



计算机图形学

COMPUTER GRAPHICS

VISUAL COMPUTING RESEARCH CENTER

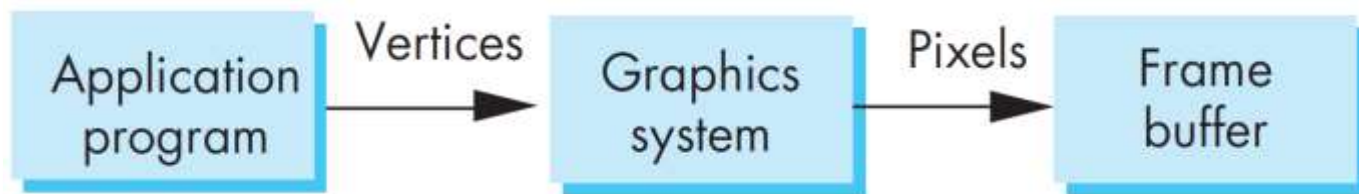


1-8 Chapter

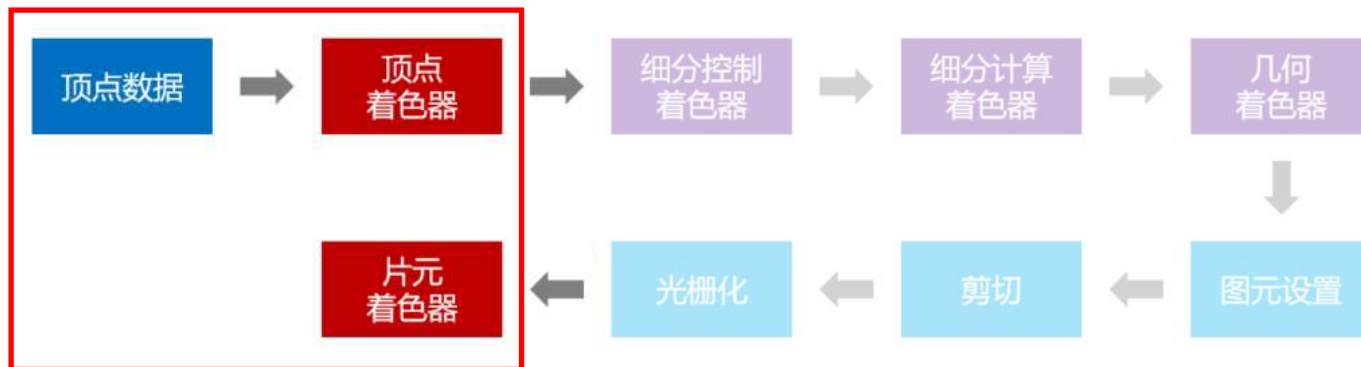
重要知识点回顾和巩固

◆ 计算机图形绘制的高层观点：

- 以应用程序为起点，而以图像的生成作为终点
- 输入：顶点和状态变量，即几何对象、属性和相机参数等
- 输出：位于帧缓存中的彩色像素阵列



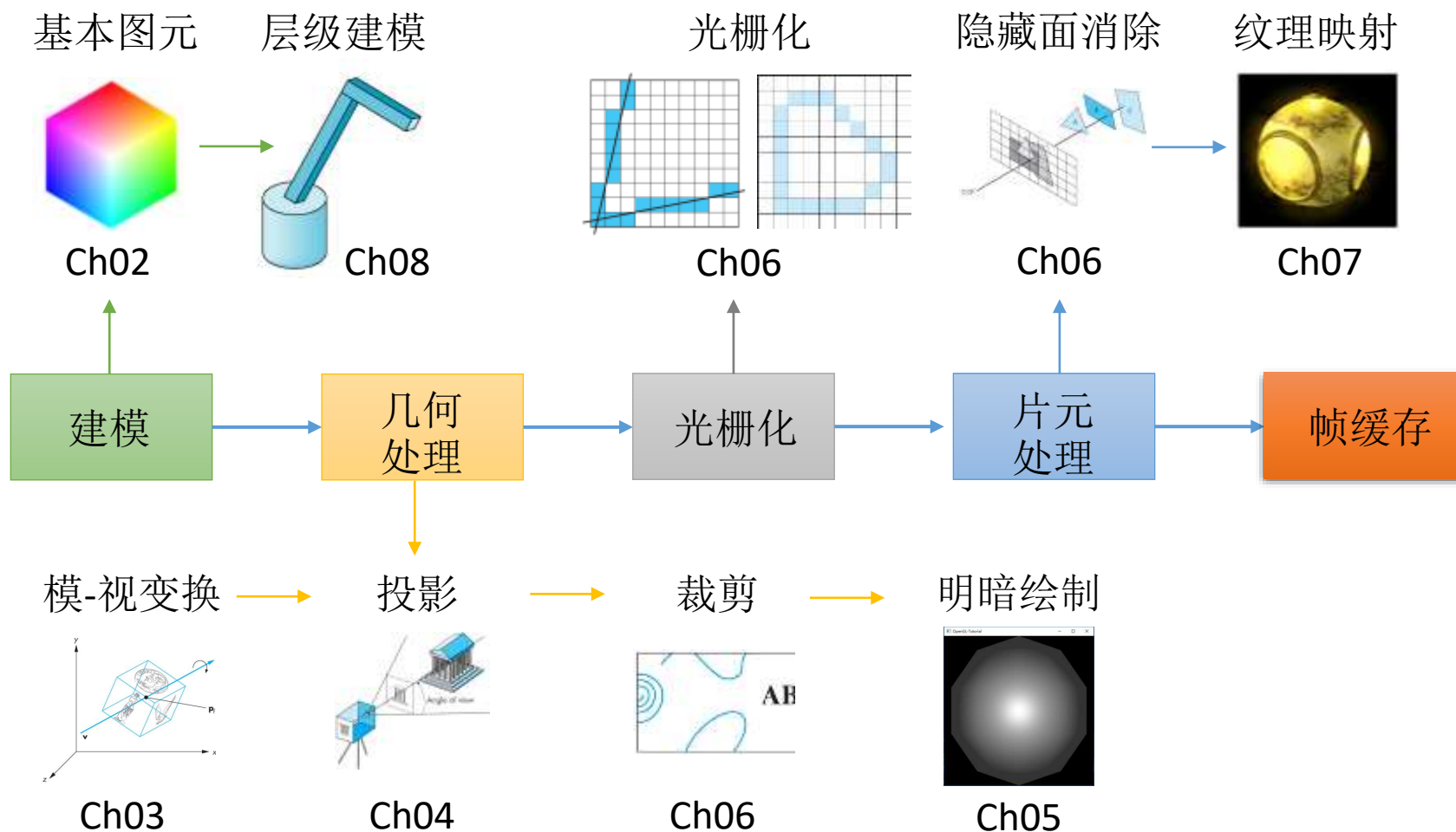
◆ OpenGL渲染管线：



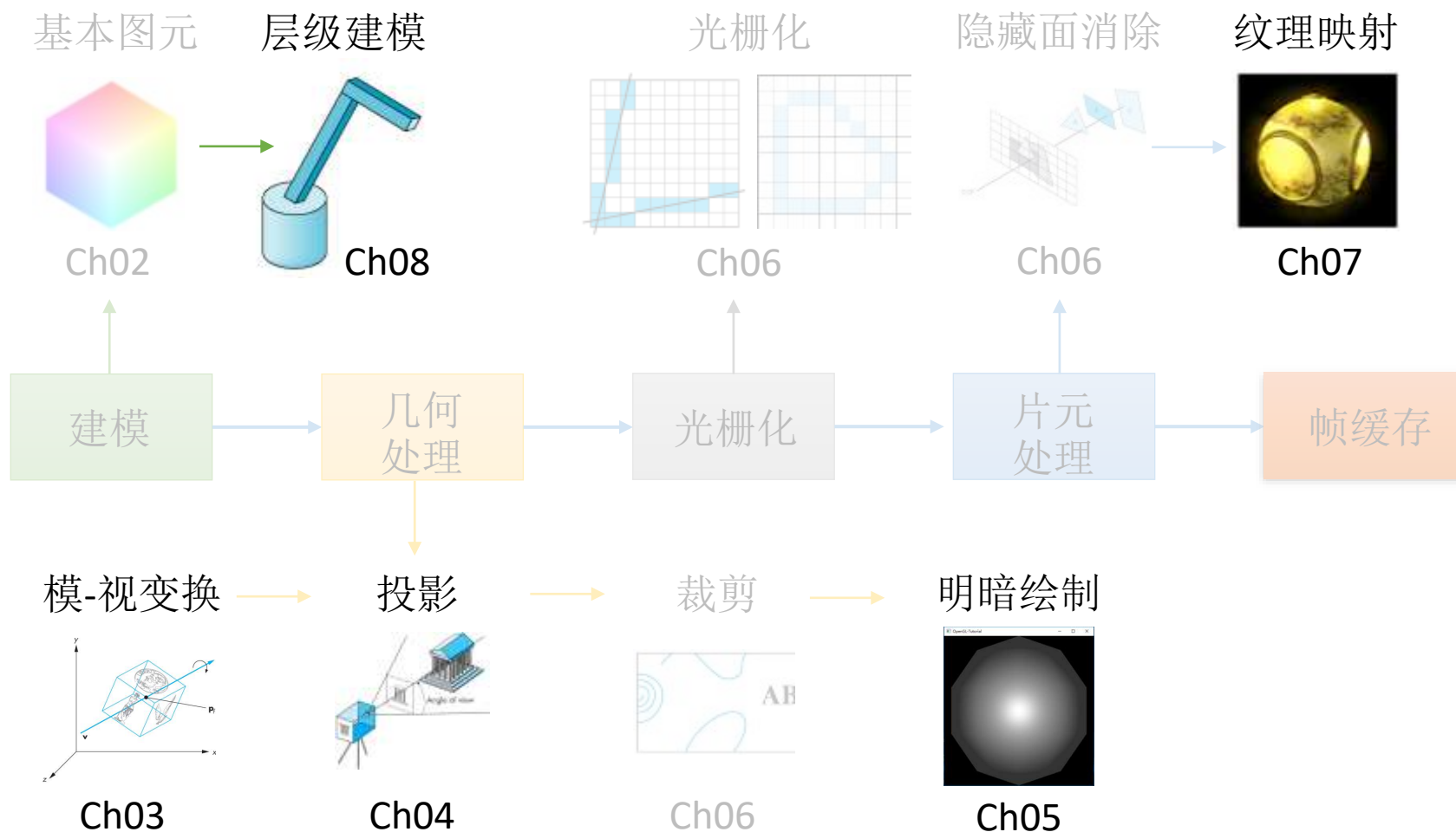
编程处理

使用默认设置

◆ 图形绘制系统的4个主要任务：



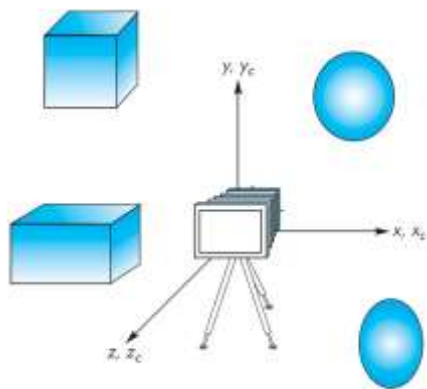
◆ 图形绘制系统的4个主要任务：



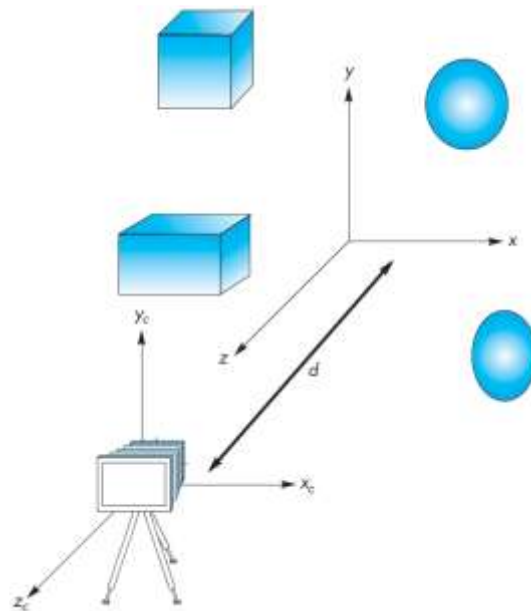
Section **1**

模-视变换

- ◆ 目标:
 - 确定几何对象和相机的相对位置关系
- ◆ 方式:
 - 通过标架之间的变换实现



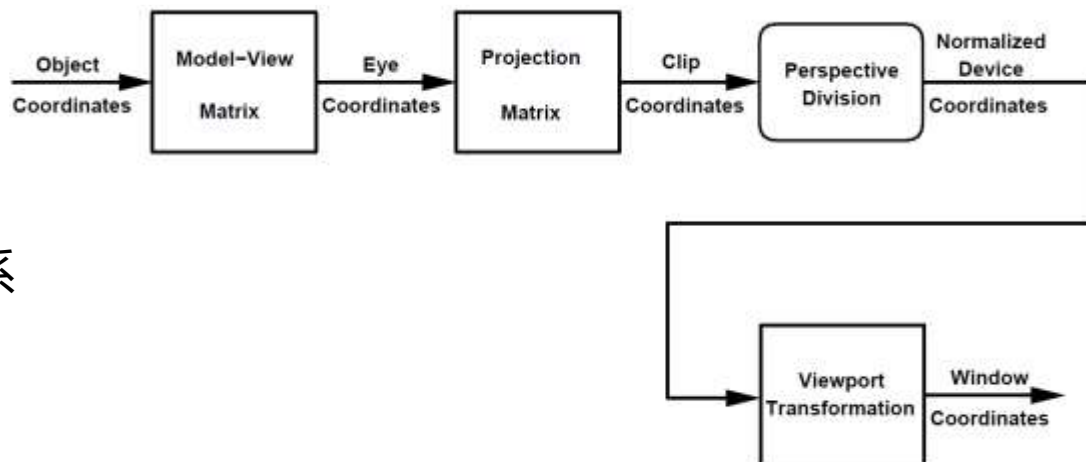
初始标架



变换后标架

◆ OpenGL固定管线中的6个标架:

- 模型坐标系
- 世界坐标系
- 相机坐标系
- 裁剪坐标系
- 规范化的设备坐标系
- 屏幕坐标系



- ◆ 标架中的坐标采用齐次坐标表示
- ◆ 标架之间的变换用仿射变换矩阵实现

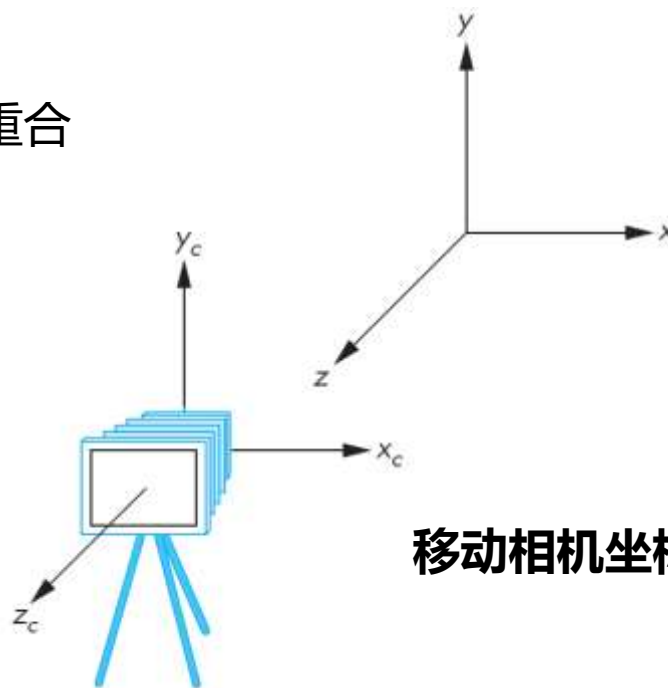
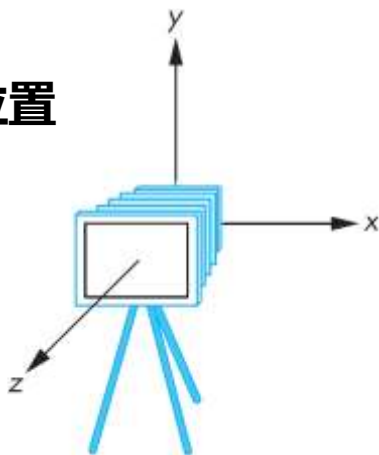
◆ 在OpenGL图形管线中顶点通常在三个坐标系间转换：

