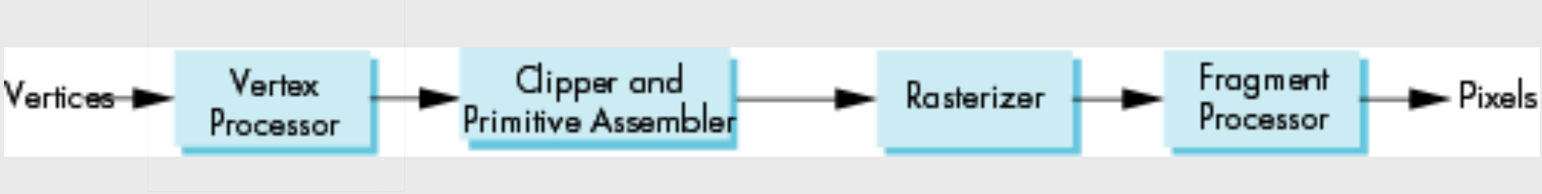
Comp 410/510

Computer Graphics
Spring 2023

Geometry & Transformations

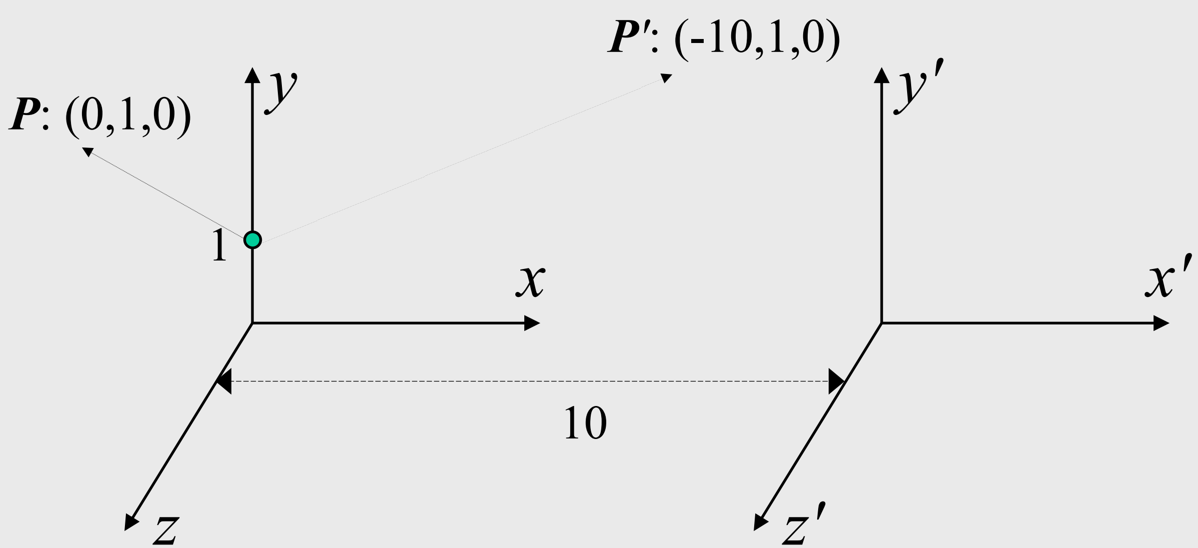


Basic Elements

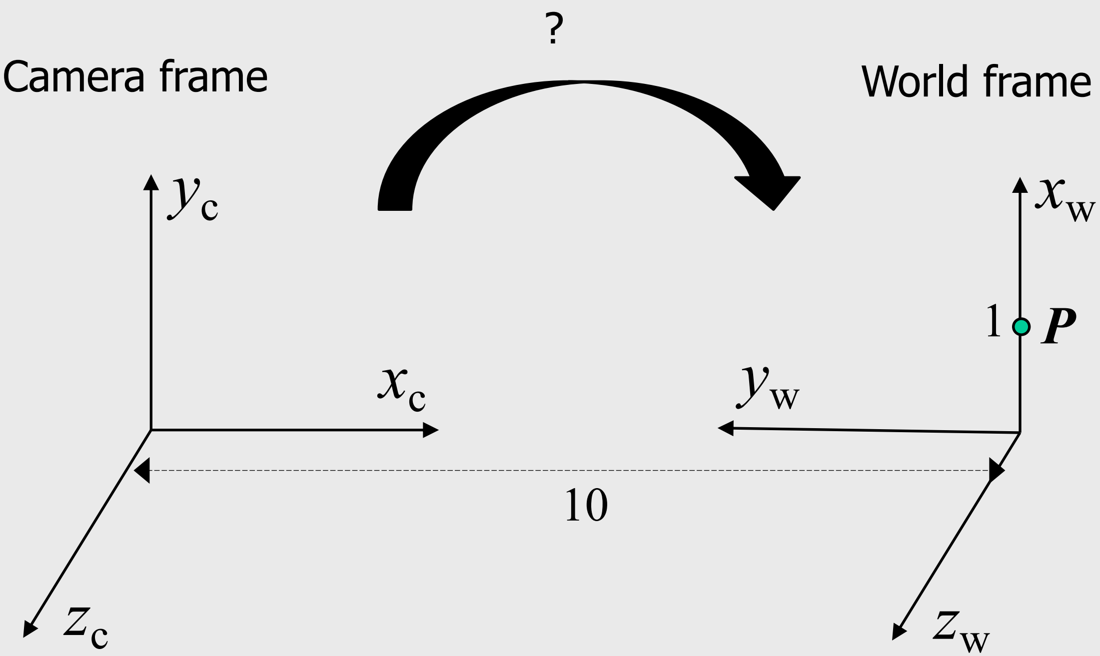
- Geometry is the study of spatial properties of objects and their relationships in an n-dimensional space
  - In computer graphics, we are interested in objects that exist in three dimensions
- We want a minimum set of elements from which we can build more sophisticated objects
- We will need three basic elements
  - Scalars
  - Vectors
  - Points

How to represent points?

