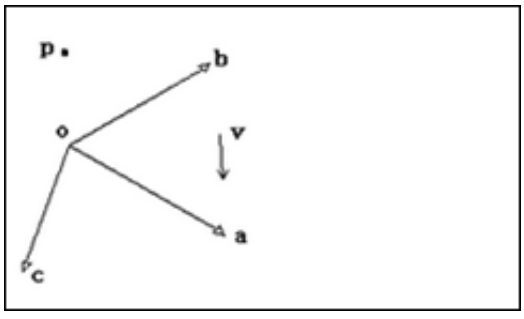
透视投影(Perspective Projection)变换推导

透视投影是3D固定流水线的重要组成部分，是将相机空间中的点从视锥体(frustum)变换到规则观察体(Canonical View Volume)中，待裁剪完毕后进行透视除法的行为。在算法中它是通过透视矩阵乘法和透视除法两步完成的。

透视投影变换是令很多刚刚进入3D图形领域的开发人员感到迷惑乃至神秘的一个图形技术。其中的理解困难在于步骤繁琐，对一些基础知识过分依赖，一旦对它们中的任何地方感到陌生，立刻导致理解停止不前。

没错，主流的3D APIs如OpenGL、D3D的确把具体的透视投影细节封装起来，比如gluPerspective(,,) 就可以根据输入生成一个透视投影矩阵。而且在大多数情况下不需要了解具体的内幕算法也可以完成任务。但是你不觉得，如果想要成为一个职业的图形程序员或游 戏开发者，就应该真正降伏透视投影这个家伙么？我们先从必需的基础知识着手，一步一步深入下去（这些知识在很多地方可以单独找到，但我从来没有在同一个地 方全部找到，但是你现在找到了）。

我们首先介绍两个必须掌握的知识。有了它们，我们才不至于在理解透视投影变换的过程中迷失方向（这里会使用到向量几何、矩阵的部分知识，如果你对此不是很熟悉，可以参考



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，我们可以看出为了在坐标系中表示一个点（如

p），我们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），我们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。如果我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:

$$v = (a, b, c, o) \times \begin{pmatrix} v1 \\ v2 \\ v3 \\ 0 \end{pmatrix}$$

$$p = (a, b, c, o) \times \begin{pmatrix} p1 \\ p2 \\ p3 \\ 1 \end{pmatrix}$$

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：

$$\begin{pmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+Tx \\ y+Ty \\ z+Tz \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$$

而旋转和缩放对于向量和点都有意义，你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。
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：

$$\frac{x-a}{b-a}=\frac{y-c}{d-c}$$

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。

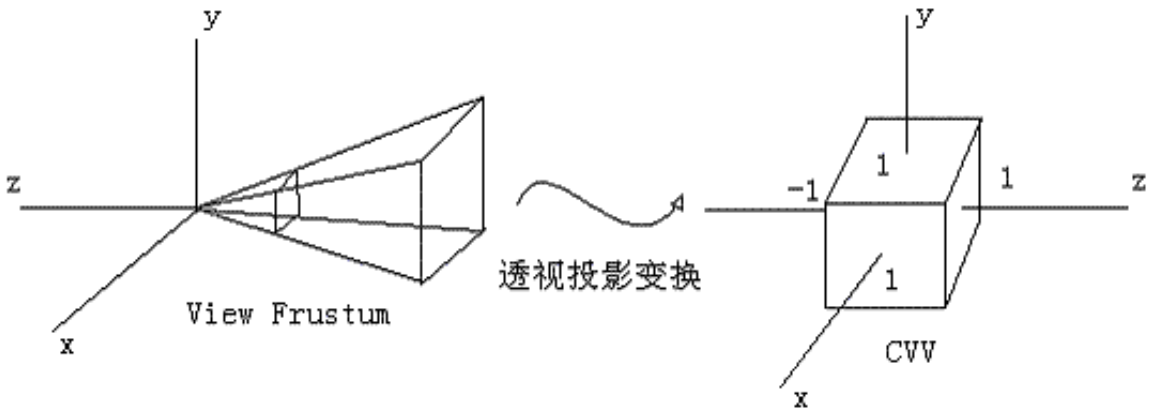
此外，如果x不在[a, b]内，比如x < a或者x > b，则得到的y也是符合y < c或者y > d，比例仍然不变，插值同样适用。