- 模型坐标系：以物体为中心的局部坐标系
- 世界坐标系：在该坐标系中构建多个物体之间的三维空间关系
- 相机坐标系：相机成像坐标系

◆ OpenGL中的默认相机：

- 朝向-z方向
- 默认世界坐标系与相机坐标系重合

初始相机位置



移动相机坐标系

◆ LookAt函数:

```
mat4 LookAt(point4 eye, point4 at, vec4 vup)
```

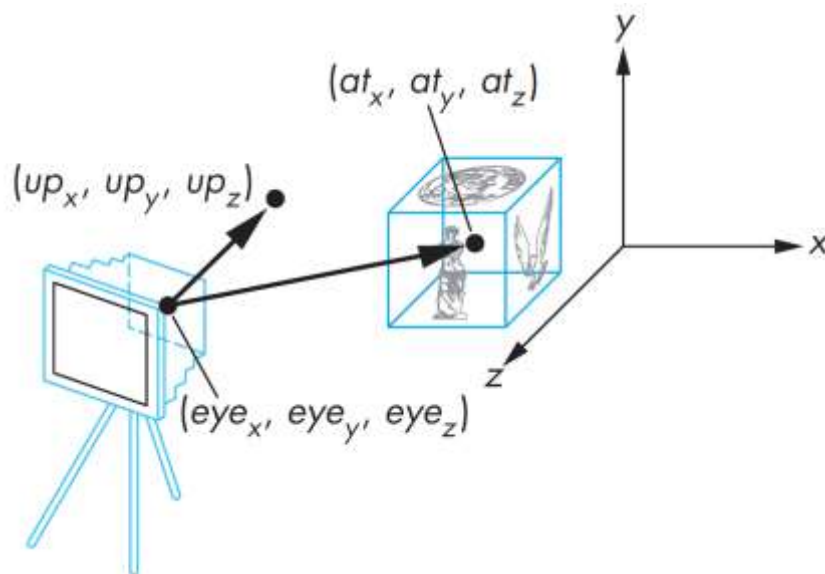
```
mat4 LookAt(GLfloat eyex, GLfloat eyey, GLfloat eyez,  
            GLfloat atx, GLfloat aty, GLfloat atz,  
            GLfloat upx, GLfloat upy, GLfloat upz)
```

◆ 函数解读:

- **eye** point: 视点
- **at** point: 参考点
- **vpn = eye - at**

$$\mathbf{n} = \frac{\mathbf{vpn}}{|\mathbf{vpn}|} \quad \mathbf{u} = \frac{\mathbf{v}_{up} \times \mathbf{n}}{|\mathbf{v}_{up} \times \mathbf{n}|}$$

$$\mathbf{v} = \frac{\mathbf{n} \times \mathbf{u}}{|\mathbf{n} \times \mathbf{u}|}$$



- ◆ 设VRP点 $\mathbf{p} = [x, y, z, 1]^T$, 视平面法向 $\mathbf{n} = [n_x, n_y, n_z, 0]^T$, 上方向量为 $\mathbf{v}_{up} = [v_{upx}, v_{upy}, v_{upz}, 0]^T$
- ◆ $\mathbf{v} = \mathbf{v}_{up} - (\mathbf{v}_{up} \cdot \mathbf{n})/(\mathbf{n} \cdot \mathbf{n})\mathbf{n}$, 把 \mathbf{v}, \mathbf{n} 单位化
- ◆ $\mathbf{u} = \mathbf{v} \times \mathbf{n}$
- ◆ 模型视图矩阵为:

$$\begin{bmatrix} u_x & u_y & u_z & -xu_x - yu_y - zu_z \\ v_x & v_y & v_z & -xv_x - yv_y - zv_z \\ n_x & n_y & n_z & -xn_x - yn_y - zn_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\blacklozenge M^T = [u, v, n, p]$$

$$\blacklozenge a = M^T b, b = (M^T)^{-1} a$$

$$\blacklozenge a = M^T b = [u, v, n, p] b = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} b$$

$$\blacklozenge b = \begin{bmatrix} R^T & -R^T * t \\ 0 & 1 \end{bmatrix} a$$

注意 R 必须为正交矩阵

$$(M^T)^{-1} = [u, v, n, p]^{-1} = \begin{bmatrix} u_x & u_y & u_z & -xu_x - yu_y - zu_z \\ v_x & v_y & v_z & -xv_x - yv_y - zv_z \\ n_x & n_y & n_z & -xn_x - yn_y - zn_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

◆ 线性变换

- 线性变换: $f(\alpha p + \beta q) = \alpha f(p) + \beta f(q)$
- 可理解为对物体的变换或者对空间的变换
- 可表示为: $v = Cu$, 其中 C 是4x4的矩阵

◆ 仿射变换

- 定义为只有12个自由度的线性变换:

$$C = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 仿射变换的直观特性: 保持线性, 将直线映射为直线

- ◆ 考虑同一个向量相对于两组不同基的表示
- ◆ 假设表示分别是

$$\begin{aligned} \mathbf{a} &= [a_1 \ a_2 \ a_3]^T & \mathbf{v} &= a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + a_3 \mathbf{v}_3 = [a_1 \ a_2 \ a_3] [\mathbf{v}_1 \ \mathbf{v}_2 \ \mathbf{v}_3]^T \\ \mathbf{b} &= [b_1 \ b_2 \ b_3]^T & &= b_1 \mathbf{u}_1 + b_2 \mathbf{u}_2 + b_3 \mathbf{u}_3 = [b_1 \ b_2 \ b_3] [\mathbf{u}_1 \ \mathbf{u}_2 \ \mathbf{u}_3]^T \end{aligned}$$



$$\mathbf{a} = M^T \mathbf{b}$$


$$\begin{aligned} u_1 &= \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3 \\ u_2 &= \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3 \\ u_3 &= \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3 \end{aligned}$$

with $M = ?$

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

- ◆ 考虑同一个向量相对于两组不同基的表示
- ◆ 假设表示分别是

$$\begin{aligned} \mathbf{a} &= [a_1 \ a_2 \ a_3]^T & \mathbf{v} &= a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + a_3 \mathbf{v}_3 = [a_1 \ a_2 \ a_3] [\mathbf{v}_1 \ \mathbf{v}_2 \ \mathbf{v}_3]^T \\ \mathbf{b} &= [b_1 \ b_2 \ b_3]^T & &= b_1 \mathbf{u}_1 + b_2 \mathbf{u}_2 + b_3 \mathbf{u}_3 = [b_1 \ b_2 \ b_3] [\mathbf{u}_1 \ \mathbf{u}_2 \ \mathbf{u}_3]^T \end{aligned}$$


$$\mathbf{a} = M^T \mathbf{b}$$

$$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$$

$$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$$

$$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$$

$$Q_0 = \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + P_0$$

齐次坐标表示下的 M ?

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

- ◆ 可以用在齐次坐标中一个 4×4 的平移变换 T 表示平移：
 $p' = Tp$

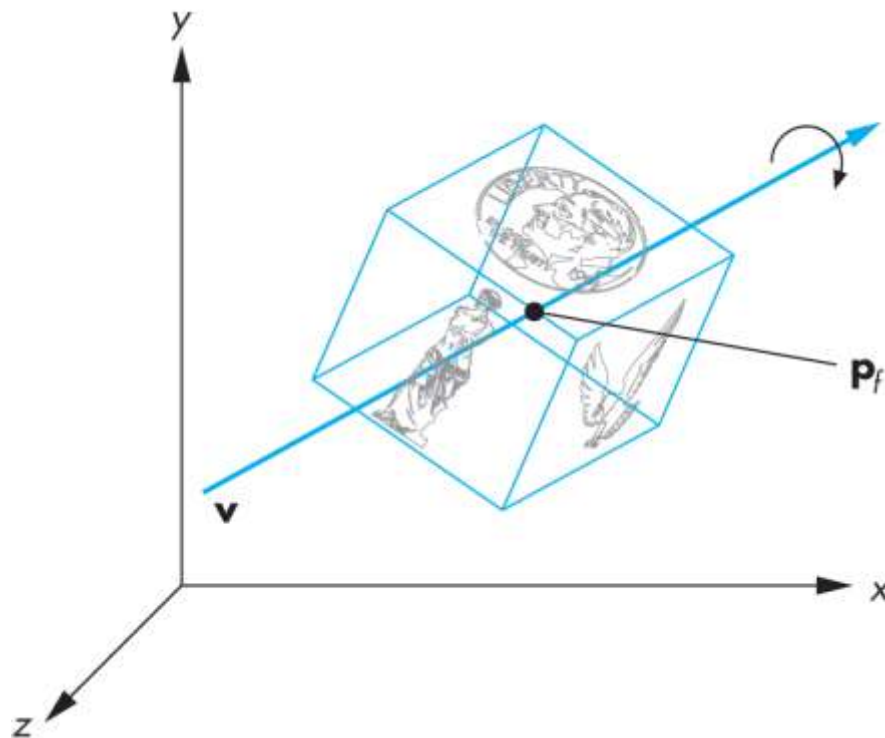
$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

◆ 三维对象的旋转

- 不动点
- 旋转轴
- 旋转角度

◆ 几种特殊情形

- 分别绕 x , y , z 轴的旋转
- 绕过原点的轴旋转
- 绕任一轴旋转



◆ 在三维空间中绕z轴旋转，点的z坐标不变

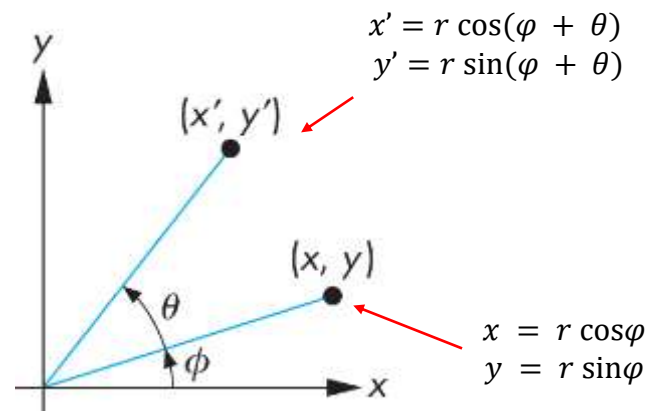
- 等价于在 $z=\text{常数}$ 的平面上进行二维旋转

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta$$

$$z' = z$$

- 其齐次坐标表示为 $p' = R_z(\theta)p$
- 其中旋转矩阵为



$$R = R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

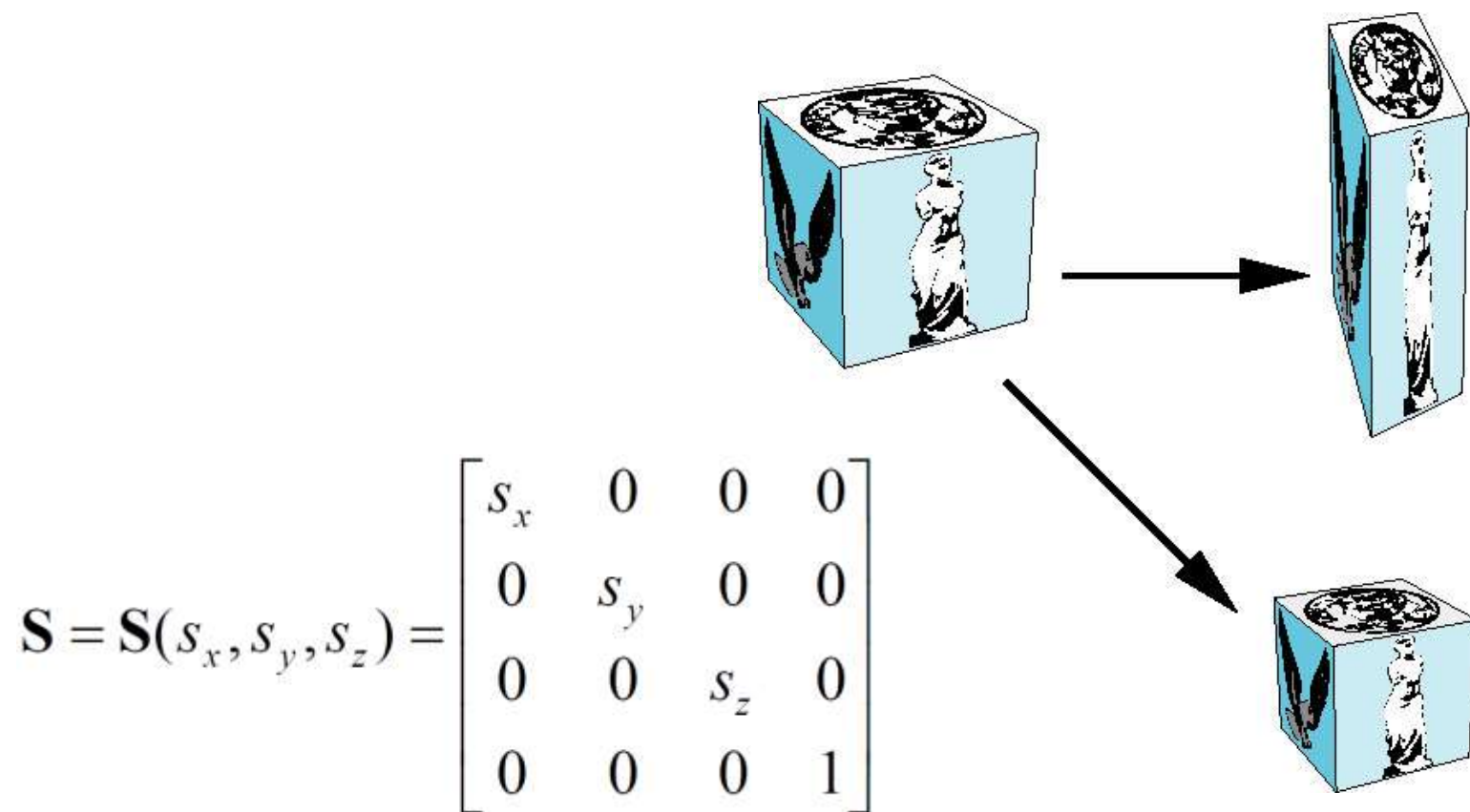
◆ 与绕z轴的旋转完全类似

- 对于绕x轴的旋转, x坐标不变
- 对于绕y轴的旋转, y坐标不变

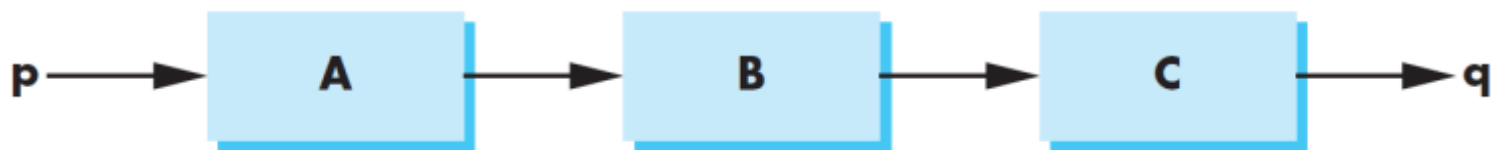
$$\mathbf{R} = \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{R}_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