- Until now we have been able to work with geometric entities without explicitly using any frame of reference or a coordinate system
- Need frame(s) of reference to relate points and objects in our physical world
- Given coordinates of a point, we can't really know where the point is without a reference system

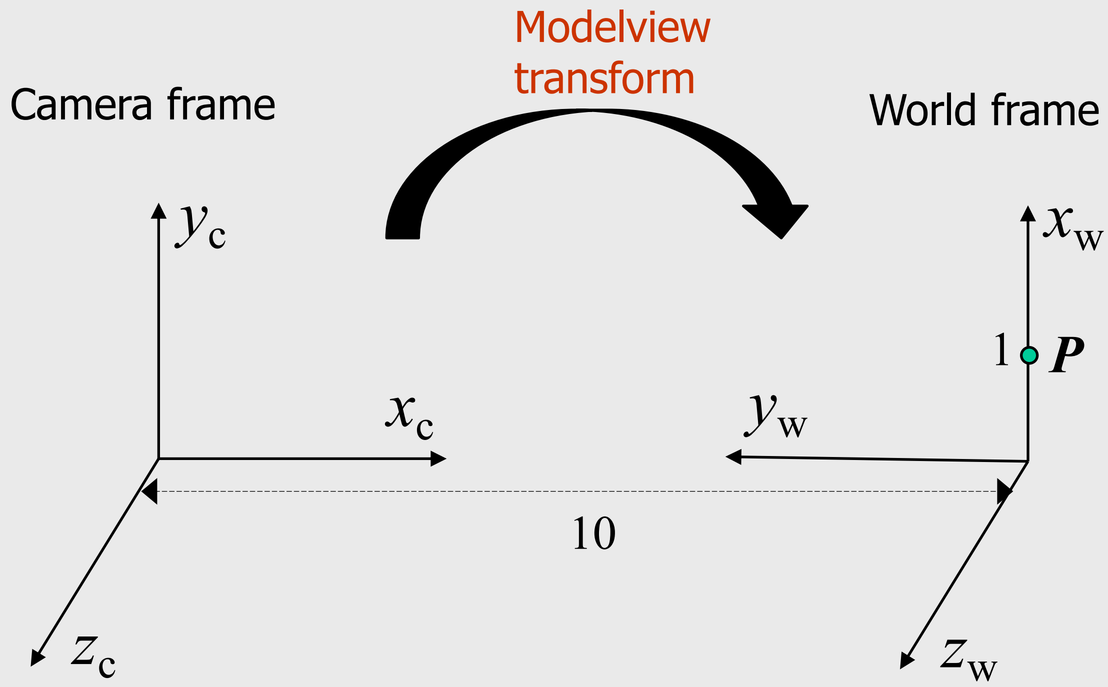


Frames of reference (in graphics)



- What is the transformation between camera and world frames in the above example?
- What is the representation of point $\boldsymbol{P}$ in world and camera frames?
- In world coordinates: (1,0,0)
- In camera coordinates: ?

Change of frames



- Transformation between camera and world frames: $\underbrace{\text{Rotation + Translation}}_{\text{Modelview transform}}$

- The representation of point $\boldsymbol{P}$:

- In world coordinates: (1,0,0)
- In camera coordinates: (10,1,0)

Transformations

To understand transformations, we need to understand

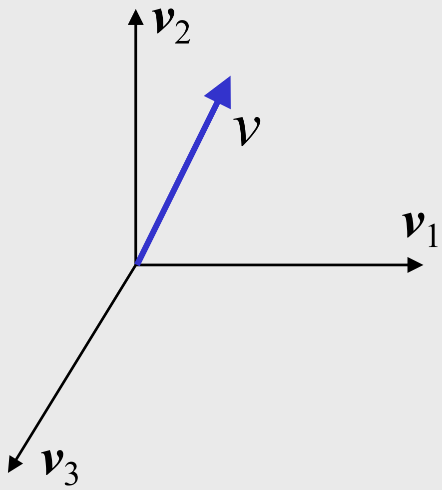
- what a coordinate system is
- what a frame of reference is
- how to change a coordinate system or a frame of reference
- what homogeneous coordinate representation is

Coordinate Systems

- Consider a basis: $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ (n vectors)
- An n-dimensional vector can then be written as a linear combination of these basis vectors: $\boldsymbol{v} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \ldots + \alpha_n\boldsymbol{v}_n$
- The list of scalars $\{\alpha_1, \alpha_2, \ldots. \alpha_n\}$ is the representation of $\boldsymbol{v}$ with respect to the given basis
- We can write the representation as a column array of scalars:

$$\boldsymbol{\alpha} = [\alpha_1 \;\; \alpha_2 \;\; \ldots \;\; \alpha_n]^{\mathrm{T}} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

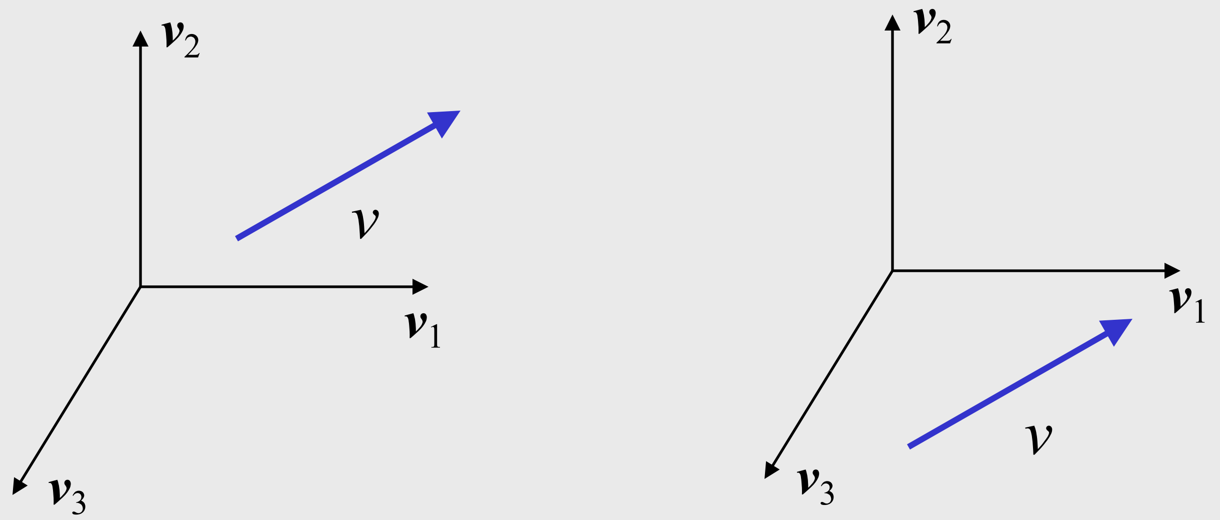
Example



- Vector: $\boldsymbol{v} = 2\boldsymbol{v}_1 + 3\boldsymbol{v}_2 - 4\boldsymbol{v}_3$
- Its coordinate representation: $\boldsymbol{\alpha} = [2\ 3\ {-4}]^T$
- Note that this representation is with respect to a particular basis
- For example, in OpenGL we start by representing geometry using the world basis but later the system needs a representation in terms of the camera (or eye) basis

Vectors in Coordinate Systems

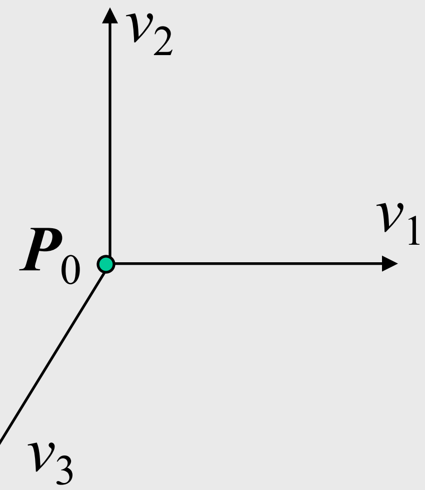
- Which placement is correct for a vector $\boldsymbol{v}$?



