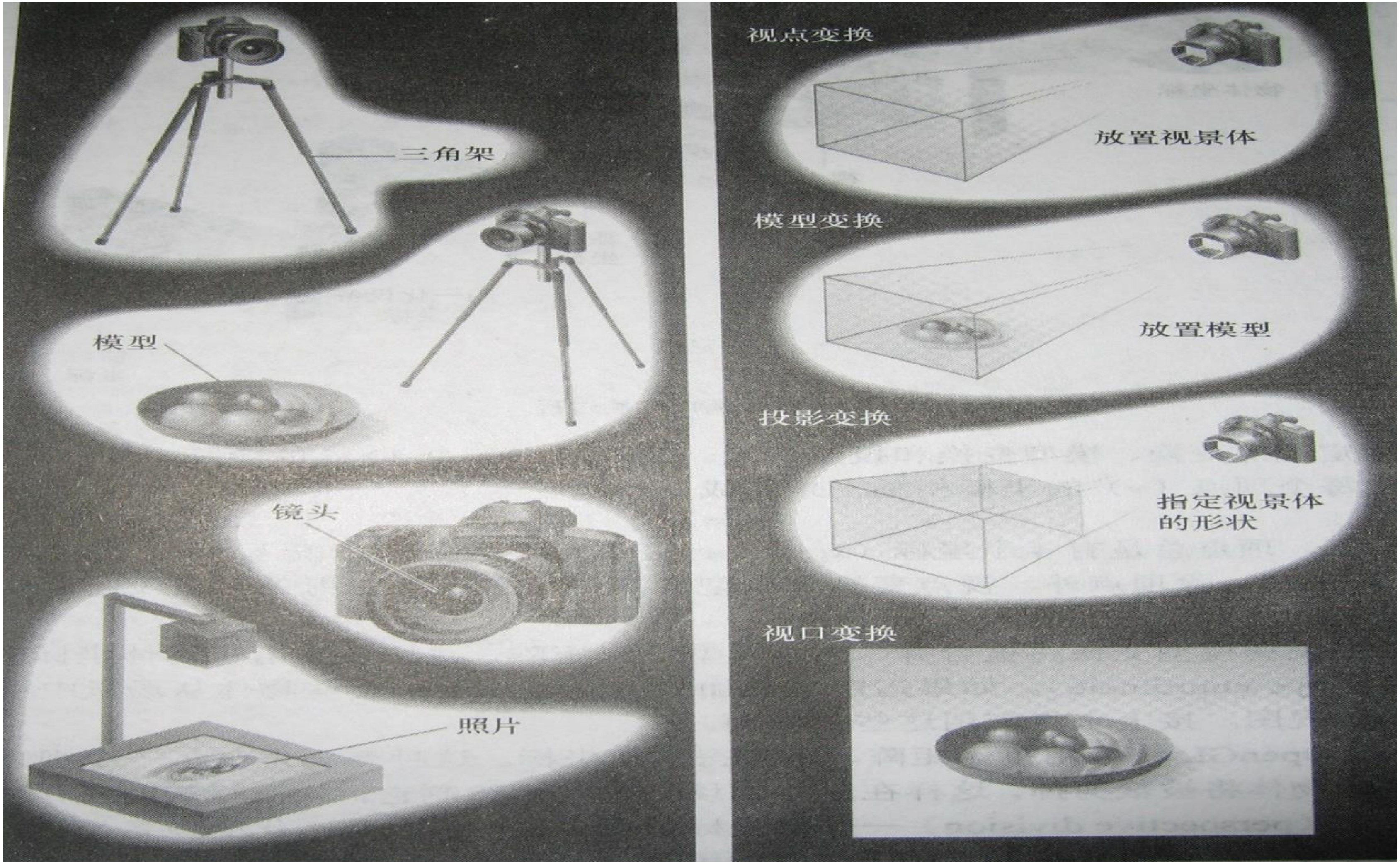
Outlines

- Viewing Pipeline

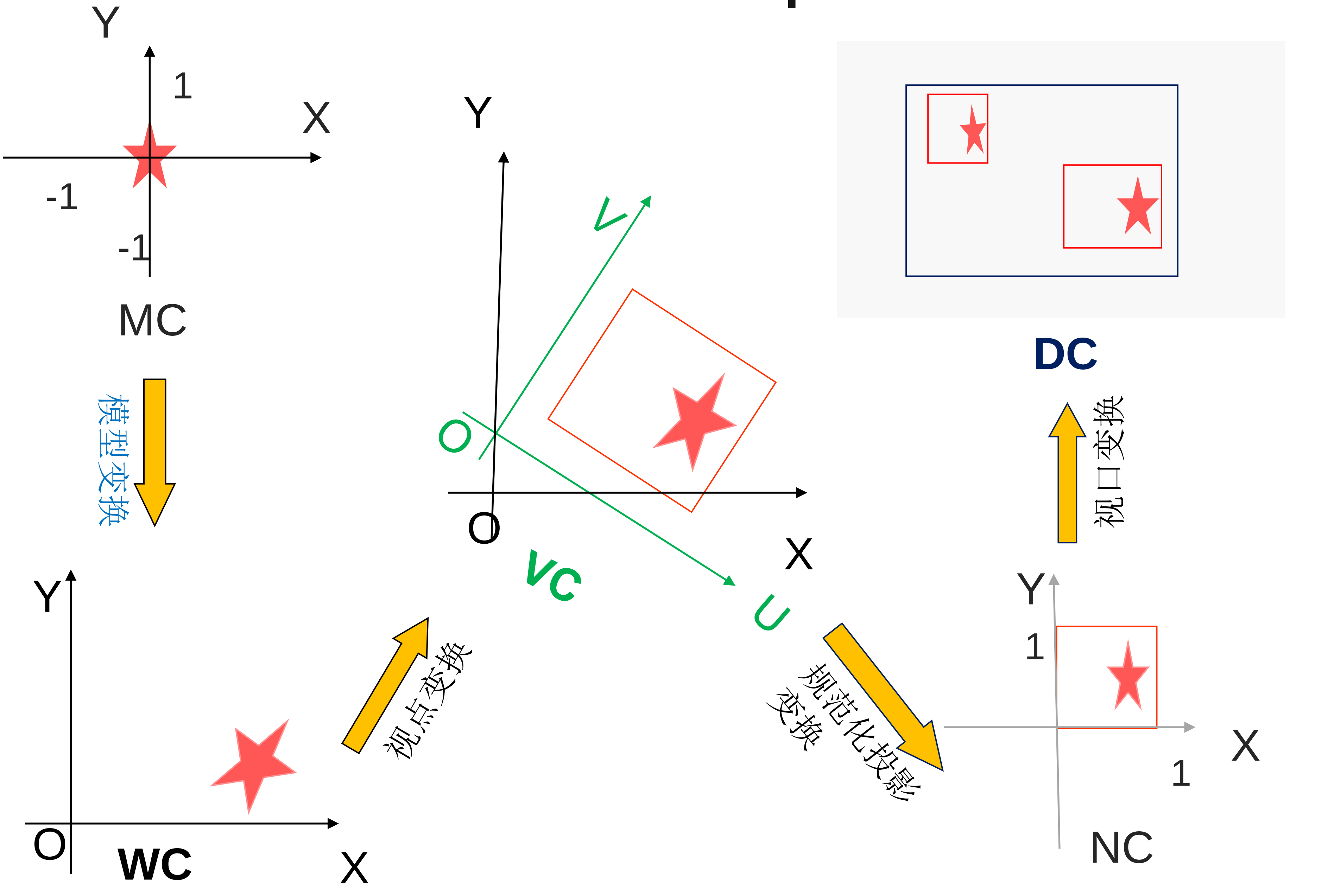
- Transformations in Viewing Pipeline
  - ModelView Transformation
    - Model Transformation建模变换
    - View Transformation视点/相机/眼变换
  - Projection Transformation
  - Viewport Transformation

类比相机成像过程中的四步变换

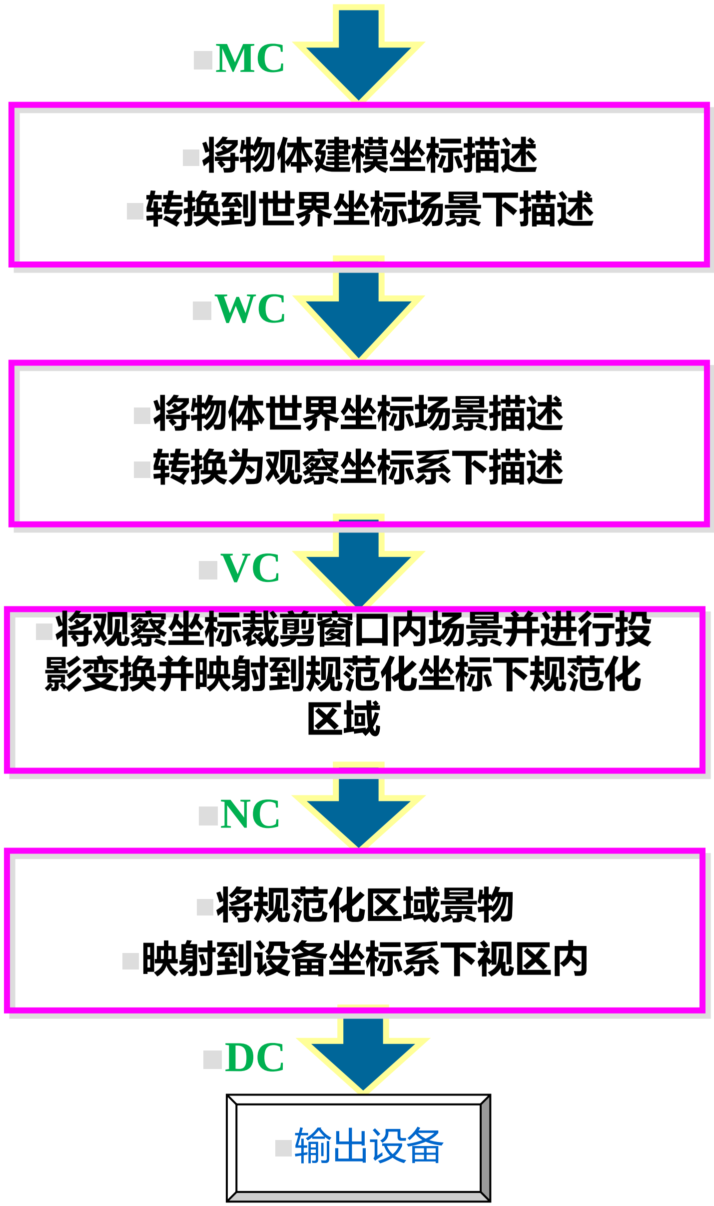
- 视点变换+模型变换->投影变换->视口变换



2D Example



Viewing Pipeline



- MC/OC 模型/对象坐标系

—>Model模型变换

- WC世界坐标系

—>Viewpoint视点变换(相机/眼/观察变换)

