**http://blog.csdn.net/wangdingqiaoit/article/details/51531002**

**OpenGL中的MVP矩阵**

<http://www.tuicool.com/articles/2IJvQfj>

<http://blog.sina.com.cn/s/blog_4bce4aa301011ebe.html>

二维坐标系之间的变换：

下图给出了一个在笛卡尔坐标系统xy中用坐标原点（x0，y0）及方向角θ指定的笛卡尔坐标系统x'y'，为了将对象描述从xy坐标变换到x'y'坐标，必须建立把x'y' 轴叠加到xy轴的变换，这需要分两步进行（平移和旋转都已经证明）：

1）将x'y'系统的坐标原点（x0，y0）平移到xy系统的原点（0,0）

2）将x'轴旋转到x轴上

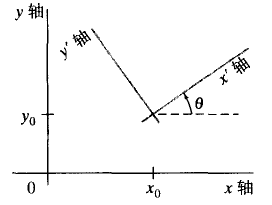
即点从A坐标系转换到B坐标系：

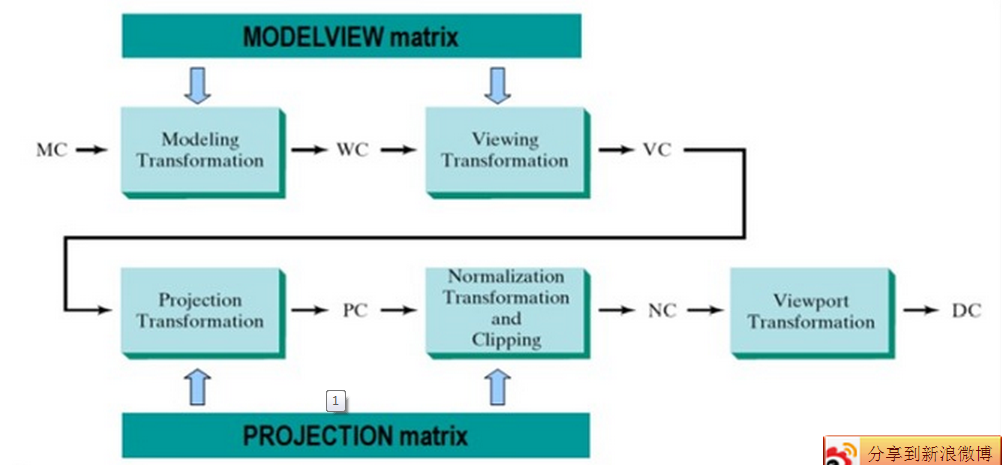
将B坐标系变到与A坐标系重合，比如下图中从x'y'坐标系转换到xy坐标系，则先将xy坐标系平移，然后旋转，即T(x0,y0)\*R(θ)\*v(x,y)，也就是将x'y'坐标系中与xy对应的点先R(θ)然后T(x0,y0)，这一切的变换都是在xy坐标系上做的

为了使观察处理独立与输出设备，图形系统将对象描述转换到规范化设备坐标系并提供裁剪程序。有些系统的规范化设备坐标范围是0到1，而有些是-1到1，裁剪工作通常在规范化坐标系中进行。这使得我们可以在此之前合并变换矩阵从而减少计算时间

应用程序要得到特殊的裁剪效果，可通过选择裁剪窗口的不同形状、大小和方向来实现、例如，可以使用椭圆、星型等形状作为裁剪窗口，但是使用凹多边形或非线性边界裁剪窗口来比用矩形裁剪要花费更多的时间。确定对象与圆的焦点比确定它与直线的交点需要更多的计算。最简单的用户裁剪的窗口边界是用坐标轴平行的直线段。因此，图形软件一般仅允许使用平行于x和y轴的矩形裁剪窗口。

世界坐标系裁剪窗口，图形系统程序库中一般均提供定义标准矩形裁剪窗口的函数。我们可以指定世界坐标系中的两点作为标准矩形的两个对角顶点。一旦建立了裁剪窗口，观察函数就对场景描述进行处理并将结果送到输出设备