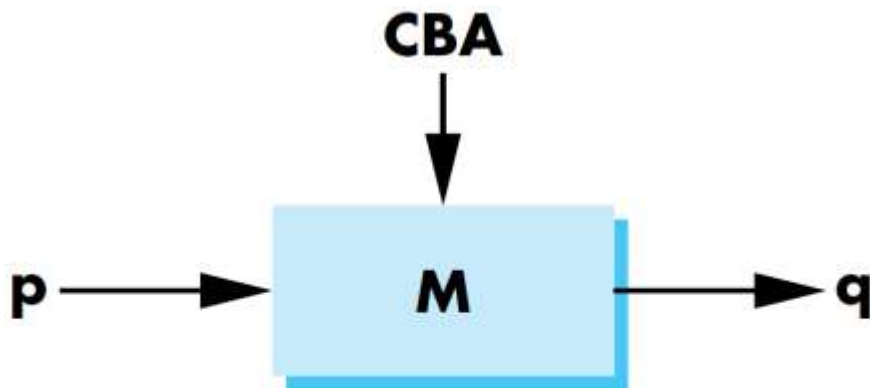
- ◆ 沿每个坐标轴伸展或收缩（**原点为不动点**）



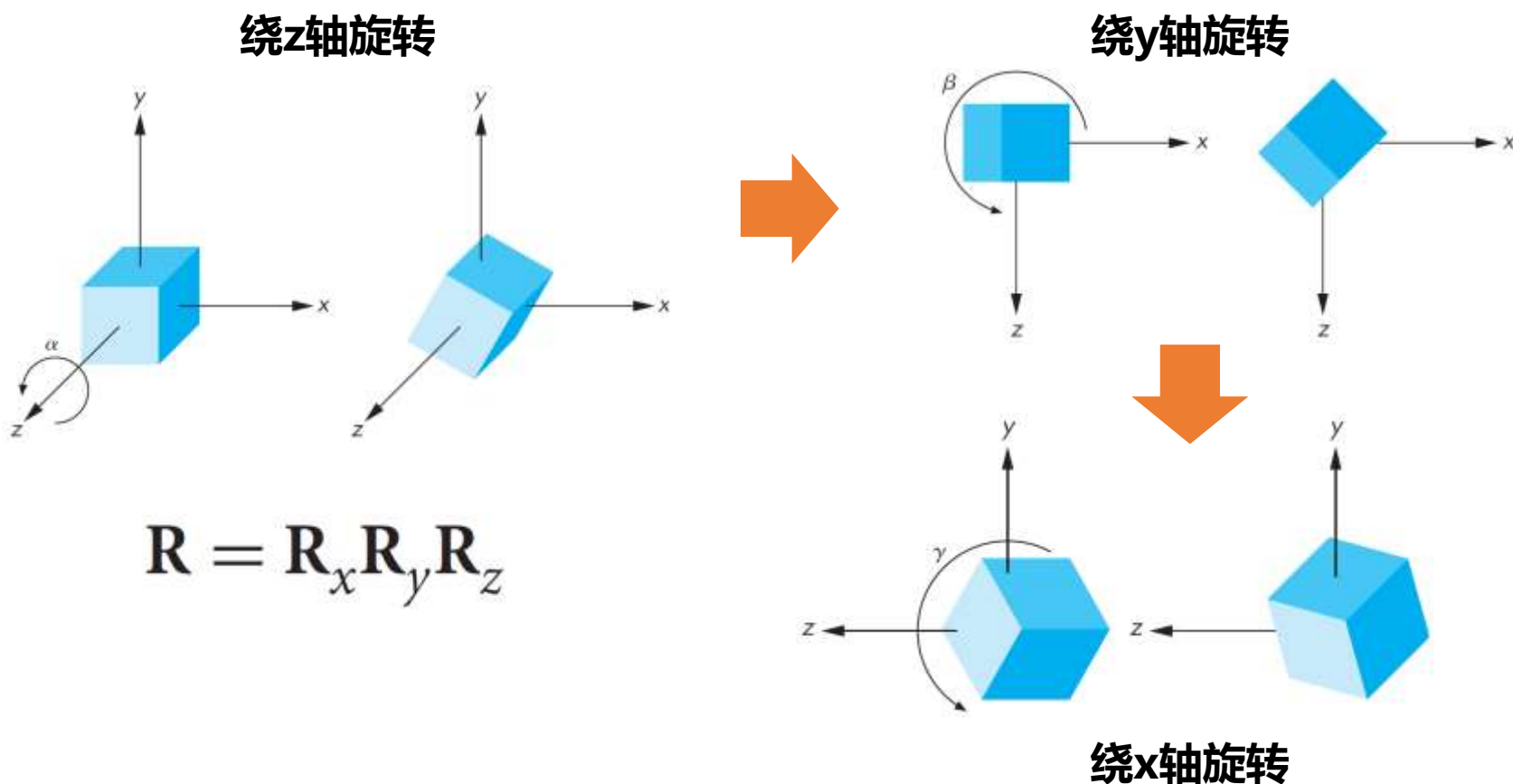
- ◆ 多个变换依次作用于点 p : $q = \mathbf{CBA}p$



- ◆ 变换的级联: $M = \mathbf{CBA}$, $q = \mathbf{M}p$

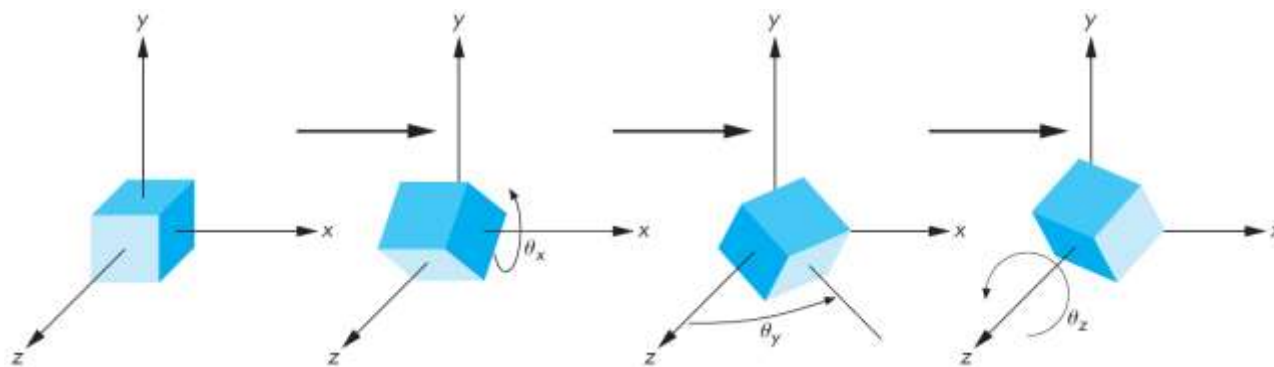


- 绕原点的任意旋转根据选择的顺序不同可**不唯一**地分解为绕x轴、y轴、z轴的三个旋转的级联

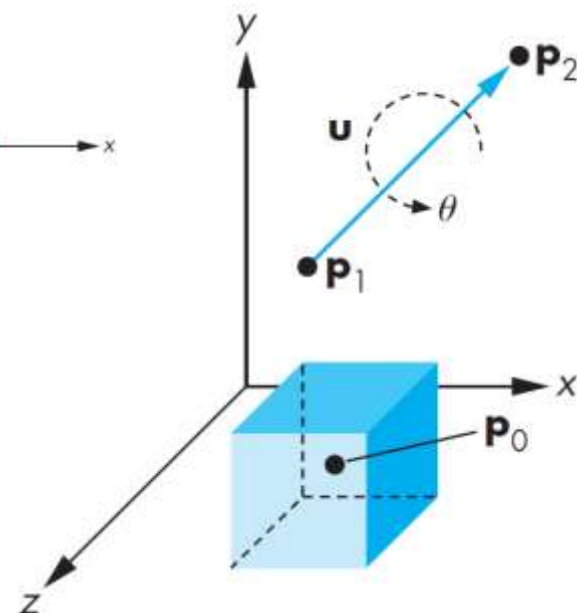


◆ 绕任意轴旋转变换的构造步骤：

1. 使用两个旋转变换 $R_x(\theta_x)$ 、 $R_y(\theta_y)$ 将轴向量 u 与 z 轴**对齐**
2. 施加绕 z 轴的旋转变换 $R_z(\theta)$
3. 使用两个逆旋转变换 $R_y(-\theta_y)$ 、 $R_x(-\theta_x)$ 将向量 u 转至原位



$$R = R_x(-\theta_x)R_y(-\theta_y)R_z(\theta)R_y(\theta_y)R_x(\theta_x)$$



Section 2

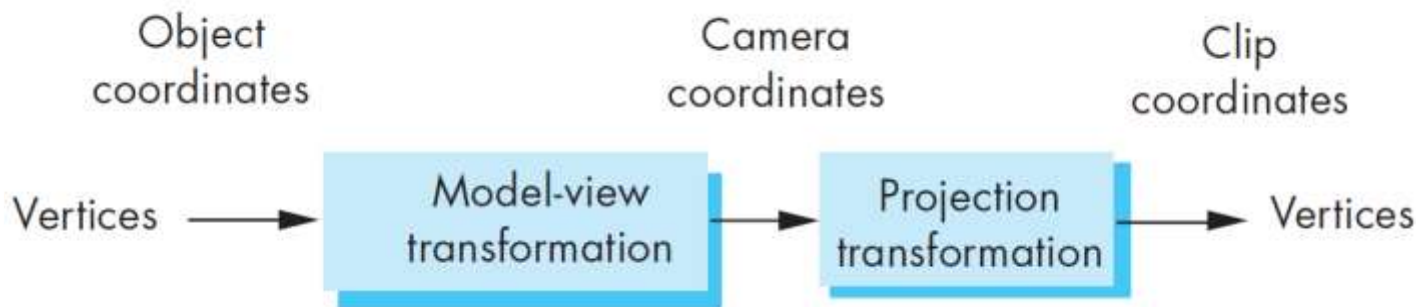
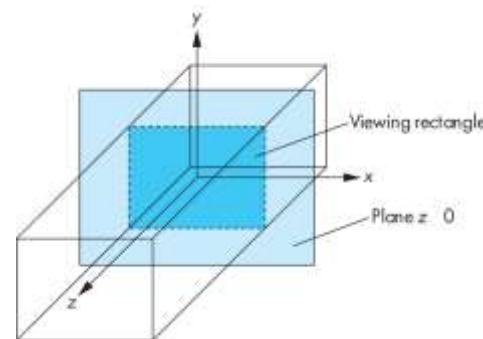
投影变换

◆ 计算视图的构建原理

- 应用程序分别指定物体形状与相机参数，由图形管线计算视图
- 视图在OpenGL中通过**坐标系转换与裁剪**实现
- 核心机制：**仿射变换、投影规范化**(Perspective Normalization)

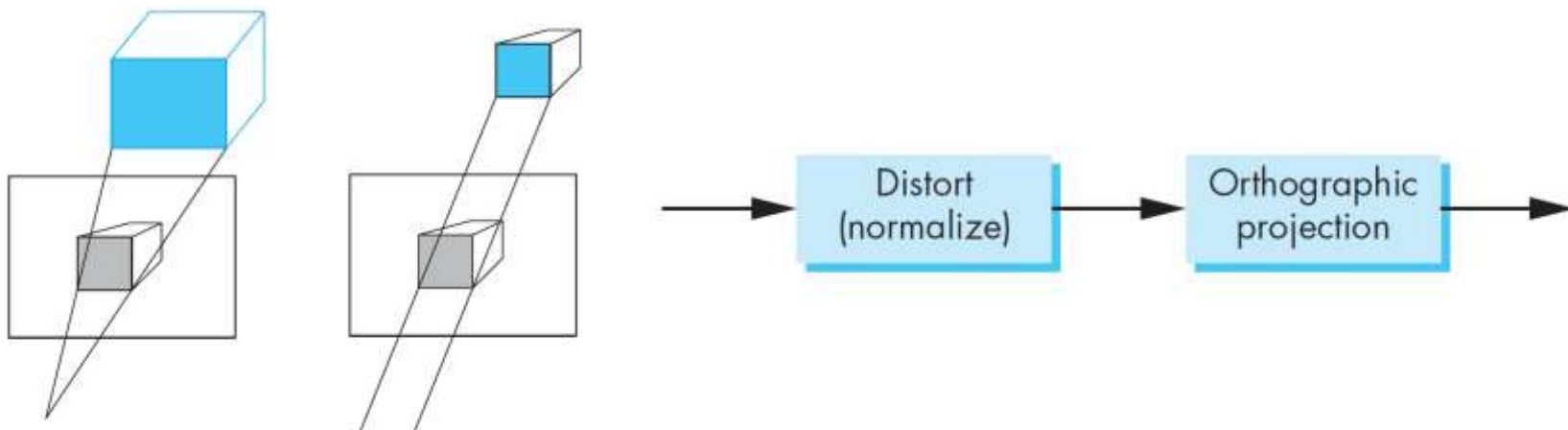
◆ OpenGL程序实现步骤

- 定位照相机：设置模 - 视图矩阵
- 设置镜头：设置投影矩阵，规范化视见体
- **裁剪**：裁剪规范视见体之外的几何图元

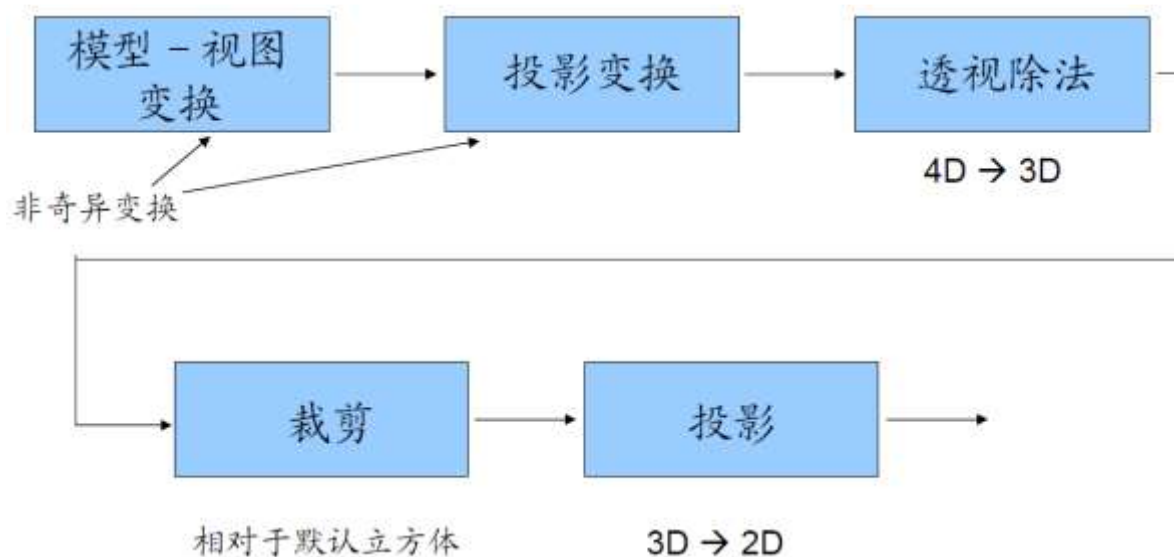


◆ 把所有的投影都转化为正交投影

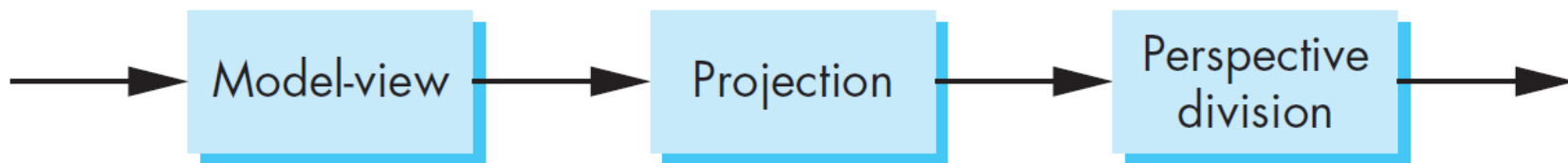
- 把对象变形使得它的正交投影图与原来想要得到的对象的投影图相同
- 适用于图形管线的另外一种处理方式：**射影畸变+正交投影**
- 不想为每种类型的投影设计不同的投影矩阵，所以把所有的投影转化为具有默认视见体的正交投影：**规范视见体 $x = \pm 1, y = \pm 1, z = \pm 1$**
- 这种策略可以使我们在流水线中应用标准变换，同样的绘制流水线可以既支持透视投影又支持平行投影