- Both are equivalent because vectors have no fixed location

Frames of Reference

- We can represent vectors in coordinate systems
- But coordinate system is insufficient to represent points
- We can add a single point, the origin, to the basis vectors so as to form a frame of reference



Representation in a Frame

- A frame of reference is determined by $(\boldsymbol{P}_0, \boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3)$

- Within this frame, every vector can be written as

  $\boldsymbol{v} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3$

- Every point can be written as

  $\boldsymbol{P} = \boldsymbol{P}_0 + \beta_1\boldsymbol{v}_1 + \beta_2\boldsymbol{v}_2 + \beta_3\boldsymbol{v}_3$

Confusing Points and Vectors

- Consider the point and the vector

$$\boldsymbol{P} = \boldsymbol{P}_0 + \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3$$

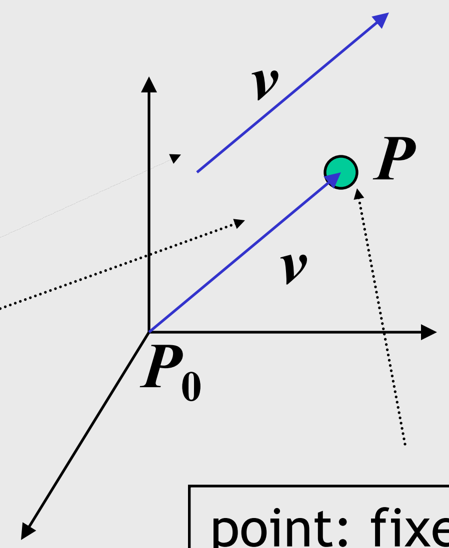
$$\boldsymbol{v} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3$$

- They appear to have similar representations:

$$\boldsymbol{P} \rightarrow [\alpha_1\ \alpha_2\ \alpha_3]^T \qquad \boldsymbol{v} \rightarrow [\alpha_1\ \alpha_2\ \alpha_3]^T$$

which confuses the point with the vector.

- A vector has no position, but a point has!



vector: can place anywhere

point: fixed

A Single Representation

Thus we write

$$\boldsymbol{v} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3 = [\alpha_1\ \alpha_2\ \alpha_3\ 0\ ]\ [\boldsymbol{v}_1\ \boldsymbol{v}_2\ \boldsymbol{v}_3\ \boldsymbol{P}_0]^T$$

$$\boldsymbol{P} = \boldsymbol{P}_0 + \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3 = [\alpha_1\ \alpha_2\ \alpha_3\ 1\ ]\ [\boldsymbol{v}_1\ \boldsymbol{v}_2\ \boldsymbol{v}_3\ \boldsymbol{P}_0]^T$$

where we define
$0 \cdot \boldsymbol{P} = \mathbf{0}$ and $1 \cdot \boldsymbol{P} = \boldsymbol{P}$

And we obtain the four-dimensional homogeneous coordinate representation:

$\boldsymbol{v} \rightarrow [\alpha_1\ \alpha_2\ \alpha_3\ 0\ ]^T$: for vectors

$\boldsymbol{P} \rightarrow [\alpha_1\ \alpha_2\ \alpha_3\ 1\ ]^T$: for points

Homogeneous Coordinates

The general form of four dimensional homogeneous coordinates is

$$[x\ y\ z\ w]^{\mathrm{T}}$$

We return to a three dimensional point (for $w \neq 0$) by perspective division

$$x \leftarrow x/w$$

$$y \leftarrow y/w$$

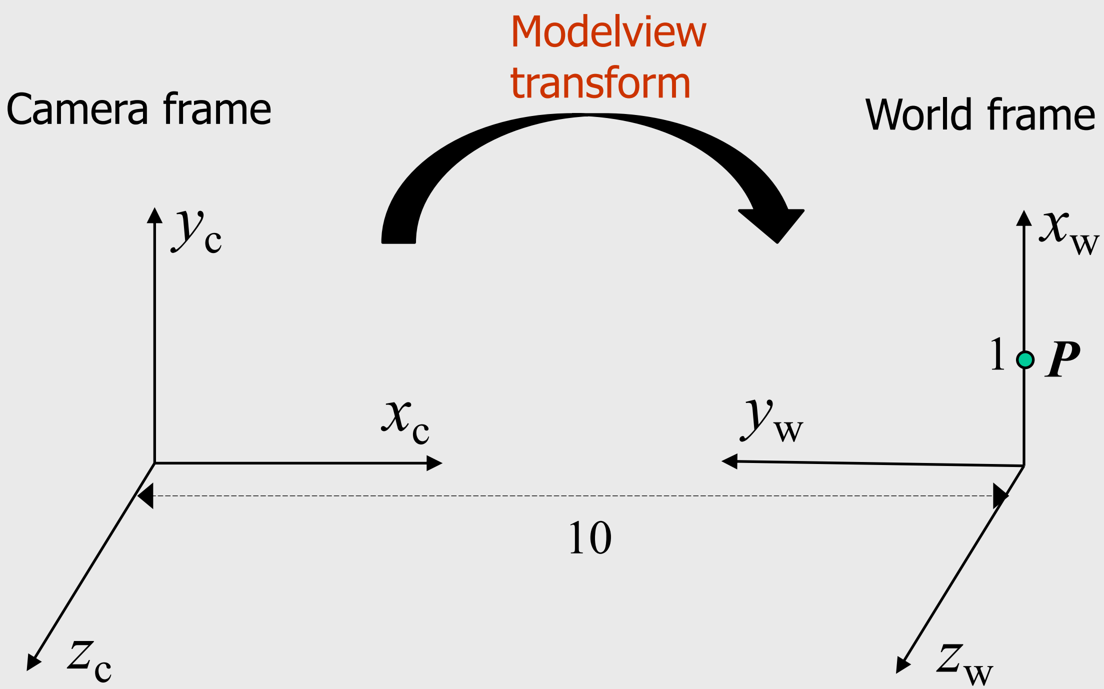
$$z \leftarrow z/w$$

If $w = 0$, the representation is that of a vector.