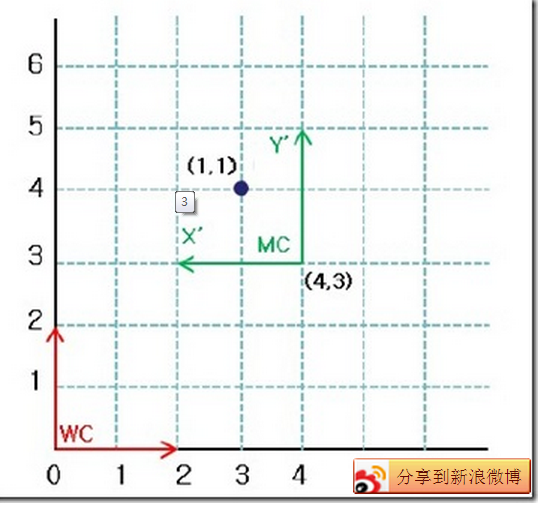
****

1）模型视图变换

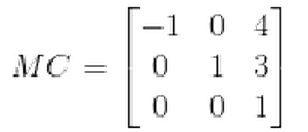
图中显示了opengl图形管线的主要部分，也是我们进行图形编程时常常要用到的部分。一个顶点数据从左上角（MC）进行管线，最后从图的右下角（DC）输出。MC是Model Coordinate的简写，表示模型坐标，DC是Device Coordinate的简写，表示设备坐标。当然DC有很多，比如显示器、打印机等，这里的坐标我们理解为屏幕坐标。MC是3D坐标（注意：这里所说的是3D坐标，而不是世界坐标），这个3D坐标就是模型坐标，也说成本地坐标（相对于世界坐标）。MC要经过模型变换（Modeling Transformation）才变换到世界坐标。

变换到世界坐标WC（World Coordinate）简单点说就是如何用世界坐标系来表示本地坐标系中的坐标

举例：

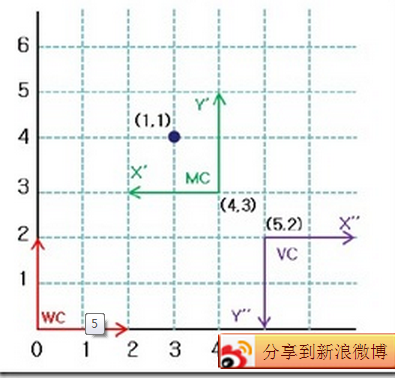


上图中红色的是世界坐标（WC），绿色的是模型坐标（MC）,首先要将模型坐标系MC在世界坐标系中表示出来，得到的变换矩阵为：（先将xy坐标系仿射，后平移，参考上面的二维坐标系之间的变换）

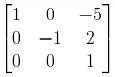


其中，矩阵的第一列为MC中的X轴在WC中的向量表示，第二列为MC中Y轴在WC中的向量表示，第三列为MC中的原点在WC中的表示

现在我们将相机坐标也加进去，相机坐标也成为观测坐标（View Coordinate）

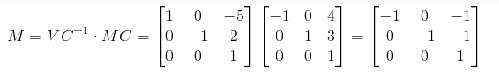


可以得到世界坐标转化为眼坐标的矩阵为（先将x''y''平移，后仿射）：



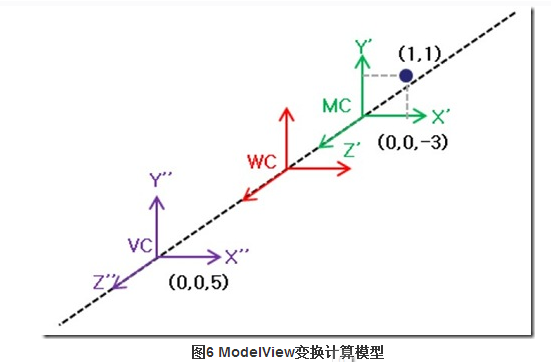
**记住一点：所有的平移、旋转、放缩都必须基于世界坐标做变化**

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。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，在上面的例子中，就是将上面的两个矩阵相乘。这就是opengl中的ModelView变换矩阵。这也是ModelView变换的名字的由来，就是通过场面的两个步骤得到的。

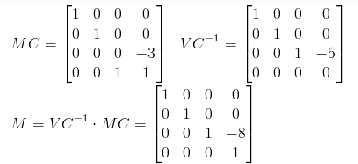


现在只要用上面的模型观测矩阵M乘以模型坐标系MC中的坐标就可以得到相机坐标系中的坐标了。模型观测变换的关键就是要得到相机坐标系中的坐标，**因为光照等计算都是在这个坐标系中完成的**

以下图为例进行验证：



通过上图可以得出



void testModelViewMatrix()

{

GLfloat matrix[16] = {0};

int size = GET\_ARRAY\_LENGTH(matrix);

glMatrixMode(GL\_MODELVIEW\_MATRIX);

glLoadIdentity();

glGetFloatv(GL\_MODELVIEW\_MATRIX, matrix);

showMatrix(matrix, size);

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

glGetFloatv(GL\_MODELVIEW\_MATRIX, matrix);

showMatrix(matrix, size);

glTranslatef(0.0f, 0.0f, -3.0f);

glGetFloatv(GL\_MODELVIEW\_MATRIX, matrix);

showMatrix(matrix, size);