透视投影变换

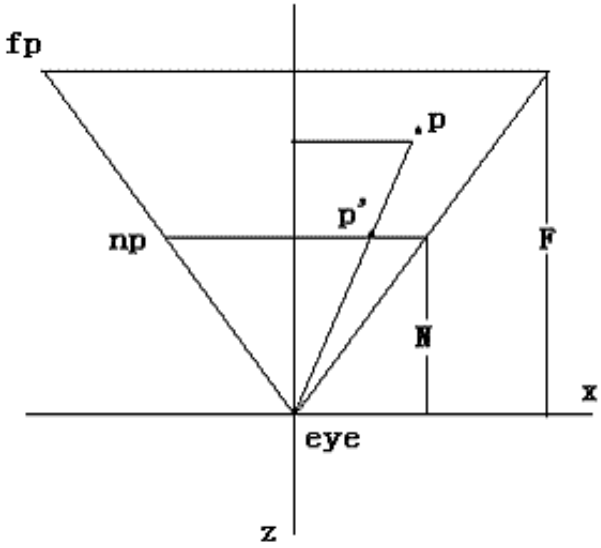
好，有了上面两个理论知识，我们开始分析这次的主角——透视投影变换。这里我们选择OpenGL的透视投影变换进行分析，其他的 APIs会存在一些差异，但主体思想是相似的，可以类似地推导。经过相机矩阵的变换，顶点被变换到了相机空间。这个时候的多边形也许会被视锥体裁剪，但在这个不规则的体中进行裁剪并非那么容易的事情，所以经过图形学前辈们的精心分析，裁剪被安排到规则观察体(Canonical View Volume, CVV)中进行，CVV是一个正方体，x, y, z的范围都是[-1，1]，多边形裁剪就是用这个规则体完成的。所以，事实上是透视投影变换由两步组成：

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：

$$\frac{x}{x'} = \frac{z}{z'} = \frac{z}{-N}$$

$$x' = -N\frac{x}{z}$$

同理，有

$$y' = -N\frac{y}{z}$$

这样，我们便得到了P投影后的点P’

$$p' = \left(-N\frac{x}{z} \quad -N\frac{y}{z} \quad -N \right)$$

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：

$$p' = \left(-N\frac{x}{z} \quad -N\frac{y}{z} \quad z \right)$$

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：

$$p' = \left(-N\frac{x}{z} \quad -N\frac{y}{z} \quad -\frac{az+b}{z} \right)$$

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：

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} Nx \\ Ny \\ az+b \\ -z \end{pmatrix} \rightsquigarrow \begin{pmatrix} -Nx/z \\ -Ny/z \\ -(az+b)/z \\ 1 \end{pmatrix}$$

齐次坐标
变普通坐标

其中

$$P = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad P' = \begin{pmatrix} -Nx/z \\ -Ny/z \\ -(az+b)/z \\ 1 \end{pmatrix}$$

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

$$\begin{pmatrix} Nx \\ Ny \\ az+b \\ -z \end{pmatrix}$$

变成

$$\begin{pmatrix} -Nx/z \\ -Ny/z \\ -(az+b)/z \\ 1 \end{pmatrix}$$

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

$$\begin{pmatrix} Nx \\ Ny \\ az+b \\ -z \end{pmatrix}$$

，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

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

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

$$-\frac{az+b}{z}$$

有两个原因：
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

$$-\frac{az+b}{z}$$

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：

$$-\frac{az+b}{z} = \begin{cases} -1, & 当z = -N \\ 1, & 当z = -F \end{cases}$$

$$a = -\frac{F+N}{F-N}$$

$$b = \frac{-2FN}{F-N}$$

这样我们就得到了透视投影矩阵的第一个版本：

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad a = -\frac{F+N}{F-N} \quad b = \frac{-2FN}{F-N}$$

投影矩阵ver1

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：

$$\begin{pmatrix} -Nx/z \\ -Ny/z \\ -(az+b)/z \\ 1 \end{pmatrix}$$

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：

$$\begin{cases} \dfrac{\frac{-Nx}{z} - left}{right - left} = \dfrac{x-(-1)}{1-(-1)} \\ \dfrac{\frac{-Ny}{z} - bottom}{top - bottom} = \dfrac{y-(-1)}{1-(-1)} \end{cases}$$

$$\begin{cases} x = \dfrac{2Nx/-z}{right-left} - \dfrac{right+left}{right-left} \\ y = \dfrac{2Ny/-z}{top-bottom} - \dfrac{top+bottom}{top-bottom} \end{cases}$$

