OpenGL 图形管线和坐标变换

OpenGL Rendering Pipeline & Coordinate Transformation

DONGGUK UNIVERSITY GAME ENGINERRING

张赐 2010/1/26

E-mail：zhangci226@hotmail.com

Blog：http://blog.csdn.net/zhangci226

目　录

1. 引言

学习计算机图形学首先要搞清楚计算机图形渲染管线。当今两大图形 API，OpenGL 和 Direct3D 都有自己的渲染管线。但是对于刚刚接触计算机图形学的同学来说，“图形管线”这个抽象的概念很是不好理解。在 CSDN 论坛中也看到很多同学问到有关于管线的问题。比如：屏幕坐标是怎么转换成世界坐标的等等。似乎这个问题是刚接触计算机图形学必问的问题。国内 OpenGL 的教科书很少，国外这类书虽然多，但是也许是我们和老外思维方式的不同，有的内容不好理解。所以我特别写了这篇教学来为大家解释在 OpenGL 中，它的渲染管线是什么样的，以及渲染管线是如何在程序中反映出来的，最后再详细讲一下屏幕坐标是怎样转换成世界坐标的。希望这篇教学对大家在今后的学习中有所帮助。如果发现该教程中有什么错误或有什么好的建议，请发送到我的邮箱 zhangci226@hotmail.com。

2. 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。

2.1 模型观测变换

让我们先来弄清楚 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 2.1。图中显示了 OpenGL 图形管线的主要部分，也是我们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，图 2.2。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 2.3，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)：

$$MC = \begin{bmatrix} -1 & 0 & 4 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

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，请先学习游戏数学方面的知识。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:

