**这句话很重要：**

世界坐标系与眼坐标系都是基于右手坐标系下定义的坐标系，而不是说本身是右手坐标系，也就是说两者的基都是基于右手坐标系定义的

**2）投影变换矩阵推导**

http://www.songho.ca/opengl/gl\_projectionmatrix.html（重要）

<http://blog.csdn.net/lyx2007825/article/details/8792475>

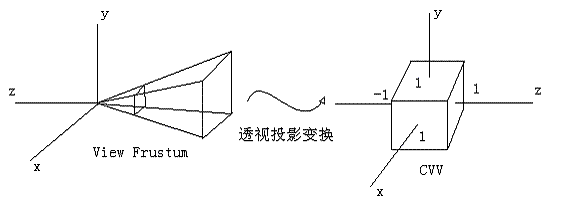
http://blog.csdn.net/popy007/article/details/1797121

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

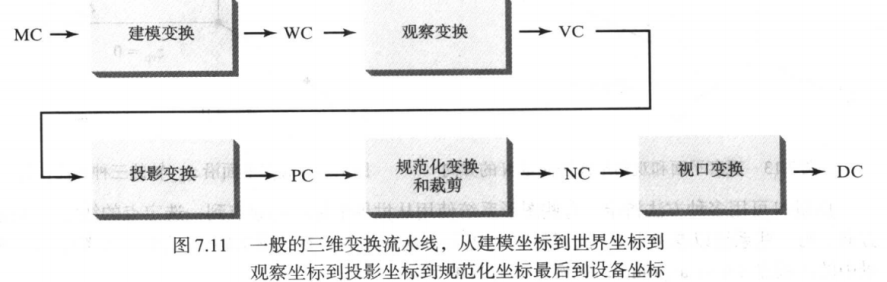
要注意的是，生成该矩阵的时候，OpenGL会把右手坐标系转换为左手坐标系

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，但是在这个不规则体中进行裁剪并非那么容易，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（Cannonical View Volume，CVV）中进行，CVV是一个正方体，xyz的范围都是-1到1，多边形的裁剪就是用这个规范体完成的。所以，事实上是头是投影变换由两步组成：

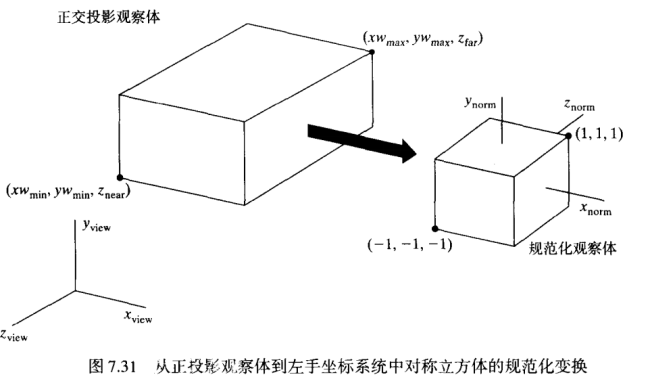
1. 用透视变换矩阵把顶点从视锥体中变换到建材空间的CVV中
2. CVV裁剪完成后进行透视除法

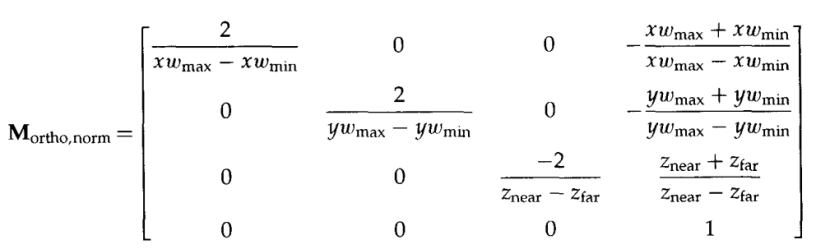


完成向规范化投影坐标的转换后，可高效地对对称立方体（或单位立方体）进行裁剪。裁剪以后，规范化观察体的内容转变到屏幕坐标系，对于x和y的变换和之前的等比变换相同，但三维观察体的所有位置还有一个深度，我们需要将深度信息留给可见性检测和表面绘制时使用。因此，现在可以把视口变换看成向三维屏幕坐标系的映射。



裁剪操作可以在所有与设备无关的坐标变换（从世界坐标系到规范化坐标系）完成之后进行，这样，坐标变换可以合并以便最大限度的提高效率





在规范化变换之后可高效地进行裁剪处理，在观察流水线的这个阶段，所有与设备无关的变换均已完成，并且已经合并到一个组合矩阵中。裁剪后，再应用可见性测试，表面绘制和视视口变换来生成最后的场景屏幕显示

Libgdx的实现：

**public** **void** update (**boolean** updateFrustum) {

projection.setToOrtho(zoom \* -viewportWidth / 2, zoom \* viewportWidth / 2, zoom \* -viewportHeight / 2, zoom

\* viewportHeight / 2, Math.*abs*(near), Math.*abs*(far));

view.setToLookAt(position, tmp.set(position).add(direction), up);

combined.set(projection);

Matrix4.*mul*(combined.val, view.val);

**if** (updateFrustum) {

invProjectionView.set(combined);

Matrix4.*inv*(invProjectionView.val);

frustum.update(invProjectionView);

