

# COMP 371 – Winter 2012

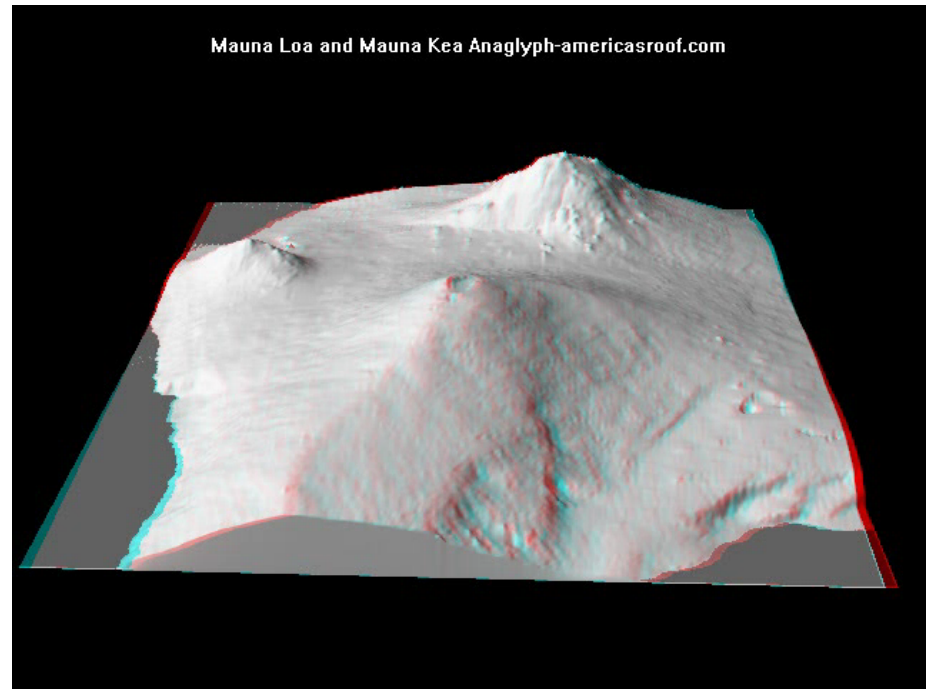
## Computer Graphics

- 3D Transformations.



# 3-D Transformations

- Types
  - Translation
  - Rotation
  - Scaling
  - Shear, reflection
- Mathematical representation
- OpenGL functions for applying



## Uses of Transformations

- **Modeling:** position and resize parts of a complex model
- **Viewing:** define and position the virtual camera
- **Animation:** define how objects move/change with time

# Review: Representing Transformations

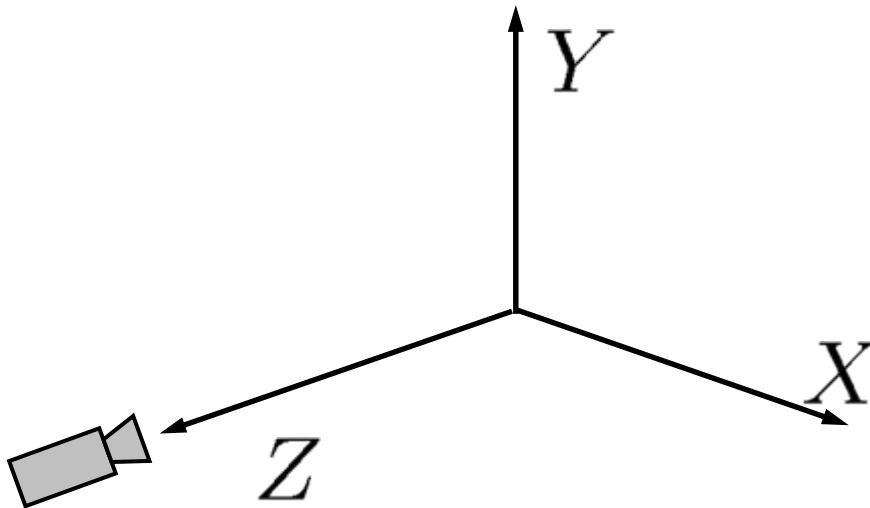
- Note that we've defined translation as a vector addition but rotation, scaling, etc. as matrix multiplications
- It's inconvenient to have two different operations (addition and multiplication) for different forms of transformation
- It would be desirable for all transformations to be expressed in a common form
  - **Solution:** Homogeneous coordinates

# Homogeneous coordinates

- Add an extra coordinate,  $W$ , to a point.
  - $P(x,y,z) \rightarrow P(xW,yW,zW,W)$ .
- Two sets of homogeneous coordinates represent the same point if they are a multiple of each other.
  - $(2,5,3,11)$  and  $(4,10,6,22)$  represent the same point.
- At least one component must be non-zero  $\Rightarrow (0,0,0,0)$  is not defined.
- If  $W \neq 0$ , divide by it to get Cartesian coordinates of point  $(x/W,y/W,z/W,1)$ .
- If  $W=0$ , point is said to be at infinity.

# OpenGL's coordinates

- The underlying form of all points/vertices is a 4-D vector  $(x_h, y_h, z_h, w)$
- If you do something in 2-D, OpenGL simply sets  $z = 0$  for you
- If the scale coordinate  $w$  is not set explicitly (recall that there is a `glVertex4()` that allows you to do so), OpenGL sets  $w = 1$  for you



# 3D Transformations.

- What is a transformation?

$$P' = T(P)$$

- Why use them?

## Modelling

- *Create objects in natural/convenient coordinates*
- *Multiple instances of a prototype shape*
- *Kinematics of linkages/skeletons - robot animation*

## Viewing

- *Windows and device independence*
- *Virtual camera: parallel and perspective projections*

# 3D Transformations

- All transformations in 3D can be expressed as combinations of translations, rotations, scaling
  - expressed using matrix multiplication

- Translation.

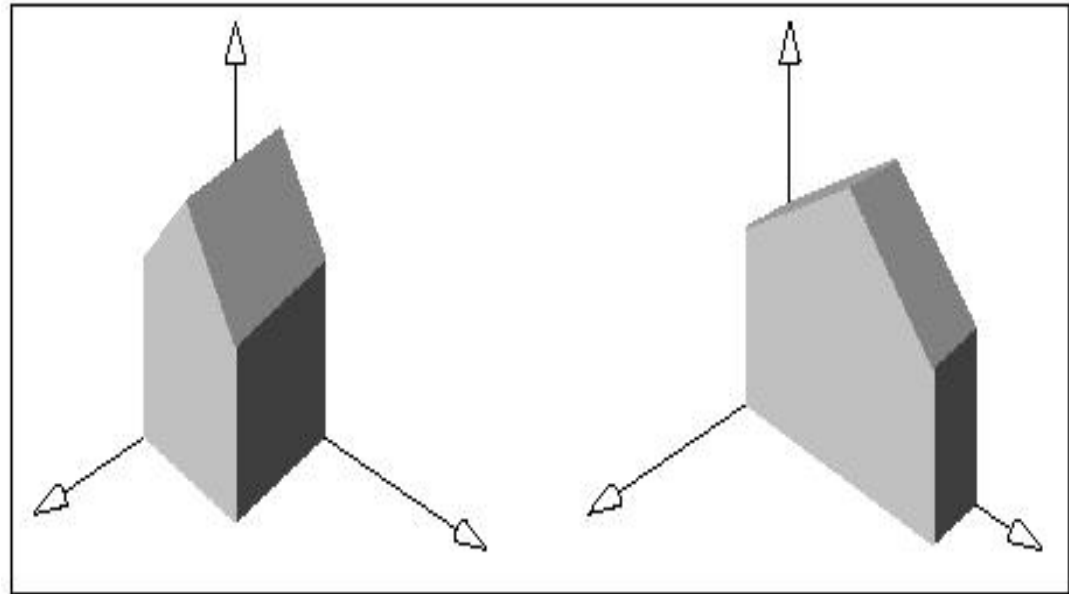
- $P' = T + P$

- Scale

- $P' = S \cdot P$

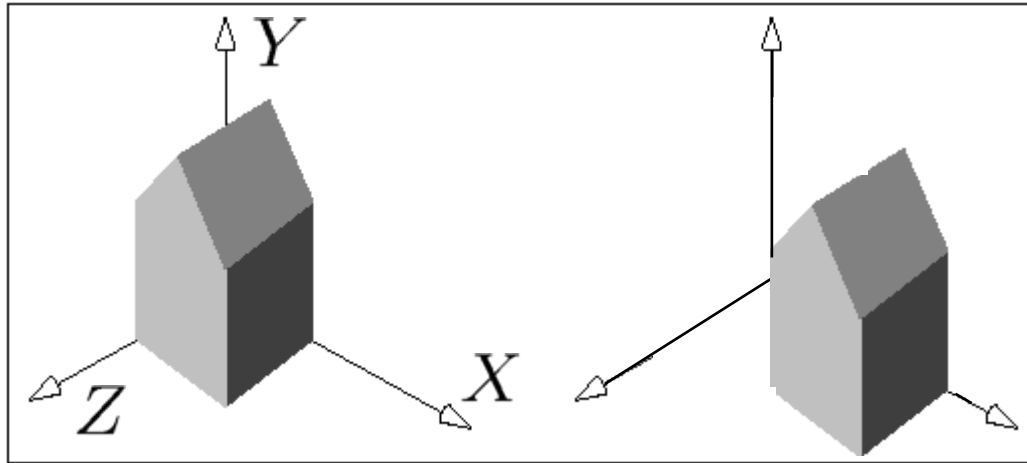
- Rotation

- $P' = R \cdot P$



- We would like all transformations to be multiplications so we can concatenate them  $\Rightarrow$  express points in **homogenous coordinates**.

# 3-D Translations

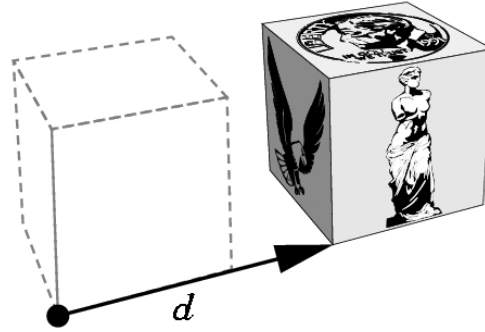


$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

# 3D Translation



(a)



(b)

$$\mathbf{T}_{\text{translate}} = \mathbf{M}^T = \begin{bmatrix} 1 & 0 & 0 & \Delta_x \\ 0 & 1 & 0 & \Delta_y \\ 0 & 0 & 1 & \Delta_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

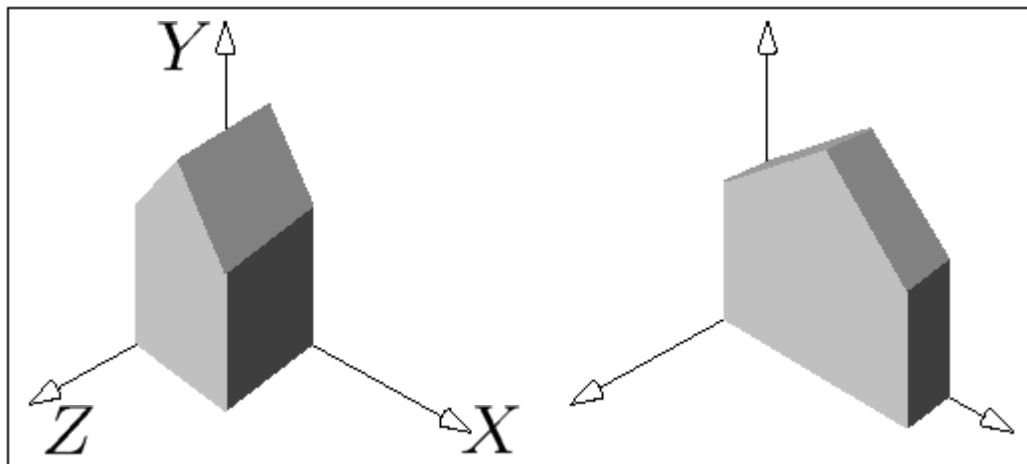
## 3D Translation with homogeneous coordinates

- Old way:
- New way:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

# 3-D Scaling



$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

# 3-D Uniform Scaling

- Note that uniform scaling can also be accomplished using homogeneous coordinate properties by manipulating scale coordinate

$$\begin{pmatrix} s_x \\ s_y \\ s_z \\ 1 \end{pmatrix} = \begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & s \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

“Standard” scaling matrix

## 3-D Uniform Scaling

Note that uniform scaling can also be accomplished using homogeneous coordinate properties by manipulating scale coordinate

$$\begin{pmatrix} x \\ y \\ z \\ 1/s \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/s \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