则我们得到了最终的投影点：

$$P' = \begin{pmatrix} \frac{2Nx / -z}{right - left} - \frac{right + left}{right - left} \\ \frac{2Ny / -z}{top - bottom} - \frac{top + bottom}{top - bottom} \\ -(az+b)/z \\ 1 \end{pmatrix}$$

式出发反推出下一个版本的透视投影矩阵。注意到

$$\begin{pmatrix} -Nx/z \\ -Ny/z \\ -(az+b)/z \\ 1 \end{pmatrix}$$

是

$$\begin{pmatrix} Nx \\ Ny \\ az+b \\ -z \end{pmatrix}$$

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

$$\begin{pmatrix} \frac{2Nx}{right - left} + \frac{right + left}{right - left} z \\ \frac{2Ny}{top - bottom} + \frac{top + bottom}{top - bottom} z \\ az + b \\ -z \end{pmatrix}$$

而这个结果又是这么来的：

$$M \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{2Nx}{right - left} + \frac{right + left}{right - left} z \\ \frac{2Ny}{top - bottom} + \frac{top + bottom}{top - bottom} z \\ az + b \\ -z \end{pmatrix}$$

则我们最终得到：

$$\begin{cases} M = \begin{pmatrix} \frac{2N}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2N}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \\ a = -\frac{F + N}{F - N} \\ b = \frac{-2FN}{F - N} \end{cases}$$

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

3D APIs都提供了诸如gluPerspective(fov, aspect, near, far)或者D3DXMatrixPerspectiveFovLH(pOut, fovY, Aspect, zn, zf)这样的函数为用户提供快捷的透视矩阵生成方法。我们还是用OpenGL的相应方法来分析它是如何运作的。

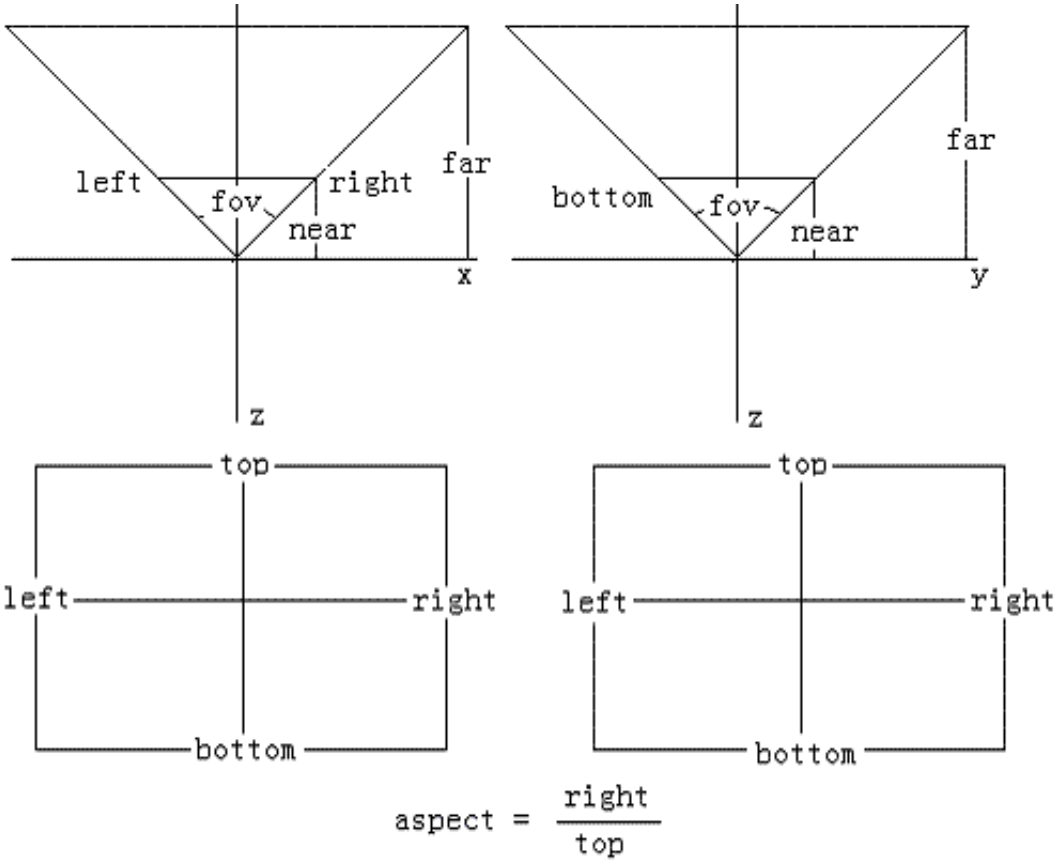
gluPerspective(fov, aspect, near, far)