}

}

获取正交投影归一化矩阵

**public** Matrix4 setToOrtho (**float** left, **float** right, **float** bottom, **float** top, **float** near, **float** far) {

**this**.idt();

**float** x\_orth = 2 / (right - left);

**float** y\_orth = 2 / (top - bottom);

**float** z\_orth = -2 / (far - near);

**float** tx = -(right + left) / (right - left);

**float** ty = -(top + bottom) / (top - bottom);

**float** tz = -(far + near) / (far - near);

val[*M00*] = x\_orth;

val[*M10*] = 0;

val[*M20*] = 0;

val[*M30*] = 0;

val[*M01*] = 0;

val[*M11*] = y\_orth;

val[*M21*] = 0;

val[*M31*] = 0;

val[*M02*] = 0;

val[*M12*] = 0;

val[*M22*] = z\_orth;

val[*M32*] = 0;

val[*M03*] = tx;

val[*M13*] = ty;

val[*M23*] = tz;

val[*M33*] = 1;

**return** **this**;

}

**透视投影矩阵**

即将每一个棱锥体的每个切面都转换到正方体中，可以看做平行于眼坐标切一个面做映射

也可以看作先投影了再做规范化，映射到CVV中（注意：透视投影的裁剪是在齐次坐标系下进行的，也就是说将投影的齐次坐标在CVV立方体中进行裁剪）

先映射到观察面（取近平面），设原始点为p(x,y,z)，投影点为p1(x1,y1,z1)（其实这里的投影点x1,y1,z1会变为齐次坐标）通过三角形的相似性我们可以得到:

x1=-x\*near/z y1=-y\*near/z z1=-near

由于z值以后要进行深度测试，故可以将z1取为之前的z，即z1=z，z值与x和y无关

此时可以将x1,y1,z1用齐次坐标式表示为

near 0 0 0

0 near 0 0 \* (x,y,z,1)^T

0 0 A B

0 0 -1 0

因此可以得到：（注意是除以-z之后映射到了-1-1，而不是直接将near与far映射到-1-1,也就是真实的坐标点要映射到-1-1），**也就是假设了AZ+B已经是z的齐次坐标了，然后归一化，在x和y的求解过程中，是转换为正常坐标-归一化坐标-齐次坐标，而z的求解是假设了齐次坐标，然后将其进行归一化**

**由于ndc坐标是左手坐标系，因为两点为（-near，-1）（-far，1）**

z1 = (A\*z+B)/(-z)

此时将z轴方向映射到-1到1

由于opengl是右手坐标系，而ndc是左手坐标系

则当z=-near时，z1=-1 当z=-far时，z1=1 求解得到：

A=(near+far)/(near-far) B=(2\*near\*far)/(near-far)

但此时x和y还没有归一化到-1到1之间

即(near\*x-left)/(right-left)=(x2+1)/2

得到x2=(2\*near\*x-right-left)/(right-left)

同理得到y2=(2\*near\*y-top-bottom)/(top-bottom)

最终得到的矩阵为：

2\*near/(right-left) 0 (right+left)/(right-left) 0

0 2\*near/(top-bottom) (top+bottom)/(top-bottom) 0

0 0 (near+far)/(near-far) (2\*near\*far)/(near-far)

0 0 -1 0

由常见的公式可得（注意这里的a角度是在垂直方向上的角度）：

Left=-near\*aspect\*tan(a/2)

Right=near\*aspect\*tan(a/2)

bottom=-near\*tan(a/2)

top= near\*tan(a/2)

带入得到：

Cot(a/2)/aspect 0 0 0

0 Cot(a/2) 0 0

0 0 (near+far)/(near-far) (2\*near\*far)/(near-far)

0 0 -1 0

Libgdx的实现

**public** Matrix4 setToProjection (**float** near, **float** far, **float** fov, **float** aspectRatio) {

idt();

**float** l\_fd = (**float**)(1.0 / Math.*tan*((fov \* (Math.*PI* / 180)) / 2.0));

**float** l\_a1 = (far + near) / (near - far);

**float** l\_a2 = (2 \* far \* near) / (near - far);

val[*M00*] = l\_fd / aspectRatio;

val[*M10*] = 0;

val[*M20*] = 0;

val[*M30*] = 0;

val[*M01*] = 0;

val[*M11*] = l\_fd;

val[*M21*] = 0;

val[*M31*] = 0;

val[*M02*] = 0;

val[*M12*] = 0;

val[*M22*] = l\_a1;

val[*M32*] = -1;

val[*M03*] = 0;

val[*M13*] = 0;

val[*M23*] = l\_a2;

val[*M33*] = 0;

**return** **this**;

}

**另外一种理解推导方式（关于x和y）**

首先我们通过映射关系可以得到正常坐标x1=-near\*x/z y1=-near\*y/z

接下来我们需要将正常坐标x1和y1映射到-1-1区域中

即(x1-left)/(right-left)=(x2+1)/2 (x2为我们所求的归一化后的坐标)

得到x2=(-2\*near)\*x/(-z)\*(right-left)-(right+left)/(right-left);

然后将x2转换为齐次坐标，也就是用-z乘以x2，则可以得到变换矩阵的第一行为：

(-2\*near/(right-left) 0 -(right+left)/(right-left) 0)