↑  
must normalize to 1

“Homogeneous” scaling matrix

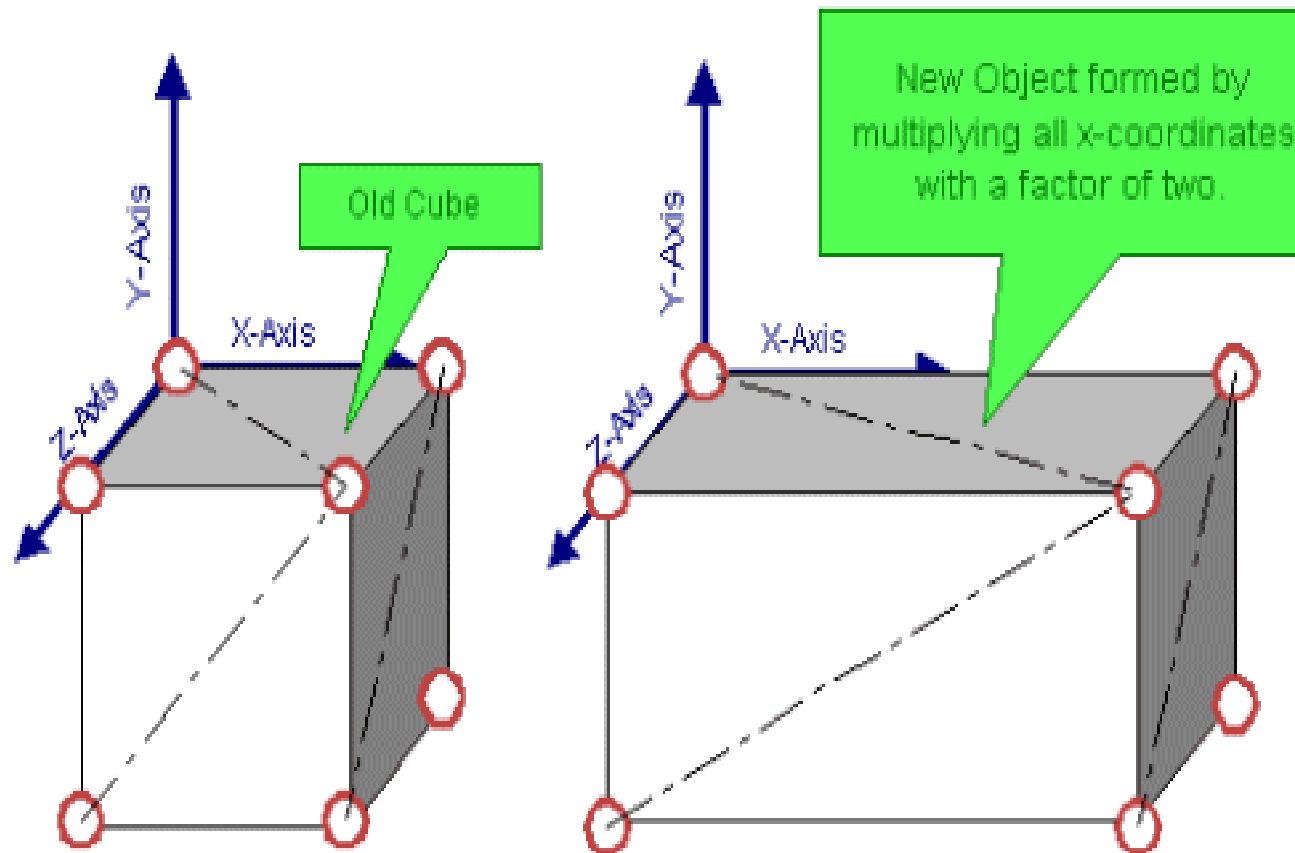
## 3-D Uniform Scaling

Note that uniform scaling can also be accomplished using homogeneous coordinate properties by manipulating scale coordinate

$$\begin{pmatrix} sx \\ sy \\ sz \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/s \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

“Homogeneous” scaling matrix

# Scaling along x-direction



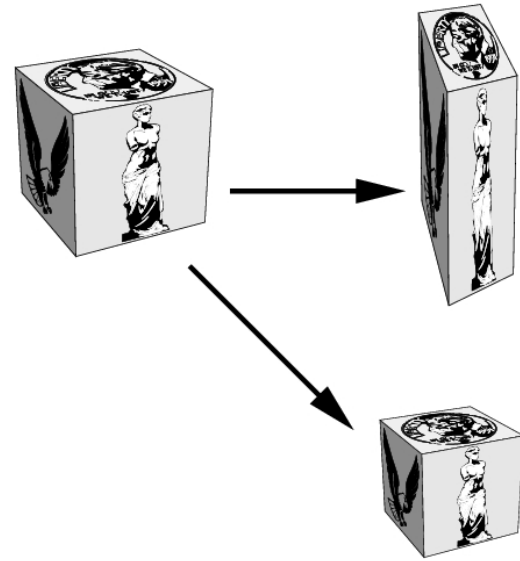
# Scaling

Scale the new frame:

$$x' = s_x x$$

$$y' = s_y y$$

$$z' = s_z z$$

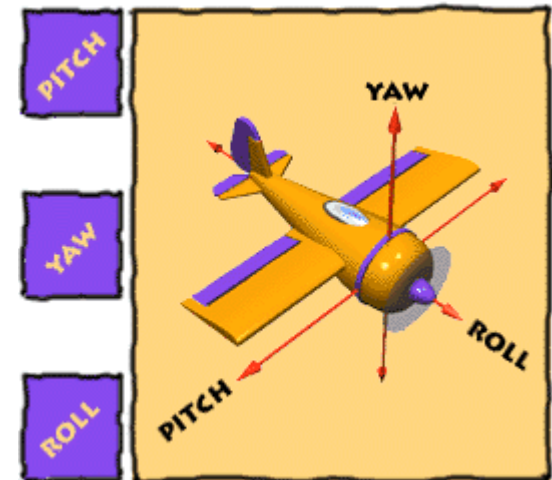
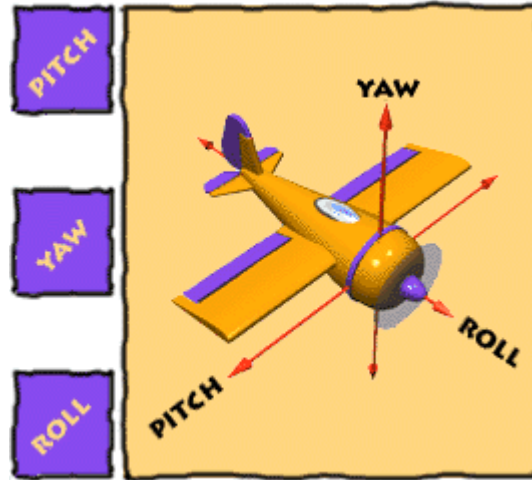
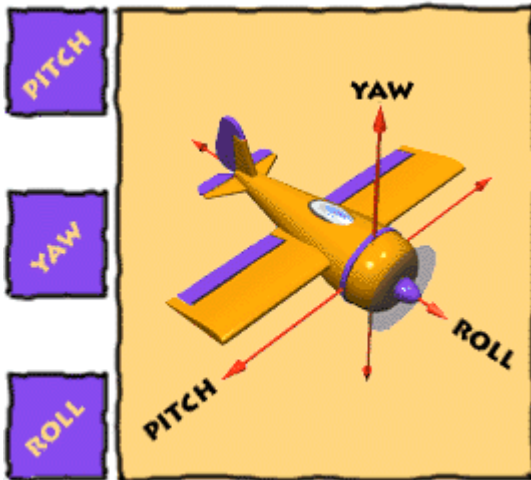
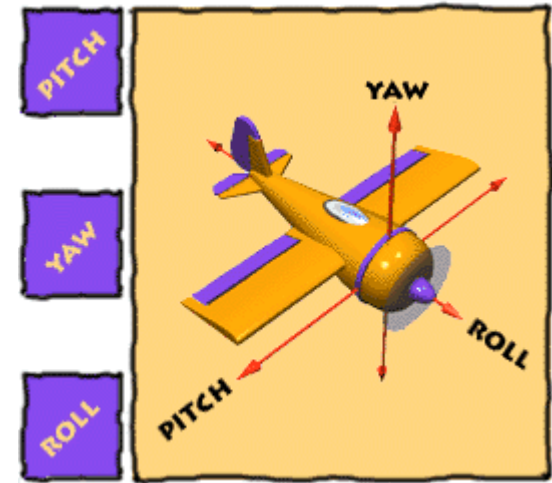


$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

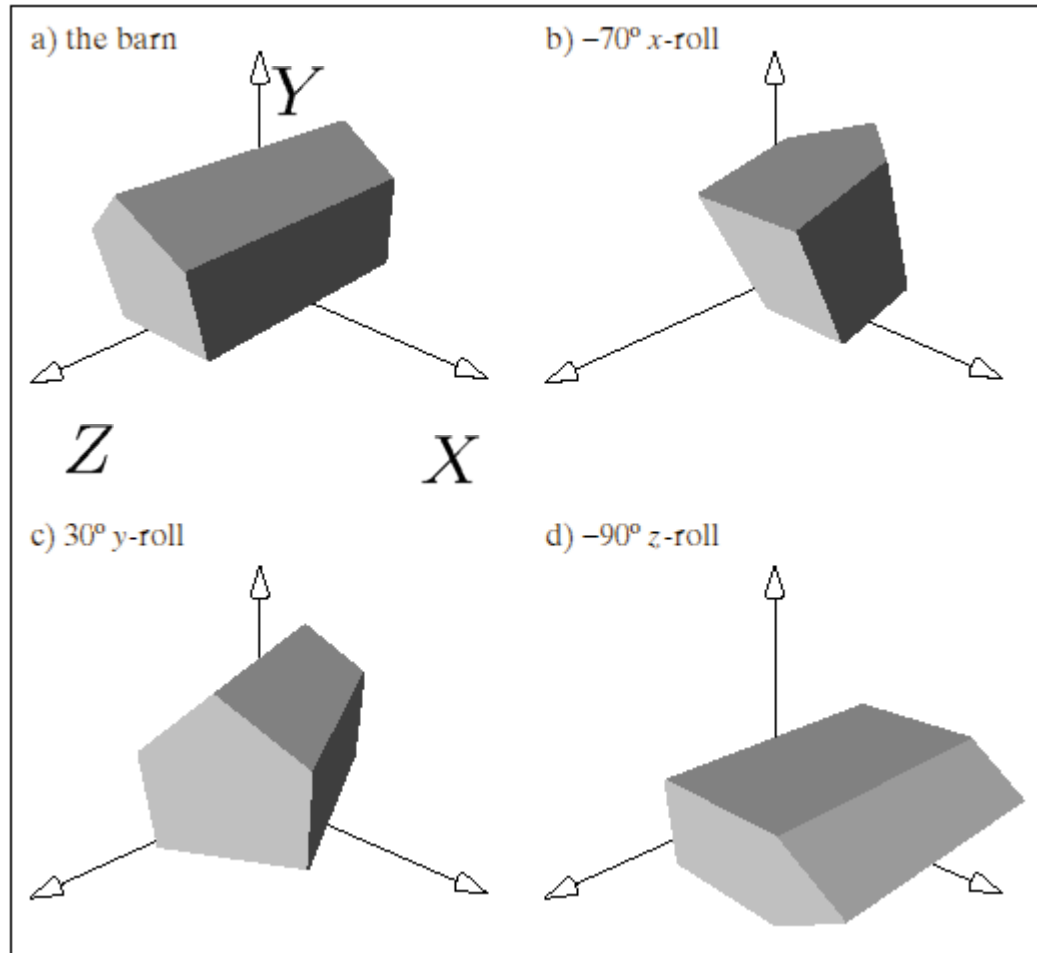
$$\mathbf{T}_{scale} = \mathbf{M}^T = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# 3-D Rotations

- In 2-D, we are always rotating in the plane of the image, but in 3-D the axis of rotation itself is a variable
- Three canonical rotation axes are the coordinate axes X, Y, Z
- These are sometimes referred to in aviation terms: pitch, yaw or heading, and roll, respectively



# Examples: 3-D Rotations

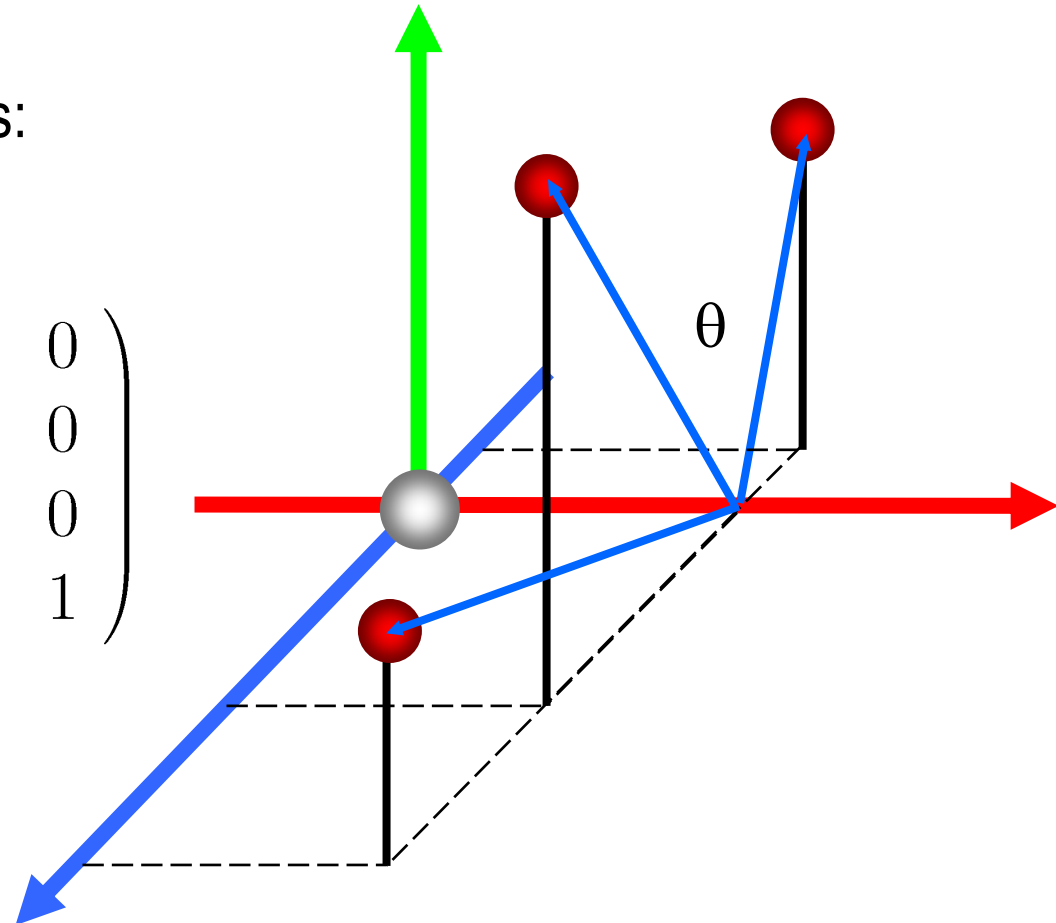


The term “roll” is also sometimes used generically, as a synonym for “rotation”

