OpenGL变换

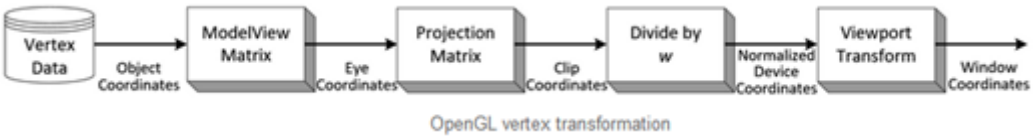
学无止境
猪八戒肚子鼓鼓的!

> **原文地址如下**

OpenGL Transformation
www.songho.ca/opengl/gl_transform.htm...

综述

在渲染管线中，几何数据(例如，顶点位置和法线向量)首先经过顶点变换(Vertex Operation)和图元组装操作(Primitive Assembly Operation)，然后进入到光栅化阶段。



OpenGL顶点变换

- **物体坐标(Object Coordinates)**

物体坐标即局部坐标，定义了未经过变换的物体的初始坐标和方位。为了对物体进行变换，我们可以使用glRotatef()、glTranslatef()、glScalef()等接口。

- **观察坐标(Eye Coordinates)**

利用GL_MODELVIEW矩阵[9]我们可以将物体坐标变换到观察坐标。OpenGL通过GL_MODELVIEW矩阵将物体从物体空间变换到观察空间。GL_MODELVIEW矩阵是Model矩阵和View矩阵的组合 $(M_{view} * M_{model})$ 。模型变换(Model Transform)可以将物体从物体空间变换到世界空间，而观察变换(View Transform)可以将物体从世界空间变换到观察空间。

$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modelView} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix} = M_{view} \cdot M_{model} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix}$$

局部坐标与观察坐标变换关系

我们注意到OpenGL中没有单独的观察矩阵(Camera/View Matrix)，因此为了模拟摄像机的变换效果，我们对场景(3D物体和光照)进行与观察变换相反的变换。也就是说，OpenGL观察空间中的摄像机被固定在原点(0, 0, 0)的位置，朝向Z轴的反方向，而且摄像机无法进行位置等相关变换。从下文的ModelView Matrix可以了解到更多关于GL_MODELVIEW的信息。

为了进行光照计算，法线向量也需要从物体空间变换到观察空间。向量的变换与坐标变换有所不同。将法线从物体空间变换到观察空间需要乘以GL_MODELVIEW矩阵的逆变换的转置。可以从Normal Vector Transformation了解更多关于法线变换的信息。

$$\begin{pmatrix} nx_{eye} \\ ny_{eye} \\ nz_{eye} \\ nw_{eye} \end{pmatrix} = \left(M_{modelView}{}^{-1}\right)^{T} \cdot \begin{pmatrix} nx_{obj} \\ ny_{obj} \\ nz_{obj} \\ nw_{obj} \end{pmatrix}$$

顶点法线变换与顶点坐标变换的差异

- **裁剪坐标(Clip Coordinates)**

利用GL_PROJECTION矩阵可以将观察坐标变换到裁剪坐标。GL_PROJECTION矩阵定义了视椎体(viewing volume/frustum)以及物体的投影方式(正交投影还是透射投影)。由于顶点(x, y, z)会被裁剪到[-w +w]的范围内，所以将其称为裁剪空间。

- **标准设备空间(Normalized Device Coordinates, NDC)**

通过将裁剪坐标除以w分量(通常称为透射除法)可以得到NDC。由于没有被平移缩放成屏幕的像素点，所以没有并没有称它为屏幕坐标(window/screen coordinates)。坐标(x, y, z)都落在[-1, 1]范围内。

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

裁剪坐标与NDC坐标的变换关系

- **屏幕坐标(Screen Coordinates)**

NDC通过视口变换可以得到屏幕坐标。为了适应屏幕大小，NDC坐标需要经过平移缩放操作。OpenGL渲染管线最终会将屏幕坐标送到光栅化阶段来得到片段。图像最终会显示到屏幕上，通过glViewport()指令可以获取屏幕的尺寸。而glDepthRange()指令用来获取屏幕坐标的z分量，屏幕坐标可以通过这两个指令计算得到。