- VC观察坐标系

—>N-Projection规范化投影变换

- NC规范化坐标系

—>Viewport视口变换(窗区视区变换)

- DC设备坐标系

5 type Coordinates

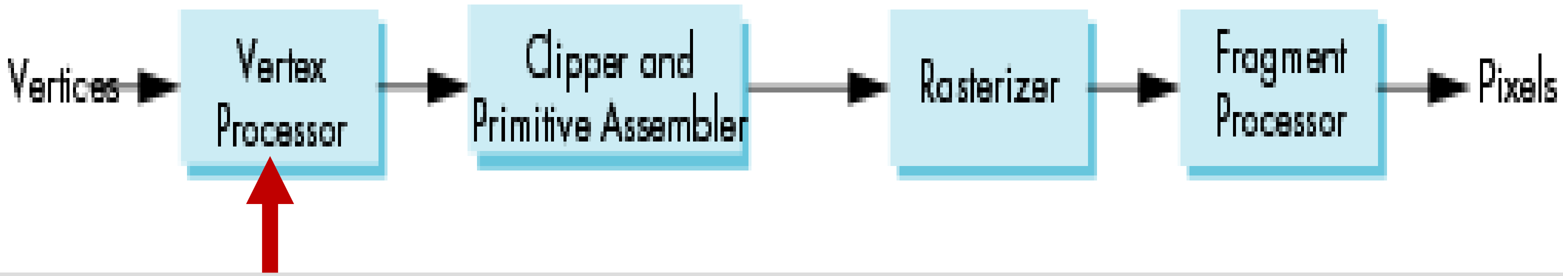
- **MC(Modeling Coordinates)**
  - 对象坐标：依物体而建,单位用户自定,坐标取值范围是整个实数域。
- **WC(World Coordinates)**
  - 世界坐标：场景坐标, 单位由用户自定,取值范围是整个实数域。
- **VC(View Coordinates)：**
  - 观察坐标，单位与WC一致，取值范围是整个实数域
- **NC(Normalized Coordinates)：**
  - 规范化坐标,一种虚拟的坐标系，作用是使坐标值去量纲化，使得图形软件包独立于输出设备
- **DC(Device Coordinates)：**
  - 设备坐标，原点，轴单位等依具体设备的不同而不同．长度以光栅单位（两个象素之距）为单位．取值范围是整数的有限区域

4 Type Transformations

- 模型变换Model Transformation
  - MC转换到WC
- 视点变换Viewpoint Transformation
  - 又称相机变换，WC转换到VC
- 投影变换Projection Transformation

  包含规范化变换和视见体变换等，VC转换为NDC
- 视口变换Viewport Transformation
  - 又称窗区视区变换，NDC转换为DC

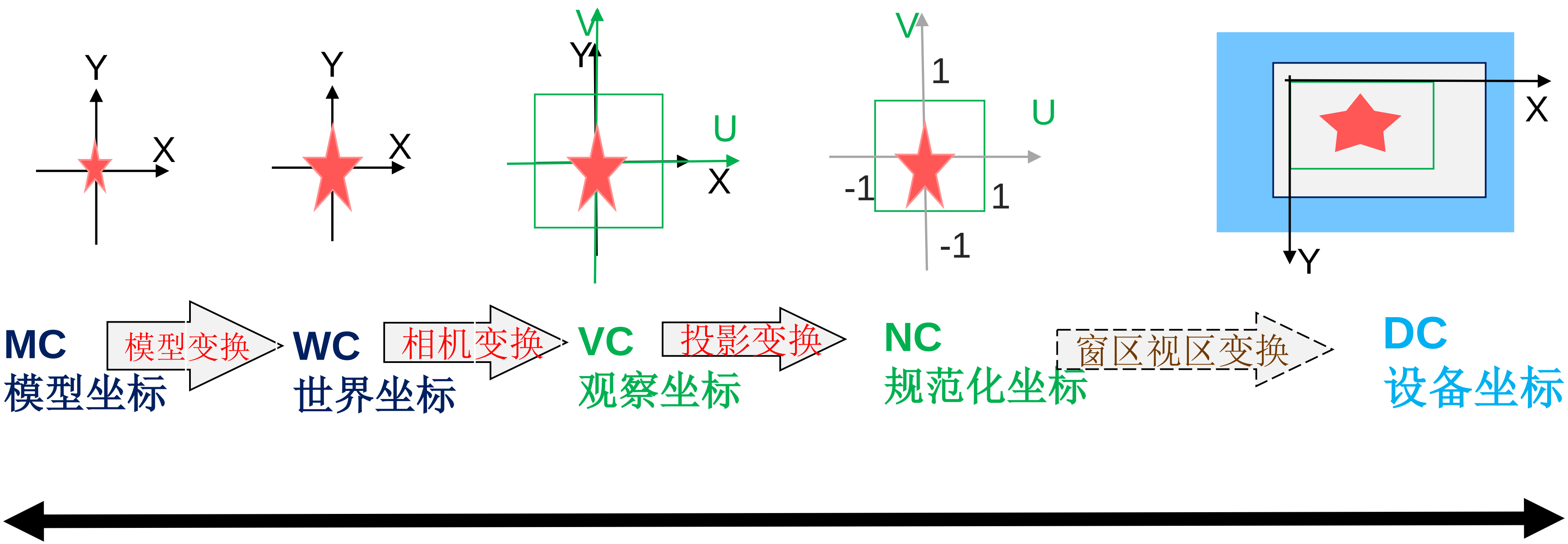
1.Vertex Processing



- **Much of the work in the pipeline is in converting object representations from one coordinate system to another**
  - **Object coordinates(OC/MC)**
  - **World coordinates(WC)**
  - **Camera (eye) coordinates(VC)**
  - **Screen coordinates（DC）**

- Every change of coordinates is equivalent to a matrix transformation
- Vertex processor also computes vertex colors

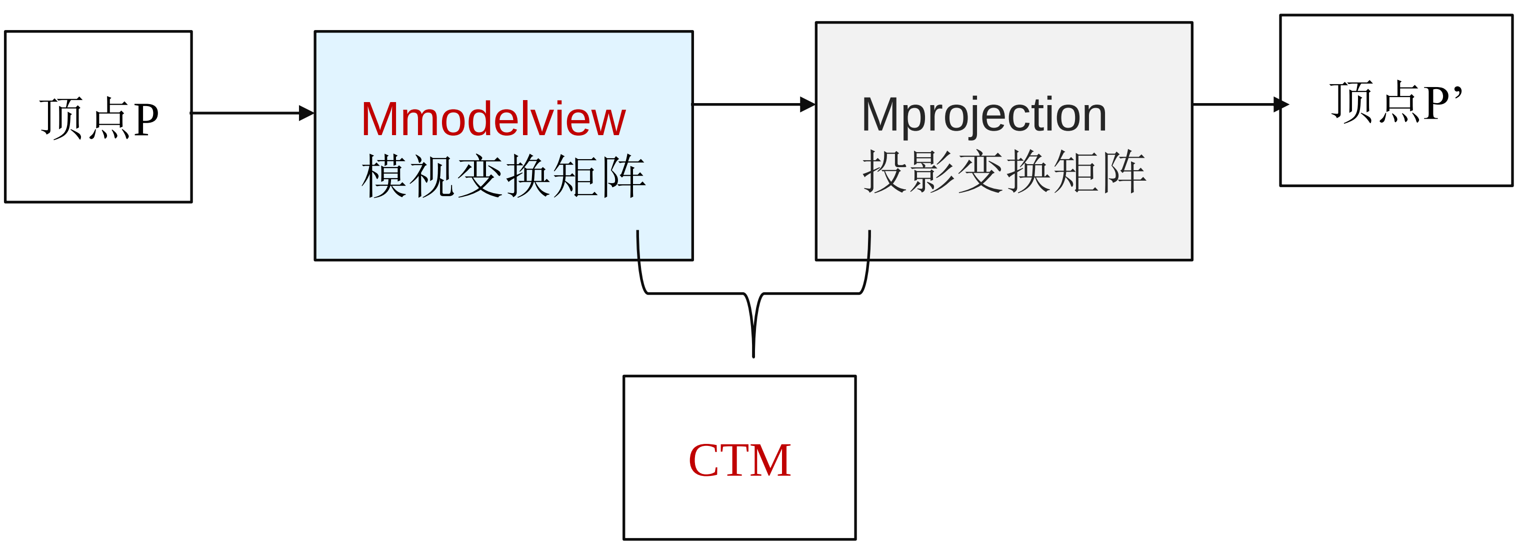
Viewing Pipeline



- *几何变换通常由顶点着色器完成，输出NC下的顶点坐标*
  - *包含MVP三种变换: 模型变换M，视点变换V，投影变换P，*
  - *gl_position=P*V*M*position*
    - *之前示例，是默认M,V,P为单位矩阵，或者说默认MC,WC,VC,NC重合而直接在NC下定义顶点坐标，所以 gl_position=position*