- ◆ 在模 - 视和投影变换过程中，始终使用四维齐次坐标系：
 - 这些变换都是非奇异的，即可逆的
 - 默认值为单位阵（正交视图）
- ◆ 规范化使得任意投影类型，都相对于默认立方体裁剪
 - 三维到二维的投影直到最后时刻才进行
 - 从而可以尽可能的保留深度信息，这对隐藏面消除是非常重要的



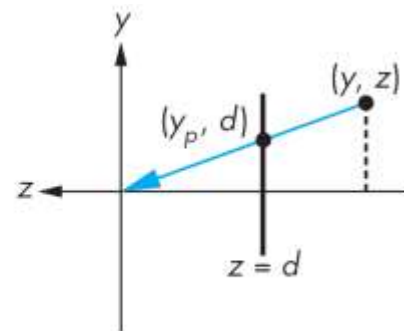
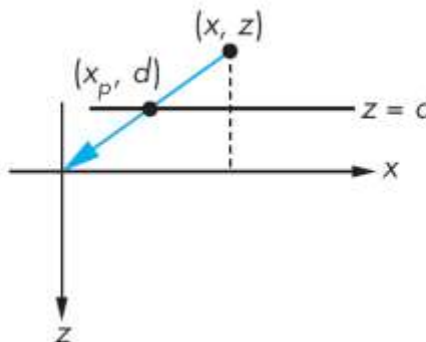
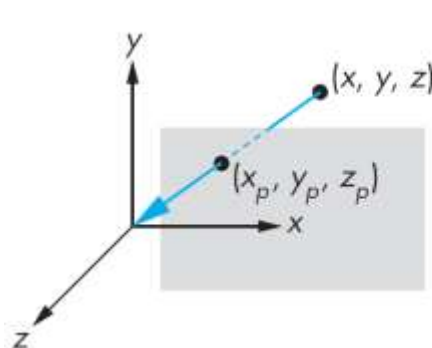
- ◆ 如果 $w \neq 1$, 那么必须从齐次坐标中除以 w 而得到所表示的点, 这就是透视除法



$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \quad \Rightarrow \quad \mathbf{q}' = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

◆ 透视投影的概念

- 投影中心在原点
- 投影平面为 $z = d, d < 0$
- 近大远小



◆ 透视方程：

$$x_p = \frac{x}{z/d},$$

$$y_p = \frac{y}{z/d},$$

$$z_p = \frac{z}{z/d} = d,$$

齐次坐标表达

$$\mathbf{p} = \mathbf{M}\mathbf{q}$$

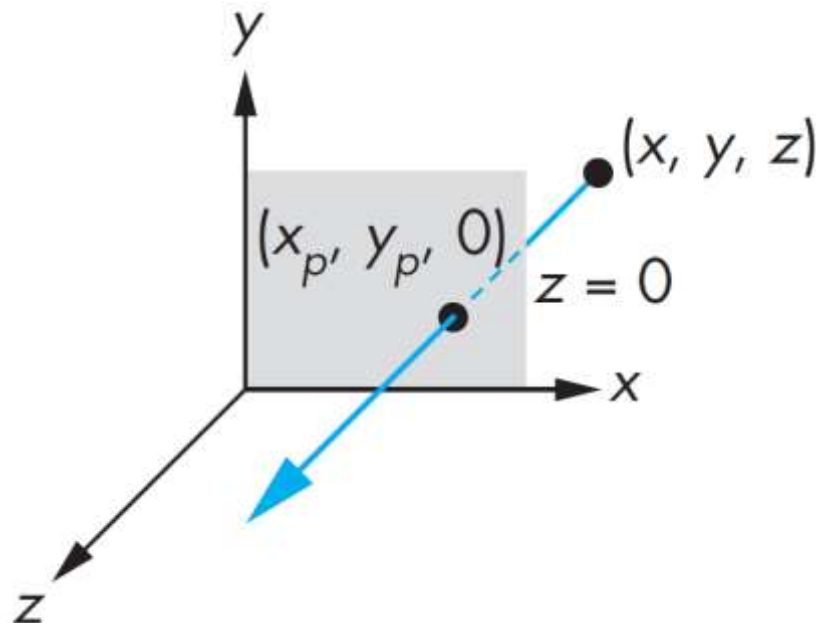
$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

◆ 正交投影原理

- 平行投影的特殊情况
- 投影方向与投影平面垂直
- 投影矩阵（投影平面为 $z = 0$ ）：

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



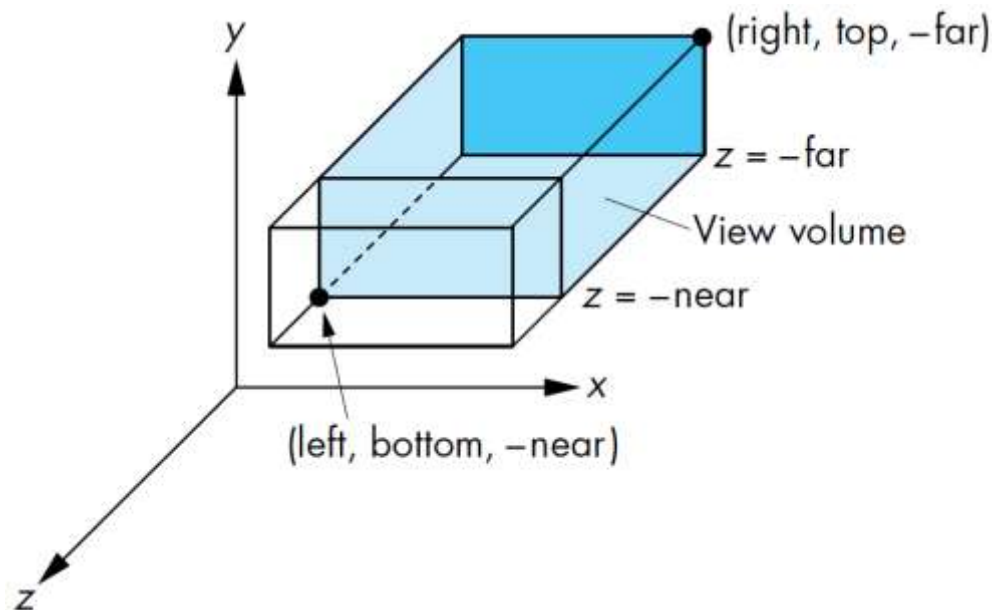
◆ 特点：

- 保持 x , y 方向上尺寸不变
- 默认观察范围是以原点为中心，变长为2的立方体区域

- ◆ 在OpenGL中采用下述函数指定正交投影：

```
mat4 Ortho(GLfloat left, GLfloat right, GLfloat bottom,  
           GLfloat top, GLfloat near, GLfloat far)
```

- ◆ 正交投影的视见体：

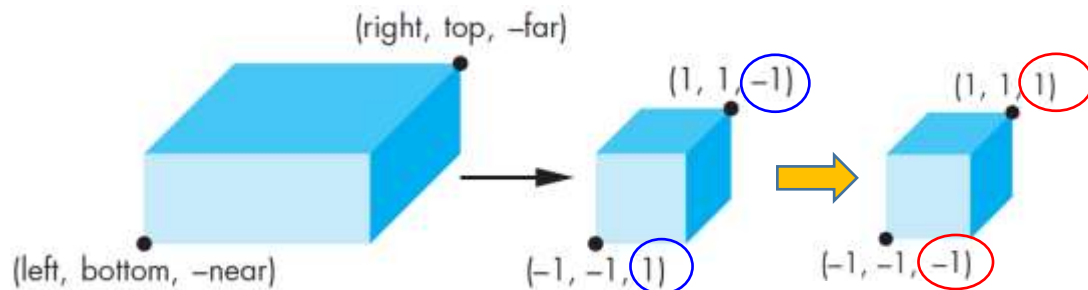


mat4 N = Ortho(left, right, bottom, top, near, far);

-near \rightarrow -1

-far \rightarrow 1

◆ 正交矩阵的构造:

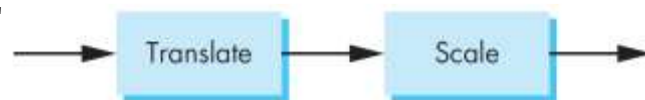


- 把中心移到原点, 对应的变换为:

$$T\left(-\frac{left+right}{2}, -\frac{bottom+top}{2}, \frac{near+far}{2}\right)$$

- 进行放缩从而使视见体的边长为2:

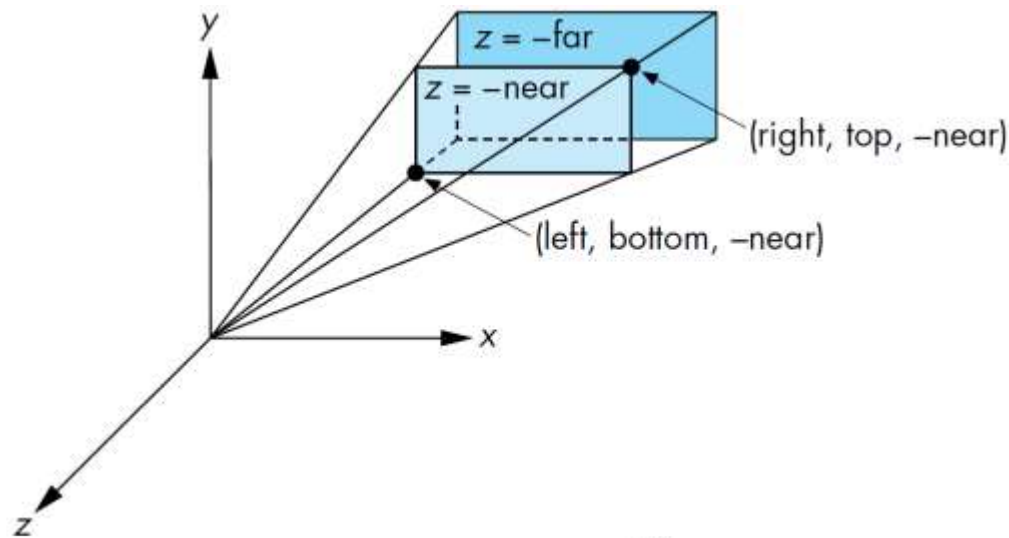
$$S\left(\frac{2}{right-left}, \frac{2}{top-bottom}, -\frac{2}{far-near}\right)$$



$$ST = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{left+right}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

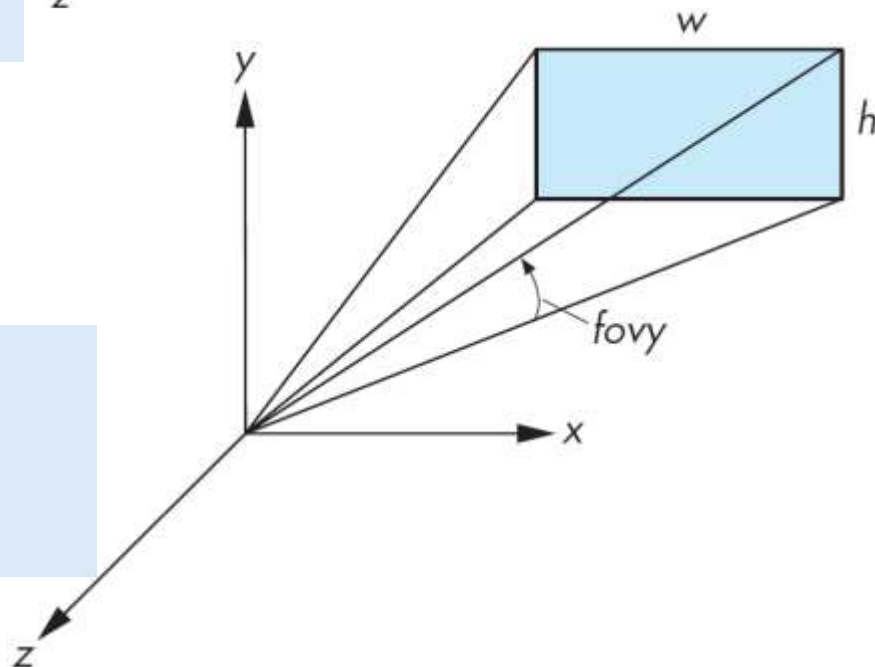
◆ 视见体函数

```
mat4 Frustum(GLfloat left,  
             GLfloat right,  
             GLfloat bottom,  
             GLfloat top,  
             GLfloat near,  
             GLfloat far);
```



◆ 透视函数:

```
mat4 Perspective(GLfloat fovy,  
                GLfloat aspect,  
                GLfloat near,  
                GLfloat far);
```

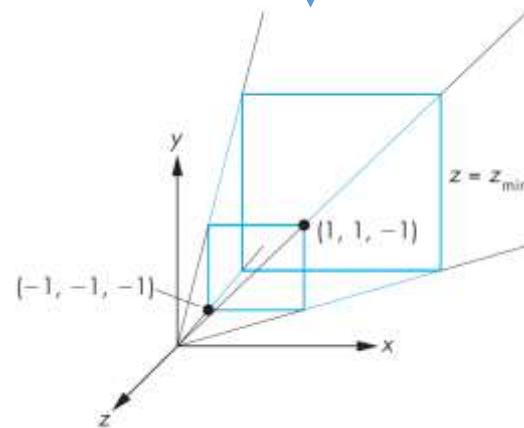
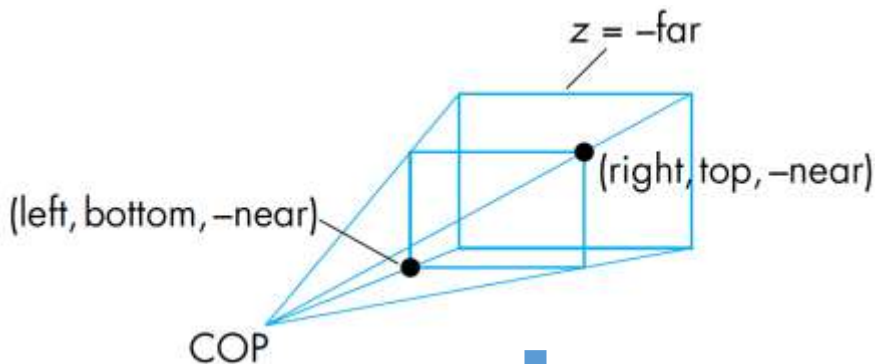
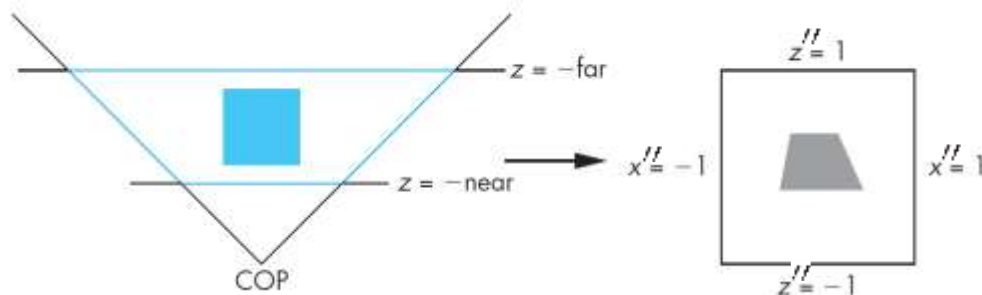


◆ 视见体函数的矩阵?

```
mat4 Frustum(GLfloat left,  
             GLfloat right,  
             GLfloat bottom,  
             GLfloat top,  
             GLfloat near,  
             GLfloat far);
```

注意: **Frustum**函数没有限制视见体一定是对称的棱台!