# 3-D Rotation Matrices

- Similar form to 2-D rotation matrices, but with coordinate corresponding to rotation axis held constant
- E.g., a rotation about the X axis:

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

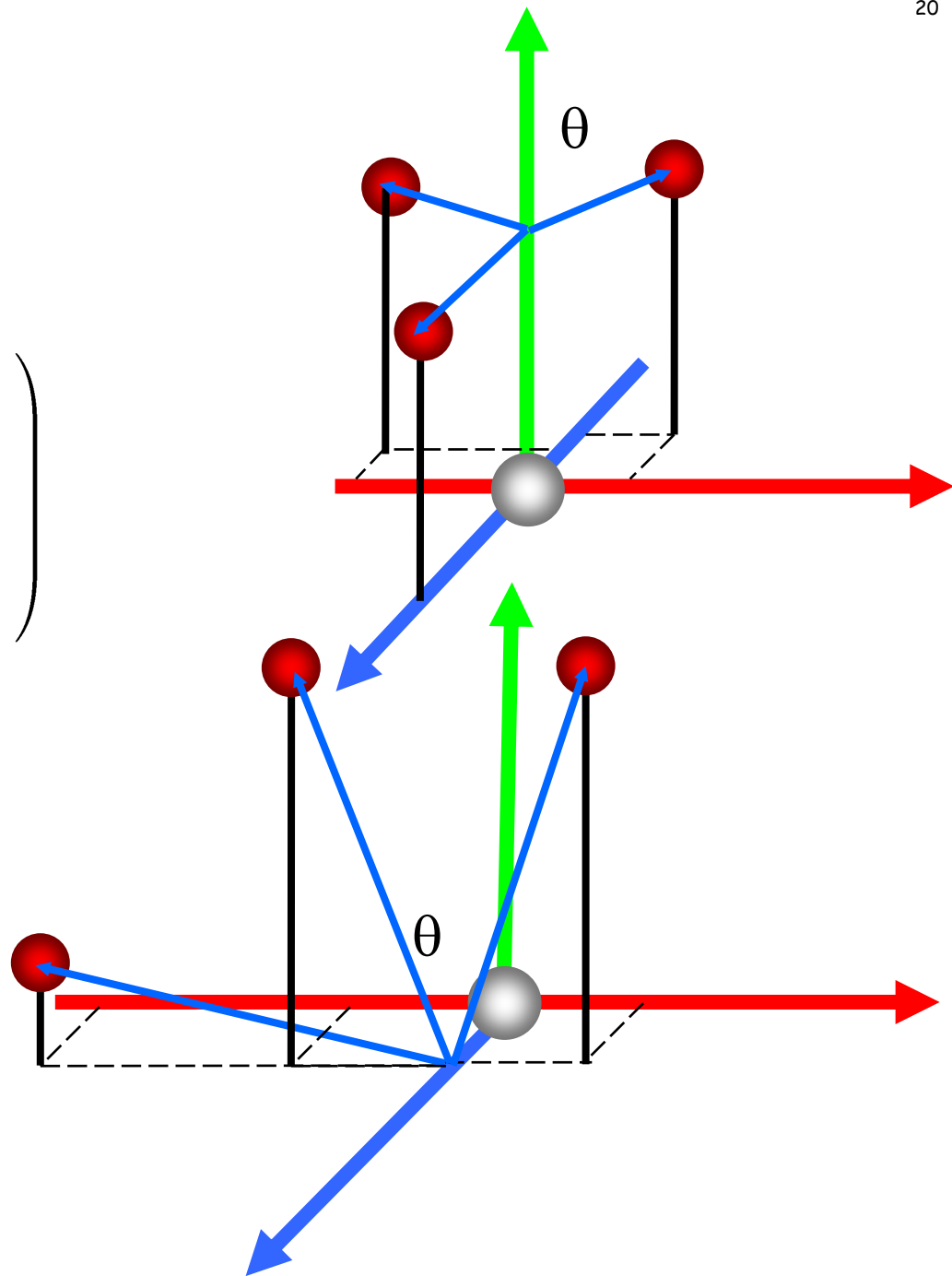


# 3-D Rotation Matrices

- Similarly:

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

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



# 3-D Rigid Transformations

- Combination of rotation and translation without scaling
- “Moves” an object from one 3-D pose to another

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & \Delta x \\ r_{21} & r_{22} & r_{23} & \Delta y \\ r_{31} & r_{32} & r_{33} & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

**T**
**R**
**M**

# Homogeneous coordinates in 3D

The translation and scaling are very similar in 3D:

Point

$$\vec{p} \equiv \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Translation

$$T = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T \cdot \vec{x} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{bmatrix} = \vec{x} + \vec{d}$$

Scaling

$$T = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T \cdot \vec{p} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} S_x \cdot x \\ S_y \cdot y \\ S_z \cdot z \\ 1 \end{bmatrix} = S \cdot \vec{p}$$

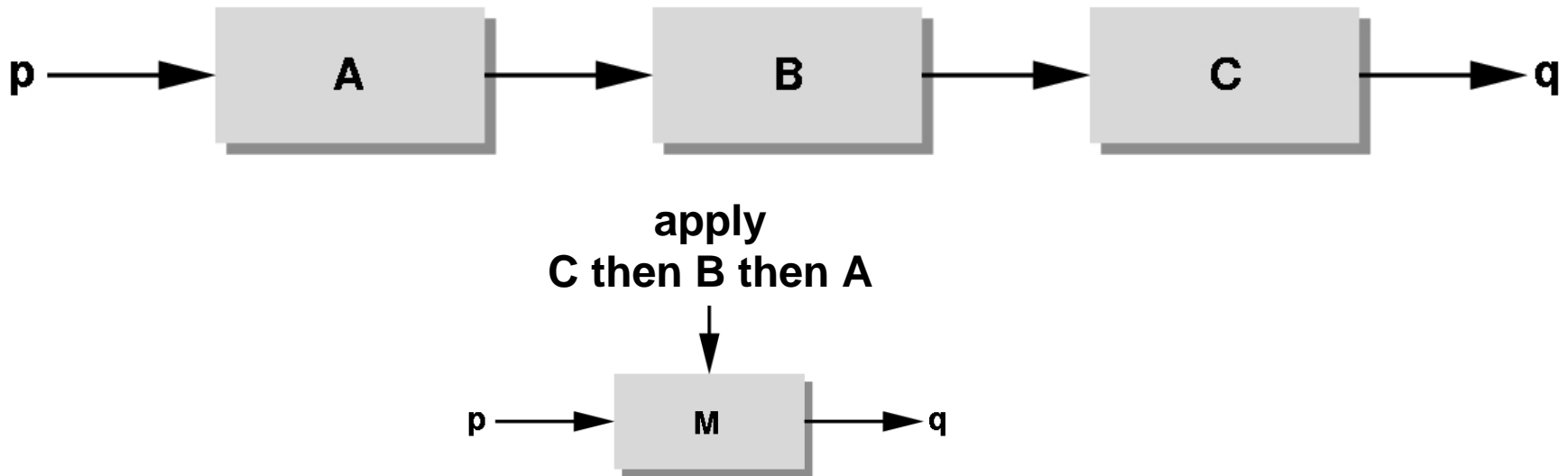
# Properties of Transformations

- Rotation and translation
  - angles and distances are preserved
  - Unit cube is always unit cube
  - *Rigid-Body* transformations.
- Rotation, translation and scale.
  - angles and distances not preserved.
  - But parallel lines are.
  - *Affine* transformations.

# Concatenation of Transformations

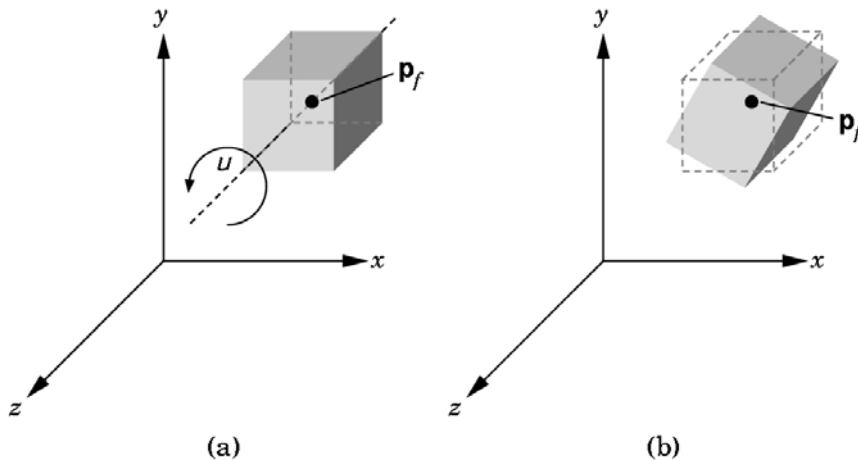
$$\mathbf{T}_1(\mathbf{T}_2(\mathbf{p})) = (\mathbf{T}_1\mathbf{T}_2)(\mathbf{p})$$

$$\mathbf{T}_1(\mathbf{T}_2(\mathbf{T}_3(\mathbf{p}))) = \mathbf{T}_1(\mathbf{T}_2\mathbf{T}_3)(\mathbf{p}) = (\mathbf{T}_1\mathbf{T}_2\mathbf{T}_3)(\mathbf{p})$$



# Rotation About a Fixed Point

- Strategy:
  - translate the cube to the origin,
  - rotate it about the origin,
  - translate it back again at  $p_f$



```

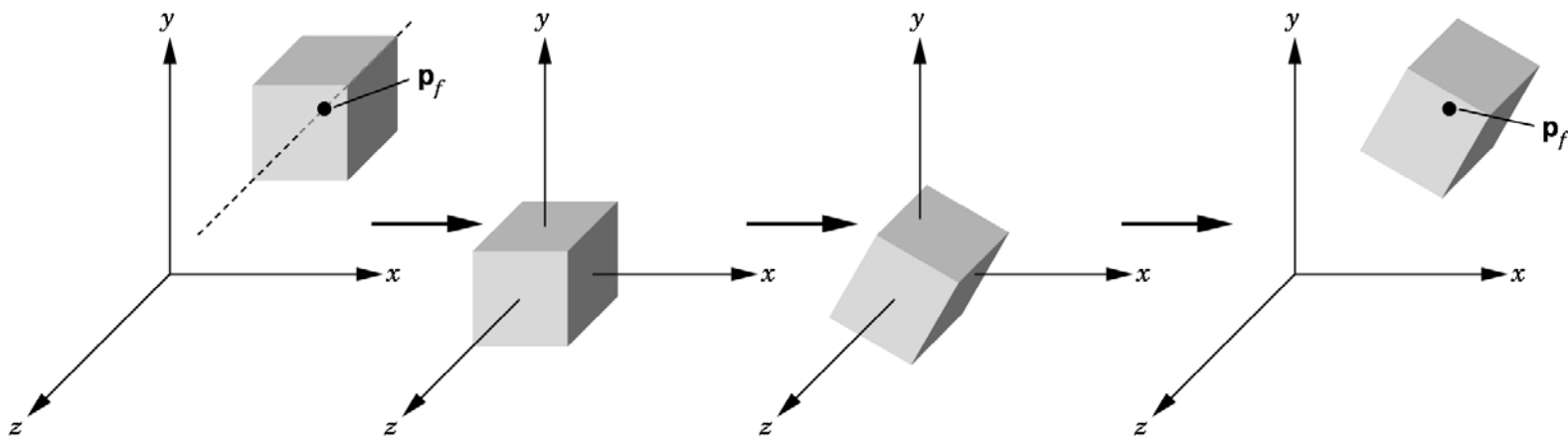
glMatrixMode (GL_MODELVIEW)
glLoadIdentity();
glTranslatef( p_f );
glRotatef( theta );
glTranslatef(- p_f );
DrawCube();