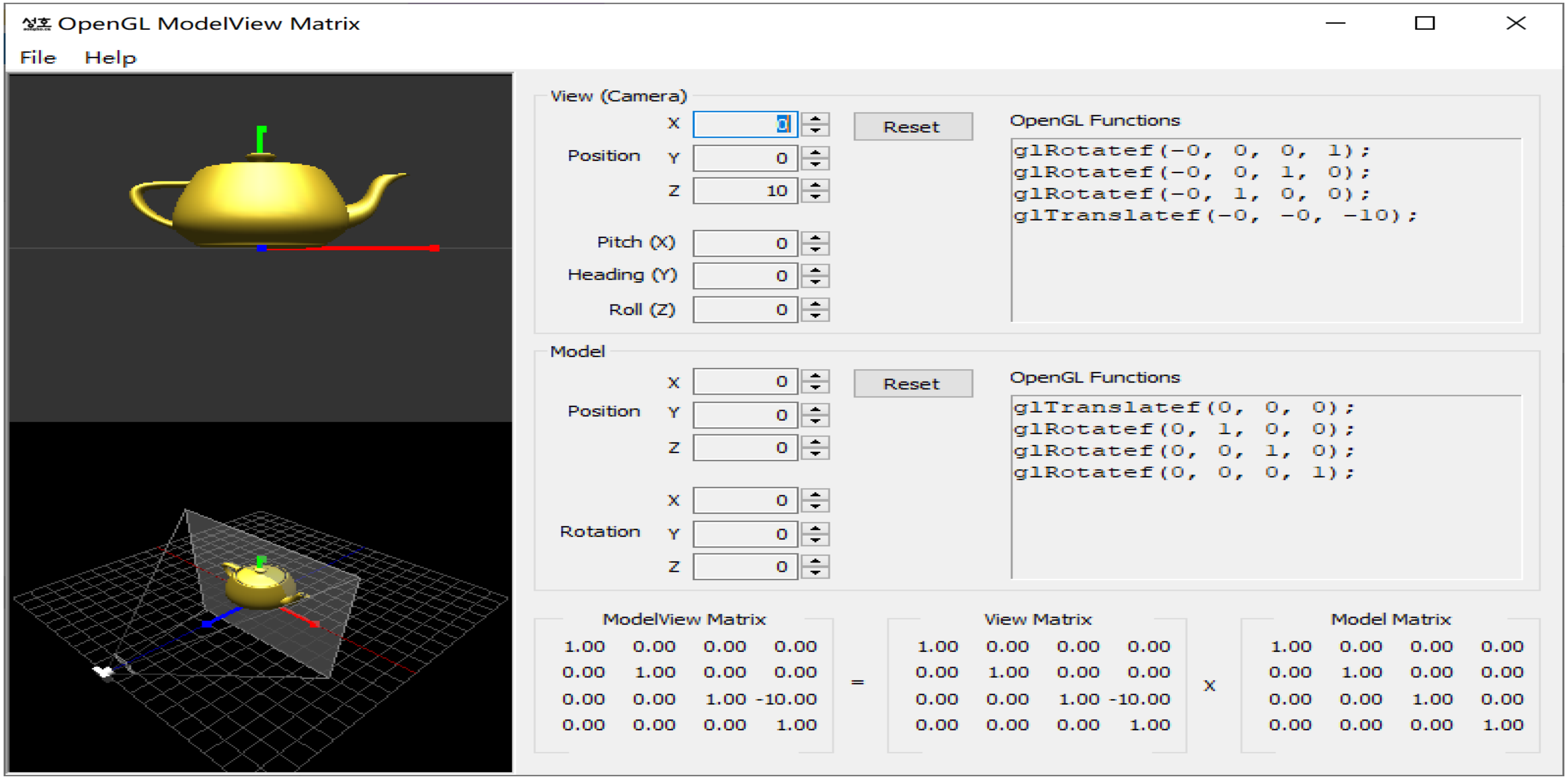
CTM当前变换矩阵

- Position'= CTM * Position
  - Current Transformation Matrix(CTM:当前变换矩阵)
  - CTM=Mprojection * Mmodelview= P*V*M;   (MVP:几何变换)
  - 当前变换矩阵分为模视变换和投影变换两个复合变换
  - 模视变换又进一步分为模型变换和视点变换这对偶变换



模视变换modelview

- CTM=MVP= P*V*M=P*(V*M);
- ModelView=V*M; //模视变换矩阵



Outlines

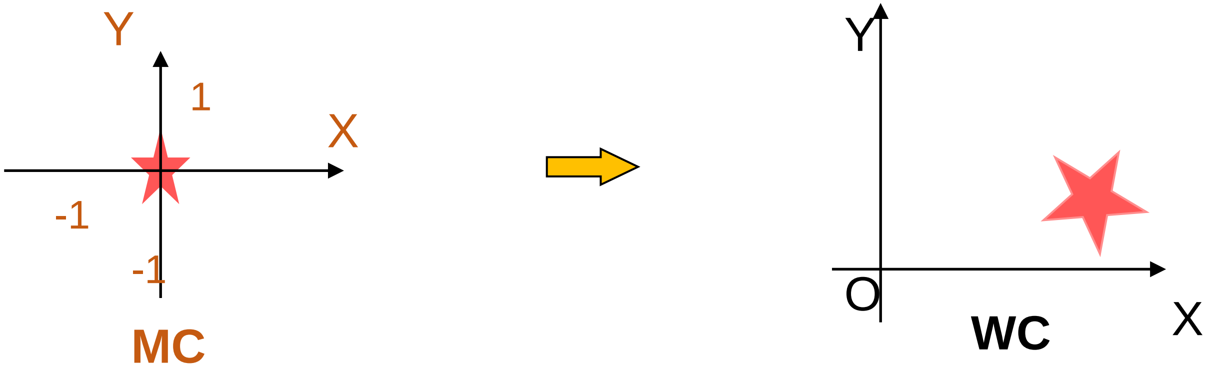
- Viewing Pipeline
- Transformations in Viewing Pipeline
  - ModelView Transformation
    - Model Transformation建模变换
    - View Transformation视点变换/相机变换
  - Projection Transformation
  - Viewport Transformation

Model Transformation模型变换

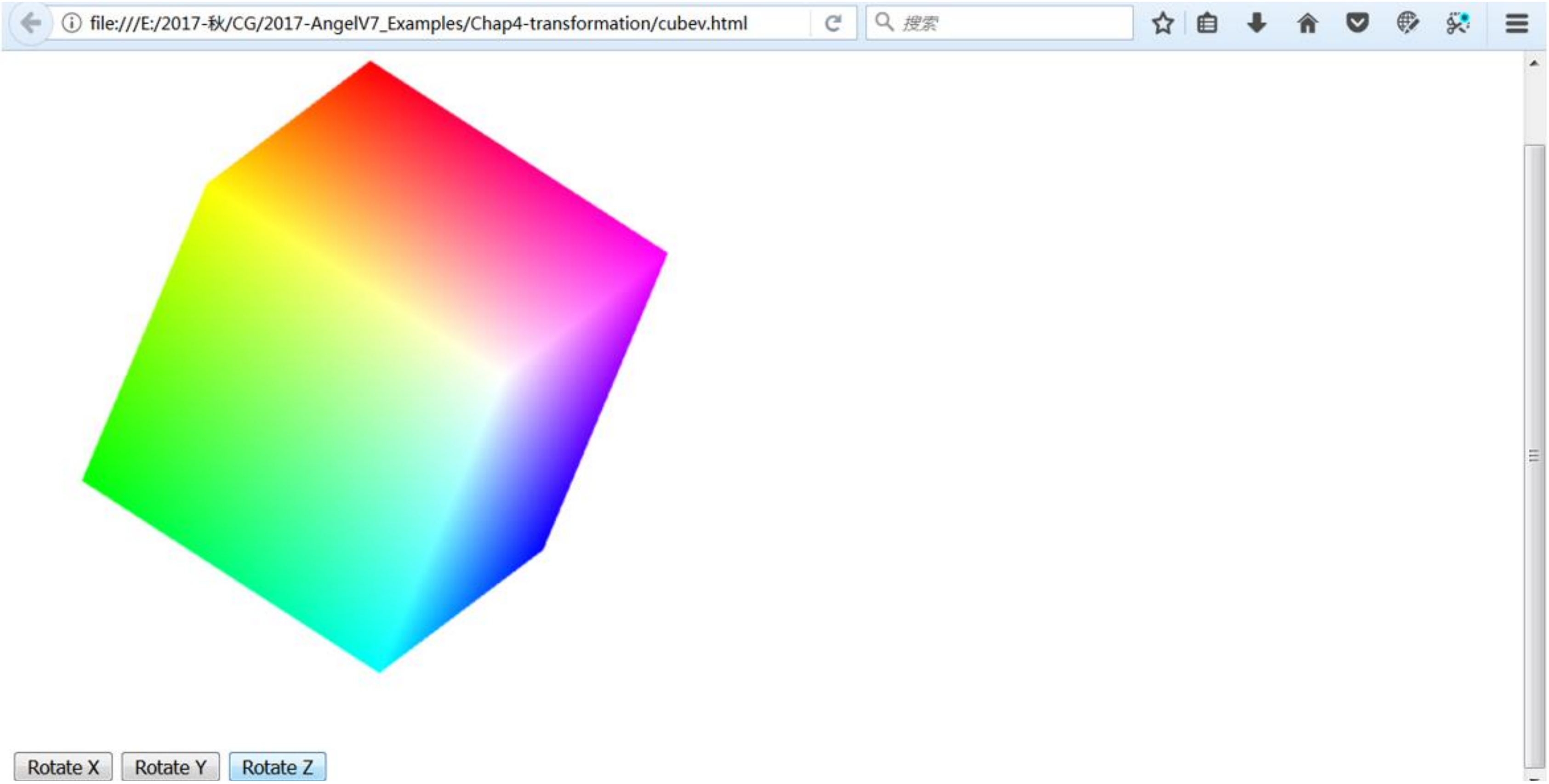
“摆放物体” ： MC to WC

从物体自身描述的坐标系OC/MC→对场景描述的世界坐标系WC.