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和 D3D都使用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。



$$aspect = \frac{right}{top}$$

right = near x tan(fov / 2)

left = -right

top = right / aspect

bottom = -top

top = near x tan(fov/2)

bottom = -top

right = top x aspect

left = -right

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！

在上一篇文章中我们讨论了透视投影变换的原理，分析了OpenGL所使用的透视投影矩阵的生成方法。正如我们所说，不同的图形API因为左右手坐标系、行向量列向量矩阵以及变换范围等等的不同导致了矩阵的差异，可以有几十个不同的透视投影矩阵，但它们的原理大同小异。这次我们准备讨论一下Direct3D（以下简称D3D）以及J2ME平台上的JSR184（M3G）（以下简称M3G）的透视投影矩阵，主要出于以下几个目 的:

（1） 我们在写图形引擎的时候需要采用不同的图形API实现，当前主要是OpenGL和D3D。虽然二者的推导极为相似，但D3D的自身特点导致了一

些地方仍然需要澄清。

（2） DirectX SDK的手册中有关于透视投影矩阵的一些说明，但并不详细，甚至有一些错误，从而使初学者理解起来变得困难，而这正是本文写作的目的。

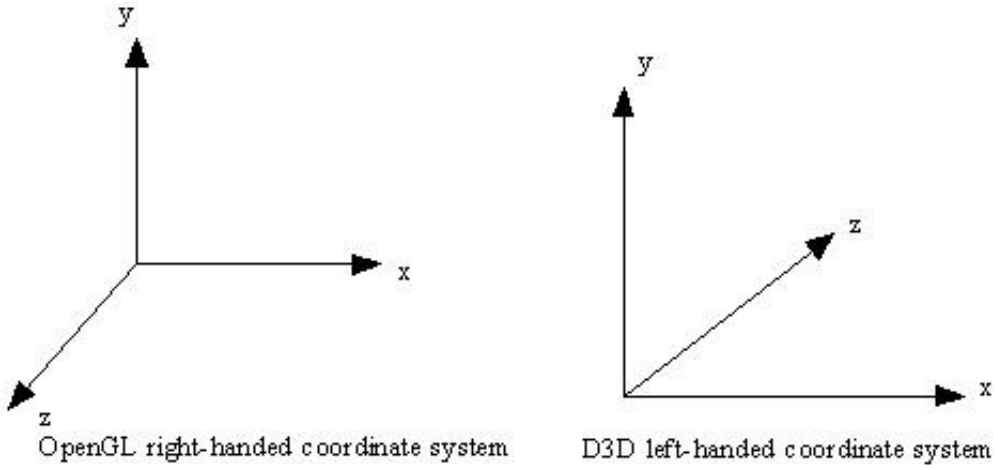
（3） M3G是J2ME平台上的3D开发包，采用了OpenGL作为底层标准进行封装。它的透视投影矩阵使用OpenGL的环境但又进行了简化，值得一提。

本文努力让读者清楚地了解D3D与M3G透视投影矩阵的原理，从而能够知道它与OpenGL的一些差别，为构建跨API的 图形引擎打好基础。需要指出的一点是为了完全理解本文的内容，请读者先理解上一篇文章《深入探索透视投影变换》的内容，因为OpenGL和它们的透视投影 矩阵的原理非常相似，因此这里不会像上一篇文章从基础知识讲起，而是对比它们的差异来推导变换矩阵。我们开始！

OpenGL与D3D的基本差异

前面提到，不同API的基本差异导致了最终变换矩阵的不同，而导致OpenGL和D3D的透视投影矩阵不同的原因有以下几个：

（1） OpenGL默认使用右手坐标系，而D3D 默认使用左手坐标系。



OpenGL right-handed coordinate system　　D3D left-handed coordinate system

（2） OpenGL使用列向量矩阵乘法而D3D使用行向量矩阵乘法。

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

OpenGL Column-Vector Multiplication

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}^{T} \times \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