```

$$M = T(p_f)R_z(\theta)T(-p_f)$$

# Sequence of transformations

$$M = T(p_f)R_z(\theta)T(-p_f)$$



# Rotation About a Fixed Point in OpenGL

```
glMatrixMode(GL_MODELVIEW)  
glLoadIdentity();
```

- *Move the fixed point to the origin:*

```
glTranslatef(4.0, 5.0, 6.0);
```

- *Rotate about the origin:*

```
glRotatef(45.0, 1.0, 2.0, 3.0);
```

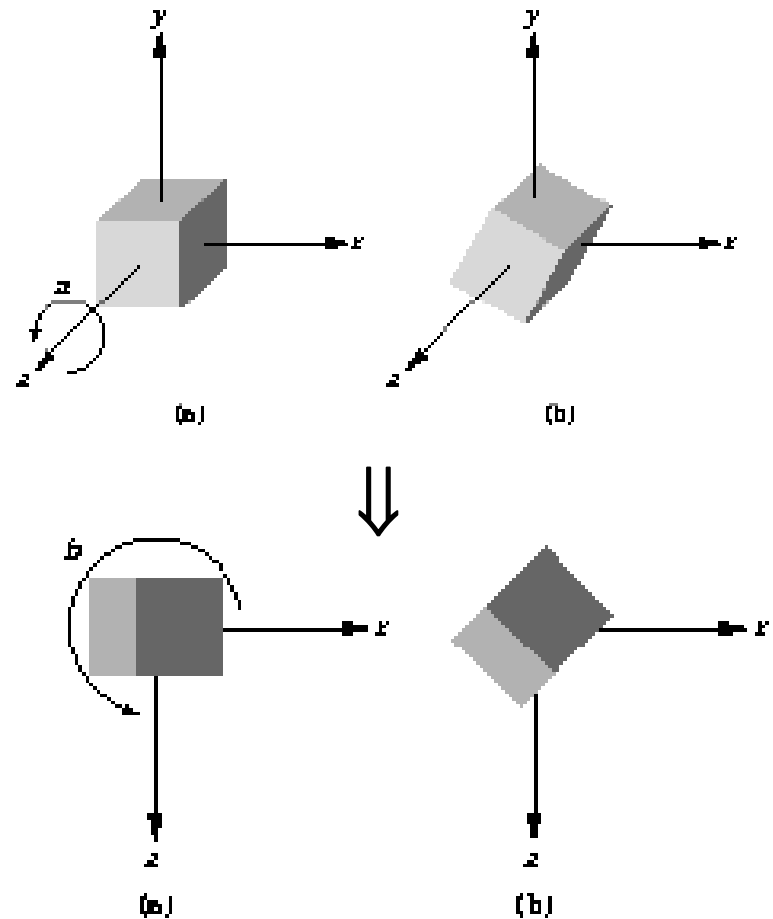
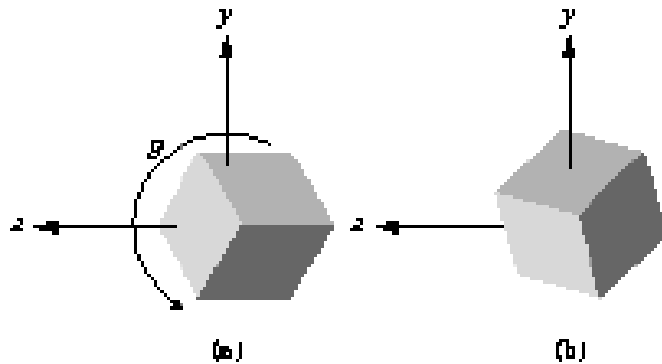
- *Move the fixed point back again:*

```
glTranslatef(-4.0, -5.0, -6.0);
```

# General Rotation

- Any rotation about the origin is equivalent to 3 successive rotations about 3 axes,  $x$ ,  $y$ , and  $z$ .

$$R = R_x R_y R_z$$



# Rotation About an Arbitrary Axis Passing Through the Origin:

- We derive the Euler-Rodrigues matrix  $R(\theta)$  for rotation about an arbitrary axis passing through the origin in the unit direction  $u=(a,b,c)$  by angle  $\theta$

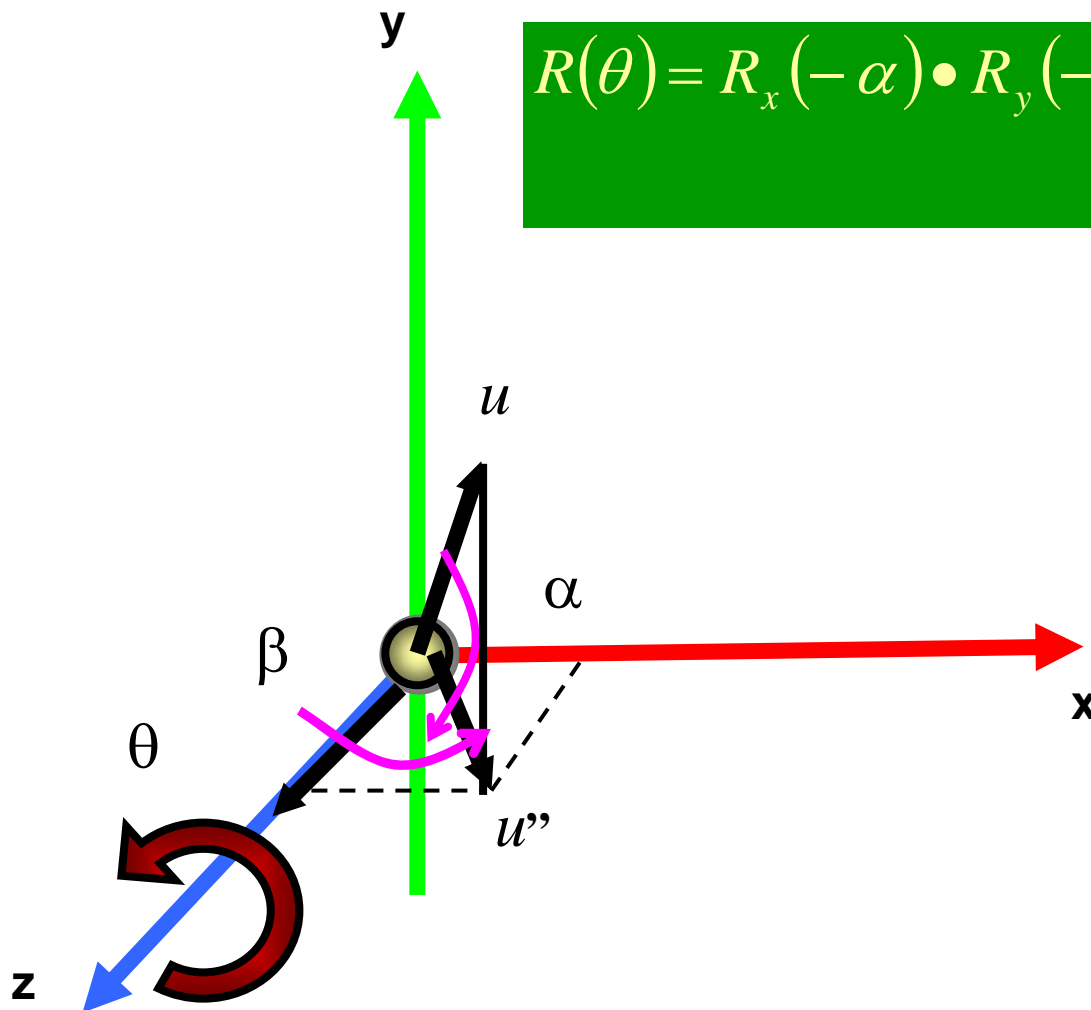
$$R(\theta) = \begin{bmatrix} a^2 + (1 - a^2) \cos \theta & ab(1 - \cos \theta) - c \sin \theta & ac(1 - \cos \theta) + b \sin \theta \\ ab(1 - \cos \theta) + c \sin \theta & b^2 + (1 - b^2) \cos \theta & bc(1 - \cos \theta) - a \sin \theta \\ ac(1 - \cos \theta) - b \sin \theta & bc(1 - \cos \theta) + a \sin \theta & c^2 + (1 - c^2) \cos \theta \end{bmatrix}$$

where

$$a^2 + b^2 + c^2 = 1$$

# Rotation About an Arbitrary Axis Passing Through the Origin

$$R(\theta) = R_x(-\alpha) \cdot R_y(-\beta) \cdot R_z(\theta) \cdot R_y(\beta) \cdot R_x(\alpha)$$



## Steps:

1. Normalize vector  $u=(a,b,c)$
2. Compute  $\alpha$  (counterclockwise rotation angle about  $x$ -axis bringing  $u$  to  $u'' = (a,0,d)$  on  $xz$  plane)
3. Compute  $\beta$  (counterclockwise rotation angle about  $y$ -axis bringing unit vector  $u''$  to  $yz$  plane)
4. Create rotation matrix

# Details

Let  $j=(0,1,0)$ ,  $k=(0,0,1)$ ,  $u'=(0,b,c)$ -projection of  $u$  on  $yz$ -plane,  $u''=(a,0,d)$ - $u$  vector in  $xz$ -plane after counterclockwise rotation about  $x$ -axis by angle  $\alpha$

$$\cos \alpha = \frac{u' \cdot k}{|u'| \cdot |k|} = \frac{c}{\sqrt{b^2 + c^2}} = \frac{c}{d}$$

$$u' \times k = |u'| |k| \sin \alpha \cdot i = d \sin \alpha \cdot i \quad \text{and} \quad u' \times k = \begin{bmatrix} i & j & k \\ 0 & b & c \\ 0 & 0 & 1 \end{bmatrix} = b \cdot i$$

Hence  $\sin \alpha = \frac{b}{d}$

and

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & \frac{-b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Details

$$\cos \beta = \frac{u'' \cdot k}{|u''| \cdot |k|} = d$$

$$u'' \times k = |u''| |k| \sin \beta \cdot j = \sin \beta \cdot j \quad \text{and} \quad u'' \times k = \begin{bmatrix} i & j & k \\ a & 0 & d \\ 0 & 0 & 1 \end{bmatrix} = -a \cdot j$$

Hence  $\sin \beta = -a$

and

$$R_y(\beta) = \begin{bmatrix} \cos \beta & 0 & \sin \beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Computing the Rotation Matrix about an arbitrary axis:

- 1. Normalize  $u$
- 2. Compute  $R_x(\alpha)$
- 3. Compute  $R_y(\beta)$
- 4. Generate Rotation Matrix

$$R(\theta) = R_x(-\alpha) \cdot R_y(-\beta) \cdot R_z(\theta) \cdot R_y(\beta) \cdot R_x(\alpha)$$

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & \frac{-b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$u = (a, b, c)$$

$$d = \sqrt{b^2 + c^2}$$

$$u_z = (0, 0, 1)$$

$$u' = (0, b, c)$$

$$\cos \alpha = \frac{u' \cdot u_z}{|u'| |u_z|} = \frac{c}{d}$$

$$\sin \alpha = \frac{b}{d}$$

$$\cos \beta = d$$

$$\sin \beta = -a$$

# Rotation Matrix About an Arbitrary Axis:

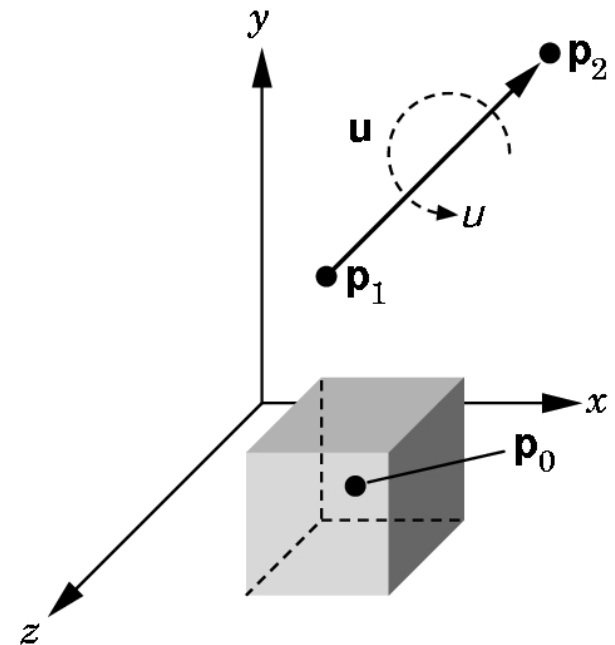
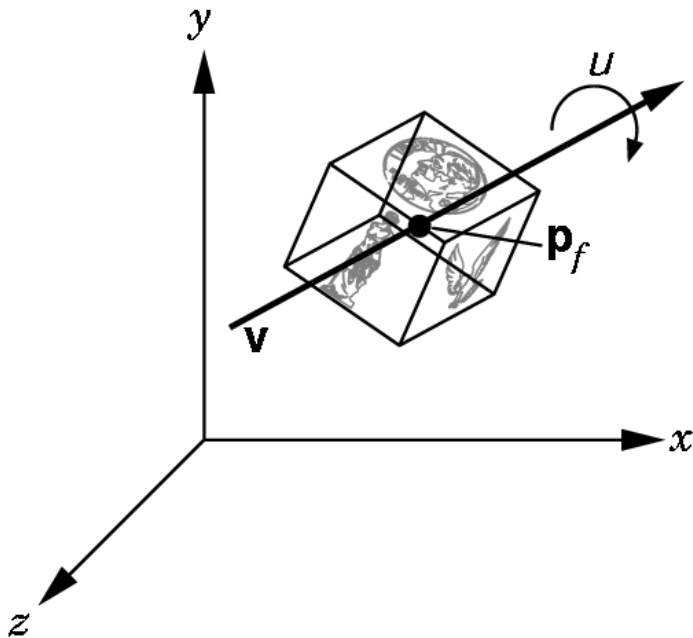
- The final matrix for rotation about an arbitrary axis passing through the origin in the direction  $u=(a,b,c)$  by angle  $\theta$  is given by Rodrigues Formula

$$R(\theta) = \begin{bmatrix} a^2 + (1 - a^2) \cos \theta & ab(1 - \cos \theta) - c \sin \theta & ac(1 - \cos \theta) + b \sin \theta \\ ab(1 - \cos \theta) + c \sin \theta & b^2 + (1 - b^2) \cos \theta & bc(1 - \cos \theta) - a \sin \theta \\ ac(1 - \cos \theta) - b \sin \theta & bc(1 - \cos \theta) + a \sin \theta & c^2 + (1 - c^2) \cos \theta \end{bmatrix}$$

where  $a^2 + b^2 + c^2 = 1$  .