⇒ 需要先将棱台变为对称标准的棱台, 再做透视归一化变换



◆ 视见体函数的矩阵?

- 错切变换: 把不对称棱台变成对称+对称平面过原点

$$H(\theta, \phi) = H \left(\cot^{-1} \left(\frac{\text{left} + \text{right}}{2\text{near}} \right), \cot^{-1} \left(\frac{\text{top} + \text{bottom}}{2\text{near}} \right) \right)$$

- 缩放变换: 把棱台的侧面变成 $x = \pm z$ 和 $y = \pm z$

$$S(2 * \text{near} / (\text{right} - \text{left}), 2 * \text{near} / (\text{top} - \text{bottom}), 1)$$

- 透视归一化变换

$$N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix} \Rightarrow P = NSH = \begin{bmatrix} \frac{2 * \text{near}}{\text{right} - \text{left}} & 0 & \frac{\text{right} + \text{left}}{\text{right} - \text{left}} & 0 \\ 0 & \frac{2 * \text{near}}{\text{top} - \text{bottom}} & \frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} & 0 \\ 0 & 0 & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} & \frac{-2\text{far} * \text{near}}{\text{far} - \text{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

◆ 透视函数的矩阵?

```
mat4 Perspective(GLfloat fovy,  
                 GLfloat aspect,  
                 GLfloat near,  
                 GLfloat far);
```

left = -right bottom = -top



top = near*tan(fovy)

right = top*aspect

$$\begin{bmatrix} \frac{2*near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2*near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & \frac{-2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



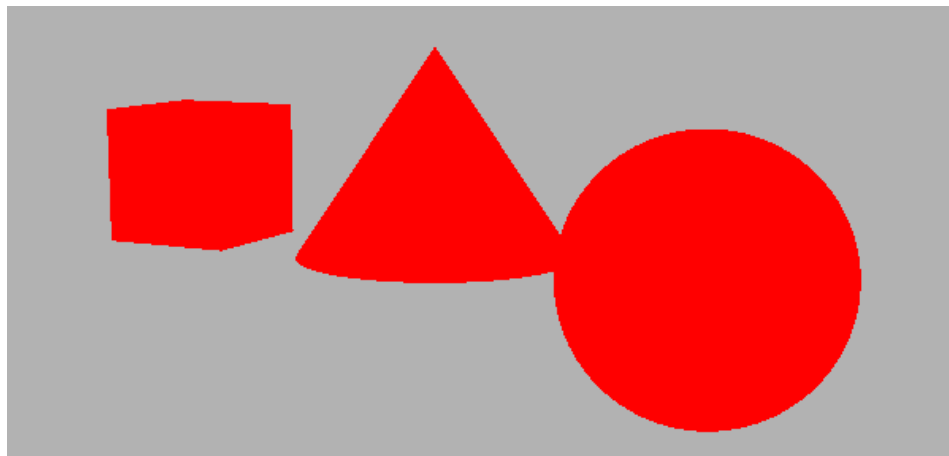
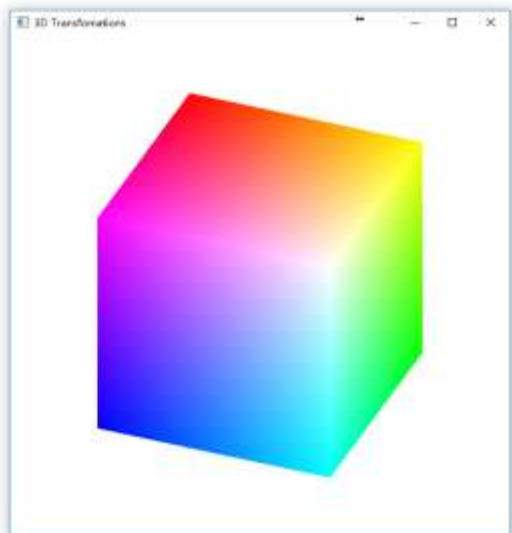
$$\begin{bmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Section 3

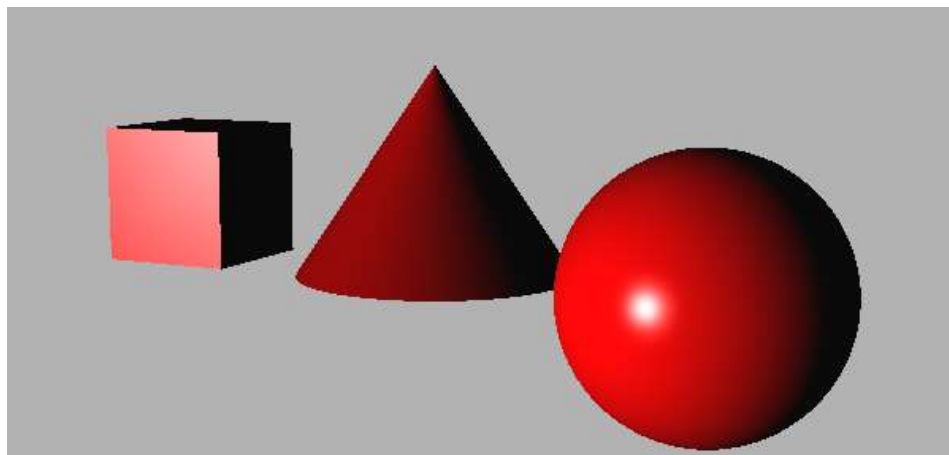
明暗绘制

◆ 绘制

- 几何对象
- 观察
 - 相机位置
 - 投影方式



光照和明暗绘制



- ◆ 曲面上某一点的颜色取决于光照与材质之间的多次作用
- ◆ 相互作用的三大分类：
 - **镜面反射**：如镜子
 - 可能有一部分入射光线被吸收，但是所有反射出去的光线都沿着同一个方向，反射光线的方向服从入射角等于反射角这一规律
 - **漫反射**：如墙面
 - 向各个方向散射的光线强度都相等
 - **折射**：如玻璃和水
 - 半透明表面，允许入射光线的一部分进入表面并且从对象的另一个位置再发射出去



镜面反射



漫反射



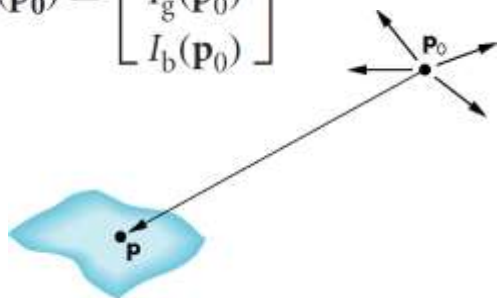
折射

◆ 基本光源:

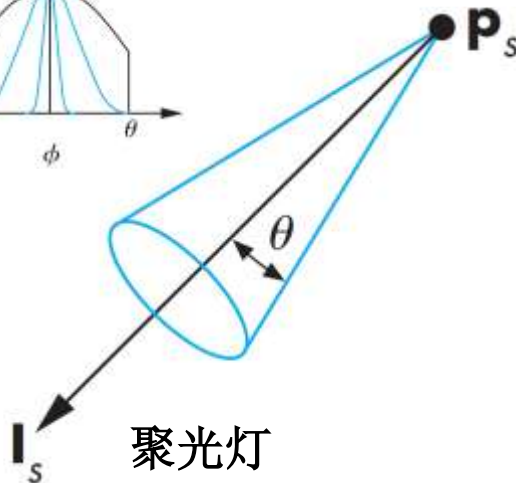
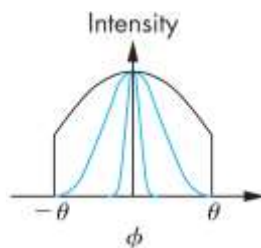
- 环境光: 环境中每个位置的环境光强度完全相同
- 点光源: 向各个方向发射的光线的强度相等
- 聚光灯: 具有一个比较的窄的照明范围, 通常为圆锥形半无穷区域
- 远距离光源: 在无穷远处的光源, 光线为平行线

$$\mathbf{I}_a = \begin{bmatrix} I_{ar} \\ I_{ag} \\ I_{ab} \end{bmatrix}$$

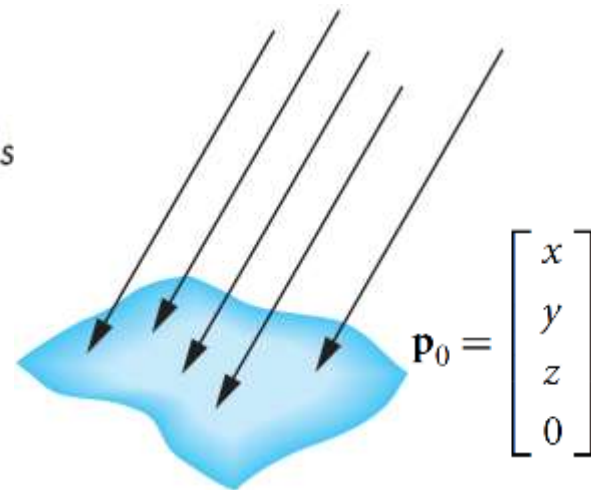
$$\mathbf{I}(\mathbf{p}_0) = \begin{bmatrix} I_r(\mathbf{p}_0) \\ I_g(\mathbf{p}_0) \\ I_b(\mathbf{p}_0) \end{bmatrix}$$



点光源



聚光灯



远距离光源

- ◆ Phong反射模型考虑了光线和材料之间三种相互作用：
 - 环境光反射
 - 漫反射
 - 镜面反射
- ◆ 假定有一组点光源 $\{L_i\}$, 对于每一个颜色分量 (r, g, b) , 每个点光源都有相应独立的三个光照分量：
 - 环境光分量 a
 - 漫反射光分量 d
 - 镜面反射光分量 s

$$\mathbf{L}_i = \begin{bmatrix} L_{ira} & L_{iga} & L_{iba} \\ L_{ird} & L_{igd} & L_{ibd} \\ L_{irs} & L_{igs} & L_{ibs} \end{bmatrix}$$

◆ 反射模型：

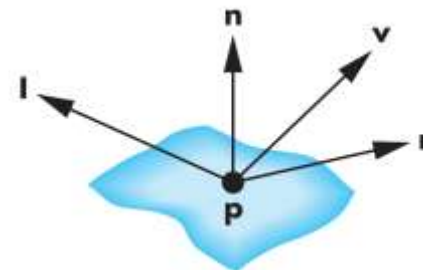
- 需要计算出入射光线的每个光照分量在曲面上 p 点的反射率

$$\mathbf{R}_i = \begin{bmatrix} R_{ira} & R_{iga} & R_{iba} \\ R_{ird} & R_{igd} & R_{ibd} \\ R_{irs} & R_{igs} & R_{ibs} \end{bmatrix} \longleftrightarrow \mathbf{L}_i = \begin{bmatrix} L_{ira} & L_{iga} & L_{iba} \\ L_{ird} & L_{igd} & L_{ibd} \\ L_{irs} & L_{igs} & L_{ibs} \end{bmatrix}$$

- p 点红色分量的光强计算公式：
$$I_{ir} = R_{ira}L_{ira} + R_{ird}L_{ird} + R_{irs}L_{irs}$$
$$= I_{ira} + I_{ird} + I_{irs}.$$
- p 点总光强为各个颜色分量光强汇总（省略光源和颜色下标）：

$$I = I_a + I_d + I_s = L_a R_a + L_d R_d + L_s R_s$$

- 反射率取决于曲面材质、光线方向、光源距离、视点位置等因素

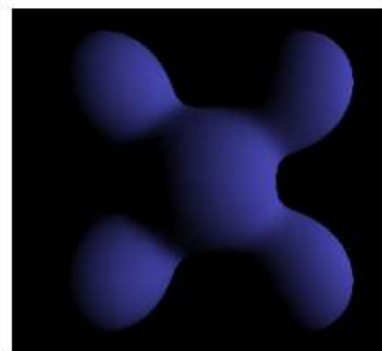


◆ 环境光反射+漫反射+镜面反射:

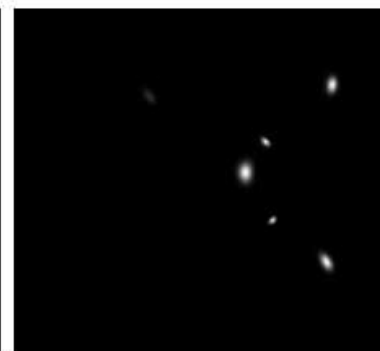
$$I = \frac{1}{a + bd + cd^2} (k_d L_d \max(\mathbf{l} \cdot \mathbf{n}, 0) + k_s L_s \max((\mathbf{r} \cdot \mathbf{v})^\alpha, 0)) + k_a L_a$$



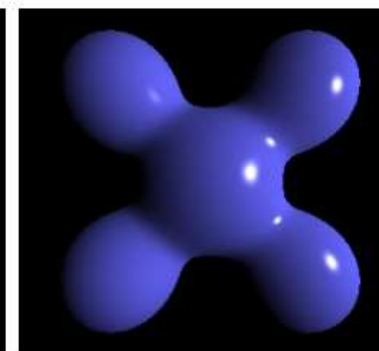
环境光反射



漫反射



镜面反射



Phong反射

+

+

=

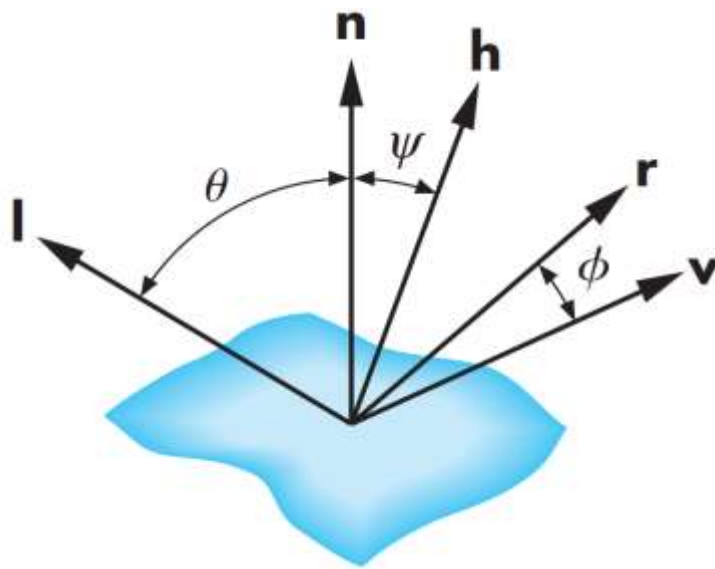
- ◆ 在Phong模型中，镜面光反射项计算量较大
 - 需要为每个顶点计算一个新的反射向量与视角方向的夹角
- ◆ Blinn-Phong模型
 - Blinn利用中**半角向量**，避免求反射角，从而使得计算效果更高
 - h 是 l 和 v 的平分单位向量，即

$$h = (l + v) / |l + v|$$

- $2\psi = \phi$ ，因此可以用 ψ 替换 ϕ :

$$(n \cdot h)^\beta \rightarrow (r \cdot v)^\alpha$$

可取适当的 β 以匹配明亮



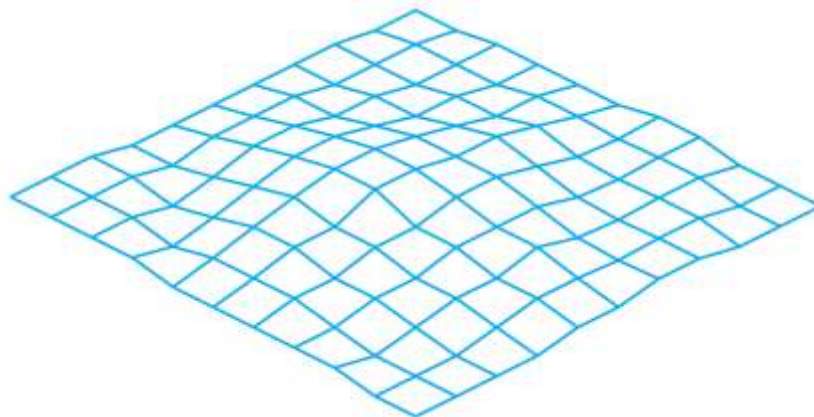


WIKIPEDIA

◆ 着色 (Shading) :