}



到这里，整个ModelView变换就完成了，通过ModelView变换后得到就是相机坐标系中的坐标。在这个坐标系内典型的计算就是法线了。

实际上也就是project\*view\*model\*vertex

在默认的情况下，glulookAt位于原点，也就是一个单位矩阵，model矩阵也是一个单位矩阵，因此最初的模型视图矩阵也可以看做是View\*Model，接下来我们要先设置View，在设置Model，实验证明如果将两者调用顺序颠倒，所得的模型视图矩阵不一样，最后得到的效果也不一样

**坐标变换：**

<http://www.niubb.net/a/2015/04-29/329440.html>

<http://www.360doc.com/content/14/1028/09/19175681_420513219.shtml>

左右手坐标系：大拇指-x轴 食指-y轴 中指-z轴

Opengl 右手坐标系 directx 左手坐标系

**OpenGL坐标系之间的变换：**

本地坐标系—>世界坐标系->眼坐标系->裁剪坐标系

一般gluLookAt用于从世界坐标系到眼坐标系的转换，但是由于OpenGL里面模型视图矩阵直接将本地坐标系转换为眼坐标系，所以gluLookAt应该被用来设置模型视图矩阵

在openGL中进行几何变换的方式为，首先通过glTranslatef、glRoratef、glScalef等函数设置好几何变换矩阵（相当于对坐标系进行了变换），然后进行绘制，那么图形的投影坐标将受到设置好的几何变换矩阵所影响而显示出几何变换的效果，而不不是首先进行绘图然后再通过几何变换函数对已经存在的图形进行变换

在OpenGL中，顶点计算方法：

**屏幕坐标点 = 3D模型点 \* 几何变换栈矩阵(n…1) \* 投影变换栈矩阵(n…1)**

openGL坐标系可以分为：世界坐标系和当前绘图坐标系

**世界坐标系：**以屏幕中心为原点，你面对屏幕，你的右边是x正轴，上面是y正轴，上面是正轴，屏幕指向你的为z正轴

**当前绘图坐标系：**是绘制物体时的坐标系，程序刚初始化时，世界坐标系和当前绘图坐标系是重合的，当用glTranslatef、glScakef、glRotatef对当前绘图坐标系进行评议、伸缩、旋转变换之后，世界坐标系和当前绘图坐标系不再重合，改变之后，再用过来Verte3f等绘图函数绘图时候， 都是在当前绘图坐标系进行绘图，所有的函数参数也都是相对当前绘图坐标系来讲的

在二维情况下，所有的坐标变换（平移、放缩和旋转）都可以表示为P1 = M1\*P + M2，其中M1是一个2\*2的表达式

想象一下，如果我们想实现一个首先将坐标位置缩放，然后将缩放后的坐标旋转，最后将旋转后的坐标平移，我们必须一步一步地计算变换的坐标。更有效的方法是将变换组合，从而直接从初始坐标得到最后的坐标位置，这样就消除了中间坐标值的计算，计算量将大大的减少，因此，变换操作的有效实现是先形成变换矩阵、合并所有变换序列，最后计算变换坐标。

如果将2\*2矩阵表达式扩充为3\*3矩阵，就可以把二维几何变换的乘法和平移项组合成单一矩阵表示。这时将变换矩阵的第三列用于平移项，而左右的变换公式可表达为矩阵乘法，这样就用到了齐次坐标。为了这样的操作，必须解释二维坐标位置到三元列向量的矩阵表示。标准的实现技术是将二维坐标位置表示（x，y）扩充到三维表示（xh，yh，h），其中

x=xh/h y=yh/h

这样，普通的二维齐次坐标表示可写为(h\*x, h\*y, h)，对于二维几何变换，可以把齐次参数h取为任何非零值，因此，对于每个坐标点(x,y)可以有无数个等价的齐次表达式

有些图形库仅仅只包含了绕坐标系原点旋转或放缩

对于旋转：可以通过平移-旋转-平移操作顺序来实现绕任意选定的基准点进行旋转

1 平移对象使其基准点位置移动到坐标原点

2 绕坐标原点旋转

3 平移对象使基准点回到其原始位置

对于放缩也一样：

1 平移对象使固定点与坐标原点重合

2 对于坐标原点进行缩放

3 使用步骤1的反向平移将对象返回到原始位置

void testGeometric()

{

GLfloat vertices[] = {100.0f, 100.0f,

120.0f, 100.0f,

120.0f, 120.0f,

100.0f, 120.0f};