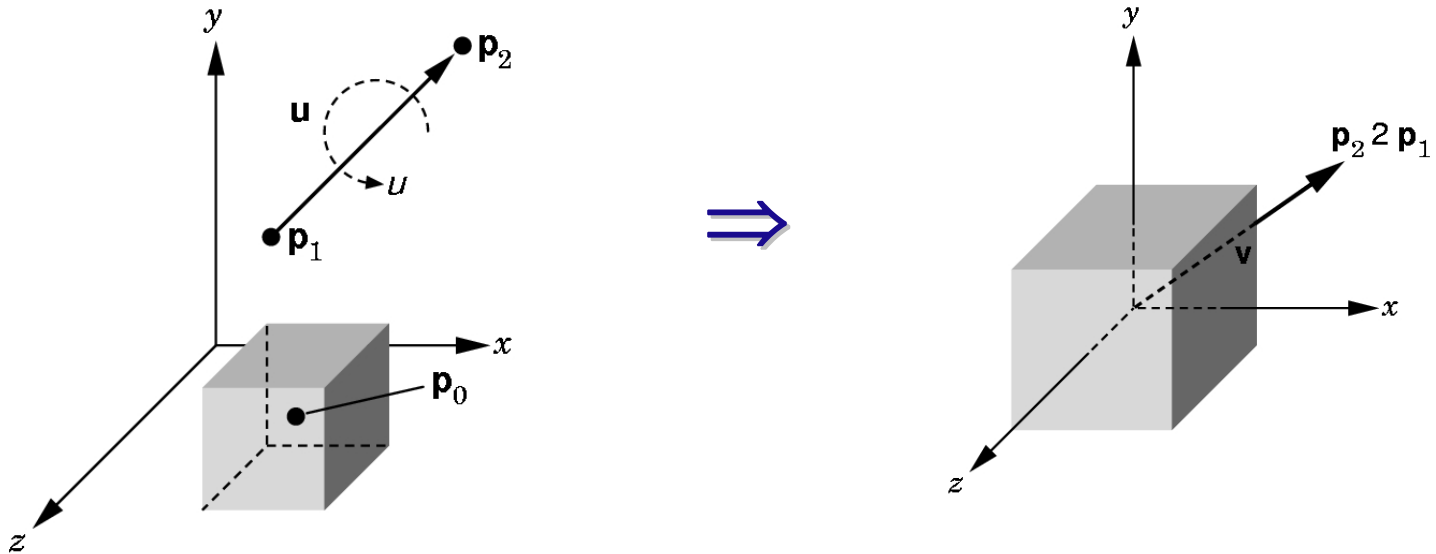
# Rotation About an Arbitrary Axis

- Given: a fixed point  $p_0 = (p_1, p_2, p_3)$ , a unit vector  $v = (\alpha_x, \alpha_y, \alpha_z)$  about which we rotate, and an angle of rotation  $\theta$ .
- Find: a sequence of rotations.



# Rotation (cont'd)

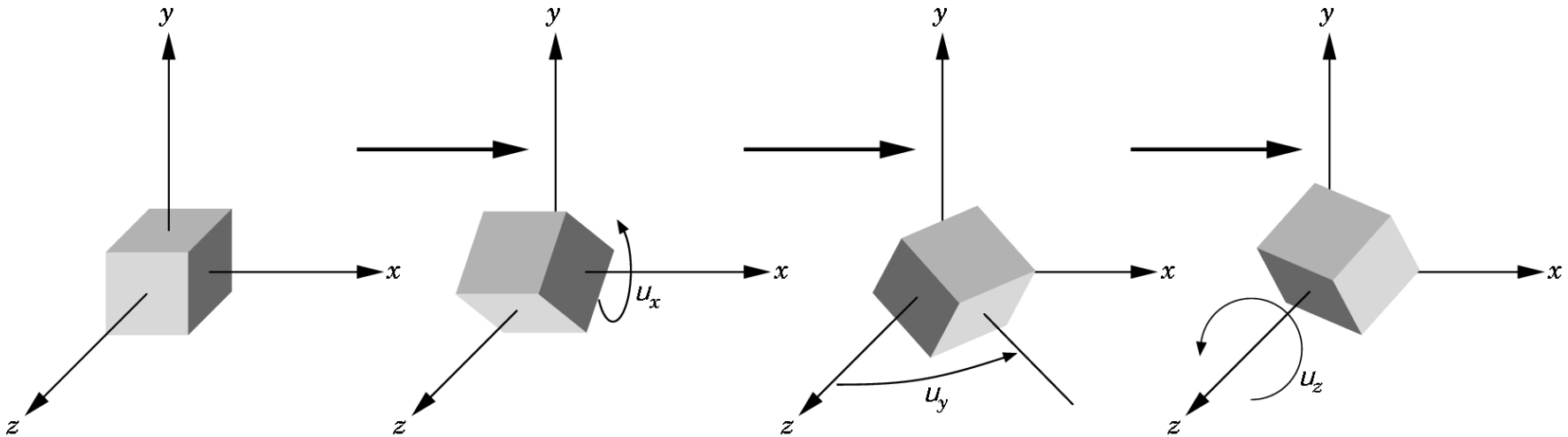
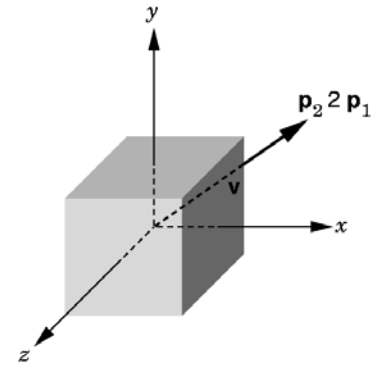
- Strategy:
  - **translate** the fixed point  $p_0$  to the origin,
  - **align** the axis of rotation with the z axis,
  - complete the rotation of the cube,
  - **undo** the rotation of the rotation axis,
  - **translate** the fixed point back again.



# Rotation (cont'd)

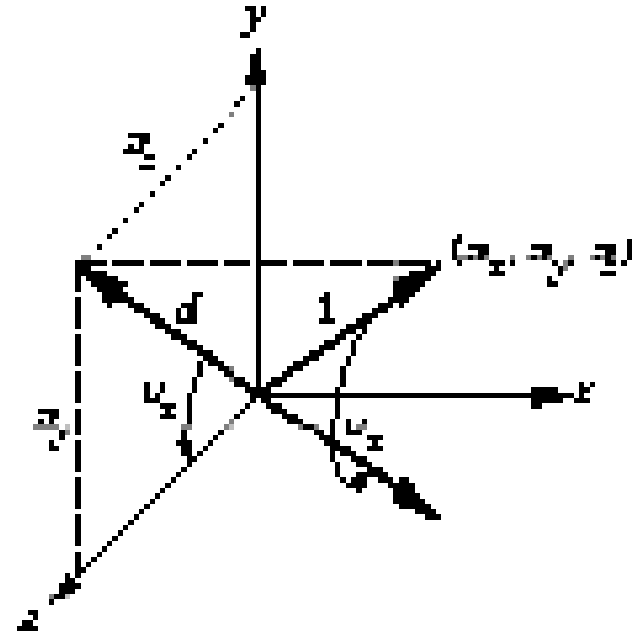
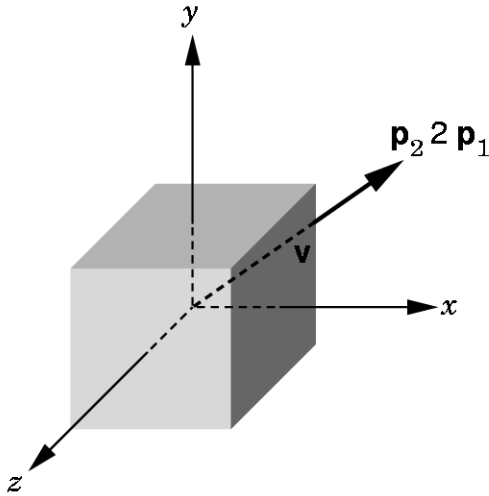
- **Rotation matrix:**

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



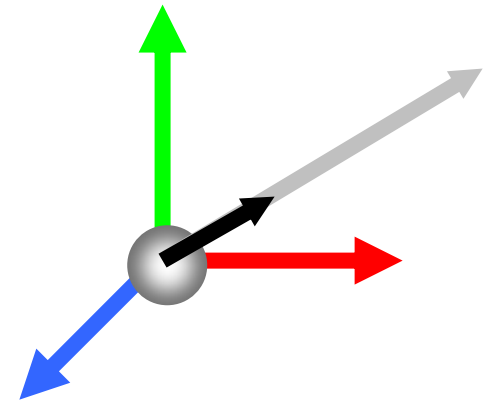
# Rotation (cont'd)

$$u = p_2 - p_1$$



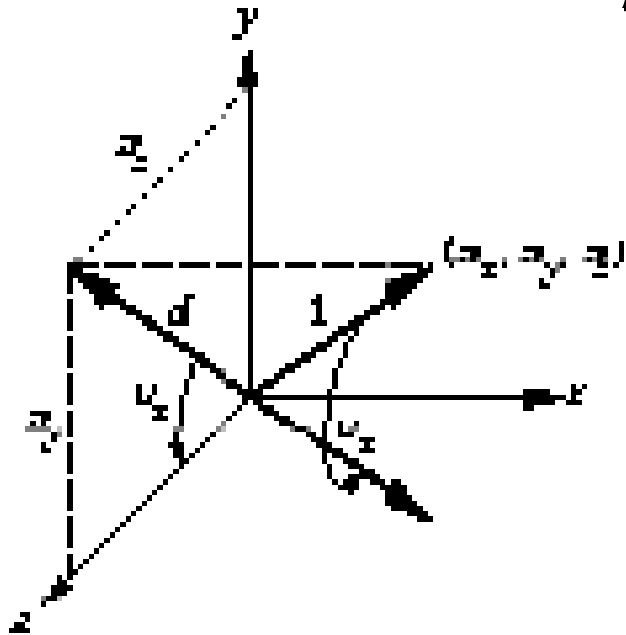
$$v = \frac{u}{\|u\|} = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \end{bmatrix}$$

$$\alpha_x^2 + \alpha_y^2 + \alpha_z^2 = 1$$



# Rotation (cont'd)

- Rotate about  $x$  by  $\theta_x$ :

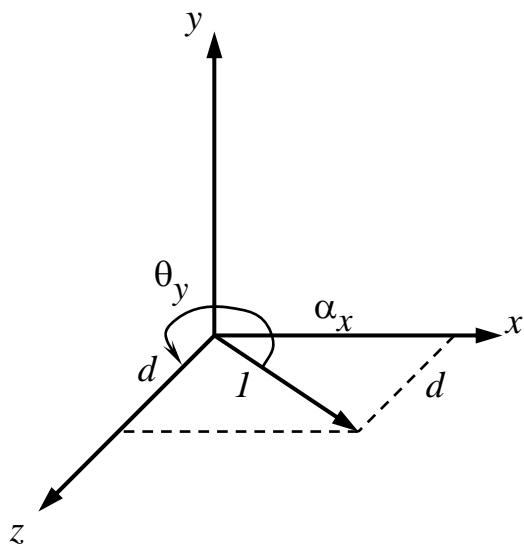


$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x & 0 \\ 0 & \sin \theta_x & \cos \theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_z / d & -\alpha_y / d & 0 \\ 0 & \alpha_y / d & \alpha_z / d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Rotation (cont'd)

Rotate about  $y$  by an angle  $\theta_y$



Complete matrix:

$$R_y(\theta_y) = \begin{bmatrix} \cos \theta_y & 0 & \sin \theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

$$M = T(p_0)R_x(-\theta_x)R_y(-\theta_y)\underline{R_z(\theta)}R_x(\theta_x)R_x(\theta_x)T(-p_0)$$