Homogeneous Coordinates & Computer Graphics

- Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented by matrix multiplications with 4 x 4 matrices
  - Hardware pipeline works with 4 dimensional representations
  - Orthographic projection maintains $w = 0$ for vectors and $w = 1$ for points; but for perspective projection, we will need a perspective division

Change of frames



- Transformation between camera and world frames: $\underbrace{\text{Rotation + Translation}}_{\text{Modelview transform}}$

- The representation of point P :

- In world coordinates: (1,0,0)
- In camera coordinates: (10,1,0)

Change of Coordinate Systems

- All standard transformations (such as rotation, translation) in computer graphics are actually change of coordinate systems (or frames or reference).

Change of Coordinate Systems

- Consider two representations of the same vector $\boldsymbol{x}$ with respect to two different bases. The representations are

$$\boldsymbol{\alpha} = [\alpha_1 \;\; \alpha_2 \;\; \alpha_3\,]^{\mathrm{T}}$$
$$\boldsymbol{\beta} = [\beta_1 \;\; \beta_2 \;\; \beta_3]^{\mathrm{T}}$$

where

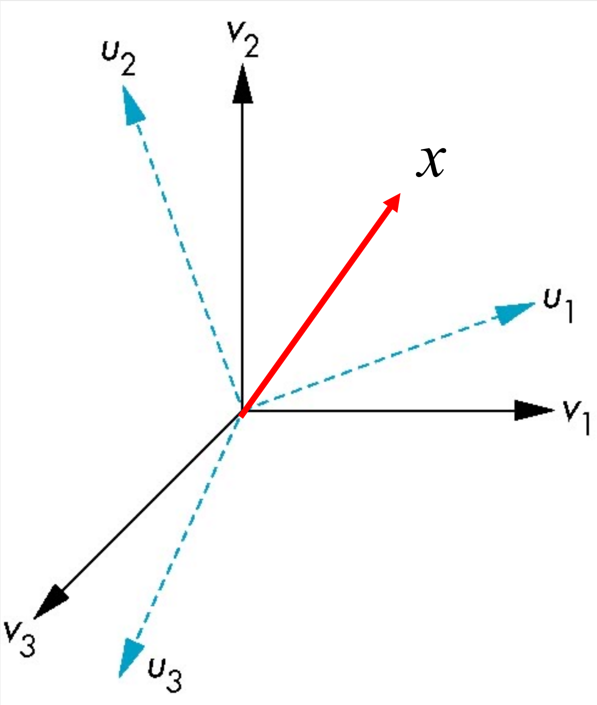
$$\boldsymbol{x} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3 = \boldsymbol{\alpha}^{\mathrm{T}}\,[\boldsymbol{v}_1 \;\boldsymbol{v}_2 \;\boldsymbol{v}_3]^{\mathrm{T}}$$

$$= \beta_1\boldsymbol{u}_1 + \beta_2\boldsymbol{u}_2 + \beta_3\boldsymbol{u}_3 = \boldsymbol{\beta}^{\mathrm{T}}\,[\boldsymbol{u}_1 \;\boldsymbol{u}_2 \;\boldsymbol{u}_3]^{\mathrm{T}}$$

Representing the second basis in terms of the first

Each of the basis vectors, $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3$, is a vector that can be represented in terms of the first basis:

$$\boldsymbol{u}_1 = \gamma_{11}\boldsymbol{v}_1 + \gamma_{12}\boldsymbol{v}_2 + \gamma_{13}\boldsymbol{v}_3$$
$$\boldsymbol{u}_2 = \gamma_{21}\boldsymbol{v}_1 + \gamma_{22}\boldsymbol{v}_2 + \gamma_{23}\boldsymbol{v}_3$$
$$\boldsymbol{u}_3 = \gamma_{31}\boldsymbol{v}_1 + \gamma_{32}\boldsymbol{v}_2 + \gamma_{33}\boldsymbol{v}_3$$



Matrix Form

These coefficients define a 3 x 3 matrix

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

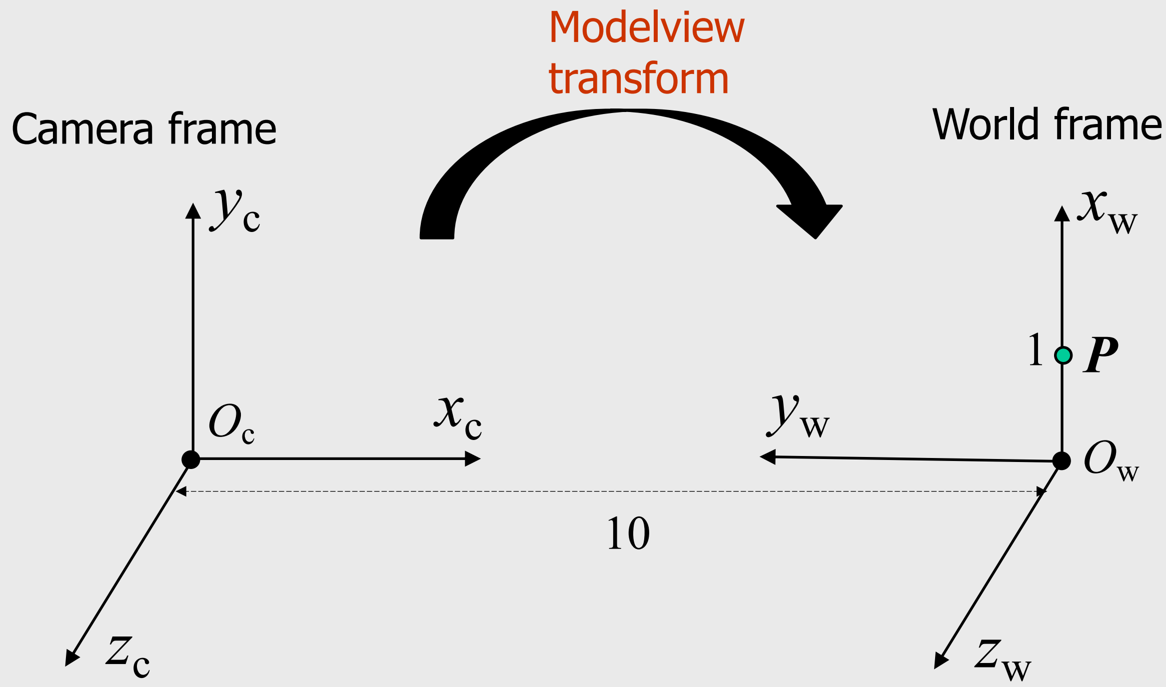
and the representations in these coordinate systems are then related by

$$\boldsymbol{\alpha} = \boldsymbol{A}^{\mathrm{T}} \boldsymbol{\beta} = \boldsymbol{M} \boldsymbol{\beta}$$