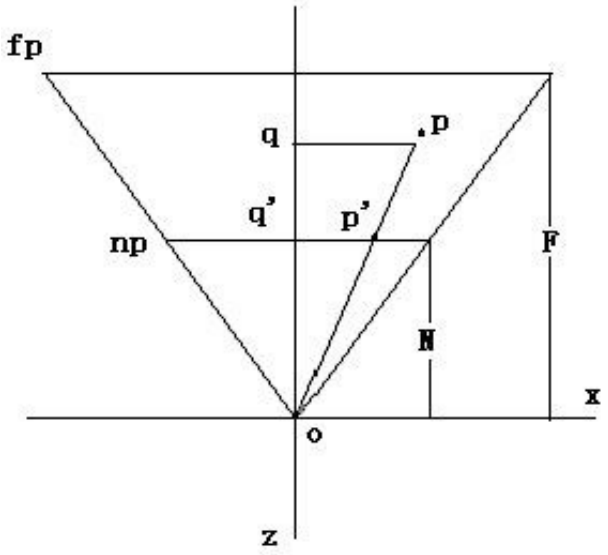
D3D Row-Vector Multiplication

(3) OpenGL的CVV的Z范围是[-1，1]，D3D的CVV的Z范围是[0，1]。

以上这些差异导致了最终OpenGL和D3D的透视投影矩阵的不同。

D3D的透视投影矩阵推导

我们先来看最最基本的透视关系图（上一篇文章开始的时候使用的图）：



这里我们考察的是xz平面上的关系，yz平面上的关系同理。这里o是相机位置。np是近裁剪平面，也是投影平面，N是它到相机的距离。fp是远裁剪平面，F 是它到相机的位置。p是需要投影的点，p’是投影之后的点。根据相似三角形定理，我们有

$$\Delta qpo \cong \Delta q'p'o$$

则有

$$\frac{x'}{x} = \frac{z'}{z} = \frac{N}{z} \Rightarrow x' = N\frac{x}{z} \text{ 同理 } y' = N\frac{y}{z}$$

注意到OpenGL使用右手坐标系，因此应该使用-N（请参考上一篇文章的

这一步），而D3D使用左手坐标系，因此使用N，这是二者的不同点之一。这样，我们得到投影之后的点

$$p' = \begin{pmatrix} N\frac{x}{z} & N\frac{y}{z} & N \end{pmatrix}$$

第三个信息点是变换之后的z在投影平面上的位置，也就是N，它已经没用了，我们把p’写成

$$p' = \begin{pmatrix} N\frac{x}{z} & N\frac{y}{z} & \frac{az+b}{z} \end{pmatrix}$$

从而用第三个没用信息点它来存储z（如果读者对这一点不太了解，请参考上一篇文章）。接下来我们求出a和b，从而在z方向 上构建CVV。请注意这里是 OpenGL和D3D的另一个不同点，OpenGL的CVV的z范围是[-1，1]，而D3D的CVV的z范围是[0，1]。也就是说，D3D 中在近裁剪平面上的点投影之后的点会处于CVV的z=0平面上，而在远裁剪平面上的点投影之后的点会在CVV的z=1平面上。这样我们的计算方程就是

$$\begin{cases} \frac{az+b}{z} = \mathbf{0}, z = N \\ \frac{az+b}{z} = \mathbf{1}, z = F \end{cases} \Rightarrow \begin{cases} a = \frac{F}{F-N} \\ b = -\frac{NF}{F-N} \end{cases}$$

