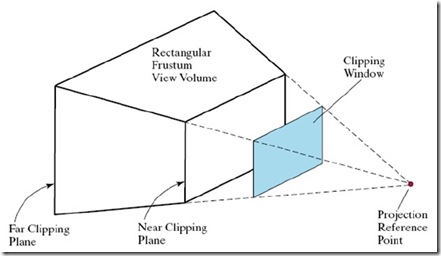
· glGetFloatv(GL\_PROJECTION\_MATRIX,m)

在****glGetFloatv(GL\_PROJECTION\_MATRIX,m)****处设定断点就可以看到下图中所显示的信息。



### 3.2.2透视投影

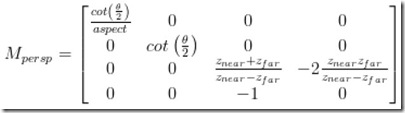
透视投影和正交投影最大的区别就是透视投影具有远近感。



透视投影采用了上图中的模型，这样的模型就是保证远的物体看起来小，近的物体看起来大。 在OpenGL中设置透视投影可以使用函数:

**void** gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);

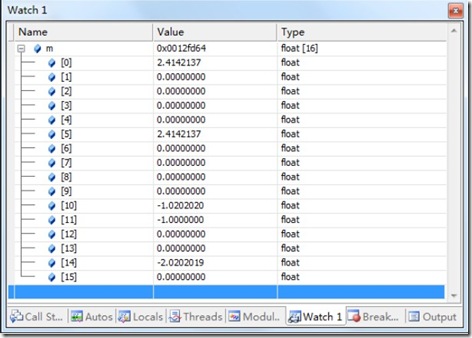
该函数也会根据给定的参数生成一个投影空间。如下图中，该投影空间是一个截头体。同样地，OpenGL会自动生成透视投影矩阵，该矩阵也会让3D坐标投影在投影平面上，并且将投影后的坐标也进行正规化。下面也直接给出OpenGL中使用的透视投影矩阵。



下面在OpenGL中添加下面代码段:

1. //投影设置
2. glMatrixMode(GL\_PROJECTION);
3. glLoadIdentity();
4. gluPerspective(45, 1.0, 1.0, 100);
5. glMatrixMode(GL\_MODELVIEW);
6. glGetFloatv(GL\_PROJECTION\_MATRIX,m)

设置断点后，我们可以看到下图中显示的数据。



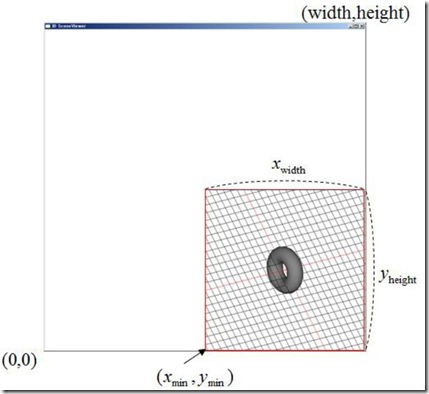
到此为止，整个投影变换就完成了。透过投影变换后得到的是正规化的投影平面坐标。这为下一个阶段的****视口变换(View port Transformation)****做好了准备。

### 3.2.3视口变换

现在到了最后一个阶段了。这个阶段叫做视口变换，它把上个阶段得到的正规化的投影坐标转化为windows 窗口坐标。视口变换会将投影平面上的画面映射到窗口上。在OpenGL中可以使用函数

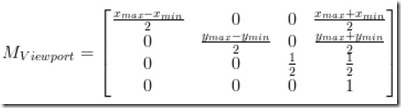
**void** glViewport (GLint x, GLint y, GLsizei width, GLsizei height)

来进行对窗口的映射，如下图：



举个例子说明，比如上个阶段中得到了一个顶点的坐标为(0,0,0.5,1)，根据这个坐标，该顶点位于投影平面的正中间。如果将该点映射到大小为50\*50的窗口上时，那么它应该位于屏幕的中间，坐标为(25,25, 0.5,1)。当然这里深度值0.5是不会改变的。有的同学肯定有疑问了，既然投影到了窗口上，那么还要深度值0.5干什么？这里要注意的是，虽然在窗口上显示时只需要x，y坐标就够了，但是要在2D窗口上显示3D图形时深度值是不可少的。这里的深度值不是用于显示，而是用于在光栅化的时候进行深度测试。