# Rotation (cont'd)

Our final matrix is given by

$$M = \begin{bmatrix} \alpha_x^2 + \cos \theta (1 - \alpha_x^2) & \alpha_x \alpha_y (1 - \cos \theta) - \alpha_z \sin \theta & \alpha_x \alpha_z (1 - \cos \theta) + \alpha_y \sin \theta & -(\alpha_x^2 + \cos \theta (1 - \alpha_x^2))p_1 - (\alpha_x \alpha_y (1 - \cos \theta) - \alpha_z \sin \theta)p_2 - (\alpha_x \alpha_z (1 - \cos \theta) + \alpha_y \sin \theta)p_3 + p_1 \\ \alpha_x \alpha_y (1 - \cos \theta) + \alpha_z \sin \theta & \alpha_y^2 + \cos \theta (1 - \alpha_y^2) & \alpha_y \alpha_z (1 - \cos \theta) - \alpha_x \sin \theta & -(\alpha_x \alpha_y (1 - \cos \theta) + \alpha_z \sin \theta)p_1 - (\alpha_y^2 + \cos \theta (1 - \alpha_y^2))p_2 - (\alpha_y \alpha_z (1 - \cos \theta) - \alpha_x \sin \theta)p_3 + p_2 \\ \alpha_x \alpha_z (1 - \cos \theta) - \alpha_y \sin \theta & \alpha_y \alpha_z (1 - \cos \theta) + \alpha_x \sin \theta & \alpha_z^2 + \cos \theta (1 - \alpha_z^2) & -(\alpha_x \alpha_z (1 - \cos \theta) - \alpha_y \sin \theta)p_1 - (\alpha_y \alpha_z (1 - \cos \theta) + \alpha_x \sin \theta)p_2 - (\alpha_z^2 + \cos \theta (1 - \alpha_z^2))p_3 + p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Quaternions

- The ring of quaternions =  $\{w + xi + yj + zk\}$  (Extension of imaginary numbers.)
  - $i^2 = j^2 = k^2 = -1$
  - $ij = k, ji = -k; \quad jk = i, kj = -i; \quad ki = j, ik = -j$
- We will denote  $q = (w, v) = w + xi + yj + zk$  where  $v = (x, y, z)$

Operations:

- $q_1 + q_2 = (w_1, v_1) + (w_2, v_2) = (w_1 + w_2, v_1 + v_2)$
- $q_1 \cdot q_2 = (w_1, v_1) \cdot (w_2, v_2) = (w_1 w_2 - \langle v_1, v_2 \rangle, w_1 v_2 + w_2 v_1 + v_1 \times v_2)$

Note that  $q_1 \cdot q_2$  is not equal to  $q_2 \cdot q_1$

- $\bar{q} = (w, -v) = w - xi - yj - zk$
- $\langle q_1, q_2 \rangle = q_1 \cdot \bar{q}_2 = w_1 w_2 + \langle v_1, v_2 \rangle$
- $\|q\|^2 = \langle q, q \rangle = w^2 + x^2 + y^2 + z^2$
- $q^{-1} = \bar{q} / \|q\|^2$  (inverse of  $q$ )

# Visualizing a Unit Quaternion

## → Rotation in 4D Space

$$\|q\| = \text{Norm}(q) = \sqrt{w^2 + x^2 + y^2 + z^2}$$

$$q = q / \|q\|$$

$$q = [w, v], \quad w = \text{scalar}, \quad v = (a, b, c),$$

Cosine of  $\frac{1}{2}$  rotation angle

Arbitrary axis of rotation  
passing through the origin

- $q$  forms a sphere of unit length in the 4D space
- You can get a 180 degree rotation of a quaternion by simply inverting the scalar ( $w$ ) component

# Rotating with quaternions

- Represent  $P$  as a vector  $[0, P]$ .
- Rotating about axis  $V = (u_x, u_y, u_z)$ ,  $\|V\| = 1$ , by angle  $\theta$ :

$$\begin{aligned}
 R_{n,\theta}(P) &= q \cdot [0, P] \cdot q^{-1} \\
 q &= (\cos(\theta/2), \sin(\theta/2) * V) \\
 &= (w, a, b, c)
 \end{aligned}$$

## Conversion From Quaternions $\rightarrow$ to Quaternion Matrix

Write out  $[0, P'] = q[0, P]q^{-1}$  and expand it like a matrix equation for the  $i, j$ , and  $k$  terms

$$\text{Matrix} = \begin{bmatrix} 1 - 2b^2 - 2c^2 & 2ab - 2cw & 2ac + 2bw \\ 2ab + 2cw & 1 - 2a^2 - 2c^2 & 2bc - 2aw \\ 2ac - 2bw & 2bc + 2aw & 1 - 2a^2 - 2b^2 \end{bmatrix}$$

# Some facts about quaternions:

- Only a unit quaternion encodes a rotation - normalize by dividing by  $\sqrt{w^2 + a^2 + b^2 + c^2}$
- Rotating by  $q$  and by  $-q$  has the same effect.
- “Rotating” by  $q$  such that  $\|q\| = \alpha$  has a scaling effect.
- So, we consider the unit quaternions only  $\|q\| = 1$
- These quaternions live on the unit sphere in four dimensions...
- So, we have 3 degrees of freedom for our rotation quaternion (just like with  $\nu, \theta$  rotation: 1 degree for  $\theta$  + 2 degrees for  $\nu$  = 3 degrees of freedom.)

# Practice

- $q_1 = [1.57 \ 0 \ 0 \ 1]$ ,  $q_2 = [.78 \ 0 \ 1 \ 0]$
- $q_1 = [.25 \ 3 \ 4 \ 4]$ ,  $q_2 = [.20 \ 2 \ 9 \ 9]$
- $q_1 + q_2?$

$$q_1 + q_2 = w_1 + w_2 + i(x_1 + x_2) + j(y_1 + y_2) + k(z_1 + z_2)$$

$$q_1 + q_2 = [w_1 + w_2, v_1 + v_2]$$

- $q_1 * q_2?$

$$q_1 * q_2$$

$$= (w_1 + i x_1 + j y_1 + k z_1) * (w_2 + i x_2 + j y_2 + k z_2)$$

$$= w_1 w_2 + i w_1 x_2 + j w_1 y_2 + k w_1 z_2 +$$

$$- x_1 x_2 + i w_2 x_1 - j x_1 z_2 + k x_1 y_2 +$$

$$- y_1 y_2 + i y_1 z_2 + j w_2 y_1 - k x_2 y_1 +$$

$$- z_1 z_2 - i y_2 z_1 + j x_2 z_1 + k w_2 z_1$$

$$[w_1 w_2 - \langle v_1, v_2 \rangle, w_1 v_2 + w_2 v_1 + v_1 \times v_2]$$

Find the inverse of  $q = [0, 6, 8, 0]$

$$q^{-1} = [w, -v] / \|q\|^2$$

$$q^{-1} = [0, -6, -8, -0] / (0 + 36 + 64 + 0) = [0, -.06, -.08, 0]$$

To check,  $q * q^{-1}$  should give us the identity quaternion.

$$q * q^{-1}$$

$$= [0 - (6 * -.06 + 8 * -.08 + 0), [0, 0, 0] + [0, 0, 0] + [0, 0, 0]]$$

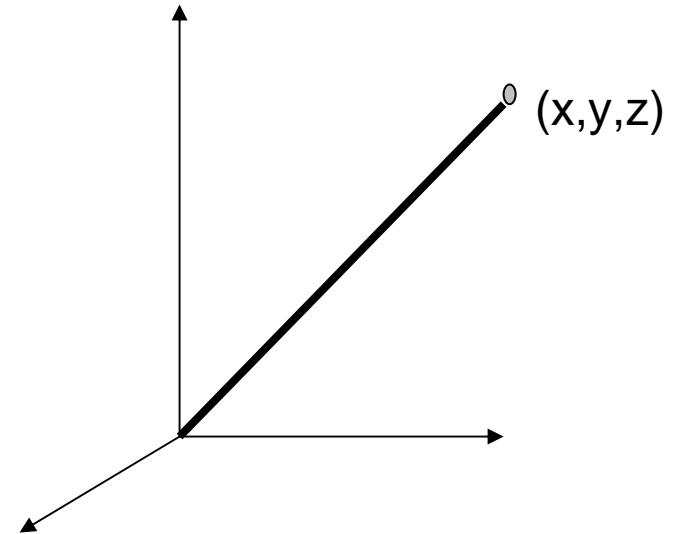
$$= [1, 0, 0, 0]$$

# Rotation about an arbitrary axis

- Let  $q$  be a unit quaternion, i.e.,  $\|q\| = 1$
- $q = [w, x, y, z]$  describes a rotation through an angle  $\theta$  where  $w = \cos(\theta/2)$  and  $[x, y, z]$  is the axis of rotation.

To create a unit quaternion that represents a rotation of  $\theta$  degrees about an arbitrary axis  $v = [x, y, z]$ .

$$q = [\cos(\theta/2), \sin(\theta/2) * v / \|v\|]$$



Find  $q$  such that  $q$  describes a rotation of 60 degrees about  $v = [3,4,0]$ .

- Remember, we need a unit quaternion.
- To get this:
  - Compute  $\cos(60/2) = \cos 30 = \sqrt{3}/2$
  - Compute  $\sin(60/2) = \sin 30 = 1/2$
  - Compute  $v' = v/\|v\| = [3,4,0]/5 = [3/5, 4/5, 0]$
- $\Rightarrow q = [\sqrt{3}/2, 1/2*[3/5, 4/5, 0]] = [\sqrt{3}/2, 3/10, 4/10, 0]$
- Check:  $\|q\|^2 = 3/4 + 9/100 + 16/100 = (75+9+16)/100 = 1$

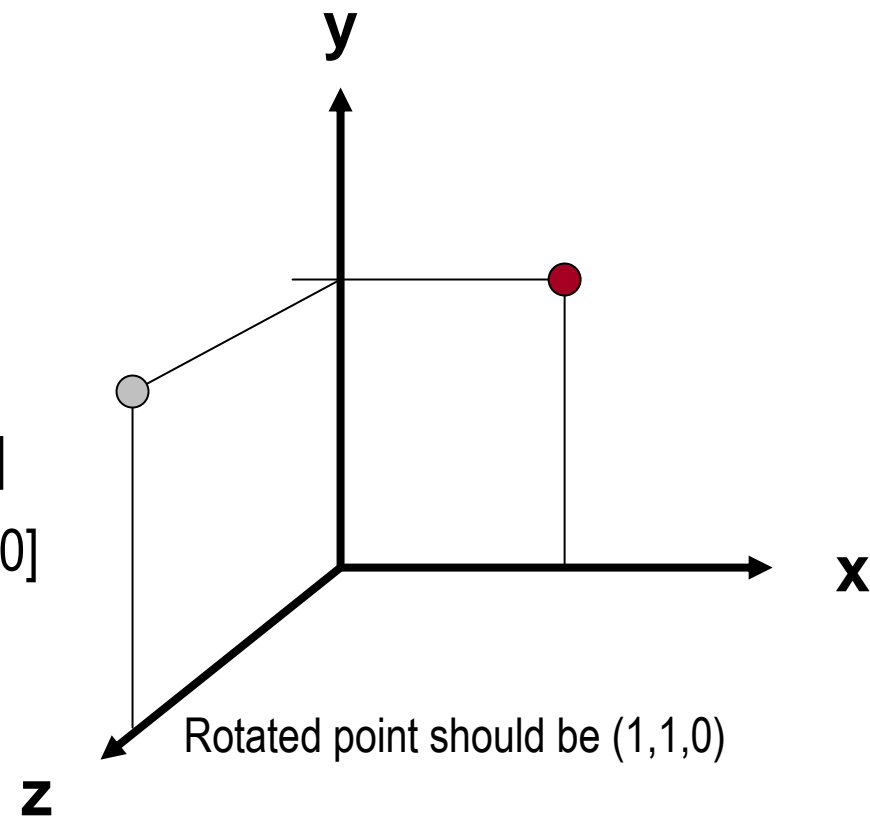
To rotate a point:  $P = (p_x, p_y, p_z)$

- $P_{\text{rotated}} = q[0, P]q^{-1}$
- Remember this is quaternion multiplication not matrix multiplication!

# Example: rotate $P = (0, 1, 1)$ 90 degrees about the vertical $y$ -axis.

- First, compute  $q$ 
  - $q = [\cos(90/2), \sin(90/2)[0, 1, 0]]$   
 $= [\cos 45, 0, \sin 45, 0]$   
 $= [\sqrt{2}/2, 0, \sqrt{2}/2, 0]$
- Next compute  $q^{-1}$ 
  - $q^{-1} = [\sqrt{2}/2, 0, -\sqrt{2}/2, 0]$  *Why?*
- $q[0, P] = [-\sqrt{2}/2, \sqrt{2}/2, \sqrt{2}/2, \sqrt{2}/2]$
- Compute  $q[0, P]q^{-1}$   
 $= [\sqrt{2}/2, 0, \sqrt{2}/2, 0][0, 0, 1, 1][\sqrt{2}/2, 0, -\sqrt{2}/2, 0]$   
 $= [-\sqrt{2}/2, \sqrt{2}/2, \sqrt{2}/2, \sqrt{2}/2][\sqrt{2}/2, 0, -\sqrt{2}/2, 0]$   
 $= [0, 1, 1, 0]$

Transformed point is  $[1, 1, 0]$ .



# Matrices in OpenGL

- Consider a transformation:  $(T^1 T^2 \dots T^n) \circ \begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix}$

(multiply object coordinate by  $T^n$  first, then  $T^{n-1} \dots$  until  $T^1$ ), To build the transformation matrix:  $(T^1 T^2 \dots T^n)$  we shall multiply identity matrix by  $T^1$  then  $T^2 \dots$  until  $T^n$ , as:  $I \circ T^1 \circ T^2 \circ \dots \circ T^n$ .