GLfloat colors[] = {1.0f, 0.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 0.0f, 1.0f,

1.0f, 1.0f, 0.0f};

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 0, vertices);

glColorPointer(3, GL\_FLOAT, 0, colors);

//glRotatef(45.0f, 0.0f, 0.0f, 1.0f);

//glScalef(1.5f, 1.0f, 1.5f);

glDrawArrays(GL\_QUADS, 0, 4);

}



void testGeometric()

{

GLfloat vertices[] = {100.0f, 100.0f,

120.0f, 100.0f,

120.0f, 120.0f,

100.0f, 120.0f};

GLfloat colors[] = {1.0f, 0.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 0.0f, 1.0f,

1.0f, 1.0f, 0.0f};

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 0, vertices);

glColorPointer(3, GL\_FLOAT, 0, colors);

**glRotatef(45.0f, 0.0f, 0.0f, 1.0f);**

**glScalef(1.5f, 1.0f, 1.5f);**

glDrawArrays(GL\_QUADS, 0, 4);

}



void testGeometric()

{

GLfloat vertices[] = {100.0f, 100.0f,

120.0f, 100.0f,

120.0f, 120.0f,

100.0f, 120.0f};

GLfloat colors[] = {1.0f, 0.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 0.0f, 1.0f,

1.0f, 1.0f, 0.0f};

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 0, vertices);

glColorPointer(3, GL\_FLOAT, 0, colors);

**glScalef(1.5f, 1.0f, 1.5f);**

**glRotatef(45.0f, 0.0f, 0.0f, 1.0f);**

glDrawArrays(GL\_QUADS, 0, 4);

}



上面的例子注意点：Sx与Sy缩放是沿着x和y方向缩放的，可以通过在应用缩放变换之前，将对象所希望的缩放方向旋转到与坐标轴一致而在其他方向上缩放对象

对可用glMatrixMode选择的4种模式（建模观察、投影、纹理和颜色）中的每一种，opengl都维护一个矩阵栈。开始，每一个栈仅包含单位矩阵，在处理场景的任何时刻，栈顶的矩阵称为该模式的“当前矩阵”。在指定观察和几何变换后，建模观察栈顶是一个应用于场景的观察变换和各种几何变换的4\*4复合矩阵。有时，需要创建多个视图和变换序列，然后分别保存复合矩阵。因此，opengl提供至少深度为32的建模观察栈，而有的实现允许在建模观察栈中保存超多32个矩阵，另外三个矩阵模式的栈深度至少为2

GLint modelViewStackDepth = 0, projectStackDepth = 0, textureStackDepth = 0;

glGetIntegerv(GL\_MAX\_MODELVIEW\_STACK\_DEPTH, &modelViewStackDepth);

glGetIntegerv(GL\_MAX\_PROJECTION\_STACK\_DEPTH, &projectStackDepth);

glGetIntegerv(GL\_MAX\_TEXTURE\_STACK\_DEPTH, &textureStackDepth);

cout << "modelViewStackDepth = " << modelViewStackDepth << endl;

cout << "projectStackDepth = " << projectStackDepth << endl;

cout << "textureStackDepth = " << textureStackDepth << endl;



如何获取当前栈中有多少个矩阵：

showModelViewStackMatrixCount();

glPushMatrix();

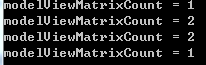
showModelViewStackMatrixCount();

glLoadIdentity();

showModelViewStackMatrixCount();

glPopMatrix();

showModelViewStackMatrixCount();



GLfloat matrix[] = {1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f};

showModelViewStackMatrixCount();

glPushMatrix();

glMultMatrixf(matrix);

glPushMatrix();

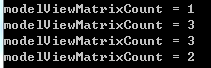
showModelViewStackMatrixCount();

glLoadIdentity();

showModelViewStackMatrixCount();

glPopMatrix();

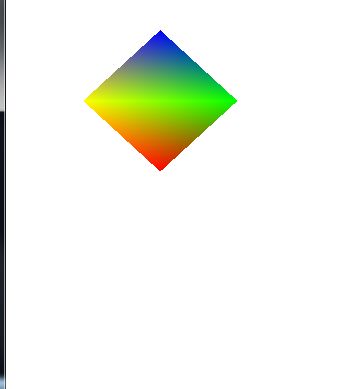
showModelViewStackMatrixCount();



glRotatef(45.0f, 0.0f, 0.0f, 1.0f);

glTranslatef(200.0f, 0.0f, 0.0f);