OpenGL也会根据glViewport函数提供的参数值生成一个视口变换矩阵

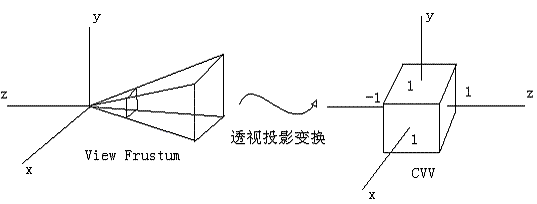


该矩阵把上个阶段得到的正规化坐标映射到窗口上，并且将正规化坐标中的深度值在转换到0到1之间。所以在深度缓冲中最大值为1，最小值为0。视口变换结束后，OpenGL中主要的图形管线阶段就算完成了，后面就是光栅化等等操作。再来回顾一下图1，现在相信大家对这个渲染管线有了一定的认识了，也明白了每一个阶段对应的变换矩阵以及如何进行坐标之间的转换的。

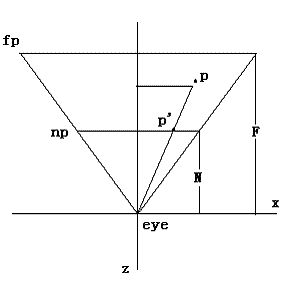
## 3.3透视投影变换推导

经过相机矩阵的变换，顶点被变换到了相机空间。这个时候的多边形也许会被视锥体裁剪，但在这个不规则的体中进行裁剪并非那么容易的事情，所以经过图形学前辈们的精心分析，裁剪被安排到规则观察体(Canonical View Volume, CVV)中进行，CVV是一个正方体，x, y, z的范围都是[-1，1]，多边形裁剪就是用这个规则体完成的。所以，事实上是透视投影变换由两步组成：

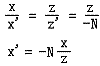
1. 用透视变换矩阵把顶点从视锥体中变换到裁剪空间的*CVV*中。
2. *CVV*裁剪完成后进行透视除法（一会进行解释）。



我们一步一步来，我们先从一个方向考察投影关系。



上图是右手坐标系中顶点在相机空间中的情形。设P(x,z)是经过相机变换之后的点，视锥体由eye——眼睛位置，np——近裁剪平面，fp——远裁剪平面组成。N是眼睛到近裁剪平面的距离，F是眼睛到远裁剪平面的距离。投影面可以选择任何平行于近裁剪平面的平面，这里我们选择近裁剪平面作为投影平面。设P’(x’,z’)是投影之后的点，则有z’ = -N。通过相似三角形性质，我们有关系：



同理，有

ty5

这样，我们便得到了*P*投影后的点*P’*

*ty6*

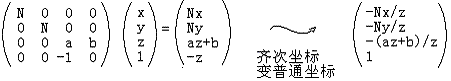
从上面可以看出，投影的结果z’始终等于-N，在投影面上。实际上，z’对于投影后的P’已经没有意义了，这个信息点已经没用了。但对于3D图形管线来说，为了便于进行后面的片元操作，例如z缓冲消隐算法，有必要把投影之前的z保存下来，方便后面使用。因此，我们利用这个没用的信息点存储z，处理成：

ty7

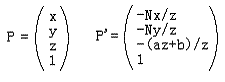
这个形式最大化地使用了*3*个信息点，达到了最原始的投影变换的目的，但是它太直白了，有一点蛮干的意味，我感觉我们最终的结果不应该是它，你说呢？我们开始结合*CVV*进行思考，把它写得在数学上更优雅一致，更易于程序处理。假入能够把上面写成这个形式：