The World and Camera Frames

- When we work with representations, we work with points, vectors and scalars
- Changes in frame of reference are then defined by 4x4 matrices
- In OpenGL, the base frame that we start with is the world frame
- Eventually entities are represented in the camera frame by changing the world representation using the model-view matrix
- So the change of frame of reference from world to camera is represented by a model-view matrix $\boldsymbol{M}$
- Initially these frames are the same ($\boldsymbol{M} = \boldsymbol{I}$)

Recap: Change of frames



- Transformation from world to camera: Rotation + Translation

$$\underbrace{\text{Rotation + Translation}}_{\text{Modelview transform } M}$$

- The representation of point P :

- In world frame: (1,0,0,1)
- In camera frame: (10,1,0,1)

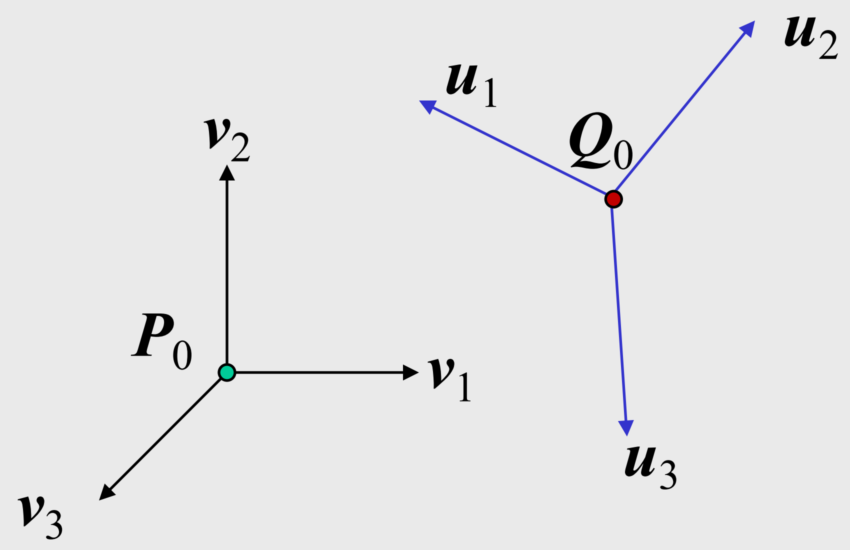
$$(10\ ,\ 1,\ 0,\ 1)^T = M_{4x4} \cdot (1,\ 0,\ 0,\ 1)^T$$

Change of Frames

- Use homogeneous coordinates

- Consider two frames:

$$(\boldsymbol{P}_0, \boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3)$$
$$(\boldsymbol{Q}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3)$$



- Any point or vector is represented differently in each frame

One Frame in Terms of the Other

Express one frame in terms of the other:

$$
\begin{aligned}
\boldsymbol{u}_1 &= \gamma_{11}\boldsymbol{v}_1 + \gamma_{12}\boldsymbol{v}_2 + \gamma_{13}\boldsymbol{v}_3 \\
\boldsymbol{u}_2 &= \gamma_{21}\boldsymbol{v}_1 + \gamma_{22}\boldsymbol{v}_2 + \gamma_{23}\boldsymbol{v}_3 \\
\boldsymbol{u}_3 &= \gamma_{31}\boldsymbol{v}_1 + \gamma_{32}\boldsymbol{v}_2 + \gamma_{33}\boldsymbol{v}_3 \\
\boldsymbol{Q}_0 &= \gamma_{41}\boldsymbol{v}_1 + \gamma_{42}\boldsymbol{v}_2 + \gamma_{43}\boldsymbol{v}_3 + \boldsymbol{P}_0
\end{aligned}
$$

We can define a 4 x 4 matrix representing a change of frames

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

Working with Representations

A point or vector can then be represented in any of the two frames using homogeneous coordinates:

$\boldsymbol{\alpha} = [\alpha_1\ \alpha_2\ \ \alpha_3\ \alpha_4\ ]^T$ in the first frame
$\boldsymbol{\beta} = [\beta_1\ \ \beta_2\ \ \beta_3\ \ \beta_4\ ]^T$ in the second frame

where $\alpha_4 = \beta_4 = 1$ for points and $\alpha_4 = \beta_4 = 0$ for vectors.

The matrix $\boldsymbol{M} = \boldsymbol{A}^T$ is 4x4 and specifies an affine transformation in homogeneous coordinates

$$\boldsymbol{\alpha} = \boldsymbol{M}\,\boldsymbol{\beta}$$

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

Affine Transformations

- Every affine transformation is equivalent to a change of frames
- Every affine transformation preserves lines
- An affine transformation has only 12 degrees of freedom because 4 of the elements in the matrix are fixed.

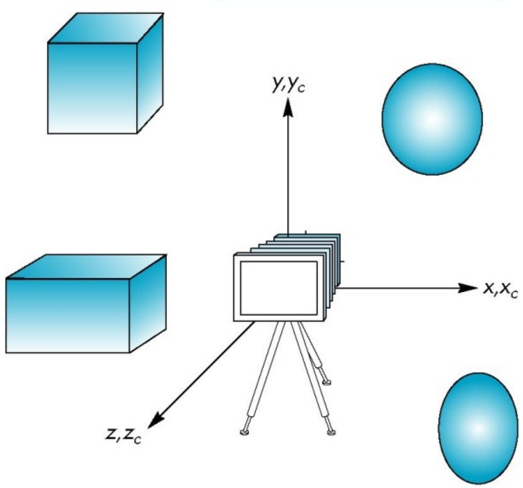
$$\boldsymbol{M} = \begin{bmatrix} \gamma_{11} & \gamma_{21} & \gamma_{31} & \gamma_{41} \\ \gamma_{12} & \gamma_{22} & \gamma_{32} & \gamma_{42} \\ \gamma_{13} & \gamma_{23} & \gamma_{33} & \gamma_{43} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Moving the World Frame