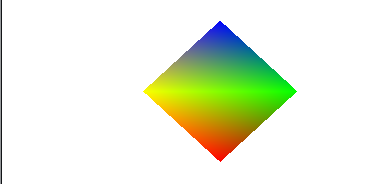
glDrawArrays(GL\_QUADS, 0, 4);



glTranslatef(200.0f, 0.0f, 0.0f);

glRotatef(45.0f, 0.0f, 0.0f, 1.0f);

glDrawArrays(GL\_QUADS, 0, 4);



上面的两种情况（理解）：

正向看，是改变局部坐标

反向看：是基于世界坐标

在选在opengl裁剪窗口和视口之前，必须建立合适的模式以便构建从世界坐标系到屏幕坐标系变换的矩阵。Opengl裁剪函数使用-1到1的规范化坐标系。gluOrtho2D函数设定变换矩阵的三维版本，将裁剪窗口中的对象映射到规范化坐标系。规范化正方形外的对象（及在裁剪窗口外的对象）不在显示的场景中出现。

Opengl可以为各种应用建立多个视口。获取当前活动视口参数的查询函数如下：

GLint\* vpArray = new GLint[4];

glGetIntegerv(GL\_VIEWPORT, vpArray);

cout << "x = " << vpArray[0] << ", y = " << vpArray[1] << endl;

cout << "width = " << vpArray[2] << ", height = " << vpArray[3] << endl;



看看libgdx中的draw函数：

**public** **void** draw (TextureRegion region, **float** x, **float** y, **float** originX, **float** originY, **float** width, **float** height,

**float** scaleX, **float** scaleY, **float** rotation) {

**if** (!drawing) **throw** **new** IllegalStateException("SpriteBatch.begin must be called before draw.");

Texture texture = region.texture;

**if** (texture != lastTexture) {

switchTexture(texture);

} **else** **if** (idx == vertices.length) //

renderMesh();

// bottom left and top right corner points relative to origin

**final** **float** worldOriginX = x + originX; **// 记录初始的锚点**

**final** **float** worldOriginY = y + originY;

**float** fx = -originX; **// 将物体的锚点移到原点**

**float** fy = -originY;

**float** fx2 = width - originX;

**float** fy2 = height - originY;

// scale

**if** (scaleX != 1 || scaleY != 1) { **// 执行放缩操作**

fx \*= scaleX;

fy \*= scaleY;

fx2 \*= scaleX;

fy2 \*= scaleY;

}

// construct corner points, start from top left and go counter clockwise

**final** **float** p1x = fx;

**final** **float** p1y = fy;

**final** **float** p2x = fx;

**final** **float** p2y = fy2;

**final** **float** p3x = fx2;

**final** **float** p3y = fy2;

**final** **float** p4x = fx2;

**final** **float** p4y = fy;

**float** x1;

**float** y1;

**float** x2;

**float** y2;

**float** x3;

**float** y3;

**float** x4;

**float** y4;

// rotate

**if** (rotation != 0) { **// 执行旋转操作**

**final** **float** cos = MathUtils.*cosDeg*(rotation);

**final** **float** sin = MathUtils.*sinDeg*(rotation);

x1 = cos \* p1x - sin \* p1y;

y1 = sin \* p1x + cos \* p1y;

x2 = cos \* p2x - sin \* p2y;

y2 = sin \* p2x + cos \* p2y;

x3 = cos \* p3x - sin \* p3y;

y3 = sin \* p3x + cos \* p3y;

x4 = x1 + (x3 - x2);

y4 = y3 - (y2 - y1);

} **else** {

x1 = p1x;

y1 = p1y;

x2 = p2x;

y2 = p2y;

x3 = p3x;

y3 = p3y;

x4 = p4x;

y4 = p4y;

}

x1 += worldOriginX; **// 恢复原点到锚点的位置**

y1 += worldOriginY;

x2 += worldOriginX;

y2 += worldOriginY;

x3 += worldOriginX;

y3 += worldOriginY;

x4 += worldOriginX;

y4 += worldOriginY;

**final** **float** u = region.u;

**final** **float** v = region.v2;

**final** **float** u2 = region.u2;

**final** **float** v2 = region.v;

vertices[idx++] = x1;

vertices[idx++] = y1;

vertices[idx++] = color;

vertices[idx++] = u;

vertices[idx++] = v;

vertices[idx++] = x2;

vertices[idx++] = y2;

vertices[idx++] = color;

vertices[idx++] = u;

vertices[idx++] = v2;

vertices[idx++] = x3;

vertices[idx++] = y3;

vertices[idx++] = color;

vertices[idx++] = u2;

vertices[idx++] = v2;

vertices[idx++] = x4;

vertices[idx++] = y4;

vertices[idx++] = color;

vertices[idx++] = u2;

vertices[idx++] = v;

}