- 应该根据实际的情况推导出实际的模型变换矩阵！
- 一般是（但不是总是）先依对象坐标原点作旋转，缩放等标准变换，然后再将物体平移到场景中相应位置
  - General： P’=T*R*S*P



建模变换实例2： Cube / Cubev



Cube中的变换

- 任何绕固定点的旋转，都可以分解成三个绕坐标轴的旋转，顺序可能不同，但是都可以分解实现

顶点位置： $Position' = R * Position$

$$R = R_X\,(thetaX) \cdot R_Y(thetaY) \cdot R_Z\,(thetaZ)$$

- 模型变换矩阵：R

Adding Buttons for Rotation

- 分别控制三个旋转方向

```
var xAxis = 0;
var yAxis = 1;
var zAxis = 2;
var axis = 0;
var theta = [ 0, 0, 0 ];
var thetaLoc;

document.getElementById( "xButton" ).onclick =
function () {axis = xAxis;    };
document.getElementById( "yButton" ).onclick =
function () {axis = yAxis;    };
document.getElementById( "zButton" ).onclick =
function () {axis = zAxis;    };
```

see cube.js

Where apply transformation?

1. in application: compute new vertexs
2. in vertex shader : send ModelView matrix
3. in vertex shader: send angles

第一种方法
在JS中计算CTM , 对图形进行变换生成新顶点，再发送给GPU显示处理。
当顶点数据量大时，IO负荷太大，不采纳。
第二种方法
在JS中计算CTM当前变换矩阵的16个数值并发送给GPU。
传送数据大大减少，着色器中只需要用当前模式变换矩阵计算新顶点。
第三种方法（cube采用方法）
在JS中计算三个方向的旋转角累积量的3个欧拉角数值，并发送给GPU。
传送的数据更少，着色器需要根据参数生成当前变换矩阵，再计算新顶

第三种方法非常适合现代GPU
因为矩阵中三角函数运算在GPU中是硬编码（计算时间几乎可以忽略不计）

方法1 JS中计算新顶点

- 第一种方法
  - 在JS中计算CTM，对图形进行变换生成新顶点再发送给GPU显示处理。

(原来的固定流水线法，数据传送量大，不采纳）

//JS 中实现的参考伪代码:

```
modelviewMatrix = mat4();
modelviewMatrix=mult(modelviewMatrix ,rotate(theta[xAxis]));
modelviewMatrix=mult(modelviewMatrix ,rotate(theta[yAxis]));
modelviewMatrix=mult(modelviewMatrix ,rotate(theta[zAxis]));
// CTM= modelviewMatrix =I* RXRYRZ

P= modelviewMatrix*P;

Render(P);
```

方法2：JS中计算矩阵并发送GPU

- 在JS中计算CTM当前变换矩阵（16个数值），并发送给GPU。

```
modelviewMatrix = mat4();//初始化未单位矩阵
```

```
Function render()
{    gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);
     theta[axis]+=2;
      modelviewMatrix=mult(modelviewMatrix ,rotate(theta[xAxis]));
      modelviewMatrix=mult(modelviewMatrix ,rotate(theta[yAxis]));
      modelviewMatrix=mult(modelviewMatrix ,rotate(theta[zAxis]));

    gl.uniformMatrix4fv(modeViewMatrixLoc,false,flatten(modelviewMatrix));

    gl.drawArrays(gl.TRIANGLES,0,numVertices);
     requesetAnimFrame(render);
}
```

注：flatten把JS中基于行主顺的mat矩阵转换为列主序的Float32Array类型数据发送给vertex shader

方法2：着色器中计算顶点

```
attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;
uniform mat4 modelViewMatrix;

Void main()
{
        gl_Position = modelViewMatrix * vPosition
}
```

方法3：JS中计算连续旋转角度并发送GPU

- 在JS中计算三个方向的旋转角累积量数组(3个数值)，并发送给shader。

```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT |gl.DEPTH_BUFFER_BIT);

    //当前旋转轴的旋转角度增加2，其它轴的角度不变
    theta[axis] += 2.0;
    gl.uniform3fv(thetaLoc, theta);//发送角度向量


    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
```

see cube.js

方法3 Shader中计算旋转矩阵

- **顶点着色器中根据输入的旋转角度计算旋转变换矩阵**

```
attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;
uniform vec3 theta;
void main() {
 // Compute the sines and cosines of theta for each of
 //  the three axes in one computation.
        vec3 angles = radians( theta );     //转换为弧度
        vec3 c = cos( angles );
        vec3 s = sin( angles );
```

see cube.html

方法3 Shader中计算旋转矩阵(cont.)

```
    //Remeber: thse matrices are column-major
    mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                    0.0,  c.x,  s.x, 0.0,
                    0.0, -s.x,  c.x, 0.0,
                    0.0,  0.0,  0.0, 1.0 );

    mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                    0.0, 1.0,  0.0, 0.0,
                    s.y, 0.0,  c.y, 0.0,
                    0.0, 0.0,  0.0, 1.0 );


    mat4 rz = mat4( c.z, s.z, 0.0, 0.0,
                   -s.z, c.z, 0.0, 0.0,
                    0.0, 0.0, 1.0, 0.0,
                    0.0, 0.0, 0.0, 1.0 );

    gl_Position = rz * ry * rx * vPosition;

    fColor = vColor;
}
```