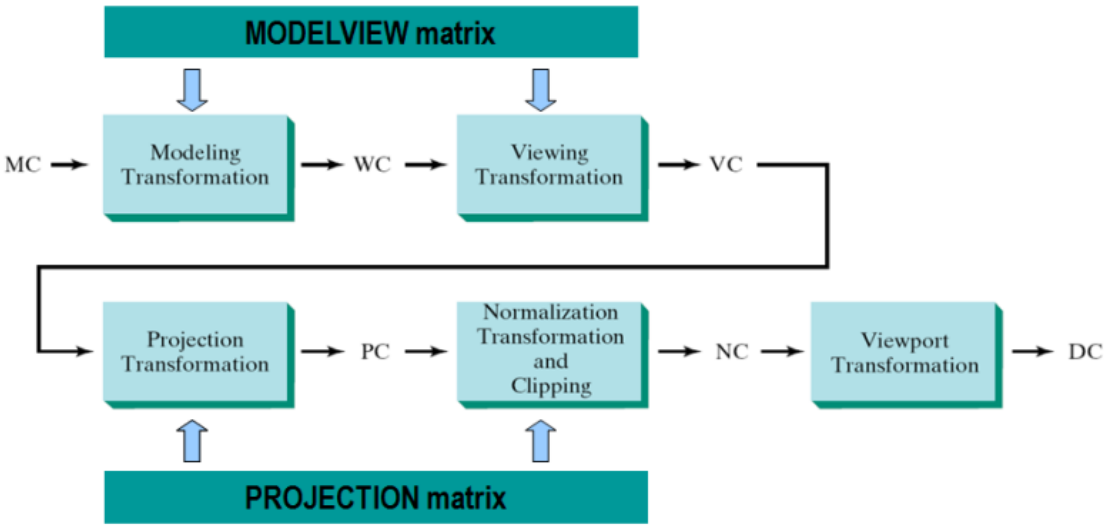

图 2.1 OpenGL 图形管线

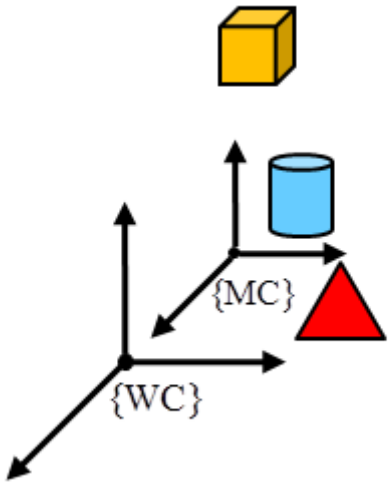

图 2.2 世界坐标系和模型坐标系

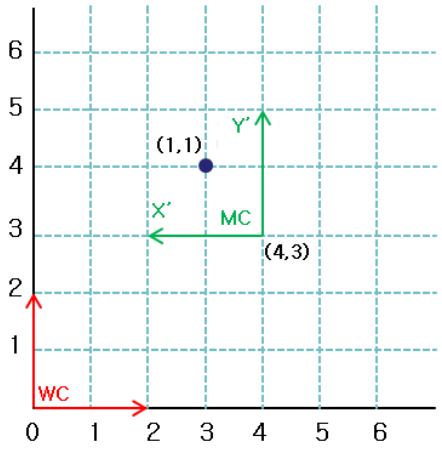

图 2.3 世界坐标系和模型坐标系的计算

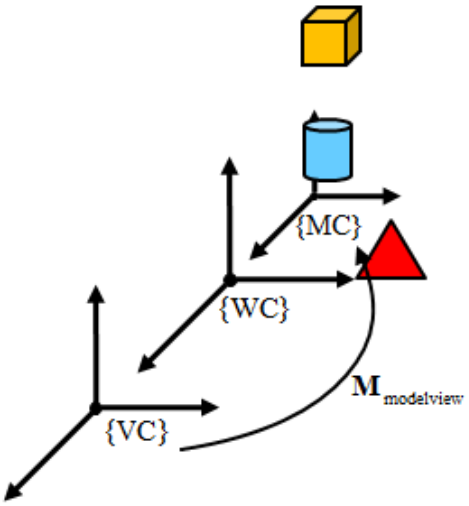

图 2.4 ModelView 变换的三个坐标系

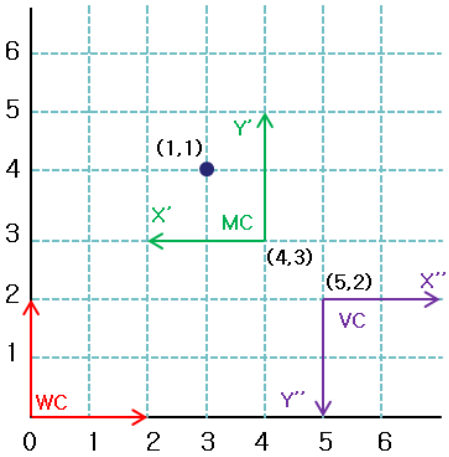

图 2.5 ModelView 变换计算

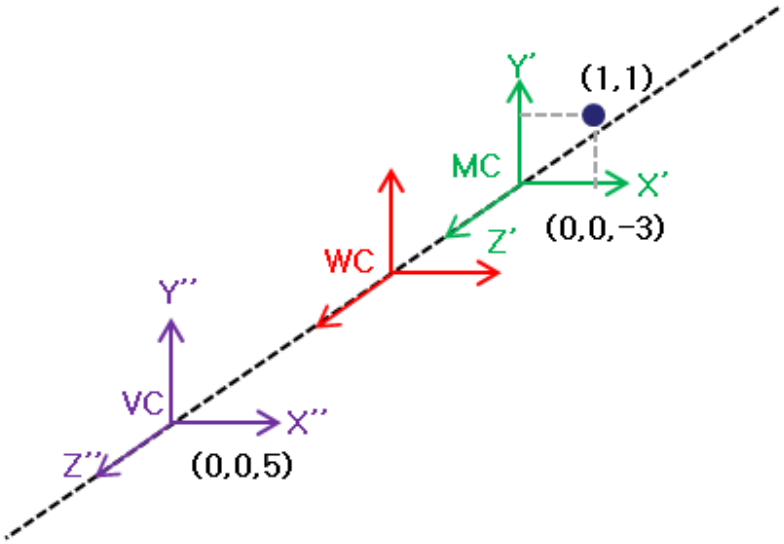

图 2.6 ModelView 变换计算模型

$$\begin{bmatrix} -1 & 0 & 4 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & -5 \\ 0 & -1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -2 \\ 1 \end{bmatrix}$$