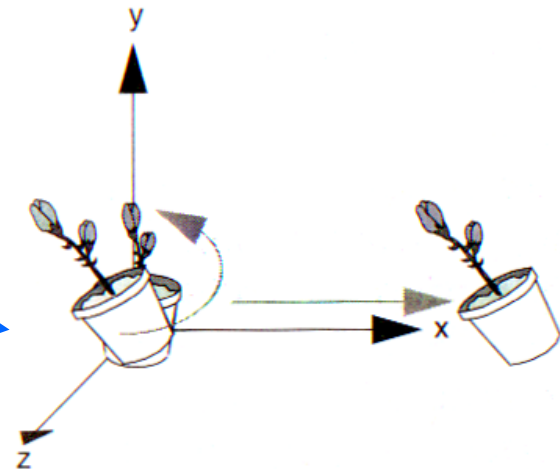
- Thus, the order of issuing commands shall be inversed.

```
//Rotate then translate:
```

```
glTranslatef( 1,0,0 );
```

```
glRotatef(45.0, 0,0,1 );
```

```
drawObject();
```



# Predefined Postmultiplier Operator in OpenGL

**glTranslatef(dx, dy, dz);**

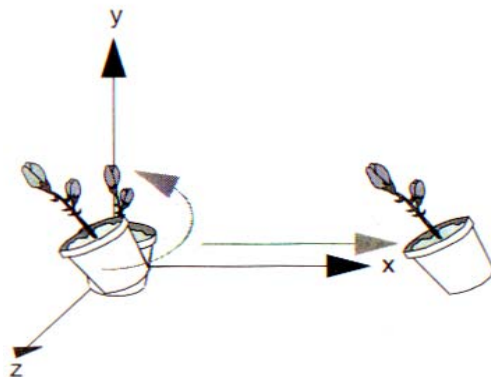
Multiplies current matrix with translation matrix.  $dx$ ,  $dy$ , and  $dz$  are translations along  $x$ ,  $y$ , and  $z$  axes.

**glRotatef(angle, x, y, z);**

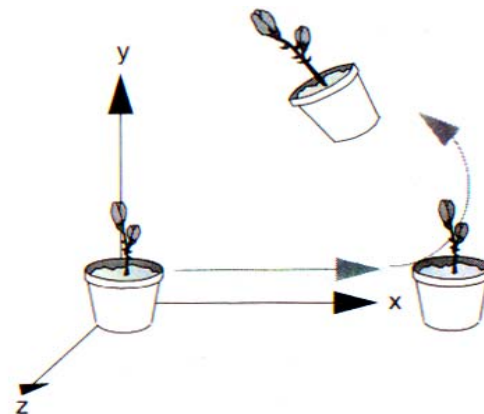
Multiplies current matrix with rotation about the line from the origin through the point  $(x, y, z)$  by angle. Right hand rule applies.

**glScalef(sx, sy, sz);**

Multiplies current matrix with scaling matrix.  $sx$ ,  $sy$ ,  $sz$  are the scale factors along the  $x$ ,  $y$ , and  $z$  axes.



Rotate then Translate

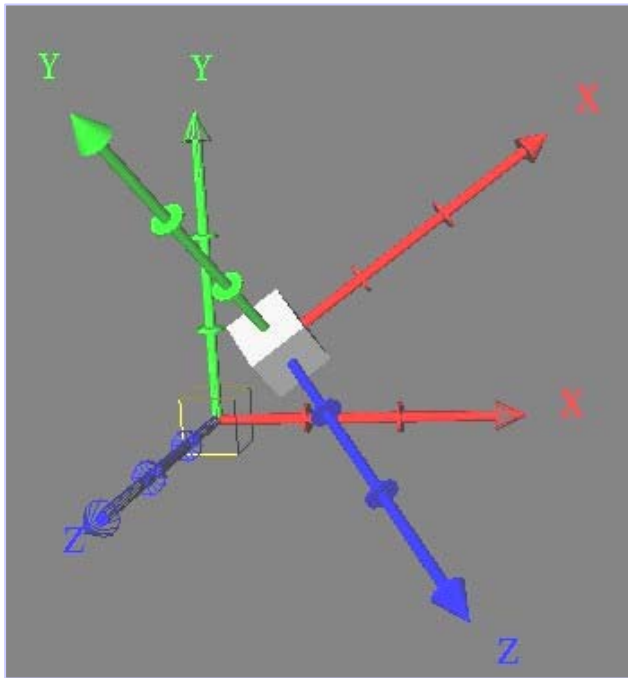


Translate then Rotate

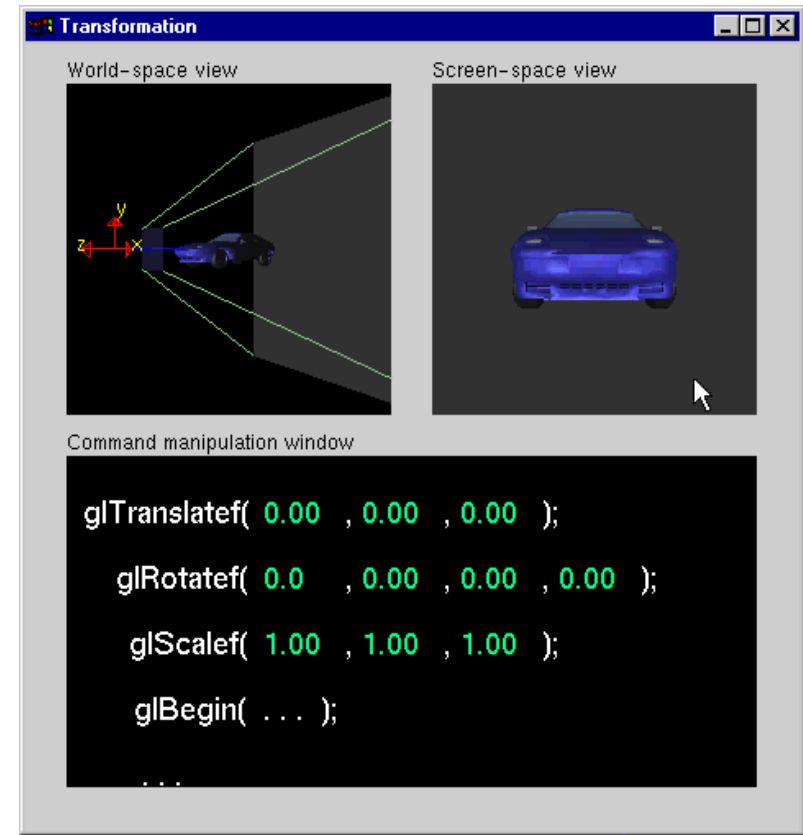
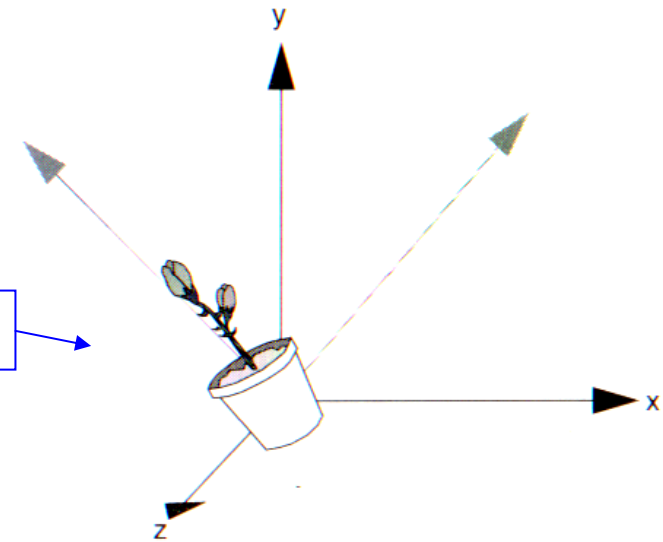
# Rotation

`glRotatef( angle, x, y, z );`

- Rotate coordinate system about an axis in space

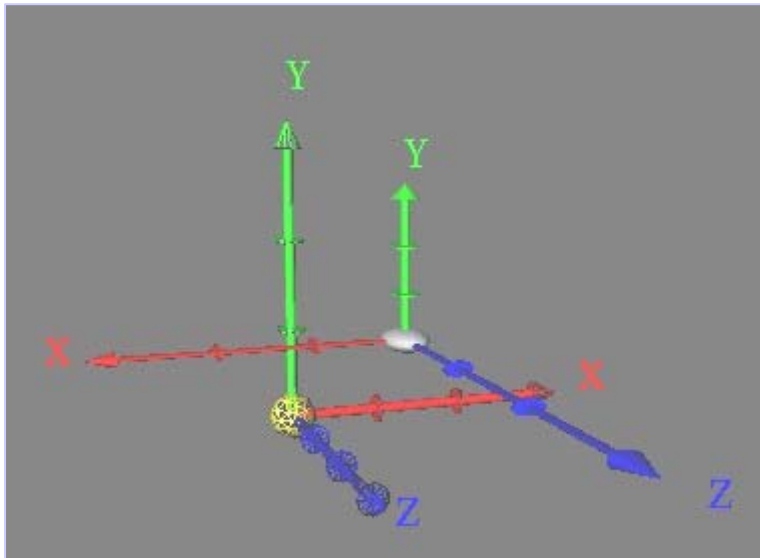


`glRotatef( 45.0, 0, 0, 1);`



# Scale

- Stretch, mirror or decimate a coordinate direction



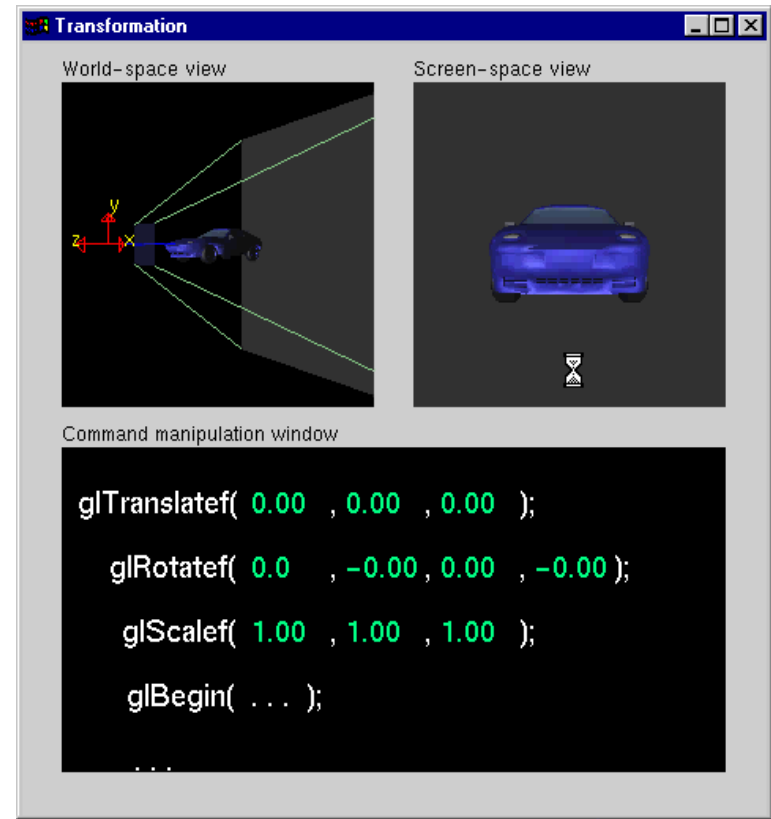
$1 < s$	stretch
$0 < s \leq 1$	shrink
$0$	decimate
$-1 \leq s < 0$	reflect/shrink
$s < -1$	reflect/stretch

Note, there's a translation applied here to make things easier to see

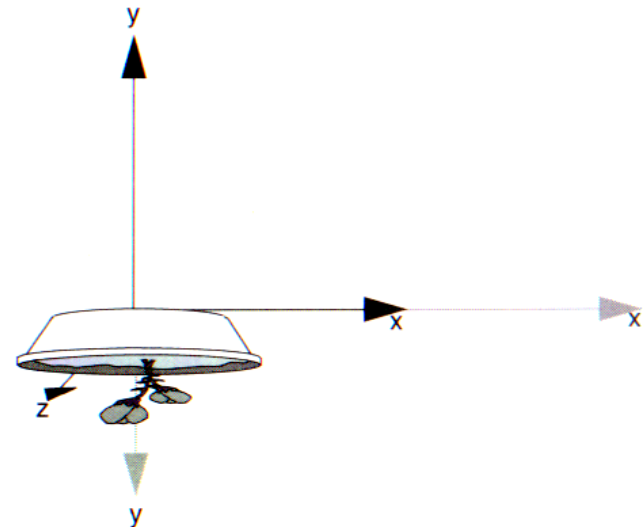
# glScale[fd]()

`glScalef( $s_x$ ,  $s_y$ ,  $s_z$ );`

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



`glScalef( 2.0, -0.5, 1.0 );`



# Rotations

$$\mathbf{glRotate}^*(\alpha, 1, 0, 0): \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