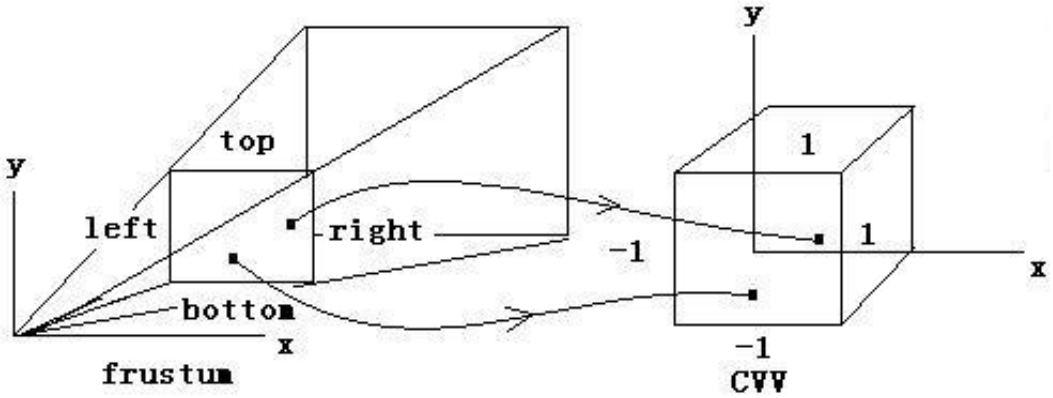
从而我们得到了透视投影矩阵的第一个版本

$$Persp\,\mathbf{Pr}oj = \begin{pmatrix} N & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & N & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & a & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & b & \mathbf{0} \end{pmatrix} \begin{cases} a = \frac{F}{F-N} \\ b = -\frac{NF}{F-N} \end{cases}$$

即

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}^T \times \begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & 1 \\ 0 & 0 & b & 0 \end{pmatrix} = \begin{pmatrix} Nx \\ Ny \\ az+b \\ z \end{pmatrix}^T \xrightarrow{Perspective\_Division(PD)} \begin{pmatrix} Nx/z \\ Ny/z \\ az+b/z \\ 1 \end{pmatrix}^T$$

这个时候第三个分量变换到CVV情形了，CVV的z范围是[0,1]。接下来根据上一篇文章所讲到的，我们要把前两个分量变成CVV情形，CVV的x和y范围是[-1，1]，如下图所示：



使用线性插值，我们有：

$$\begin{cases} \dfrac{Nx/z - left}{right - left} = \dfrac{x_{cvv} - (-1)}{1 - (-1)} \\ \dfrac{Ny/z - bottom}{top - bottom} = \dfrac{y_{cvv} - (-1)}{1 - (-1)} \end{cases}$$

这里left和right是投影平面的左右范围，top和bottom是投影平面的上下范围。xcvv和ycvv是我们需要算出的在CVV情形中的x和y，也就是我们要计算出的结果。但在算出它们之前，我们先把上面的式子写成：

$$\begin{cases} \dfrac{Nx/z}{right-left} - \dfrac{left}{right-left} = \dfrac{x_{cvv}}{2} + \dfrac{1}{2} \\ \dfrac{Ny/z}{top-bottom} - \dfrac{bottom}{top-bottom} = \dfrac{y_{cvv}}{2} + \dfrac{1}{2} \end{cases}$$

这里有一个需要注意的地方，如果投影平面在x方向上居中，则

$$-\frac{left}{right-left} = \frac{1}{2}$$

那么第一个式子就可以销掉等号两边的1/2，写成

$$\frac{Nx/z}{right-left} = \frac{x_{cvv}}{2}$$

同理，如果投影平面在y方向上居中，则第二个式子可以写成

$$\frac{Ny/z}{top-bottom} = \frac{y_{cvv}}{2}$$

则我们现在分两种情况讨论：

（1） 投影平面的中心和x-y平面的中心重合（在x和y方向上都居中）

（2） 一般情况

我们分别讨论：

（1）特殊情况方程

$$\begin{cases} \dfrac{Nx/z}{right-left} = \dfrac{x_{cvv}}{2} \\ \dfrac{Ny/z}{top-bottom} = \dfrac{y_{cvv}}{2} \end{cases}$$

这组是特殊情况，方程比较简单，但也是使用频率最高的方式（这是D3DXMatrixPerspectiveLH、 D3DXMatrixPerspectiveRH、D3DXMatrixPerspectiveFovLH、 D3DXMatrixPerspectiveFovRH四个方

法所使用的情况）。我们导出它：

$$\begin{cases} x_{cvv} = \dfrac{2Nx}{(right - left)z} \\ y_{cvv} = \dfrac{2Ny}{(top - bottom)z} \end{cases}$$