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)，如图 2.4 和图 2.5。来看看 MC 坐标中的点(1,1)如何在相机坐标中表示。从图 2.5 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：

$$VC = \begin{bmatrix} 1 & 0 & 5 \\ 0 & -1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

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：

$$VC^{-1} = \begin{bmatrix} 1 & 0 & -5 \\ 0 & -1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

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，就是可以把模型坐标变换为相机坐标。在 OpenGL 中，当我们申明顶点的时候，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，如图 2.4。现在我们已经计算得到了 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：

$$M = VC^{-1} \cdot MC$$

$$= \begin{bmatrix} 1 & 0 & -5 \\ 0 & -1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 4 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & -1 \\ 0 & -1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

现在只要用上面的模型观测矩阵 M 乘以模

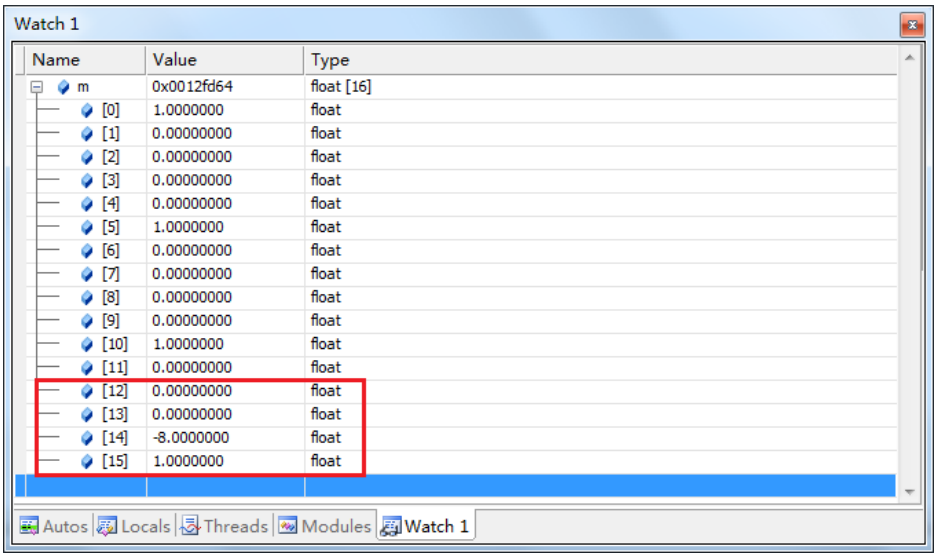

图 2.7 ModelView 变换矩阵数据

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。

$$\begin{bmatrix} -1 & 0 & -1 \\ 0 & -1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -2 \\ 1 \end{bmatrix}$$

下面我们实际 OpenGL 程序中检查一下。在程序中，为了计算方便，我们使用图 2.6 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。