<https://en.wikipedia.org/wiki/Shading>

- 根据光源与材质决定场景中
- 某一曲面点的观测颜色

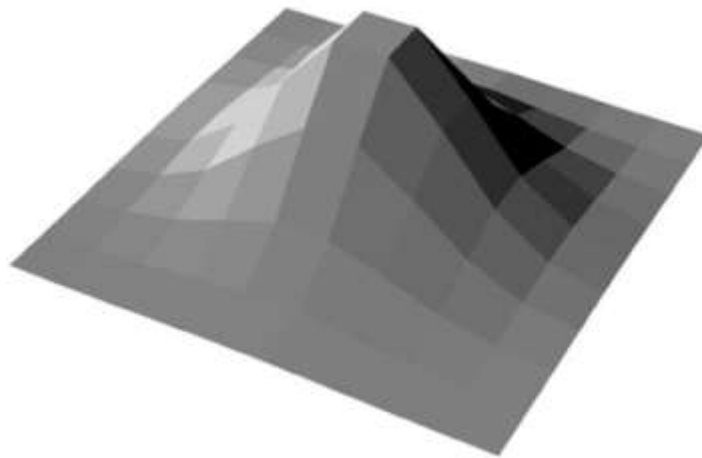


◆ 多边形着色:

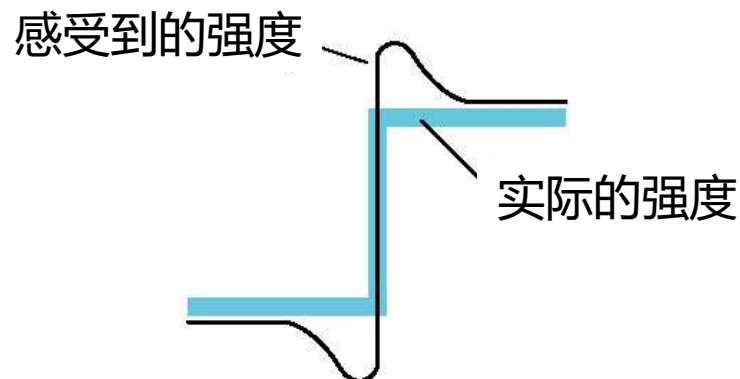
- 曲面的离散表示
- 如何获得连续or光滑的着色效果?



- ◆ 对每一个多边形根据其法向计算一个颜色

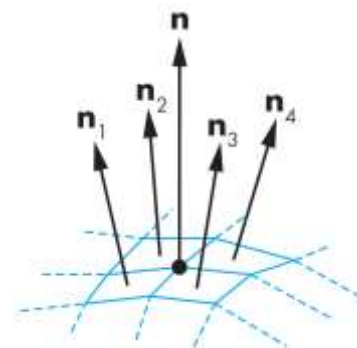


- ◆ Mach带效应
 - 边缘感知增强



- ◆ 在网格中每个顶点处有几个多边形交于该点，每个多边形有一个法向，取这几个法向的平均得到该点的法向：

$$\mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|}$$

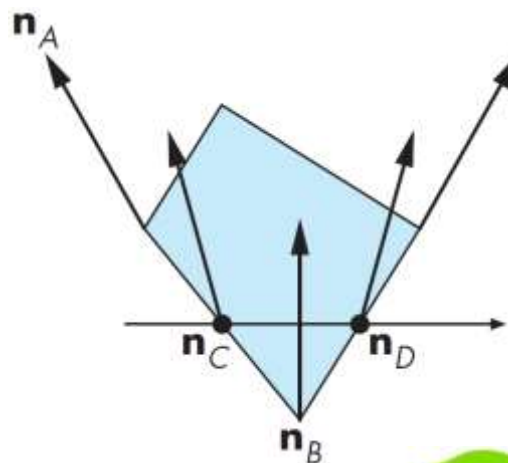
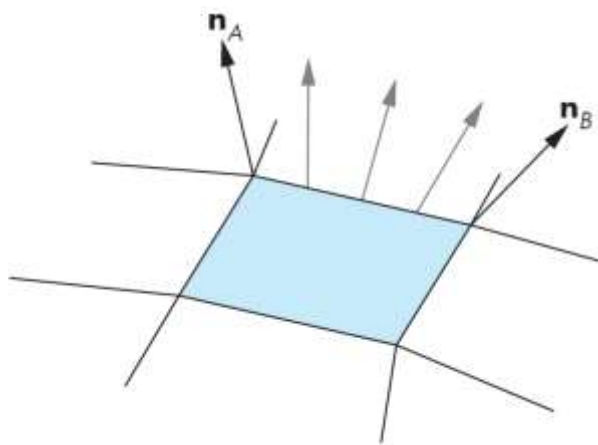


- ◆ 然后利用简单光照模型计算出顶点的颜色
- ◆ 对于多边形内的点，光栅化模块可以采用插值确定颜色
- ◆ 光滑着色在哪里实现计算？



- ◆ 与Gouraud方法不同，Phong方法是根据每个顶点的法向，插值出多边形内部各点的法向，然后基于光照模型计算出各点的颜色：

$$\mathbf{n}(\alpha, \beta) = (1 - \beta)\mathbf{n}_C + \beta\mathbf{n}_D$$

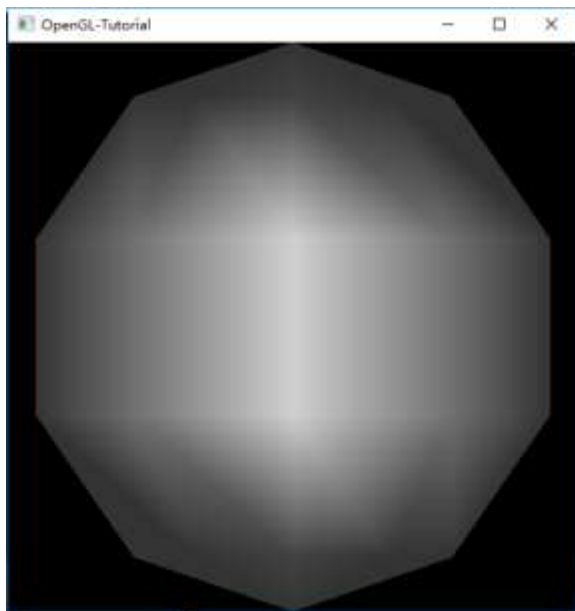


法向插值

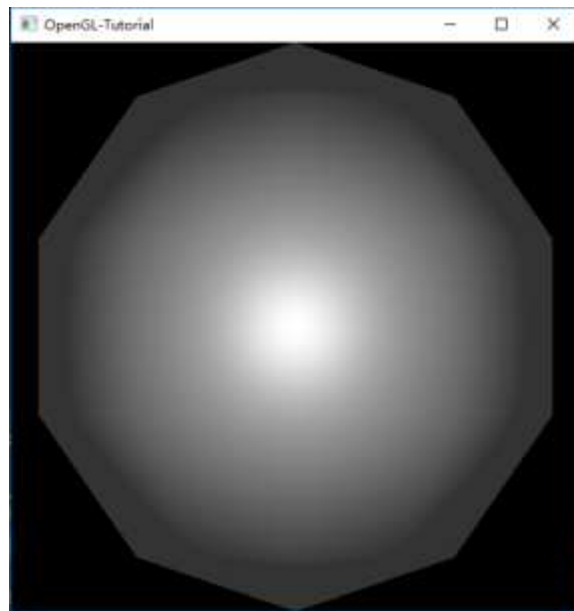
- ◆ Phong着色在哪里实现计算？



- ◆ 通常会有效地降低Mach带效应
- ◆ 得到的图形比应用Gouraud方法的结果更光滑



实验 3.3



实验 3.4

◆ 计算机产生的阴影

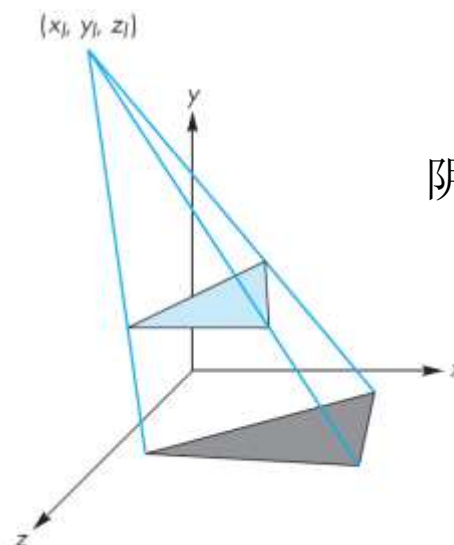
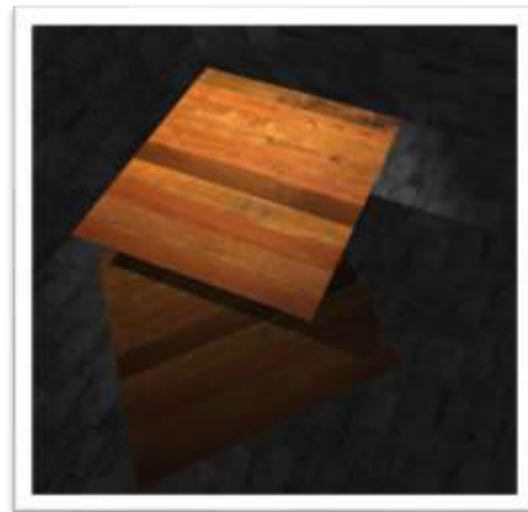
- 光源照射不到
- 视角观察不到

◆ 阴影的投影矩阵

- 向三维空间中的平面投影

光源位置: (x_l, y_l, z_l)

投影平面: $y = 0$



阴影 \Leftrightarrow 透视

$$\mathbf{p} = \mathbf{M}\mathbf{q}$$

◆ 透视投影矩阵:

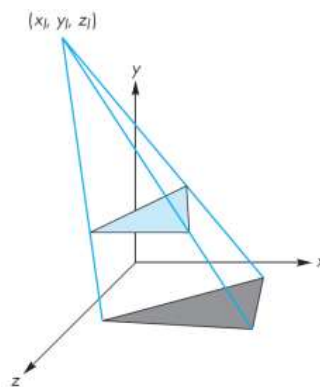
- 投影中心在原点
- 投影平面为 $z = d, d < 0$

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \Rightarrow \mathbf{q}' = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

◆ 阴影投影矩阵:

- 投影中心在 (x_l, y_l, z_l)
- 投影平面为 $y = 0$



- ➡
1. 将投影中心移至原点: $T(-x_l, -y_l, -z_l)$
 2. 投影至平面 $y = -y_l$:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix} + \text{透视除法}$$

$$x_p = x_l - \frac{x - x_l}{y - y_l} y_l$$

$$y_p = 0$$

$$z_p = z_l - \frac{z - z_l}{y - y_l} y_l$$

3. 将投影中心移回 (x_l, y_l, z_l) : $T(x_l, y_l, z_l)$

Section 4

纹理映射

- ◆ 虽然图形显示卡可以每秒钟显示多达一千万个多边形,但这个速度并不能满足模拟任何现象的要求

- 云
- 草
- 地貌
- 树皮
- 毛发
- 水波
- 火焰
- ...