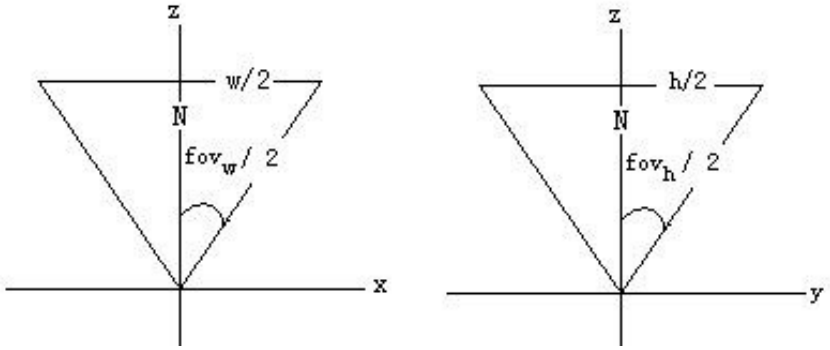
则我们反推出透视投影矩阵：

$$\begin{pmatrix} \dfrac{2Nx}{(right-left)z} \\ \dfrac{2Ny}{(top-bottom)z} \\ \dfrac{az+b}{z} \\ 1 \end{pmatrix}^T \xrightarrow{InvPD} \begin{pmatrix} \dfrac{2Nx}{(right-left)} \\ \dfrac{2Ny}{(top-bottom)} \\ az+b \\ z \end{pmatrix}^T = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}^T \times \begin{pmatrix} \dfrac{2N}{r-l} & 0 & 0 & 0 \\ 0 & \dfrac{2N}{t-b} & 0 & 0 \\ 0 & 0 & a & 1 \\ 0 & 0 & b & 0 \end{pmatrix}$$

其中

$$\begin{cases} a = \dfrac{F}{F-N} \\ b = -\dfrac{NF}{F-N} \end{cases}$$

而 r-l和t-b可以分别看作是投影平面的宽w和高h。最后那个矩阵就是D3D的透视投影矩阵之一。另外呢，如果我们不知道right、left、top以及bottom这几个参量，也可以根据视野（FOV – Field Of View）参量来求得。下面是两个平面的视野关系图：



$$\tan\frac{fov_w}{2}=\frac{w/2}{N}\Rightarrow\frac{2N}{w}=\cot\frac{fov_w}{2}$$

$$\because w=r-l$$

$$\therefore\frac{2N}{r-l}=\cot\frac{fov_w}{2}$$

$$w=\frac{2N}{\cot\frac{fov_w}{2}}$$

同理

$$\frac{2N}{h}=\cot\frac{fov_h}{2}\text{其中}h=t-b$$

$$\frac{2N}{t-b}=\cot\frac{fov_h}{2}$$

$$h=\frac{2N}{\cot\frac{fov_h}{2}}$$

其中，两个fov分别是在x-z以及y-z平面上的视野。如果只给了一个视野，也可以通过投影平面的宽高比计算出来：

$$aspect\_ratio=\frac{w}{h}$$

$$\Rightarrow w=h\times aspect\_ratio\text{或者}$$

$$\Rightarrow h=\frac{w}{aspect\_ratio}$$

用一个视野算出w或者h，然后用宽高比算出h或者w。

（2）一般情况的方程

$$\begin{cases}\dfrac{Nx/z}{right-left}-\dfrac{left}{right-left}=\dfrac{x_{cvv}}{2}+\dfrac{1}{2}\\[2ex]\dfrac{Ny/z}{top-bottom}-\dfrac{bottom}{top-bottom}=\dfrac{y_{cvv}}{2}+\dfrac{1}{2}\end{cases}$$

这组方程比较繁琐，但更具一般性（和OpenGL一般矩阵的推导一致，这也是D3DXMatrixPerspectiveOffCenterLH和D3DXMatrixPerspectiveOffCenterRH两个方法所使用的情况）。我们导

出它:

$$\begin{cases} x_{cvv} = \dfrac{2Nx/z}{right-left} - \dfrac{right+left}{right-left} \\ y_{cvv} \dfrac{2Ny/z}{top-bottom} - \dfrac{top+bottom}{top-bottom} \end{cases}$$