$$MC = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad VC^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M = VC^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -8 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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。
下面是程序段。

```
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，就可以看到图 2.7 所显示的数据。

到这里，整个 ModelView 变换就完成了。通过 ModelView 变换后得到是相机坐标系内的坐标。在这个坐标系内典型的计算就是法线了。现在再来看看后面一个阶段。

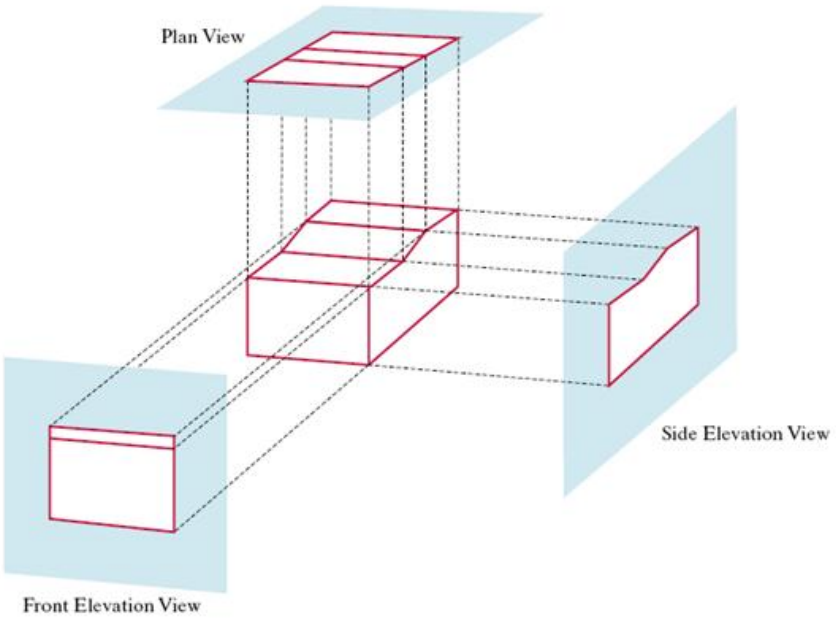

图 2.8 正交投影

2.2 投影变换

先还是复习一下 OpenGL 的渲染管线。图 2.1 中可以看到，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)。

2.2.1 正交投影

正交投影很简单，如图 2.8，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，如图 2.9。在图 2.9 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，并且该证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。该矩阵的矩阵推导这里就不详细说明了，不了解的同学可以参考游戏数学方面资料，这里只给出正交投影矩阵。

$$M_{ortho} = \begin{bmatrix} \frac{2}{x_{max}-x_{min}} & 0 & 0 & -\frac{x_{min}+x_{max}}{x_{max}-x_{min}} \\ 0 & \frac{2}{y_{max}-y_{min}} & 0 & -\frac{y_{min}+y_{max}}{y_{max}-y_{min}} \\ 0 & 0 & -\frac{2}{z_{near}-z_{far}} & \frac{z_{far}+z_{near}}{z_{near}-z_{far}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

这个矩阵看来很复杂，其实计算很简单。举

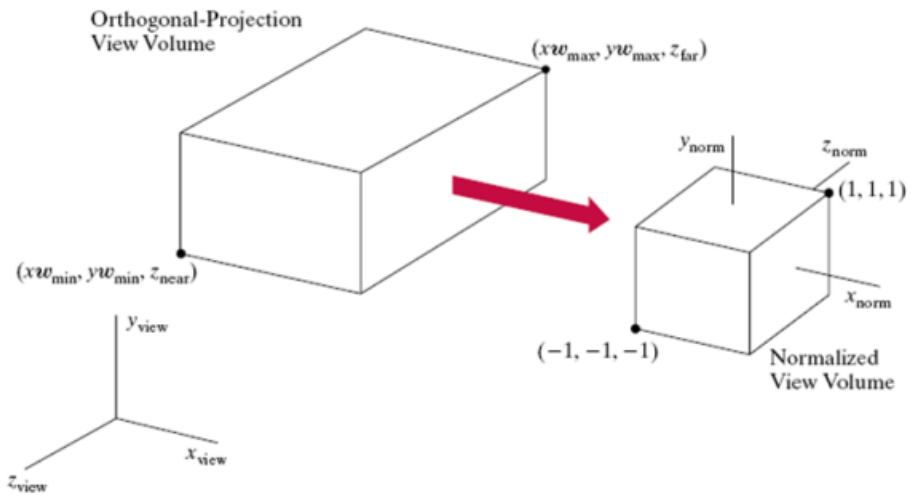

图 2.9 OpenGL 正交投影空间和投影变换

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

$$M_{ortho} = \begin{bmatrix} 0.1 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0 & 0 & -0.1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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图 2.10 中所显示的信息。

2.2.2 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。透视投影采用了图 2.11 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 APIENTRY 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。如图 2.11 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。

$$M_{persp} = \begin{bmatrix} \frac{cot\left(\frac{\theta}{2}\right)}{aspect} & 0 & 0 & 0 \\ 0 & cot\left(\frac{\theta}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{z_{near}+z_{far}}{z_{near}-z_{far}} & -2\frac{z_{near}z_{far}}{z_{near}-z_{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

下面在 OpenGL 中添加下面代码段