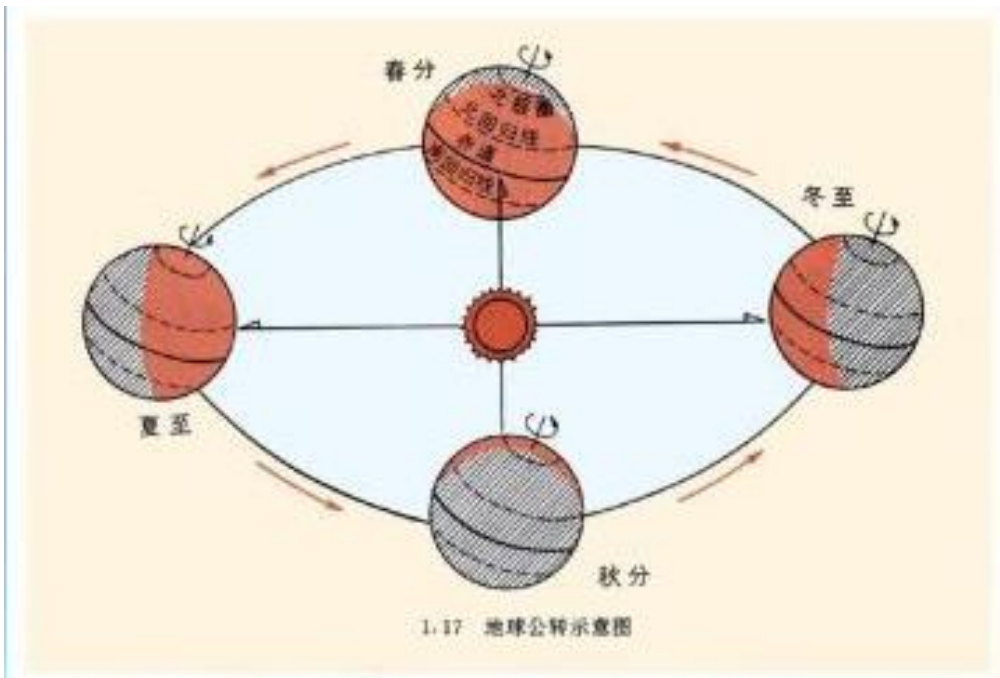
注意矩阵表示

列优先！

see cube.html

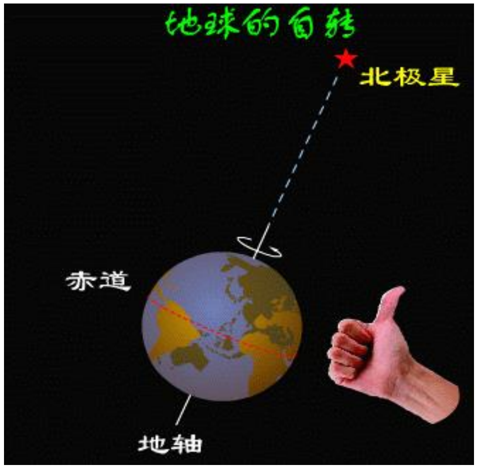
建模变换实例2： sunplanet

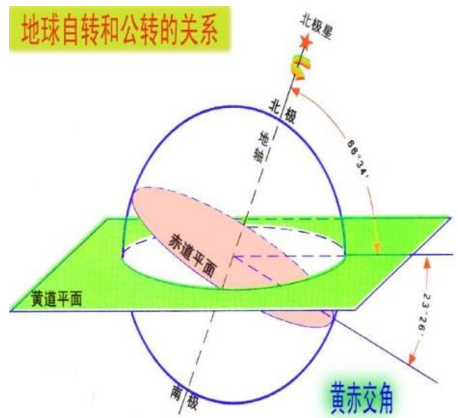
- 如模拟太阳地球，地球的模型变换

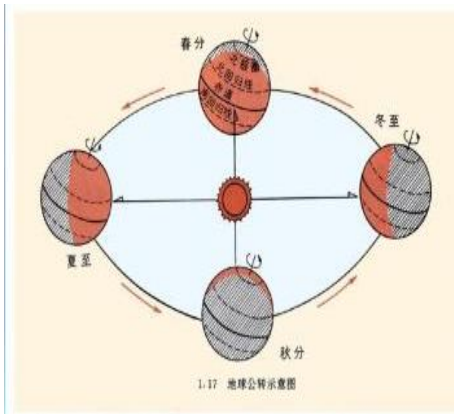


1.17 地球公转示意图

科普：地球绕着太阳旋转?
https://tv.sohu.com/v/dXMvMTIzNDgxNzM5LzUyMDEwMzMzLnNodG1s.html





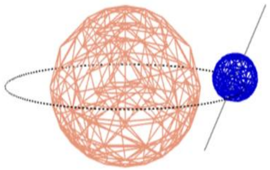


Earth'=T(公转) * Rz(-Phi) * Ry(自转) * S(缩放)* Earth

模型变换(复合变换)由以下4个标准变换得到：

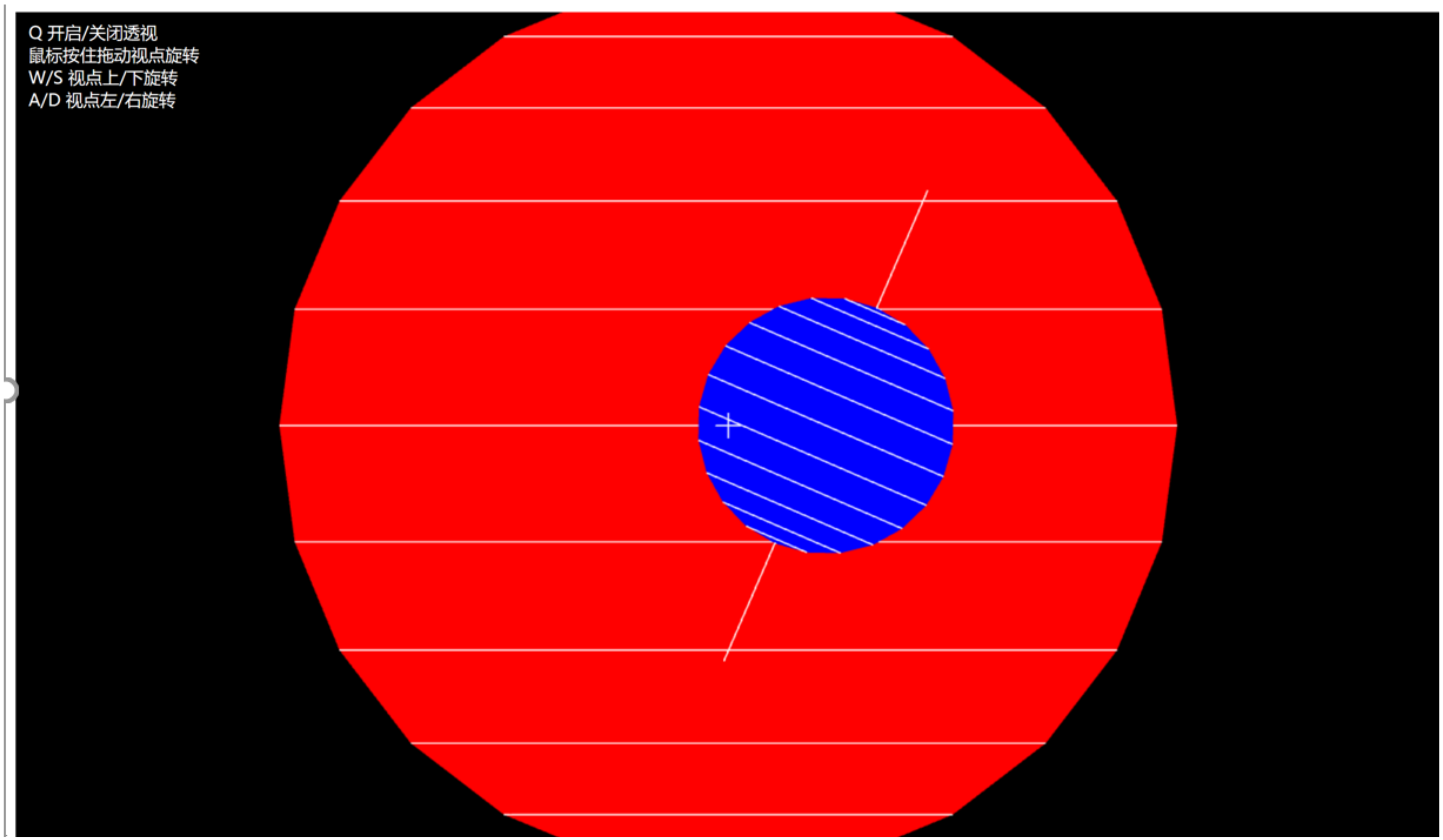
1. MC下球体进行缩放到需要的地球尺寸：S
2. MC下地球绕地轴（y轴）逆时针自转:Ry
3. MC下地球地轴偏移即绕z轴顺时针旋转R(-phi)   //(*Phi = 23.26*)
4. WC下地球平移到相应位置，实现绕太阳的公转T(r*cos(theta),rsin(theta))

参考WEBGL示例SunPlanet



```
//如果颜色是灰色olorList['grey'] = vec4(0.5, 0.5, 0.5, 1.0);绘地心轴先需要偏转再平移到公转轨道上
//如果颜色是兰色colorList['blue'] = vec4(0.0, 0.0, 0.8, 1.0);绘制地球
//其它就是红太阳和黑色轨道,没有模型变换。
if (color[0] == 0.5) //绘制地心轴
    gl_Position = projectionMatrix * ViewMatrix * rotateRevo * slideAxis_to * vPosition;

else if(color[2]== 0.8) //绘制地球
    gl_Position = projectionMatrix * ViewMatrix * rotateRevo * slideAxis_to * rotateAuto  * vPosition;

else //绘制太阳和轨道
    gl_Position = projectionMatrix * ViewMatrix* vPosition;
```

视点变换Webgl实例



MV.Js里提供了常用变换函数

- 这些函数API可以帮助简化编程，也可自己实现

```
//----------------------------------------------------------------------------
//  Basic Transformation Matrix Generators 生成变换矩阵
//  translate( x, y, z ),scalem( x, y, z )
//  rotate( angle, axis ),rotateX(theta),rotateY(theta) ,rotateZ(theta),
//  transpose( m )转置
//
/*生成齐次坐标表示,平移矩阵*/
function translate( x, y, z )
{


/**生成绕任意轴旋转的变换矩阵*/
function rotate( angle, axis )
{


/*生成绕X轴, Y轴, Z轴旋转的的标准变化矩阵*/
/***********vc右手系!修改, 原是c,s;-s,c 不对*****************/
function rotateX(theta) {
/**********vc右手系!修改, 原是c,-s;s,c 不对****************/
function rotateY(theta) {
/***********vc右手系!修改, 原是c,s;-s,c不对*****************/
function rotateZ(theta) {

/*齐次缩放矩阵: 对角线三个分量进行缩放*/
function scalem( x, y, z )
{


/*判定是矩阵后, 求它的转置, ? result[i].push( m[j][i] );*/
function transpose( m )
{
```

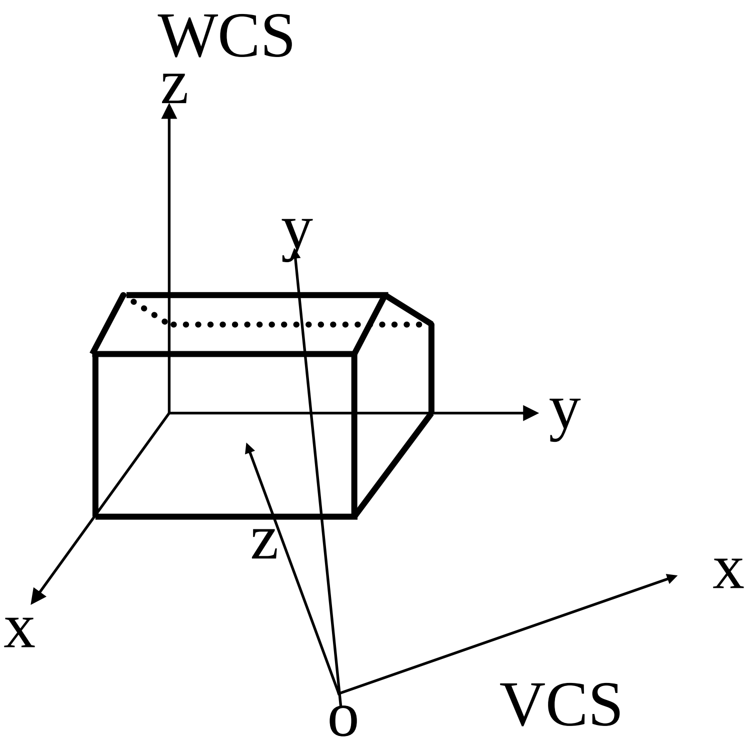
Outlines

- Viewing Pipeline

- Composite Transformations in Viewing Pipeline
  - ModelView Transformation
    - Model Transformation建模变换
    - **View Transformation视点/相机/眼/观察变换**
  - Projection Transformation
  - Viewport Transformation

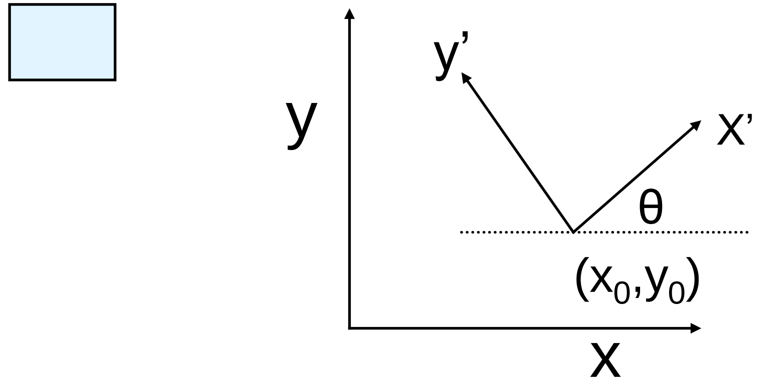
Viewpoint Transformation

- 视点变换:(WC→VC)



2D Viewpoint Trans.