◆ 纹理映射

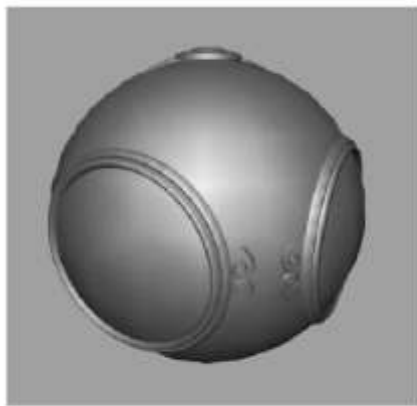
- 把图像映射到光滑表面，从而给表面增加细节特征

◆ 环境映射（或叫反射映射）

- 利用环境的图像进行纹理映射，可以模拟高度镜面曲面

◆ 凹凸映射

- 对法向量进行了扰动，从而使表面看起来有微小的起伏变化



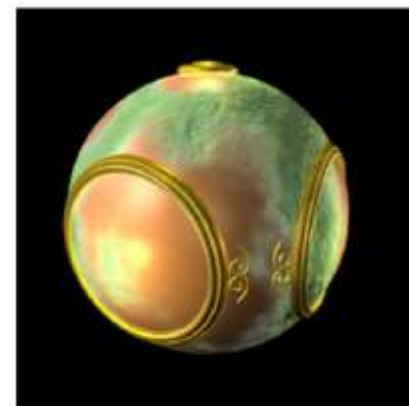
几何模型



纹理映射

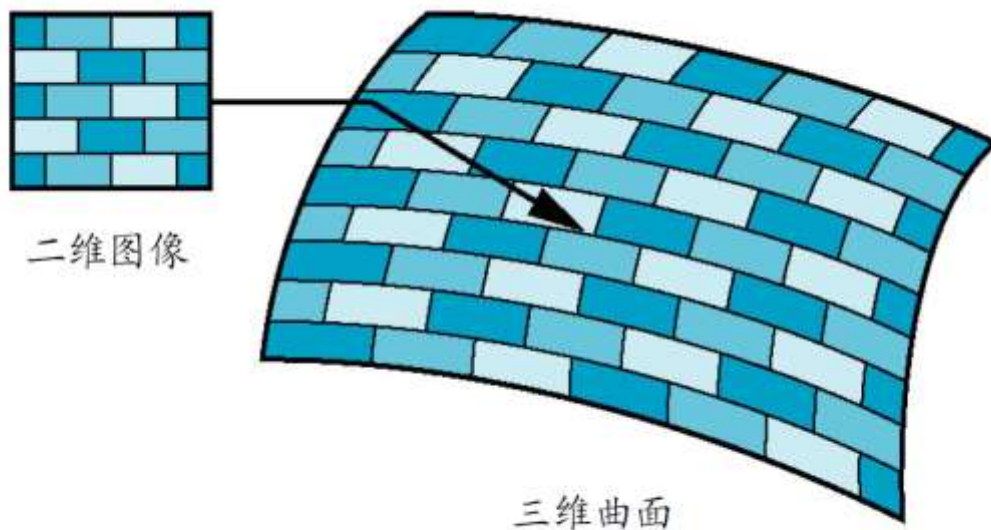


环境映射



凹凸映射

- ◆ 虽然映射的想法很简单，即把图像映射到曲面上，但由于这时要用到三四个坐标系，因此实现起来并不容易



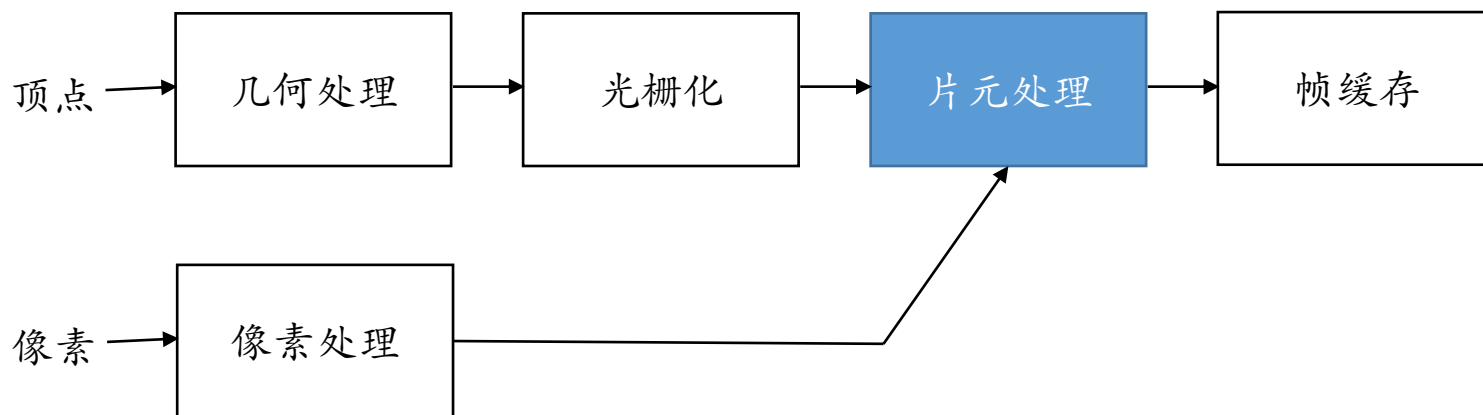
◆ OpenGL支持许多纹理映射选项

- OpenGL 1.0能将一维和二维纹理映射到一维至四维的图形对象
- 现在的版本提供了三维纹理映射，且得到大部分图形硬件的支持

◆ 本节只讨论从二维纹理到曲面的映射

◆ 图像与几何分别经过不同的流水线，在片元处理中汇合

- 复杂纹理并不影响几何的复杂性
- 非常有效，因为在经过所有的操作后，减少了许多不必要的映射



像素流水线和几何绘制流水线

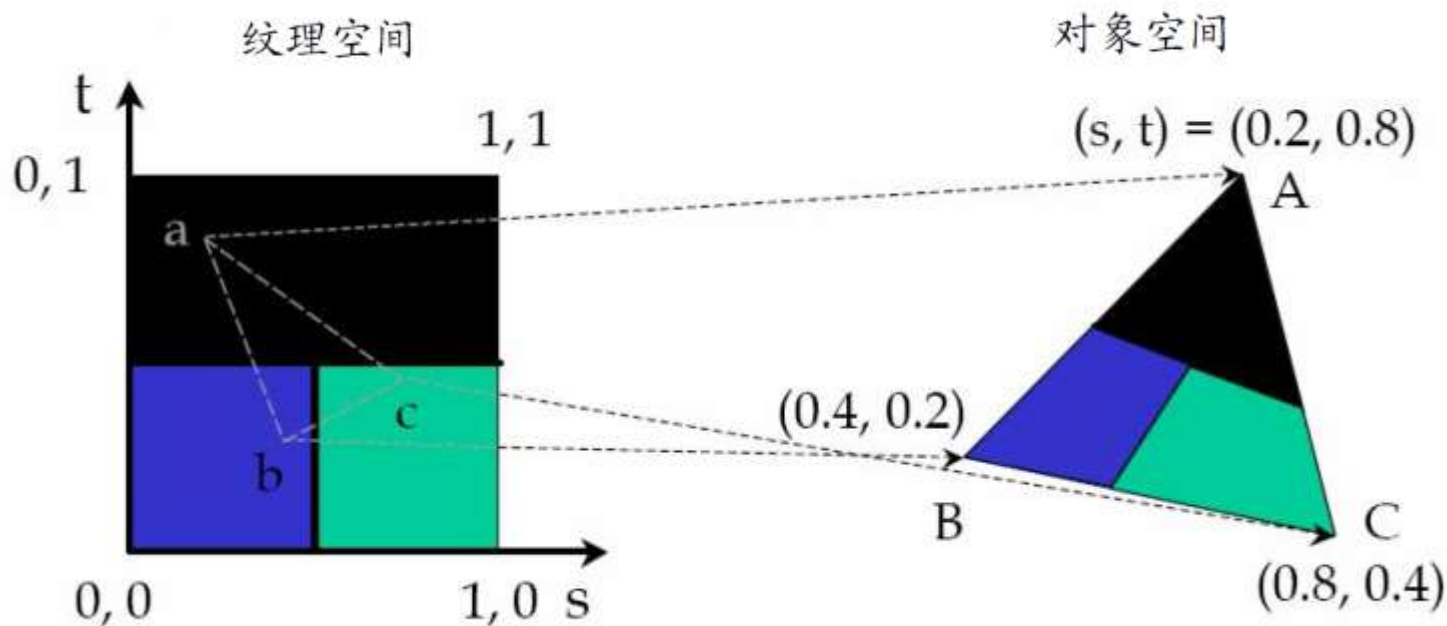
◆ 应用纹理需要下面三个基本步骤

- 指定纹理
 - 创建纹理对象
 - 读入或生成纹理图像
- 建立纹理映射
 - 将纹理坐标赋给每个片元
- 将纹理施加给每个片元
 - 点采样，线性滤波...



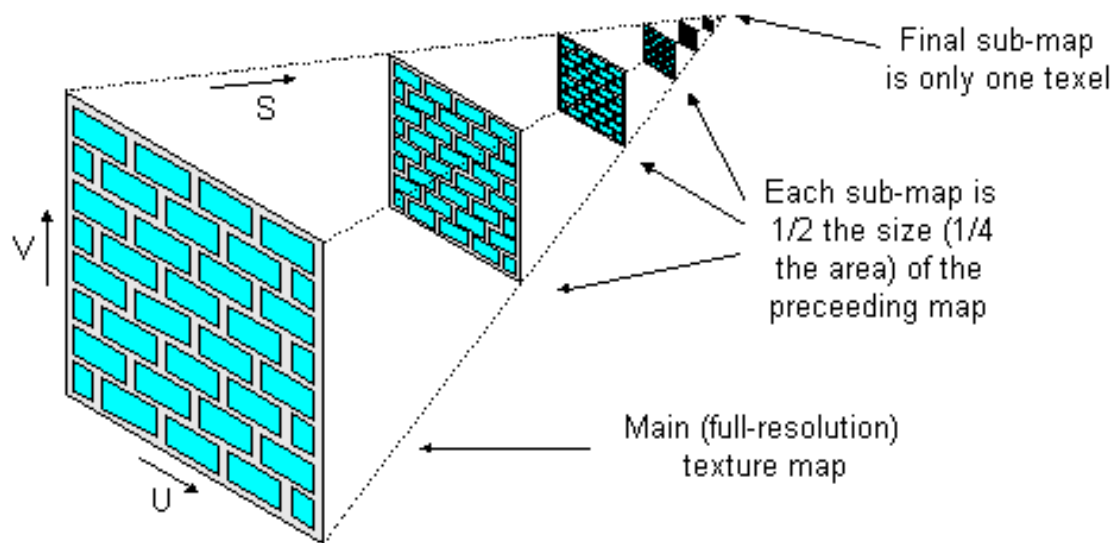
纹理（下方）是 256×256 的图像，它被映射到一个矩形上，经透视投影后的结果显示在上方

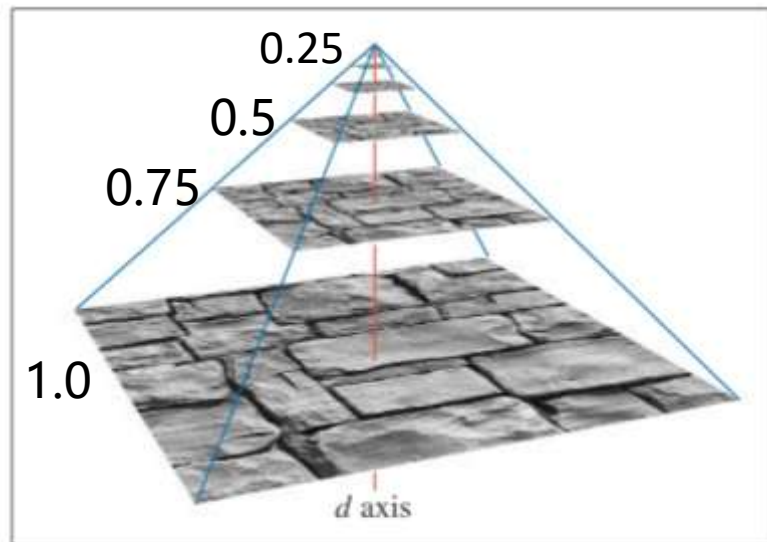
- ◆ 基于参数纹理坐标，把纹理坐标作为顶点的属性
- ◆ 将纹理坐标传送到顶点着色器中，顶点纹理坐标经过光栅化模块的插值计算得到片元的纹理坐标



- ◆ 用于处理纹理缩小的问题
- ◆ Mipmapping技术对纹理进行预先滤波，降低分辨率
- ◆ 对于非常小的要加纹理的对象，可以减小的插值误差
- ◆ 在纹理定义时声明Mipmap的层次

```
glTexImage2D(GL_TEXTURE_2D, level, ...);
```



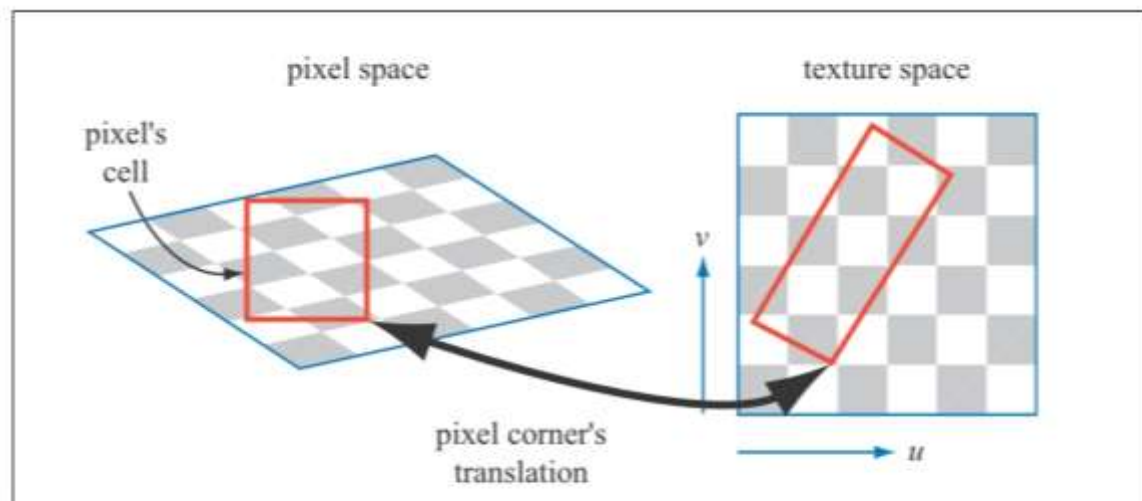


← Mipmap纹理 (纹理金字塔)

确定渲染像素所需金字塔层级

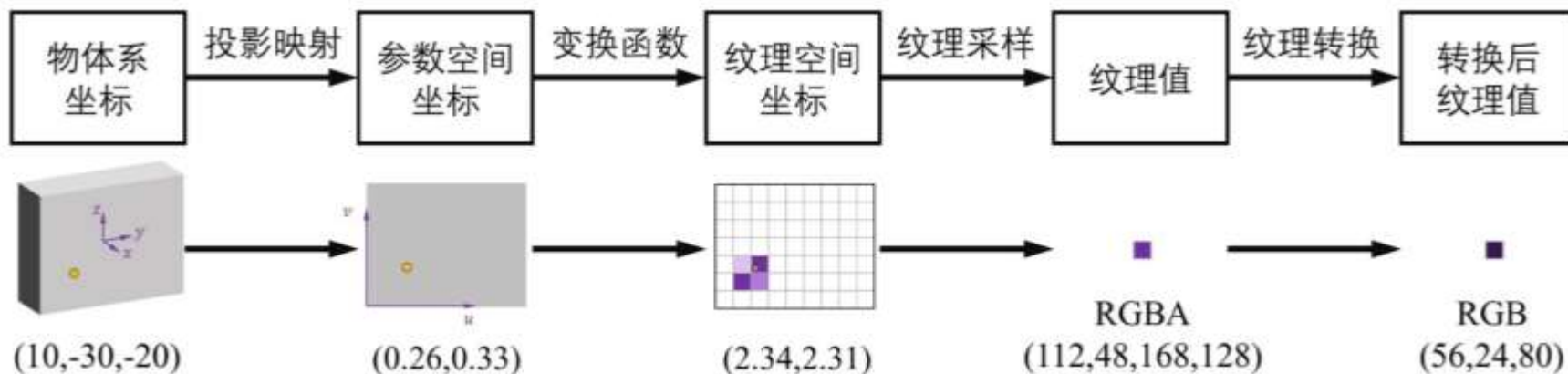


From:
Real-Time Rendering by T. Akenine-Möller, E. Haines, N. Hoffman, A. P
Iwanicki, S. Hillaire



From:
Real-Time Rendering by T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, S. Hillaire

图形学基础 - 纹理 - 纹理映射流程 - 杨鼎超的文章 - 知乎
<https://zhuanlan.zhihu.com/p/3699778>
49

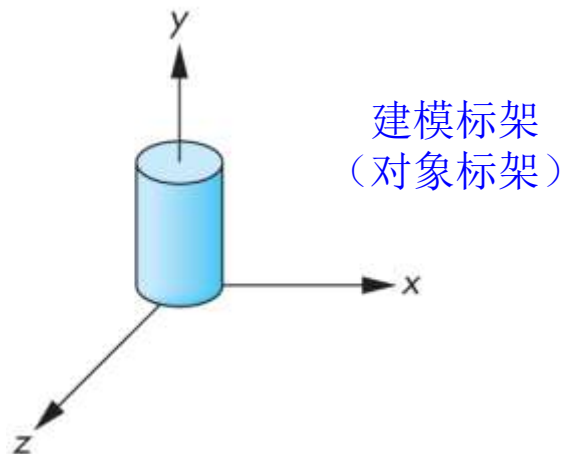


1. **投影映射**：将三维物体坐标转化为二维参数空间 uv 坐标，实时渲染中， uv 坐标通常是保存在顶点信息中
2. **变换函数**：将 uv 坐标经过处理变换后，根据实际的纹理尺寸，转化为纹理空间坐标，此时也可能有小数
3. **纹理采样**：依据纹理空间坐标，对纹理进行采样，要处理放大和缩小两个情况，其中缩小的情况更为复杂，牵涉到各向异性过滤的算法
4. **纹理转换**：通过采样得到纹理值后，往往不能直接使用，还需要进行相应转换才能使用

Section 5

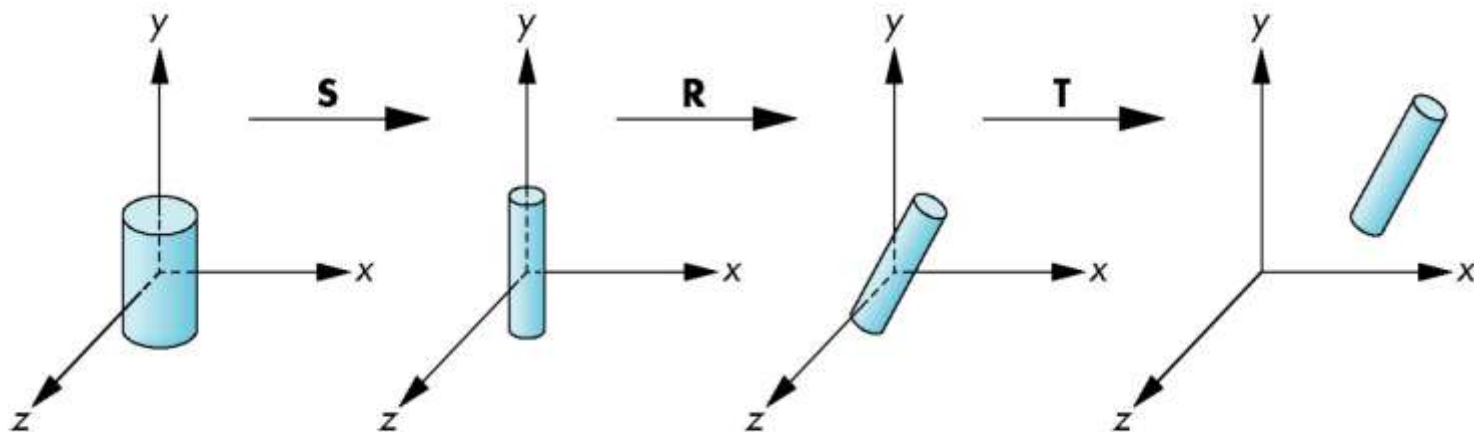
层级建模

- ◆ 大多数图形API对图元采取**最小完备**的观点：
 - 他们只包含少数的基本图元
 - 而让用户通过这些基本的图元来构建更复杂的对象
- ◆ 这些基本图元通常被视为一些**图符** (symbol) , 所要建模的世界可以看成由这些离散的图符组成
- ◆ 通常根据自身的几何特征表示为合适的大小和方位：
 - 圆柱体：
 - 中心轴与坐标系的某个主轴平行
 - 高度为1个单位
 - 半径为1个单位
 - 底面圆心与坐标系的原点重合