```
//投影设置
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45, 1.0, 1.0, 100);
glMatrixMode(GL_MODELVIEW);
glGetFloatv(GL_PROJECTION_MATRIX,m)
```

Watch 1

Name	Value	Type
m	0x0012fd64	float [16]
[0]	0.10000000	float
[1]	0.00000000	float
[2]	0.00000000	float
[3]	0.00000000	float
[4]	0.00000000	float
[5]	0.10000000	float
[6]	0.00000000	float
[7]	0.00000000	float
[8]	0.00000000	float
[9]	0.00000000	float
[10]	-0.10000000	float
[11]	0.00000000	float
[12]	-0.00000000	float
[13]	-0.00000000	float
[14]	-0.00000000	float
[15]	1.0000000	float

Call St... | Autos | Locals | Threads | Modul.. | Watch 1 | Break... | Output

图 2.10 正交变换矩阵数据

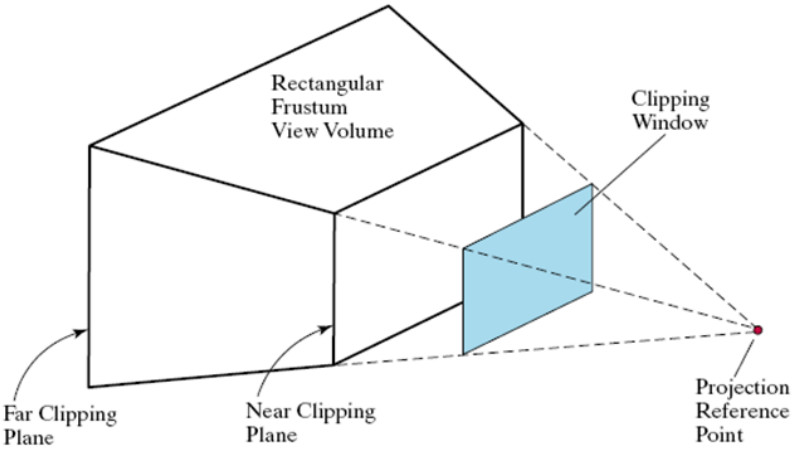

图 2.11 透视投影

Watch 1

Name	Value	Type
m	0x0012fd64	float [16]
[0]	2.4142137	float
[1]	0.00000000	float
[2]	0.00000000	float
[3]	0.00000000	float
[4]	0.00000000	float
[5]	2.4142137	float
[6]	0.00000000	float
[7]	0.00000000	float
[8]	0.00000000	float
[9]	0.00000000	float
[10]	-1.0202020	float
[11]	-1.0000000	float
[12]	0.00000000	float
[13]	0.00000000	float
[14]	-2.0202019	float
[15]	0.00000000	float

Call St... | Autos | Locals | Threads | Modul.. | Watch 1 | Break... | Output

图 2.12 透视变换矩阵数据

设置断点后，我们可以看到图 2.12 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(Viewport Transformation)做好了准备。

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