我们继续反推出透视投影矩阵:

$$\begin{pmatrix} \dfrac{2Nx/z}{right-left} - \dfrac{right+left}{right-left} \\ \dfrac{2Ny/z}{top-bottom} - \dfrac{top+bottom}{top-bottom} \\ \dfrac{az+b}{z} \\ 1 \end{pmatrix}^T \xrightarrow{InvPD} \begin{pmatrix} \dfrac{2Nx}{right-left} - \dfrac{right+left}{right-left} z \\ \dfrac{2Ny}{top-bottom} - \dfrac{top+bottom}{top-bottom} z \\ az+b \\ z \end{pmatrix}^T$$

$$= \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}^T \times \begin{pmatrix} \dfrac{2N}{r-l} & 0 & 0 & 0 \\ 0 & \dfrac{2N}{t-b} & 0 & 0 \\ \dfrac{l+r}{l-r} & \dfrac{b+t}{b-t} & a & 1 \\ 0 & 0 & b & 0 \end{pmatrix}$$

其中

$$\begin{cases} a = \dfrac{F}{F-N} \\ b = -\dfrac{NF}{F-N} \end{cases}$$

最后那个矩阵就是D3D的一般透视投影矩阵。
好了，目前为止，我们已经导出了D3D的两个透视投影矩阵。下面我把上一篇导出的OpenGL的透视投影矩阵写出来，大家可以拿它和刚刚导出的D3D的一般性透视投影矩阵做一个对比。

$$proj\_matrix(OpenGL) = \begin{pmatrix} \frac{2N}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2N}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

其中

$$\begin{cases} a = -\frac{F+N}{F-N} \\ b = -\frac{2NF}{F-N} \end{cases}$$

如果仔细观察，可以发现二者在元素的布局上是一个转置的关系，这个就是由它们使用的左右手坐标系以及使用的行列矩阵的差异造成的。而另外在一些元素的细节上也存在着差异，这是由于D3D的CVV的z范围不同造成的。可见在原理相同的情况下，细微的环境差异可以造成非常大的变 化，而这就是透视投影矩阵存在诸多不同版本的原因。一般情况的透视投影矩阵也可以使用视野方式来定义，方法和特殊情况相同。

M3G的透视投影矩阵

M3G是对OpenGL进行的一个封装，它的透视投影变换矩阵被放到了类Camera里面。因为它封装了OpenGL，因 此环境和OpenGL相同：右手坐标系、列向量乘法、CVV范围[-1， 1]。它唯一和OpenGL有些差异的地方就在于它只使用投影平面的中心和x-y平面的中心重合（在x和y方向上都居中）的情况（就是我们上面D3D的第 一种特殊情况）。我们用OpenGL透视投影矩阵最终版本来说明（再次提醒，如果读者对此感到迷惑，请参考第一篇文章）：

上面是OpenGL透视投影矩阵的最终版本，也是一般性版本，我们要把它变成特殊性，版本，非常简单，和上面D3D的特殊情况一样，我们从对x和y进行插值的那一步来看：

$$\begin{cases} \frac{-Nx/z - l}{r-l} = \frac{x'-(-1)}{1-(-1)} \\ \frac{-Ny/z - b}{t-b} = \frac{y'-(-1)}{1-(-1)} \end{cases} = \begin{cases} \frac{-Nx/z}{r-l} - \frac{l}{r-l} = \frac{x'}{2} + \frac{1}{2} \\ \frac{-Ny/z}{t-b} - \frac{b}{t-b} = \frac{y'}{2} + \frac{1}{2} \end{cases}$$

和 D3D的第一种情况一样，销掉两边的1/2，得到：

$$\begin{cases} \dfrac{-Nx/z}{r-l} = \dfrac{x'}{2} \\ \dfrac{-Ny/z}{t-b} = \dfrac{y'}{2} \end{cases}$$

则我们反推出透视投影矩阵：

$$\begin{pmatrix} \dfrac{-2Nx}{z(r-l)} \\ \dfrac{-2Ny}{z(t-b)} \\ -\dfrac{az+b}{z} \\ 1 \end{pmatrix} \xrightarrow{InvDP} \begin{pmatrix} \dfrac{2Nx}{r-l} \\ \dfrac{2Ny}{t-b} \\ az+b \\ -z \end{pmatrix} = \begin{pmatrix} \dfrac{2N}{r-l} & 0 & 0 & 0 \\ 0 & \dfrac{2N}{t-b} & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

其中

$$\begin{cases} a = -\dfrac{F+N}{F-N} \\ b = -\dfrac{2NF}{F-N} \end{cases}$$

最右边那个矩阵就是M3G的透视投影矩阵。仍然可以通过视野参数来设置透视投影矩阵，这里请读者自行推导，方法与上面D3D的完全相同。