If objects are on both sides of $z = 0$ (hence the default view volume),

- we should move (translate) objects,
- or equivalently, move the world frame with respect to camera frame

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



(a)

ld

Moving the World Frame

Change of frames:

$$
\begin{aligned}
\boldsymbol{u}_1 &= \gamma_{11}\boldsymbol{v}_1 + \gamma_{12}\boldsymbol{v}_2 + \gamma_{13}\boldsymbol{v}_3 &&= 1\boldsymbol{v}_1 + 0\boldsymbol{v}_2 + 0\boldsymbol{v}_3 \\
\boldsymbol{u}_2 &= \gamma_{21}\boldsymbol{v}_1 + \gamma_{22}\boldsymbol{v}_2 + \gamma_{23}\boldsymbol{v}_3 &&= 0\boldsymbol{v}_1 + 1\boldsymbol{v}_2 + 0\boldsymbol{v}_3 \\
\boldsymbol{u}_3 &= \gamma_{31}\boldsymbol{v}_1 + \gamma_{32}\boldsymbol{v}_2 + \gamma_{33}\boldsymbol{v}_3 &&= 0\boldsymbol{v}_1 + 0\boldsymbol{v}_2 + 1\boldsymbol{v}_3 \\
\boldsymbol{Q}_0 &= \gamma_{41}\boldsymbol{v}_1 + \gamma_{42}\boldsymbol{v}_2 + \gamma_{43}\boldsymbol{v}_3 + \boldsymbol{P}_0 &&= 0\boldsymbol{v}_1 + 0\boldsymbol{v}_2 - d\boldsymbol{v}_3 + \boldsymbol{P}_0
\end{aligned}
$$

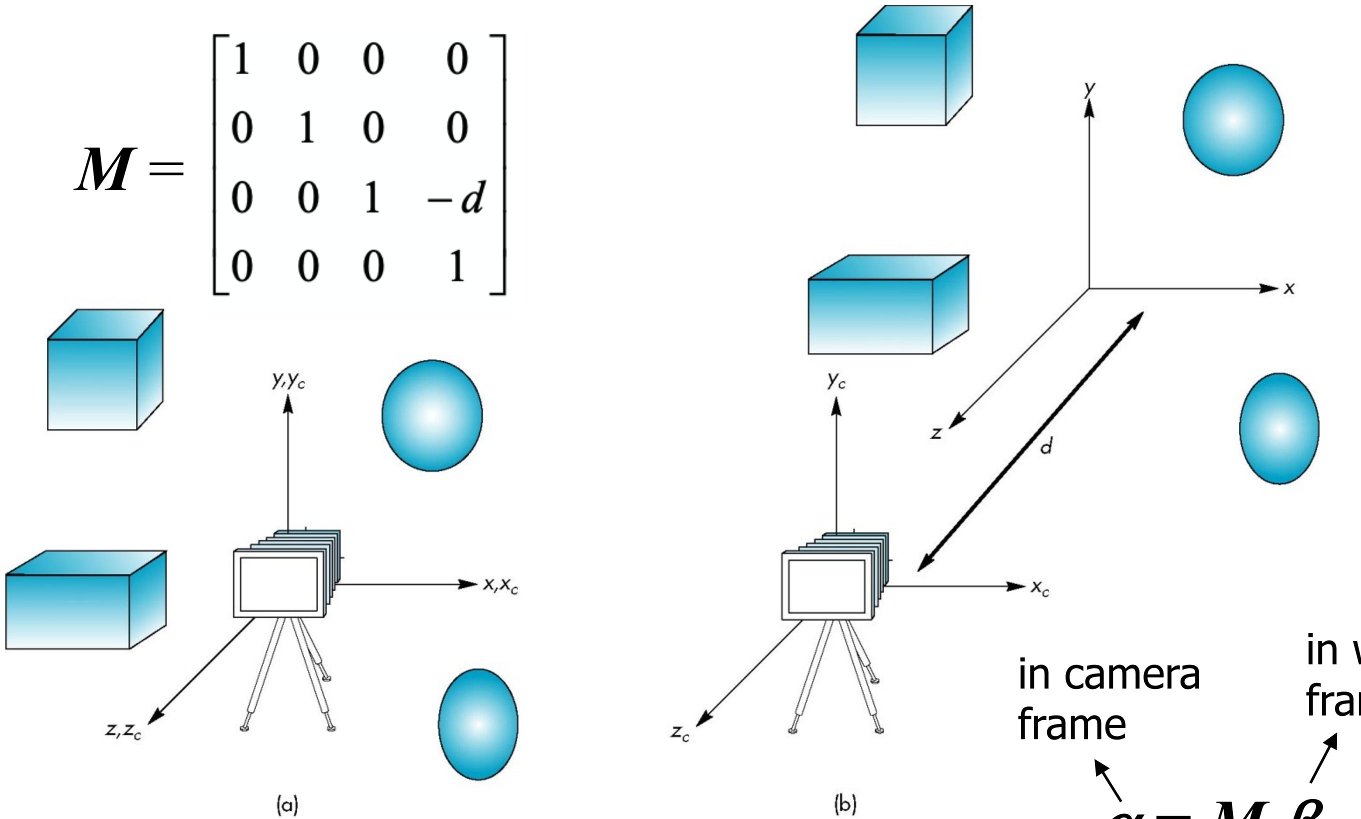
The 4x4 matrix representing the change of frames:

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

Modelview Transformation

- Modelview transformation is equivalent to change of frames of reference.
- The given example is for translation:

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



in camera frame

in world frame

$$\alpha = M\,\beta$$

Objectives

- Introduce standard transformations:
  - Rotations
  - Translation
  - Scaling
  - Shear
- Derive their homogeneous coordinate transformation matrices
- Learn to build arbitrary transformation matrices from simple transformations

Rotation around x, y and z axes

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

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

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

Scaling

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

Translation

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

We will next derive these matrices...