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{\mathbf{w}}{2} x_{ndc} + \left(\mathbf{x} + \frac{\mathbf{w}}{2}\right) \\ \frac{\mathbf{h}}{2} y_{ndc} + \left(\mathbf{y} + \frac{\mathbf{h}}{2}\right) \\ \frac{\mathbf{f}-\mathbf{n}}{2} z_{ndc} + \frac{(\mathbf{f}+\mathbf{n})}{2} \end{pmatrix}$$

NDC坐标和屏幕空间坐标对应关系

通过NDC坐标和屏幕坐标的线性关系可以得到视口变换的方程式[Q]：

$$\begin{cases} -1 & \to \mathbf{x} \\ 1 & \to \mathbf{x}+\mathbf{w} \end{cases}, \quad \begin{cases} -1 & \to \mathbf{y} \\ 1 & \to \mathbf{y}+\mathbf{h} \end{cases}, \quad \begin{cases} -1 & \to \mathbf{n} \\ 1 & \to \mathbf{f} \end{cases}$$

根据该对应关系获取上面的矩阵关系

OpenGL变换矩阵

OpenGL使用4x4矩阵[Q]进行变换操作。矩阵的16个元素以列主序的方式存储在1维数组中，通过对其进行转置操作可以变换成标准的行主序形式。

OpenGL有四个不用类型的矩阵：GL_MODELVIEW、GL_PROJECTION、GL_TEXTURE和GL_COLOR。你可以在代码中使用glMatrixMode()接口来切换当前的类型。例如，通过glMatrixMode(GL_MODELVIE)可以选中GL_MODELVIE矩阵。

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

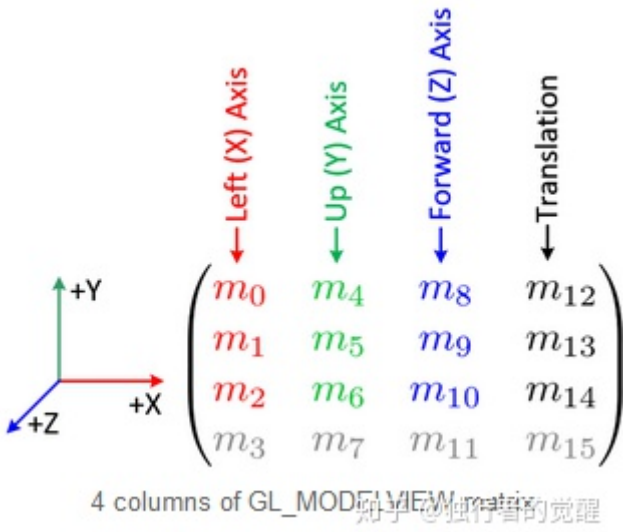
OpenGL Transform Matrix

- **Model-View Matrix(GL_MODELVIEW)**

GL_MODELVIEW矩阵包含了viewing矩阵和modeling矩阵。为了变换视角，你需要将场景进行反向变换。gluLookAt()接口专门用来进行视角变换的。

矩阵最右列的三个元素 (m_{12}, m_{13}, m_{14}) 是用来平移变换的glTranslatef()。 m_{15} 齐次坐标主要是用来进行投影变换的。

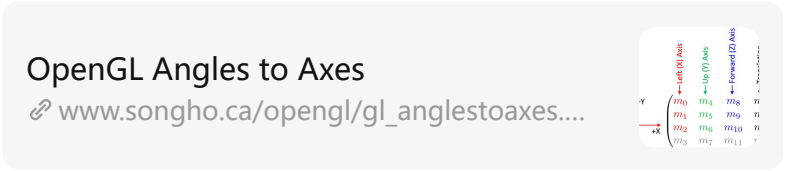
三元素组 (m_0, m_1, m_2) 、 (m_4, m_5, m_6) 和 (m_8, m_9, m_{10}) 用来进行几何变换和仿射变换的。例如，旋转glRotatef()、缩放glScalef()。注意到这三个向量组代表的其实是三个正交的坐标轴。