- How to transform the representations of objects's locations from XYZ to X'Y'Z'



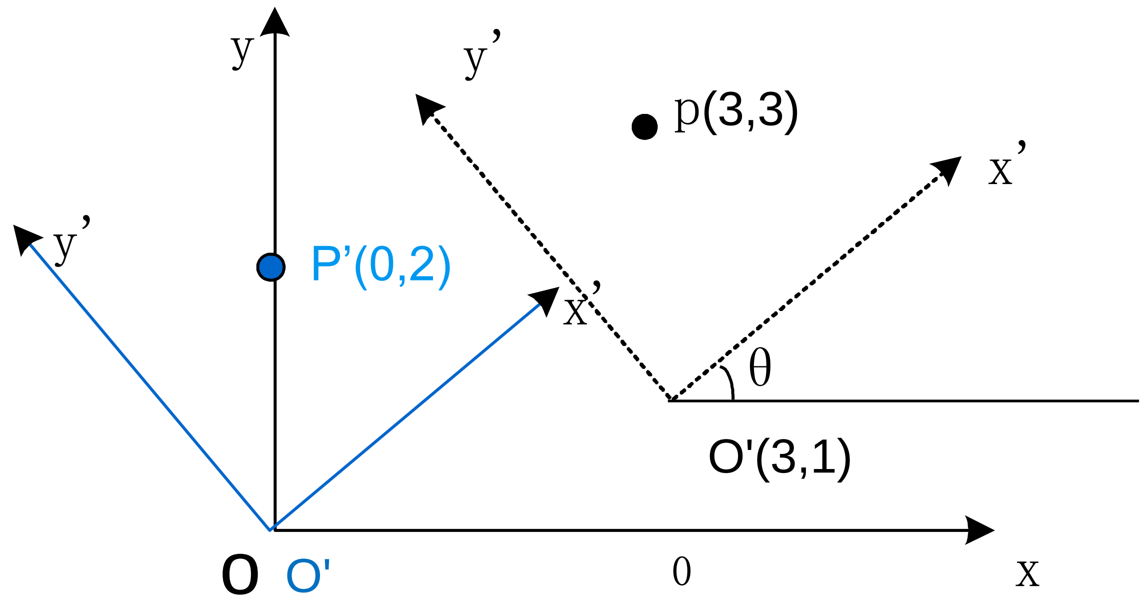
✓思路：将 X'O'Y' 变换到与XOY重合

1.平移 $(-x_0,-y_0)$到(0,0)：$T(-x_0,-y_0)$

2.旋转x'y'到xy：$R(-\theta)$

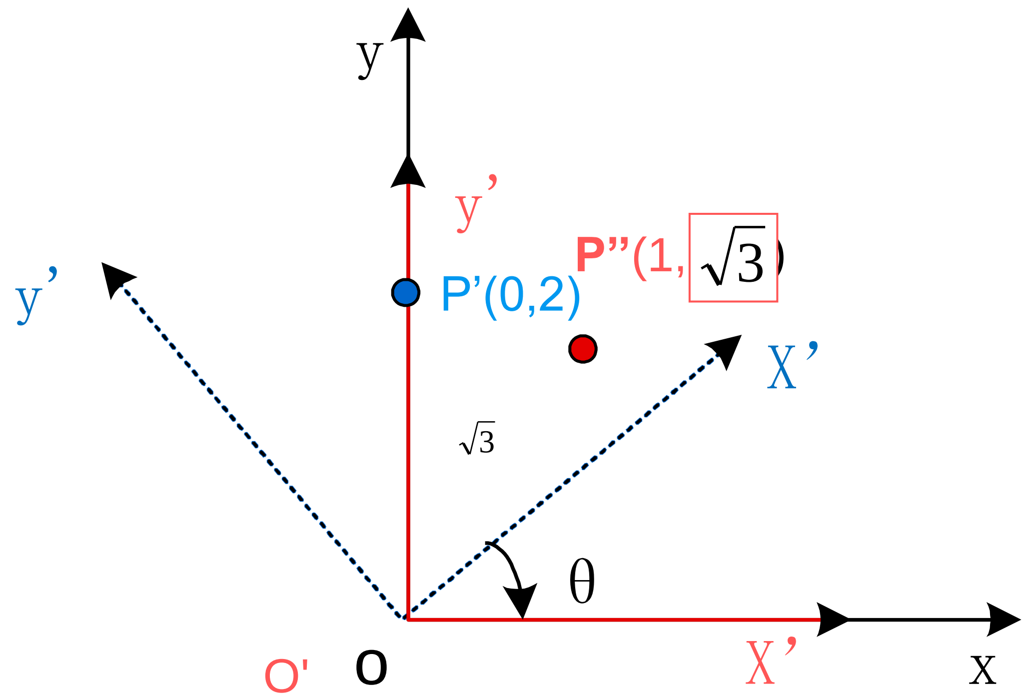
$$M=R(-\theta)T(-x_0,-y_0)$$

2D viewpoint Trans. Example1:



第1步：将x'y'坐标系的原点平移到xoy坐标系的原点

平移变换为T(Tx,Ty) 该例中Tx=-3, Ty=-1



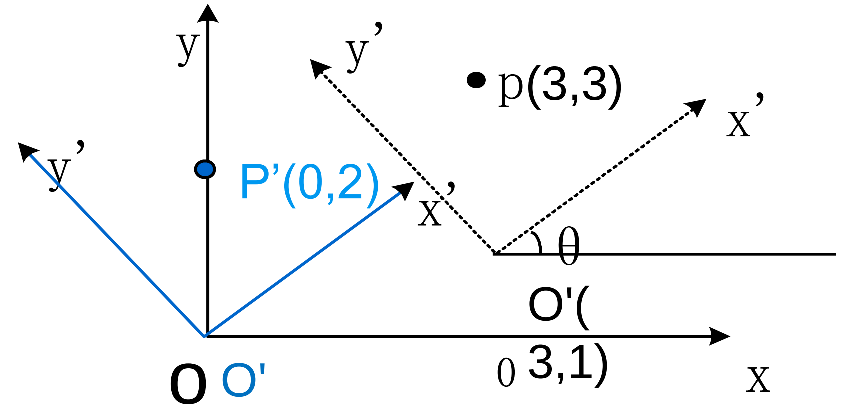
第二步： **将x‘轴旋转到x轴上，旋转变换矩阵为R(−θ)**

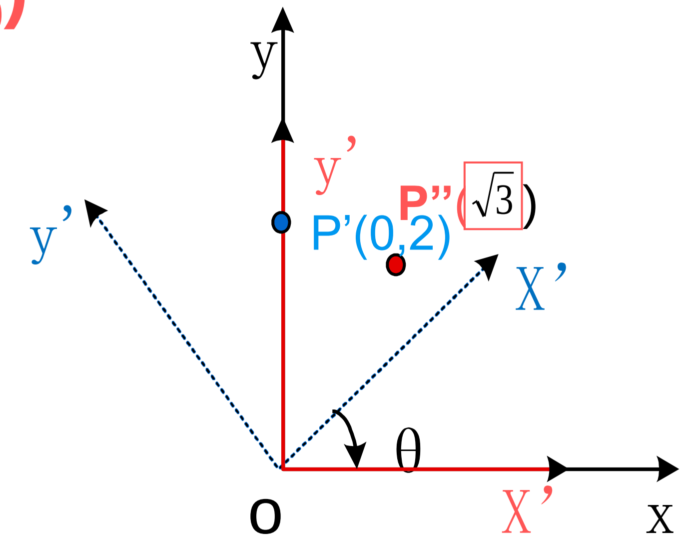
若角度＝３０度，则计算出来的坐标为（１，$\boxed{\sqrt{3}}$）

1.平移 $(-x_0,-y_0)$到(0,0)：$T(-x_0,-y_0)$

2.旋转x’y’到xy：$R(-\theta)$

求得： **$M=R(-\theta)T(-x_0,-y_0)$**

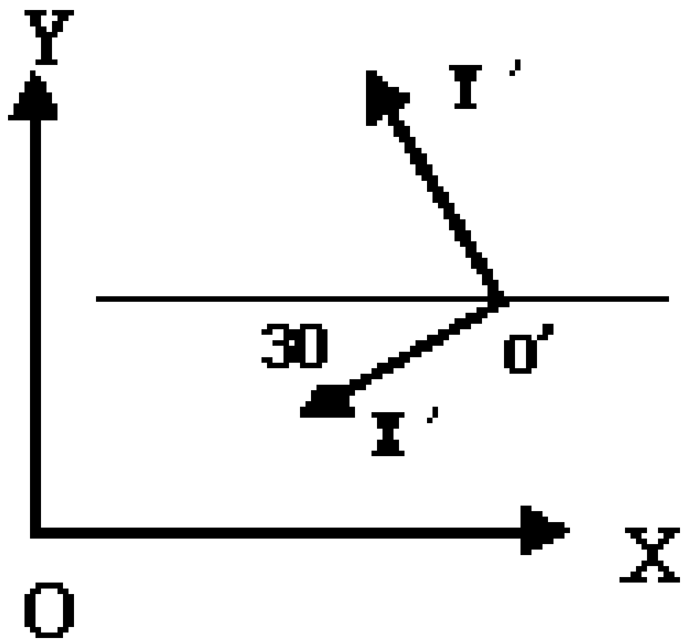




课堂练习 1

在**XOY**平面坐标系中，过**O’**点确定的新坐标系**X’O’Y’**如图所示，**O’**的齐次坐标是**(8，4，1)**。

请写出列向量表示时的坐标变换矩阵？



Another Method

前面方法求旋转矩阵必须知道旋转角，不便。

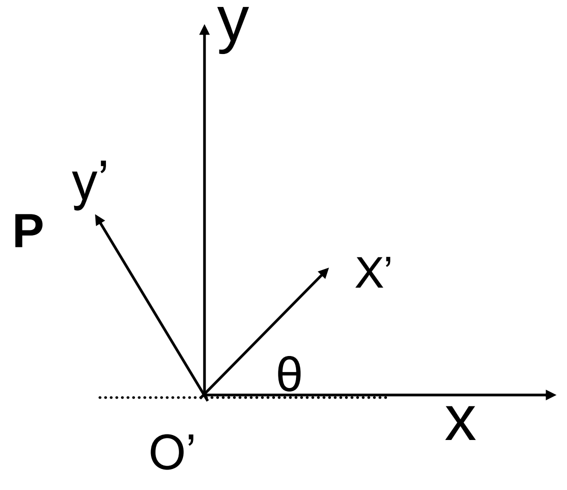
思考： 利用旋转矩阵的正交特性来构造旋转矩阵。

$$R=\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix} \Rightarrow R=\begin{pmatrix} u_x, u_y \\ v_x, v_y \end{pmatrix}$$