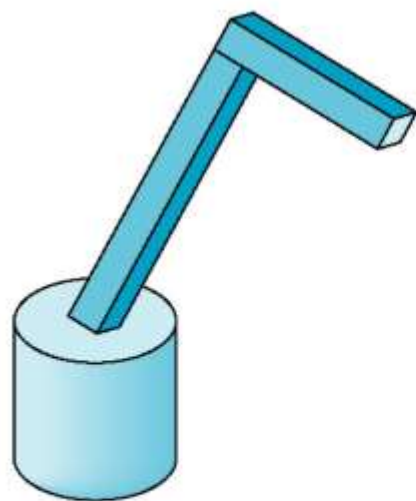


- ◆ 在OpenGL应用程序中，必须通过几何变换把图符从建模标架变换到世界标架，得到一个实例（instance）
- ◆ 实例变化把每个图符实例按照所需的大小，方向和位置放入到场景中：

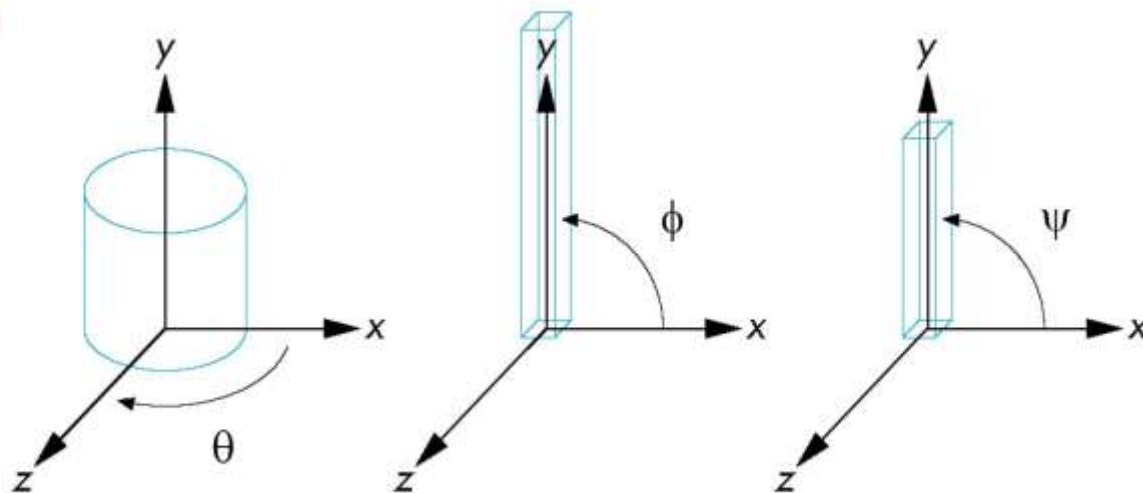
$$M = TRS$$



◆ 整体模型和各个部件：

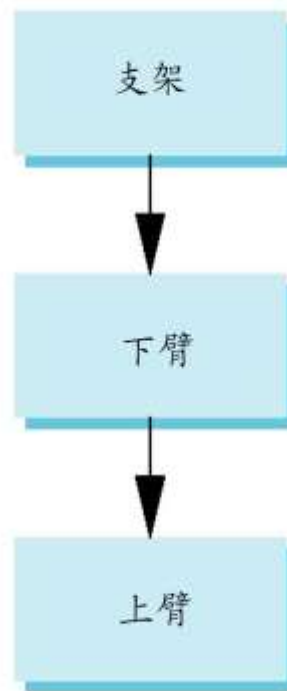
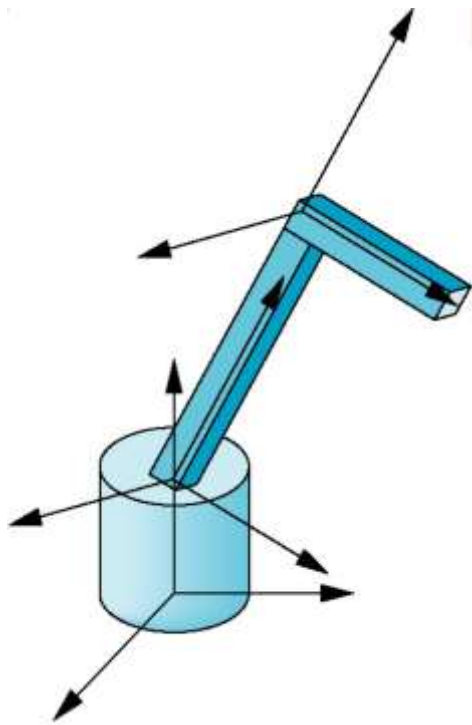


机器人手臂



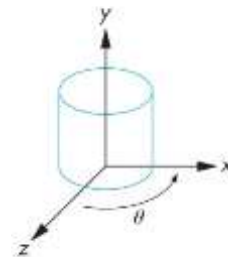
每部分在自己坐标系中的形状

- ◆ 机器人手臂就是一个**关联模型**(articulated model)
- ◆ 部件之间在关节处连接在一起
- ◆ 可以通过给定**关节角**指定模型的状态



◆ 支架的旋转: R_b

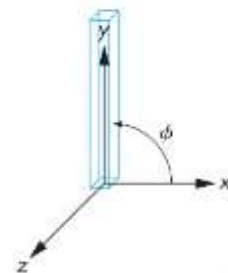
- 把 $M = R_b$ 应用到支架上



◆ 下臂相对于支架部分部分平移: T_{la}

◆ 下臂绕关节旋转: R_{la}

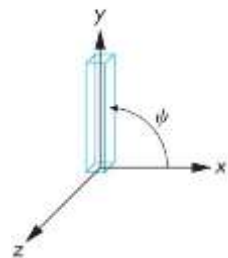
- 把 $M = R_b T_{la} R_{la}$ 应用到下臂上



◆ 上臂相对于下臂平移: T_{ua}

◆ 上臂绕关节旋转: R_{ua}

- 把 $M = R_b T_{la} R_{la} T_{ua} R_{ua}$ 应用到上臂上

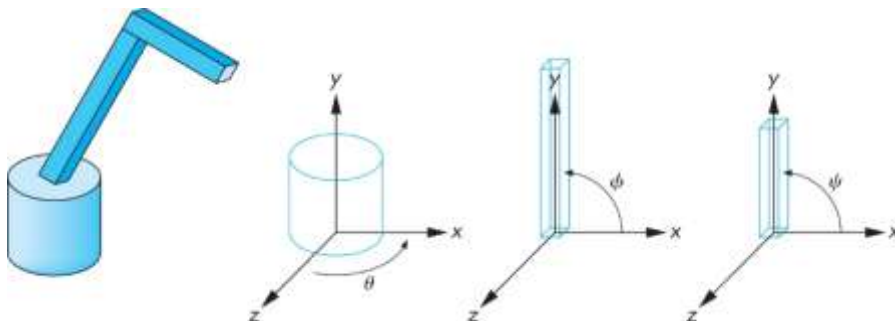


- ◆ 以增量的方式改变模-视矩阵以体现运动联动性：
 - 假设给定了三个关节角：Theta[Base], Theta[LowerArm], Theta[UpperArm]

```
model_view = RotateY(Theta[Base] ); //支架变换矩阵  
base(); //绘制支架
```

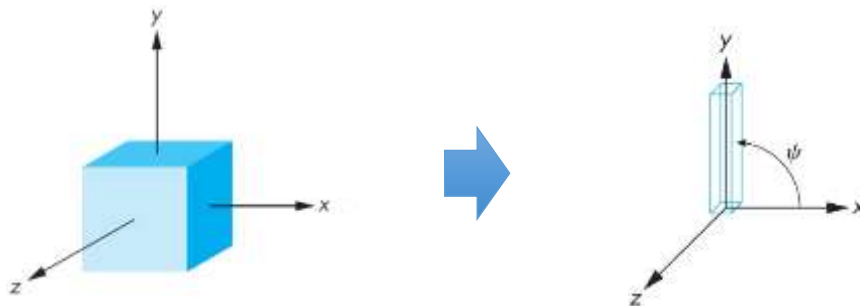
```
model_view = model_view * Translate(0.0, BASE_HEIGHT, 0.0)  
                * RotateZ(Theta[LowerArm]); //下臂变换矩阵  
lower_arm(); //绘制下臂
```

```
model_view = model_view * Translate(0.0, LOWER_ARM_HEIGHT, 0.0)  
                * RotateZ(Theta[UpperArm]); //上臂变换矩阵  
upper_arm(); //绘制上臂
```



◆ 绘制每一个部件实例：图符 + 实例变换矩阵

- 实验补充1中以单位正方体作为图符：中心在原点，边长为1
- 以上臂为例：



```
Void upper_arm()
```

```
{
```

```
    mat4 instance = Translate( 0.0, 0.5 * UPPER_ARM_HEIGHT, 0.0 )  
                      * Scale(UA_WIDTH, UA_HEIGHT, UA_WIDTH );
```

```
    //按长宽高缩放正方体，并且平移使得底面在y=0上
```

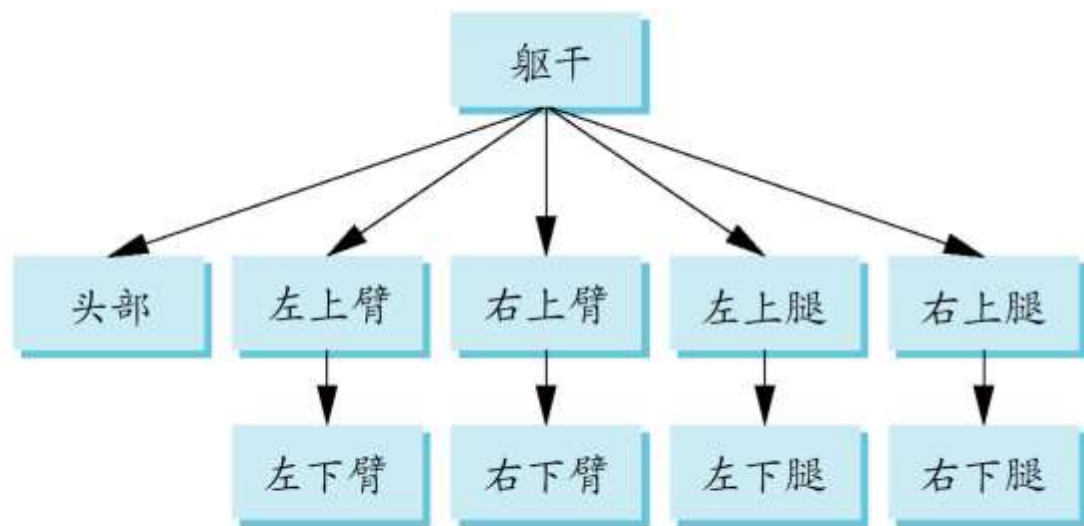
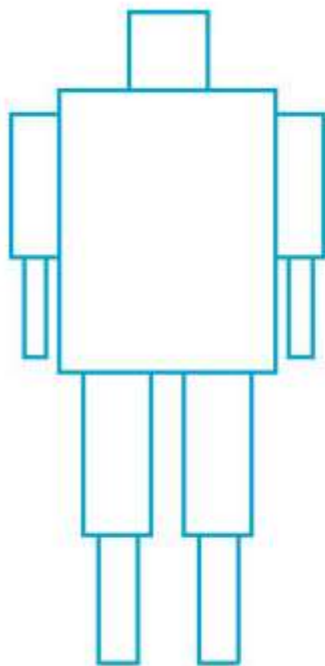
```
    glUniformMatrix4fv(ModelView, 1, GL_TRUE, model_view * instance );
```

```
    //将完整的模-视矩阵传给shader
```

```
    glDrawArrays( GL_TRIANGLES, 0, NumVertices ); //绘制相应的三角面片
```

```
}
```

- ◆ 但是对于具有更复杂的结构的模型，通过树结构的遍历来确定部件的绘制顺序会使更加便捷：
 - **先序遍历**：沿着树左子树开始访问节点，一直遍历到树叶子节点，然后回溯到树上一层访问第一个右子树，照此一直递归下去

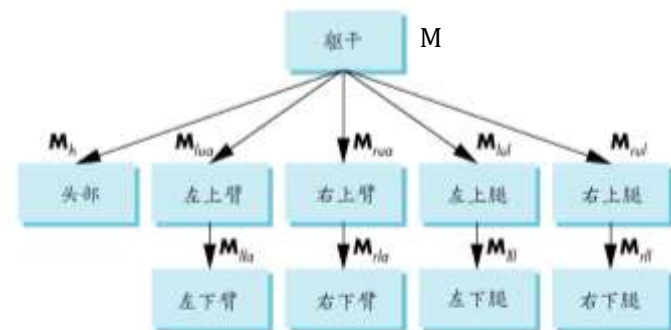
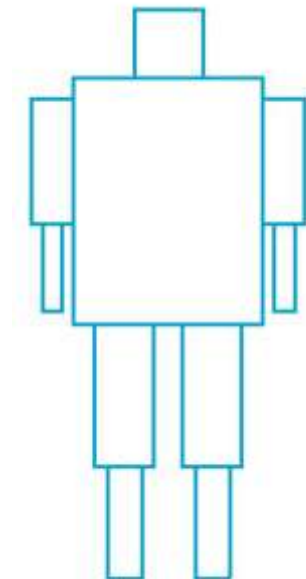


```
model_view = RotateY( theta[Torso] ); //躯干变换矩阵
torso(); //躯干绘制
```

```
mvstack.push( model_view ); //保存躯干变换矩阵
model_view *= 头结点局部变换矩阵;
head(); //头部绘制
model_view = mvstack.pop(); //恢复躯干变换矩阵
```

```
mvstack.push( model_view ); //保存躯干变换矩阵
model_view *= 左上臂局部变换矩阵;
left_upper_arm(); //左上臂绘制
model_view *= 左下臂局部变换矩阵;
left_lower_arm(); //左下臂绘制
model_view = mvstack.pop(); //恢复躯干变换矩阵
```

```
mvstack.push( model_view ); //保存躯干变换矩阵
model_view *= 右上臂局部变换矩阵;
right_upper_arm(); //右上臂绘制
model_view *= 右下臂局部变换矩阵;
right_lower_arm(); //右下臂绘制
model_view = mvstack.pop(); //恢复躯干变换矩阵
:
```



- ◆ 在示例代码中没有对状态进行修改
 - 例如没有改变颜色，可以在各个部件绘制过程中分别赋色
- ◆ 对每一个部件，同样需要一个从图符到实例的变换：
 - **push**和**pop**操作起到代码隔离作用，防止误操作

```
Void upper_arm()  
{  
    mvstack.push( model_view ); //保存当前模-视变换矩阵  
  
    mat4 instance = Translate( 0.0, 0.5 * UPPER_ARM_HEIGHT, 0.0 )  
                    * Scale(UA_WIDTH, UA_HEIGHT, UA_WIDTH );  
    //按长宽高缩放正方体，并且平移使得底面在y=0上  
  
    glUniformMatrix4fv(ModelView, 1, GL_TRUE, model_view * instance );  
    //将完整的模-视矩阵传给shader  
  
    colorCube(); // 给部件赋色  
    glDrawArrays( GL_TRIANGLES, 0, NumVertices ); //绘制相应的三角面片  
  
    model_view = mvstack.pop(); //恢复当前模-视变换矩阵  
}
```