4 columns of GL_MODELVIEW matrix

- (m_0, m_1, m_2) ：X轴，左向量，默认(1, 0, 0)。
- (m_4, m_5, m_6) ：Y轴，上向量，默认(0, 1, 0)。
- (m_8, m_9, m_{10}) ：Z轴，前向量，默认(0, 0, 1)。

GL_MODELVIEW矩阵的构造可以使用方向角或者查找向量，而不需要使用OpenGL的变换函数。下面的代码演示了如何构造GL_MODELVIEW矩阵。



我们注意到OpenGL中矩阵的乘法是逆向的，例如向量首先乘以 M_A ，然后乘以 M_B ，OpenGL先执行 $M_B \times M_A$ 然后在乘以向量。所以后面的变换先计算，前面的变换后计算。

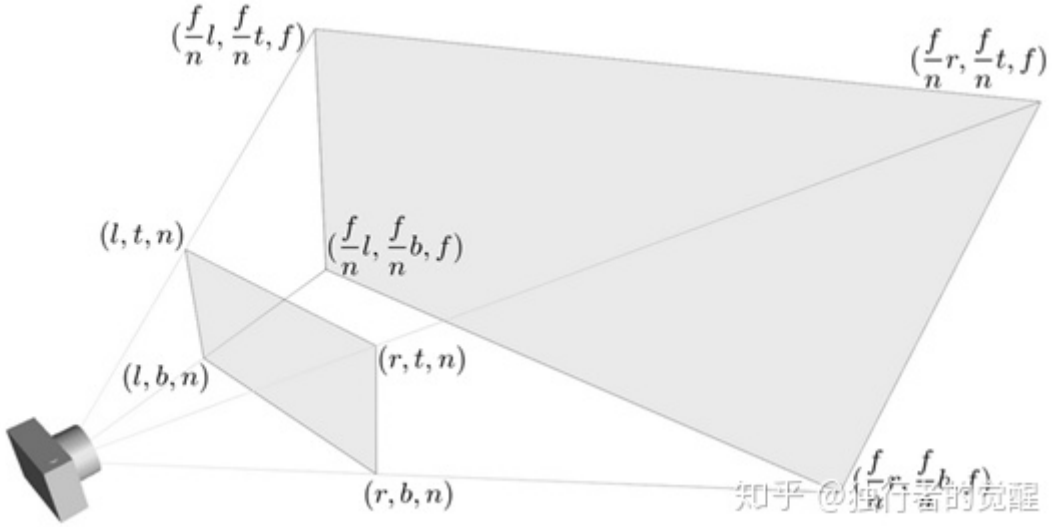
$$v' = M_B \cdot (M_A \cdot v) = (M_B \cdot M_A) \cdot v$$

```
// 物体首先进行平移操作，然后进行旋转
glRotatef(angle, 1, 0, 0);    // 绕x轴旋转angle度
glTranslatef(x, y, z);        // 移动物体
drawObject();
```


- **Projection Matrix(GL_PROJECTION)**

GL_PROJECTION矩阵用来定义视椎体。视椎体决定了物体或者物体的那些部分被剔除。此外，它还决定了3D场景如何被投射到屏幕上。

OpenGL提供了两个进行GL_PROJECTION变换的函数。glFrustum()用来构造透射投影，glOrtho()用来进行正交(平行)投影。这两个函数都需要6个参数来确定6个裁剪平面：上下左右近远。视椎体的8个顶点如下图所示。