```
GLAPI void GLAPIENTRY glViewport (GLint x, GLint y, GLsizei width, GLsizei height);
```

来进行对窗口的映射，如图 2.13。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

$$M_{Viewport} = \begin{bmatrix} \frac{x_{max}-x_{min}}{2} & 0 & 0 & \frac{x_{max}+x_{min}}{2} \\ 0 & \frac{y_{max}-y_{min}}{2} & 0 & \frac{y_{max}+y_{min}}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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 2.1，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. 屏幕坐标转换为世界坐标

通过前面的教程，以及现在大家对 OpenGL 整个渲染管线理解后，现在要将屏幕上一点坐标转换为世界坐标就比较容易了。从图形管线的开始到结束，一个模型坐标系中的坐标被转化为了屏幕坐标，那么现在把整个过程倒过来的话，屏幕上一点坐标也可以转为为世界坐标。只要在对应的阶段求得对应变换矩阵的逆矩阵，就可以得到前一个阶段的坐标。这整个过程可以用图 3.1 表示。图中显示的过程完全就是 OpenGL 渲染管线的逆过程，通过这个过程，屏幕上的点就可以转化为世界坐标系中的点了。可能又有的同学要问，当鼠标点击屏幕上一点的时候并没有深度信息，转换的时候要怎么办呢？这个时候可以使用 OpenGL 函数

```
void glReadPixels (GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels);
```

该函数能够获得屏幕上一点对应像素的深度信息。有了这个深度信息，就可以利用上面过程把屏幕上一点转换为世界坐标了。

在 OpenGL 中，上面的过程其实已经有现成的函数可以使用，那就是

```
int APIENTRY gluUnProject (
    GLdouble  winx, GLdouble  winy,
    GLdouble  winz,
    const GLdouble modelMatrix[16],
    const GLdouble projMatrix[16],
    const GLint    viewport[4],
    GLdouble  *objx,  GLdouble  *objy,
    GLdouble       *objz);
```

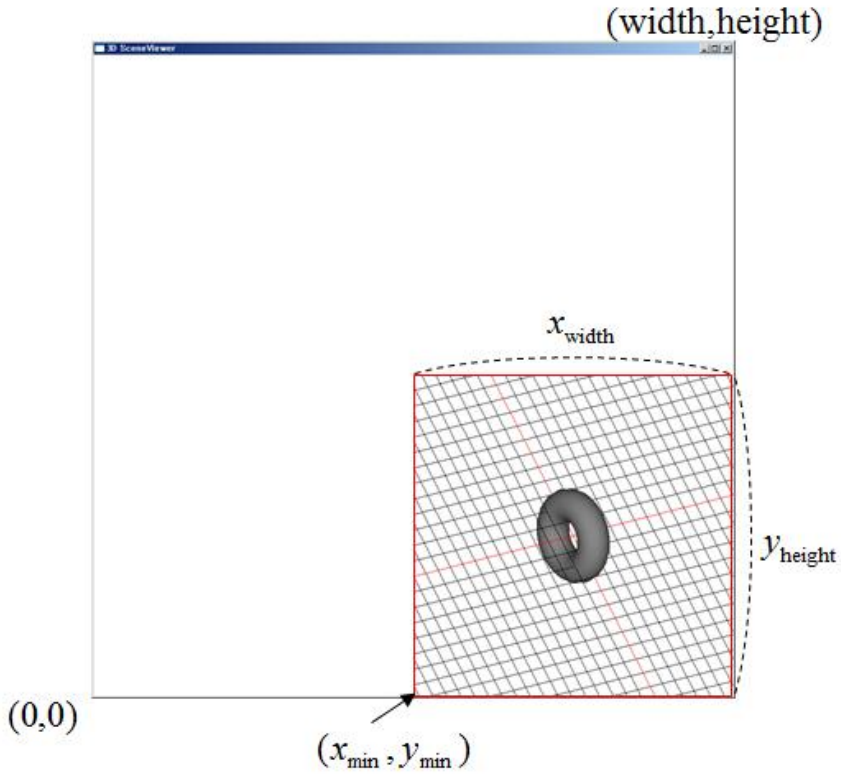

图 2.13 视口变换