ty8

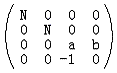
那么我们就可以非常方便的用矩阵以及齐次坐标理论来表达投影变换：



其中



哈，看到了齐次坐标的使用，这对于你来说已经不陌生了吧？这个新的形式不仅达到了上面原始投影变换的目的，而且使用了齐次坐标理论，使得处理更加规范化。注意在把ty11变成的一步我们是使用齐次坐标变普通坐标的规则完成的。这一步在透视投影过程中称为透视除法（*Perspective Division*），这是透视投影变换的第*2*步，经过这一步，就丢弃了原始的*z*值（得到了*CVV*中对应的*z*值，后面解释），顶点才算完成了投影。而在这两步之间的就是*CVV*裁剪过程，所以裁剪空间使用的是齐次坐标ty13，主要原因在于透视除法会损失一些必要的信息（如原始*z*，第*4*个*-z*保留的）从而使裁剪变得更加难以处理，这里我们不讨论*CVV*裁剪的细节，只关注透视投影变换的两步。矩阵



就是我们投影矩阵的第一个版本。你一定会问为什么要把*z*写成

ty15

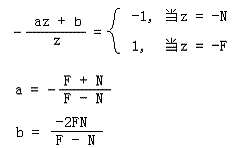
有三个原因：

1. *后面投影之后的光栅化阶段，要通过x'和y'对z进行线性插值，以求出三角形内部片元的z，进行z缓冲深度测试。在数学上，投影后的x'和y'，与z不是线性关系，与1/z才是线性关系。*

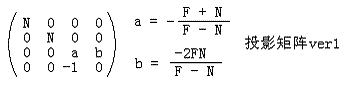
*而ty16正是1/z的线性关系，即-a+b/z。用这个1/z的线性组合值和x'、y'进行插值才是正确的。*

1. *P’*的*3*个代数分量统一地除以分母*-z*，易于使用齐次坐标变为普通坐标来完成，使得处理更加一致、高效。
2. 后面的*CVV*是一个*x,y,z*的范围都为*[-1*，*1]*的规则体，便于进行多边形裁剪。而我们可以适当的选择系数*a*和*b*，使得ty17这个式子在*z = -N*的时候值为*-1*，而在*z = -F*的时候值为*1*，从而在*z*方向上构建*CVV*。

接下来我们就求出*a*和*b*：



这样我们就得到了透视投影矩阵的第一个版本：

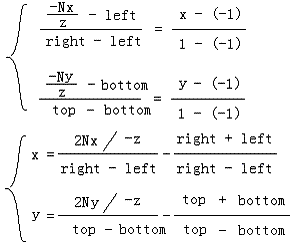


使用这个版本的透视投影矩阵可以从*z*方向上构建*CVV*，但是*x*和*y*方向仍然没有限制在*[-1,1]*中，我们的透视投影矩阵的下一个版本就要解决这个问题。

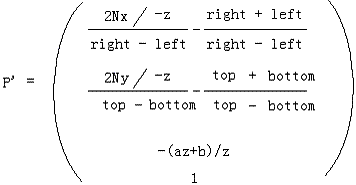
为了能在*x*和*y*方向把顶点从*Frustum*情形变成*CVV*情形，我们开始对*x*和*y*进行处理。先来观察我们目前得到的最终变换结果：



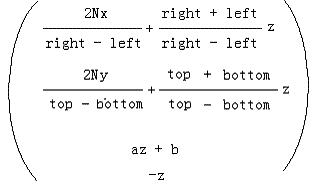
我们知道-Nx / z的有效范围是投影平面的左边界值（记为left）和右边界值（记为right），即[left, right]，-Ny / z则为[bottom, top]。而现在我们想把-Nx / z属于[left, right]映射到x属于[-1, 1]中，-Ny / z属于[bottom, top]映射到y属于[-1, 1]中。你想到了什么？哈，就是我们简单的线性插值，你都已经掌握了！我们解决掉它：