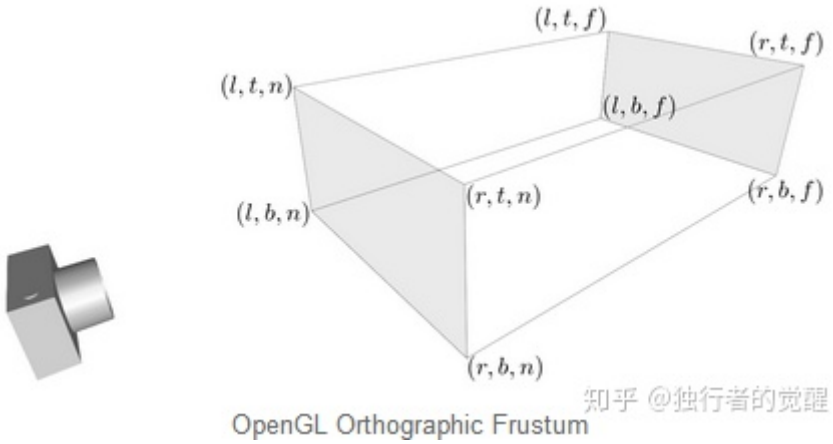
透射投影视椎体

通过相似三角形定理可以简单的计算出远平面的顶点坐标，例如，远平面坐标的坐标计算如下：

$$\frac{far}{near} = \frac{left_{far}}{left}, \quad left_{far} = \frac{far}{near} \cdot left$$

对于正交投影而言，该比例为1，因此远近平面对于坐标的xy值相同。

gluPerspective()和gluOrtho2D()这两个函数接受更少的参数。gluPerspective()函数需要4个参数：垂直视野角(FOV)，屏幕比例(aspect ratio)、远平面和近平面的距离。下面的代码说明了gluPerspective()和glFrustum()其实是等价的。



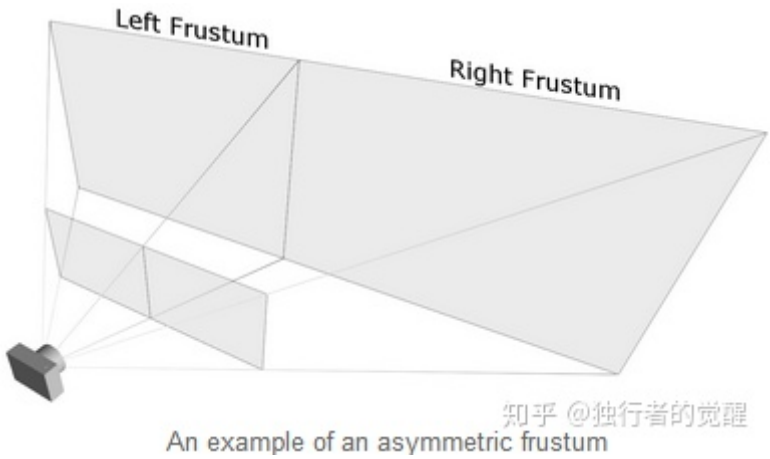
正交投影视椎体

```
// This creates a symmetric frustum.
// It converts to 6 params (l, r, b, t, n, f) for glFrustum()
// from given 4 params (fovy, aspect, near, far)
void makeFrustum(double fovY, double aspectRatio, double front, double back)
{
    const double DEG2RAD = 3.14159265 / 180;

    double tangent = tan(fovY/2 * DEG2RAD);    // tangent of half fovY
    double height = front * tangent;           // half height of near plane
    double width = height * aspectRatio;       // half width of near plane

    // params: left, right, bottom, top, near, far
    glFrustum(-width, width, -height, height, front, back);
}
```


然而，如果你需要一个非对称的视椎体，那么你需要使用glFrustum()。如果你需要将大屏幕渲染到两个相邻的屏幕上，你可以将视椎体分为左右两个非对称的视椎体，然后分别渲染到两个视椎体上。



An example of an asymmetric frustum

多视椎体实例

- **Texture Matrix(GL_TEXTURE)**

进行纹理映射前需要将纹理坐标(s, t, r, q)乘上GL_TEXTURE矩阵。默认情况下GL_TEXTURE矩阵是单位矩阵，所以纹理会根据纹理坐标准确的映射到物体上面。通过修改GL_TEXTURE矩阵我们可以实现纹理的平移、旋转、伸展以及收缩等操作。

```
// rotate texture around X-axis
glMatrixMode(GL_TEXTURE);
glRotatef(angle, 1, 0, 0);
```


- **Color Matrix(GL_COLOR)**

颜色向量(r, g, b, a)需要乘上GL_COLOR矩阵。它能够实现颜色空间的转换以及颜色分量的交换。GL_COLOR矩阵通常不使用，而且需要GL_ARB_imaging扩展。

编辑于 2019-07-31 13:22

计算机图形学 OpenGL 游戏开发