```
glViewport(width/2, 0, width/2, height/2)
```

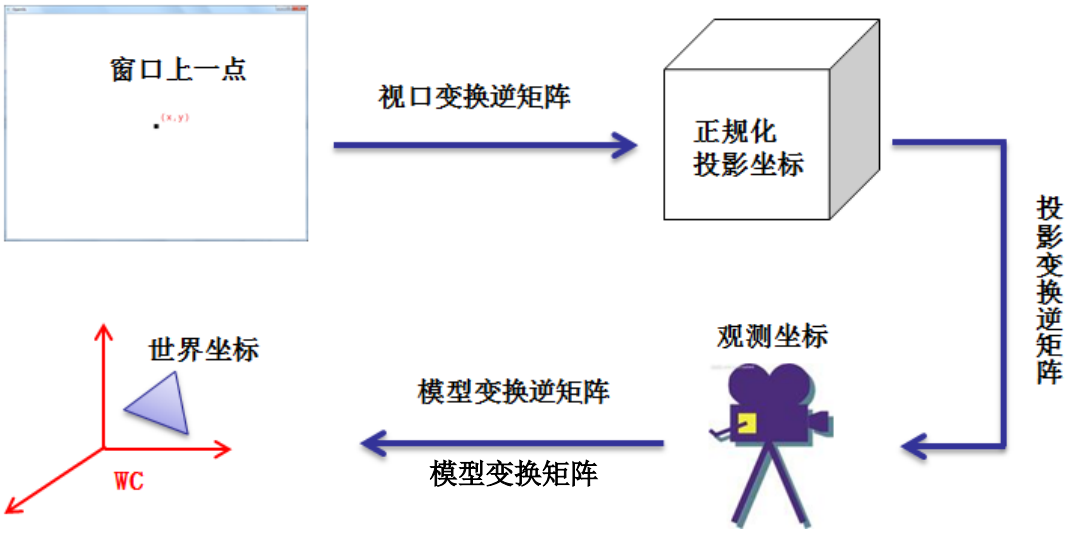

图 3.1 屏幕坐标转换为世界坐标

该函数直接将屏幕上一点转换对应的世界坐标，该函数的内部实现其实还是上面的那么逆过程。下面给出利用该函数获取世界坐标的代码段。

```
GVector screen2world(int x, int y)
{
	GLint viewport[4];
	GLdouble modelview[16];
	GLdouble projection[16];
	GLfloat winX, winY, winZ;
	GLdouble posX, posY, posZ;
	glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
	glGetDoublev(GL_PROJECTION_MATRIX, projection);
	glGetIntegerv(GL_VIEWPORT, viewport);
	winX = (float)x;
	winY = (float)viewport[3] - (float)y;
	glReadPixels(x, int(winY), 1, 1,
GL_DEPTH_COMPONENT, GL_FLOAT, &winZ);
	gluUnProject(winX, winY, winZ, modelview, projection,
viewport, &posX, &posY, &posZ);

	GVector v(4, posX, posY, posZ, 1.0);
	return v;
}
```

代码中函数返回类型 GVector 是用户定义的向量类，返回的是齐次坐标。

4. 总结

这篇教程详细地解释了 OpenGL 图形渲染管线的工作原理以及在 OpenGL 中的坐标之间的转换。利用 OpenGL 渲染管线的这种特征，在 OpenGL 中屏幕坐标和世界坐标之间的转换和 Picking 技术有很容易实现。我已经发表了一篇关于 OpenGL 中 Picking 技术的博客，有兴趣的同学可以在我的 CSDN 博客中参考《深入理解 OpenGL 拾取模式（OpenGL Picking）》这篇文章。最后希望我的所有文章对大家都有所帮助。