- **R中的每一行作为一个向量，则R的两行正好是单位正交向量组。u·v=0，|v|=1，|u|=1；$R^{-1}=R^{T}$**
- *构造旋转变换矩阵关键：求得正交基方向上单位向量！！*

2D Viewpoint Trans. Example2

- 已知 O'点（与O重合），O'Y'上P点，
- 求从XYZ到X'Y'Z'的坐标变换矩阵R



$$R=\begin{bmatrix} u_x, u_y \\ v_x, v_y \end{bmatrix}$$

1）计算O'Y'单位向量v

$v=(v_x,v_y)=((Xp-Xo)/r,\ (Yp-Yo)/r)$ 注：$r^2=(Xp-Xo)^2+(Yp-Yo)^2$

2）计算O'X'上的单位向量u

v绕原点顺时针转90度得到单位向量u， $u=(u_x,u_y)=(v_y,-v_x)$

3）将u，v的分量带入R矩阵

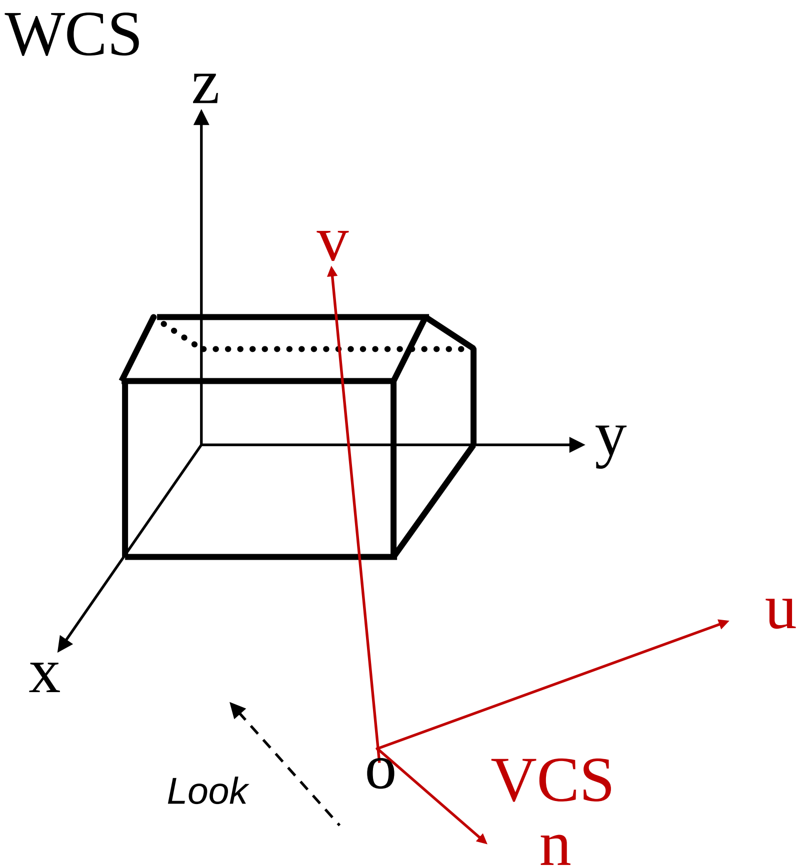
课堂练习2

- 新旧坐标都是右手系坐标系统，新坐标系统是**x’o’y’**，**O’**在**XOY**坐标系中坐标为（**0,0**），**V**是**O’ Y’**轴上的一向量，**V=(3,4)**。

请根据单位正交矩阵构造旋转矩阵的方法，求图形从**xoy**变换到**x’o’y’**的变换矩阵。

$$R=\begin{pmatrix} 4/5, & -3/5, & 0 \\ 3/5, & 4/5, & 0 \\ 0\ , & 0\ , & 1 \end{pmatrix}$$

3D Viewpoint Trans.



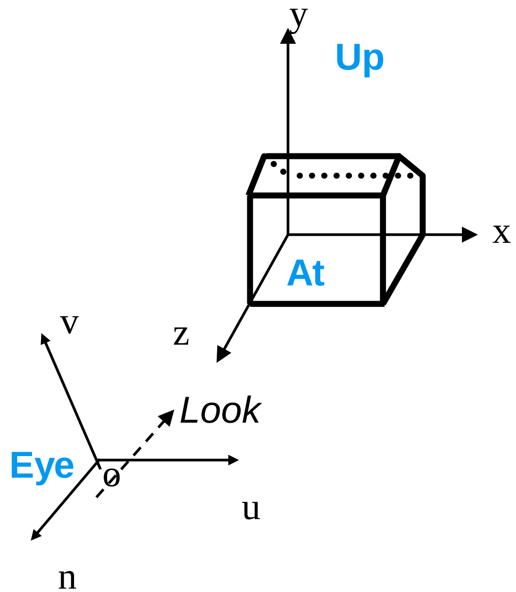
$$M_{wc->vc}=R\cdot T$$

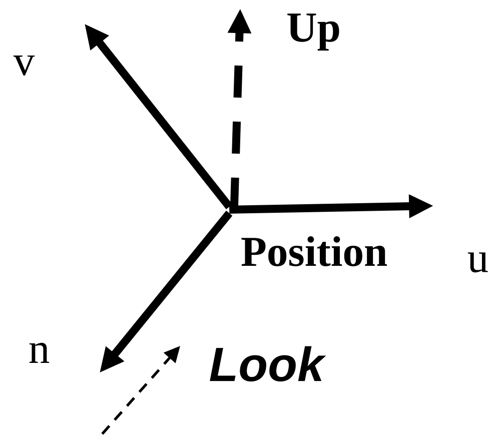
其中：R矩阵构造如下

$$\mathbf{R}=\begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

R:Finding u,v,n from *Position, Look and Up*





- N和观察方向Look向量反向（VC是右手坐标系）
  - ➢ $n=N/|N|=(n_1,n_2,n_3)$　　//N= - *Look* =At-Eye
- u必须垂直于n和Up向量所在的平面
  - ➢ $u=(V\times N)/|V\times N|=(u_1,u_2,u_3)$　　//V=Up
- v是n和u叉乘得到的单位向量
  - ➢ $v=n\times u=(v_1,v_2,v_3)$