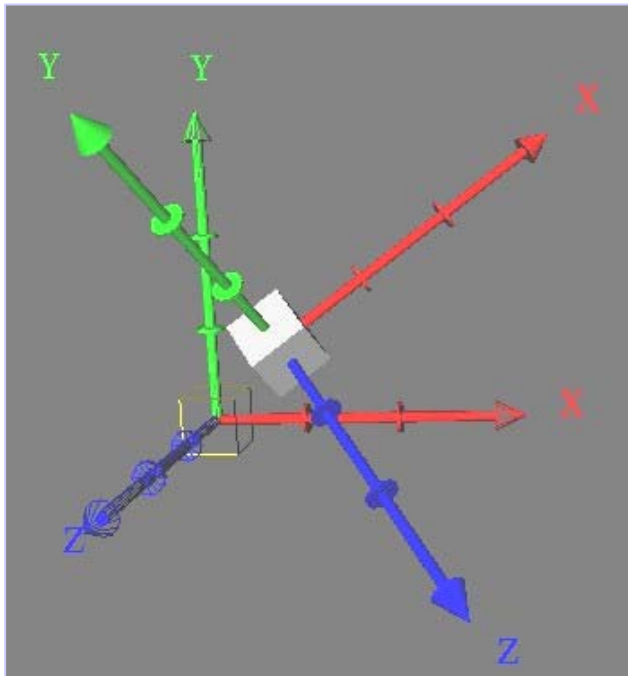
$$\mathbf{glRotate}^*(\alpha, 0, 1, 0): \begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{glRotate}^*(\alpha, 0, 0, 1): \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

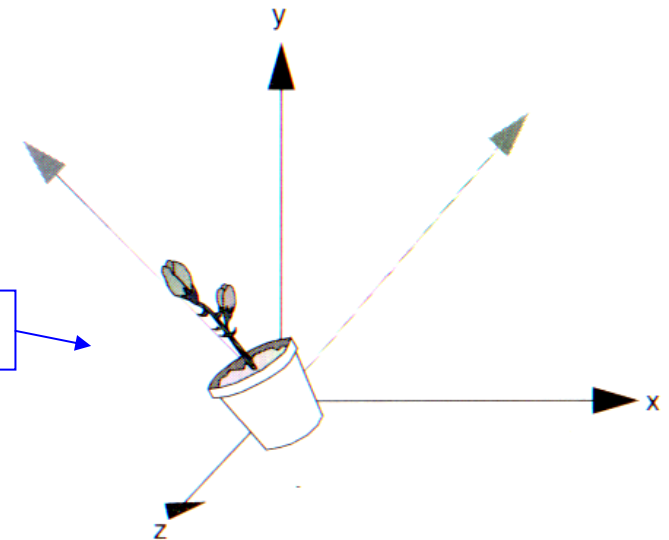
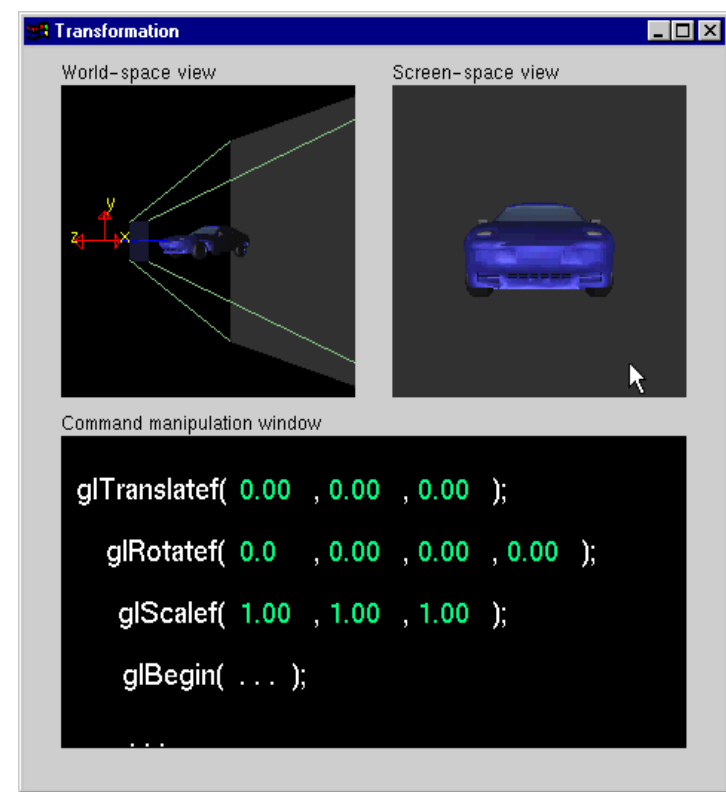
# Rotation

`glRotatef( angle, x, y, z );`

- Rotate coordinate system about an axis in space



`glRotatef( 45.0, 0, 0, 1);`



# Shear

- Scaling in one dimension depends on another
- For example

$$x' = x + f_{xy}y + f_{xz}z$$

- No OpenGL command
  - load custom matrix
- Making a shear matrix

$$H = \begin{pmatrix} 1 & f_{xy} & f_{xz} & 0 \\ f_{yx} & 1 & f_{yz} & 0 \\ f_{zx} & f_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

# Other OpenGL Matrix Commands

- `glMultMatrix(m)` multiplies the current matrix by  $m$
- `glLoadMatrix(m)` replaces the current matrix by  $m$
- `glLoadIdentity()` replace the current matrix with an identity

`GLfloat m[16];` (this is a one-dimensional array)

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

`glLoadMatrix(m);`

# Using Multiple Modeling Transforms

- Multiple modeling transforms form a *composite* modeling transform
- Individual transform matrices are multiplied together

- for example 
$$M = T(t_x, t_y, t_z) \cdot R_{\vec{v}}(\theta) \cdot S(s_x, s_y, s_z)$$

## Multiplying Matrices

- Matrix multiplication is not *commutative*
  - in general,
  - order of operations is important
- OpenGL multiplies matrices on the *right*

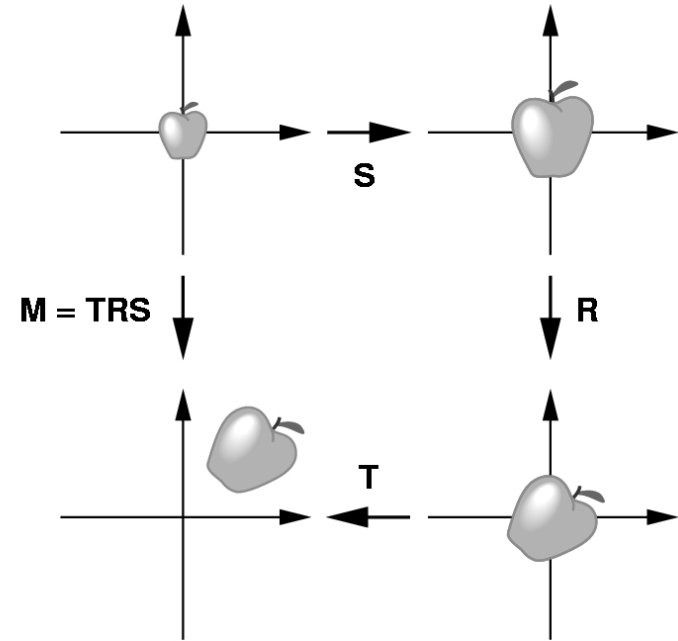
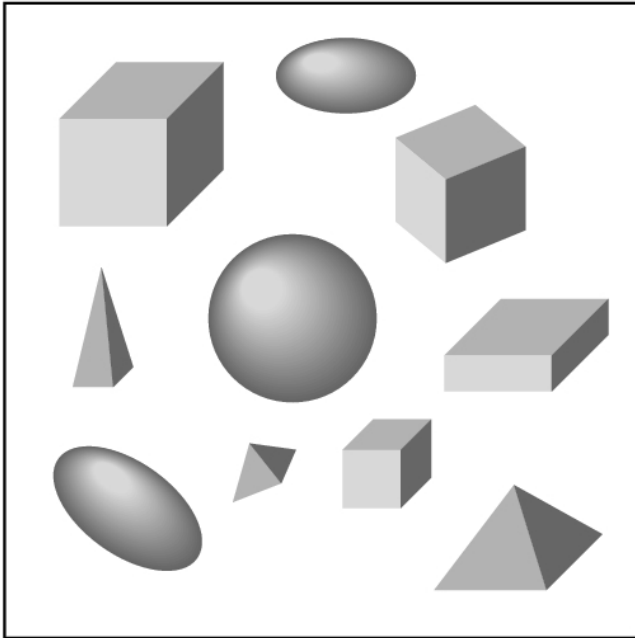
$$A \cdot B \cdot C \cdot D = (((A \cdot B) \cdot C) \cdot D)$$

# The Instance Transformation

Each occurrence of an object in the scene



an instance of the object's prototype



# 3D Transformations: summary

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

Translation

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Rotating  $R(\theta) =$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

About **x**-axis

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

About **y**-axis

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

About **z**-axis

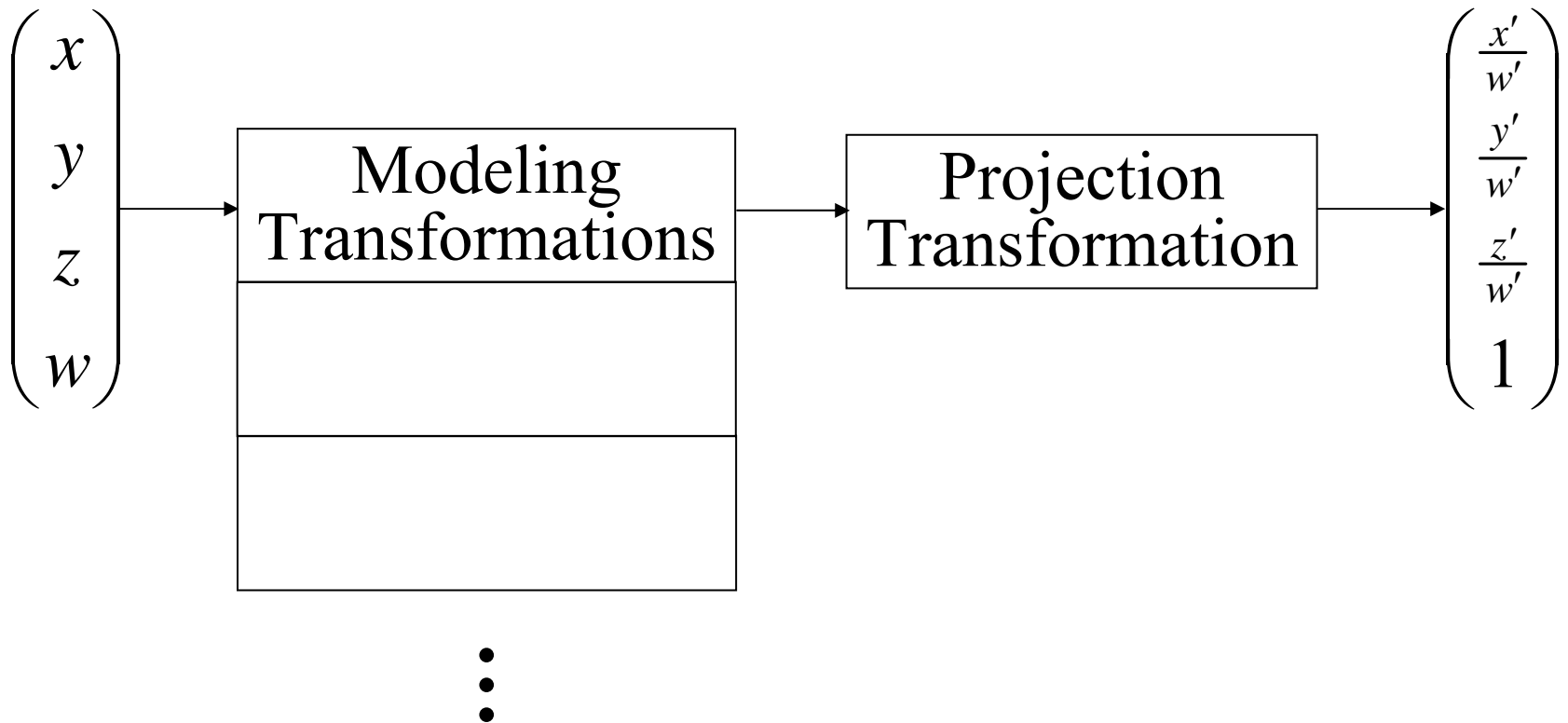
# Independent vs. Dependent Transforms

- Every modeling transform affects the model coordinate system
- Transformations accumulate
  - we record every transformation ever made
  - to undo a transform, we'd need to do the inverse transform
    - this is very inconvenient

## A Stack Based Solution

- We create a stack containing matrices
- Any transform multiplies the top of stack
- *Push* makes a copy and pushes it onto the stack
- *Pop* discards the top of stack

# A Stack Based Solution ( cont. )



# OpenGL Matrix Stack Commands

- Top of stack matrix is called the *current matrix*
- **glPushMatrix ( )**
  - copy the current matrix and pushes it
- **glPopMatrix ( )**
  - pop the current matrix

## OpenGL Matrix Stacks

- Why multiple matrix stacks?
    - certain techniques are done in different spaces
- glMatrixMode ( mode )**
- choose which stack to manipulate
    - **GL\_MODELVIEW**
    - **GL\_PROJECTION**