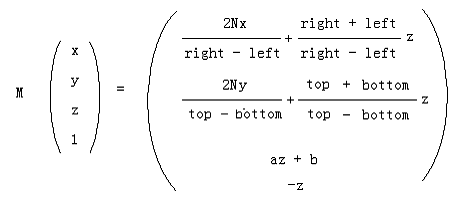
则我们得到了最终的投影点：



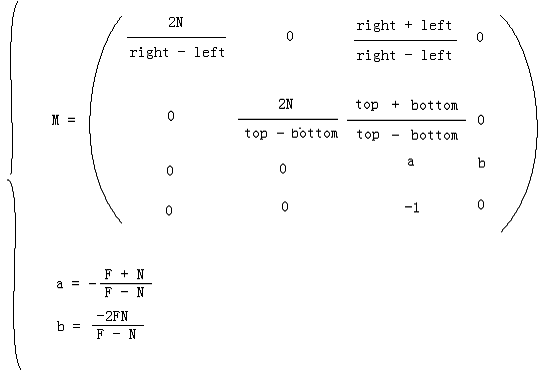
下面要做的就是从这个新形式出发反推出下一个版本的透视投影矩阵。注意到是ty24经过透视除法的形式，而*P’*只变化了*x*和*y*分量的形式，*az+b*和*-z*是不变的，则我们做透视除法的逆处理——给*P’*每个分量乘上*-z*，得到



而这个结果又是这么来的



则我们最终得到：



M就是最终的透视变换矩阵。相机空间中的顶点，如果在视锥体中，则变换后就在CVV中。如果在视锥体外，变换后就在CVV外。而CVV本身的规则性对于多边形的裁剪很有利。OpenGL在构建透视投影矩阵的时候就使用了M的形式。注意到M的最后一行不是(0 0 0 1)而是(0 0 -1 0)，因此可以看出透视变换不是一种仿射变换，它是非线性的。另外一点你可能已经想到，对于投影面来说，它的宽和高大多数情况下不同，即宽高比不为1，比如640/480。而CVV的宽高是相同的，即宽高比永远是1。这就造成了多边形的失真现象，比如一个投影面上的正方形在CVV的面上可能变成了一个长方形。解决这个问题的方法就是在对多变形进行透视变换、裁剪、透视除法之后，在归一化的设备坐标(Normalized Device Coordinates)上进行的视口(viewport)变换中进行校正，它会把归一化的顶点之间按照和投影面上相同的比例变换到视口中，从而解除透视投影变换带来的失真现象。进行校正前提就是要使投影平面的宽高比和视口的宽高比相同。

便利的投影矩阵生成函数

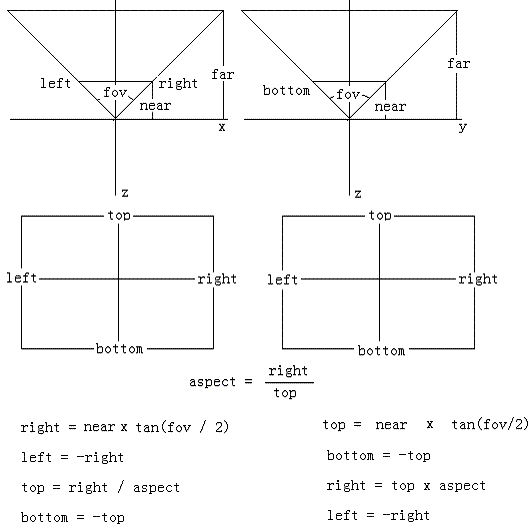
*gluPerspective(fov, aspect, near, far)*

*fov*即视野，是视锥体在*xz*平面或者*yz*平面的开角角度，具体哪个平面都可以。*OpenGL*使用*yz*平面。

*aspect*即投影平面的宽高比。

*near*是近裁剪平面的距离

*far*是远裁剪平面的距离。



上图中左边是在*xz*平面计算视锥体，右边是在*yz*平面计算视锥体。可以看到左边的第*3*步*top = right / aspect*使用了除法（图形程序员讨厌的东西），而右边第*3*步*right = top x aspect*使用了乘法，这也许就是为什么图形*APIs*采用*yz*平面的原因吧！