$$n = \frac{-Look}{\|Look\|}$$

$$u = \frac{Up\times n}{\|Up\times n\|}$$

$$\mathrm{v} = n\times u$$

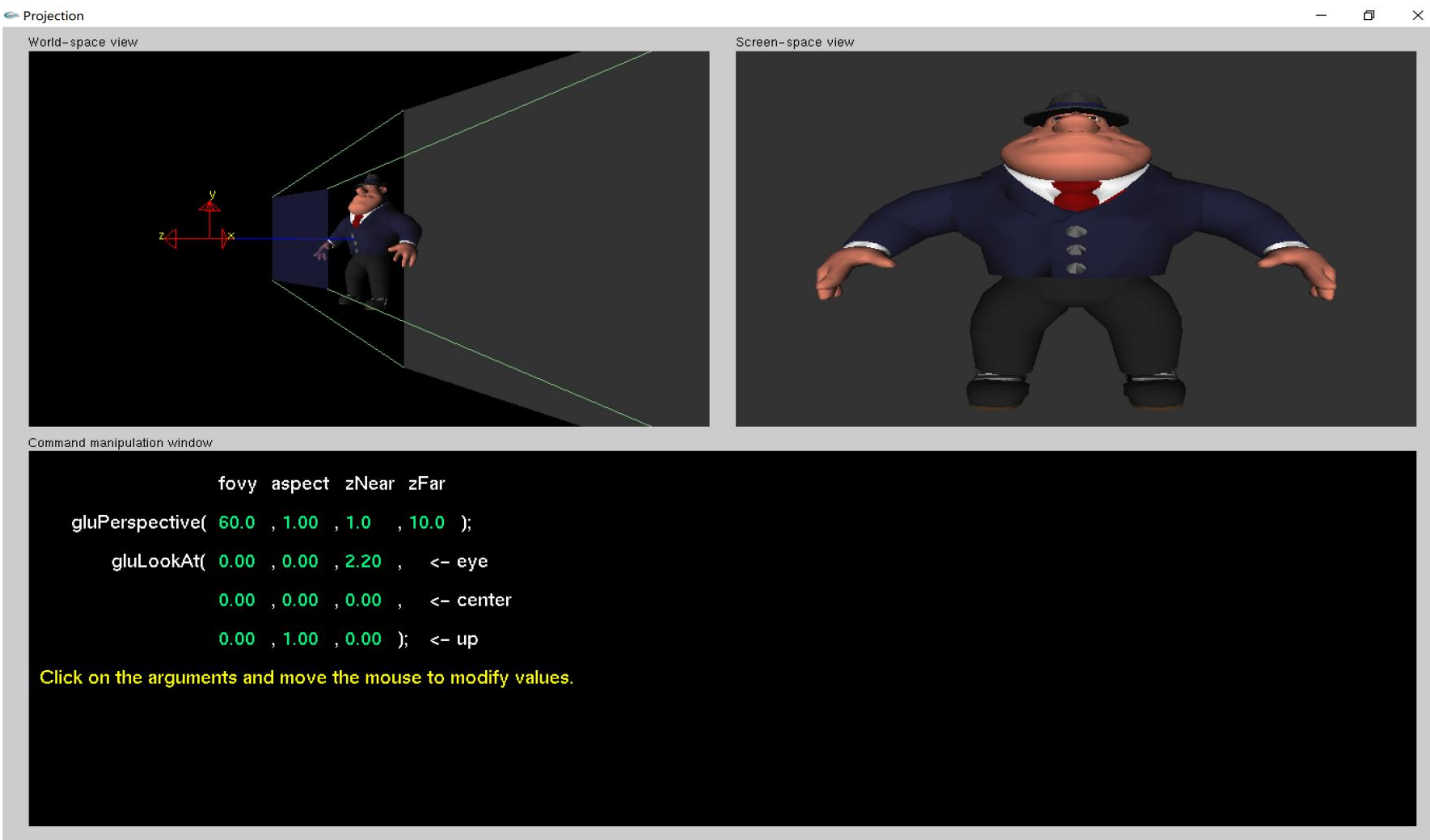
/common/MV.js实现了Lookat函数

- LookAt函数生成“观察变换矩阵”

```
function lookAt( eye, at, up )
{
    if ( !Array.isArray(eye) || eye.length != 3) {
        throw "lookAt(): first parameter [eye] must be an a vec3";
    }

    if ( !Array.isArray(at) || at.length != 3) {
        throw "lookAt(): first parameter [at] must be an a vec3";
    }

    if ( !Array.isArray(up) || up.length != 3) {
        throw "lookAt(): first parameter [up] must be an a vec3";
    }

    if ( equal(eye, at) ) {
        return mat4();
    }
    //下面的单位向量名字调整为(u,v,n), 原来ANGEL是(n,u,v),并且进行了调整
    var n = negate(normalize( subtract(at, eye) )); // view direction vector
    var u = normalize( cross(up, n) );        // perpendicular vector
    var v = normalize( cross(n, u) );         // "new" up vector

    //result=R(u,v,n)*T(-eye)
    var result = mat4(
        vec4( u, -dot(u, eye) ),
        vec4( v, -dot(v, eye) ),
        vec4( n, -dot(n, eye) ),
        vec4()
    );
    return result;
}
```

演示-观察变换

- 常用三个参数（Eye,Center,Up)或(eye,at,up)



Ref2: Chap5 实例1改变视点位置

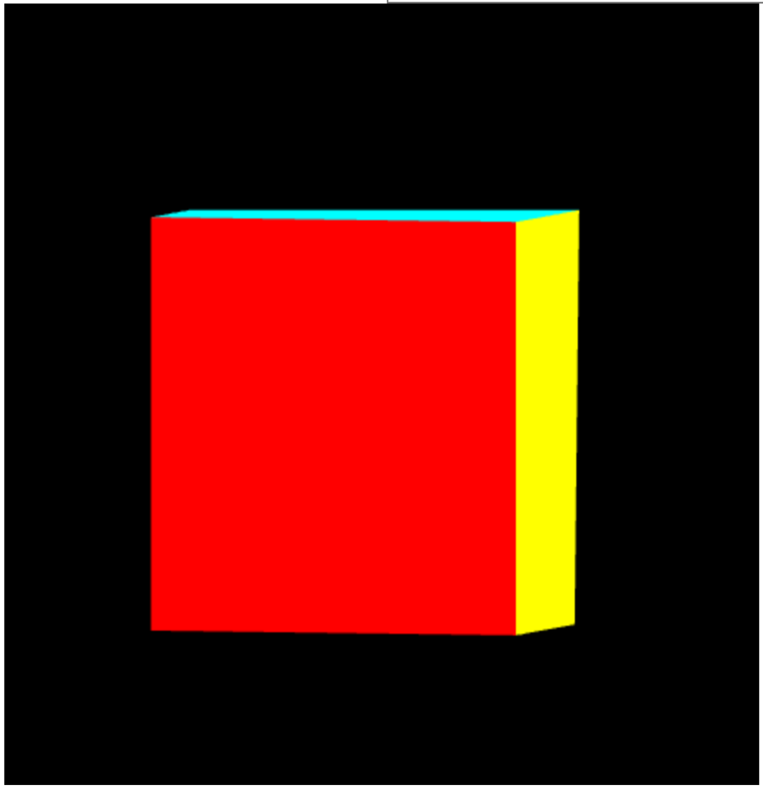
平行投影：视点参数变化

radius 0.05 2 theta -180 180 phi -90 90

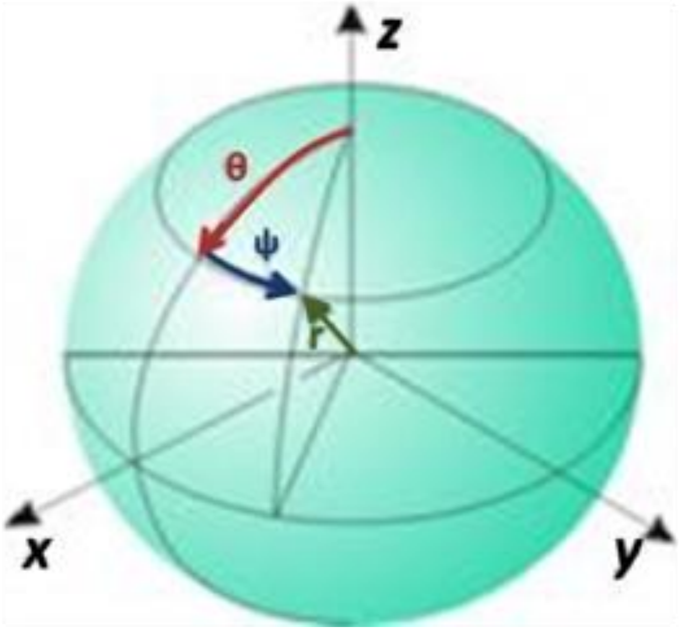
eye(radius,theta,phi): 1, 0.2617993877991494, 0.17453292519943295 eye(X,Y,Z): 0.25488700224417876, 0.044943455527547776, 0.

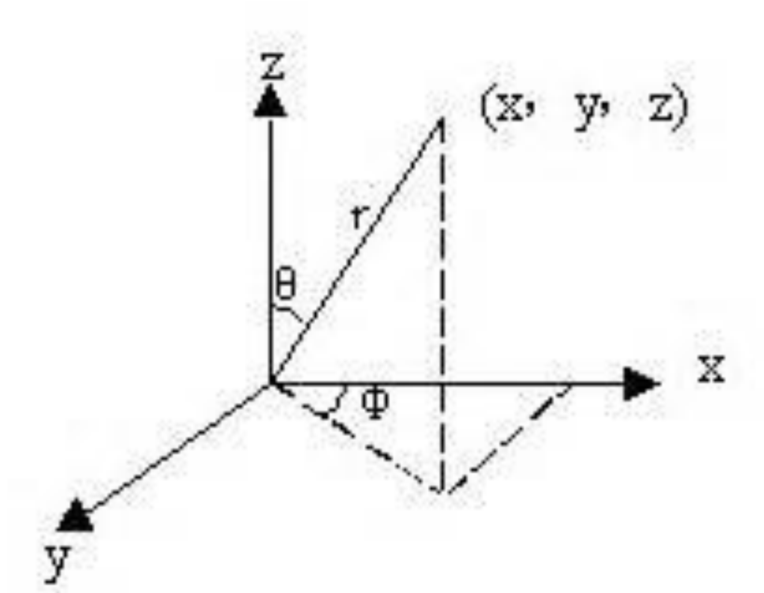
width 0.01 10 height 0.01 10 depth 0.01 2

left, right, bottom, ytop, near, far: -1, 1, -1, 1, 0, 1

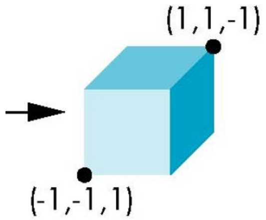


相机位置eye(x,y,z）的定位





X=r*sin(theta)cos(phi)
Y=r *sin(theta)sin(phi)
Z=r *cos(theta)



Webgl viewing functions

```
window.onload = function init() {
    ……
     gl.viewport( 0, 0, canvas.width, canvas.height );
   ……}
```

```
var render = function() {
     modelViewMatrix = lookAt(eye, at , up);
     projectionMatrix = ortho(left, right, bottom, ytop, near, far);
     // projectionMatrix = perspective(fovy, aspect, near, far);
    ….}
```

```
//vertex  shader
void main() {
      //输出的顶点坐标gl_Position，应该是裁剪坐标系下的坐标！
     gl_Position =projectionMatrix*modelViewMatrix*vPosition;
     fColor = vColor;
}
```

总结：模视变换
ModelView Transformation

- 图形包常把模型变换和观察变换合称为“模视变换”
  - 模型变换Model Trans.：
    - 摆放物体
  - 视点变换View Trans.：
    - 摆放相机
- 几何变换复合矩阵记作MVP.
- 顺序：一般先作模型变换M，后作视点变换V，最后作投影变换P：
- **Position’= P*V*M * Position**

ModelView演示

- 编程实现时，通常先作模型变换M，再作视点变换V