Concatenation

- We can form arbitrary affine transformation matrices by multiplying rotation, translation, and scaling matrices
- Since the same transformation is applied to many vertices, the cost of forming a matrix $\boldsymbol{M=ABC}$ only once is not significant compared to the cost of computing $\boldsymbol{Mp}$ for many vertices $\boldsymbol{p}$
- The difficult part is how to form a desired transformation from the specifications in the application

Order of Transformations

- Note that the matrix on the right is the first applied
- Mathematically, the following are equivalent

$$p' = ABCp = A(B(Cp))$$

Rotation, Translation, Scaling

You need to implement this general rotation function yourself, not defined in `mat.h` header file.

mat.h includes `RotateX()`, `RotateY()` and `RotateZ()` functions

Create an identity matrix:

```
mat4 m = Identity();
```

Multiply on the right :

```
mat4 r = Rotate(theta, vx, vy, vz)
m = m*r;
```

`theta` specifies angle in degrees (counter-clockwise)

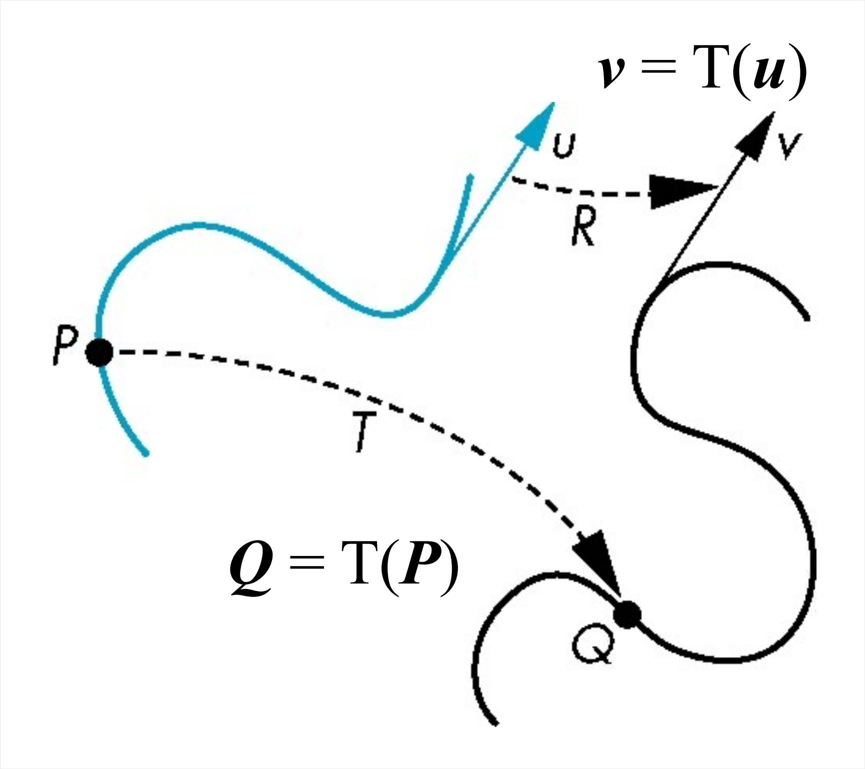
`(vx, vy, vz)` defines axis of rotation

Do the same with translation and scaling:

```
mat4 s = Scale(sx, sy, sz)
mat4 t = Translate(dx, dy, dz);
m = m*s*t;
```

General Transformations

- A transformation maps points to other points and/or vectors to other vectors:

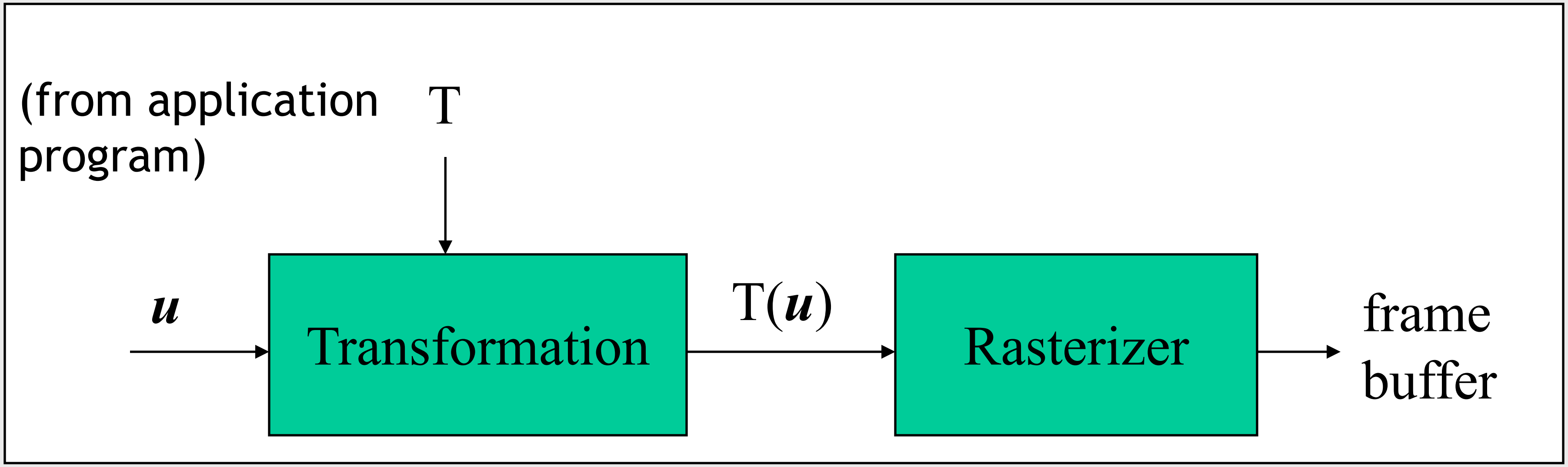


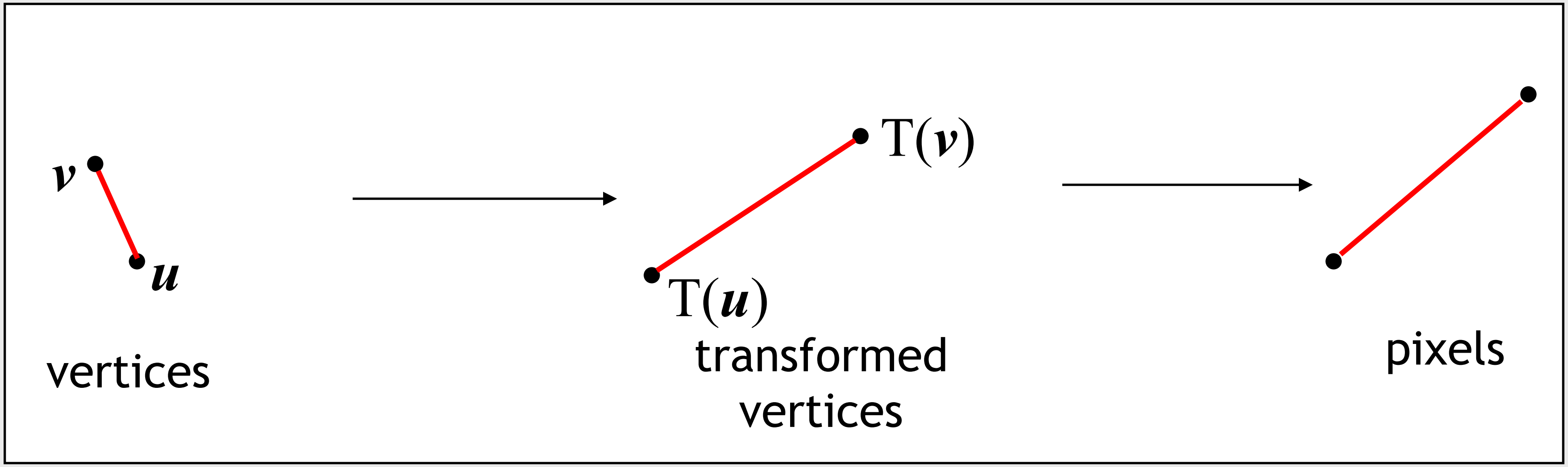
Affine Transformations

- Line preserving property
- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear
- Importance in graphics is that we need only transform the endpoints of line segments and let implementation draw the line segment between the transformed endpoints

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

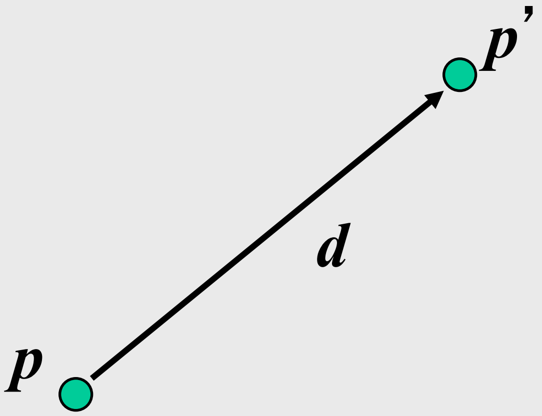
Pipeline Implementation





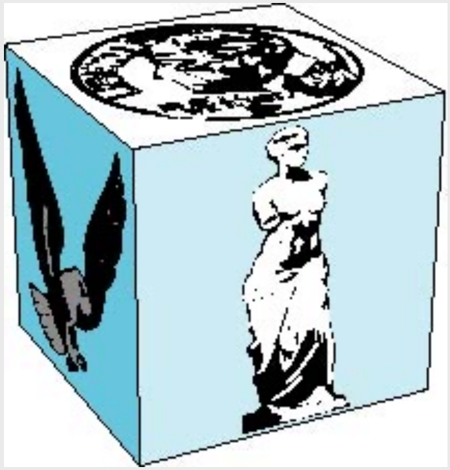
Translation

- Move (translate, displace) a point to a new location

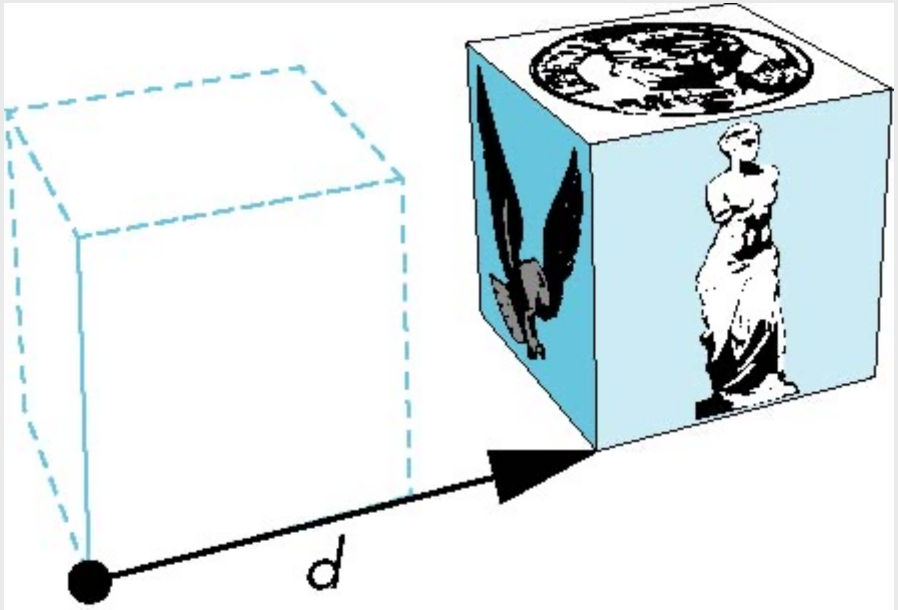


- Displacement is determined by a vector $\boldsymbol{d}$
  - 3 degrees of freedom
  - $\boldsymbol{p}' = \boldsymbol{p} + \boldsymbol{d}$

Translation



object



Translation of an object: Every point of the object is displaced by the same vector

Translation using Homogeneous Coordinates

Consider the homogeneous coordinate representation in some frame:

$$\boldsymbol{p} = [\,x\; y\; z\; 1]^{\mathrm{T}}$$
$$\boldsymbol{p}' = [x'\; y'\; z'\; 1]^{\mathrm{T}}$$
$$\boldsymbol{d} = [d_x\; d_y\; d_z\; 0]^{\mathrm{T}}$$

Hence $\boldsymbol{p}' = \boldsymbol{p} + \boldsymbol{d}$ or

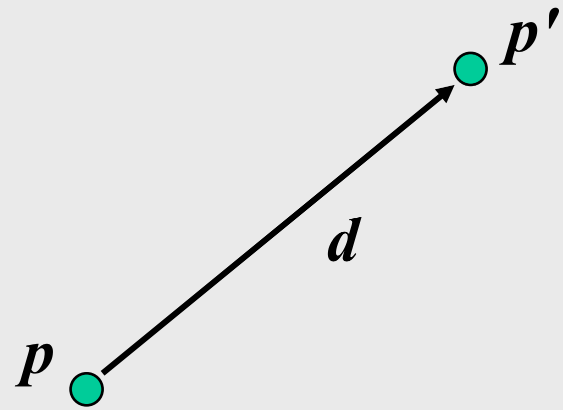
$$x' = x + d_x$$
$$y' = y + d_y$$
$$z' = z + d_z$$
$$1 = 1 + 0$$

Note that this expression is in four dimensions, thus point = vector + point

Translation Matrix

We can express translation using a 4x4 matrix $\boldsymbol{T}$ in homogeneous coordinates:

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



$\boldsymbol{p} = [x \;\; y \;\; z \;\; 1]^{\mathrm{T}}$
$\boldsymbol{p}' = [x' \; y' \; z' \; 1]^{\mathrm{T}}$

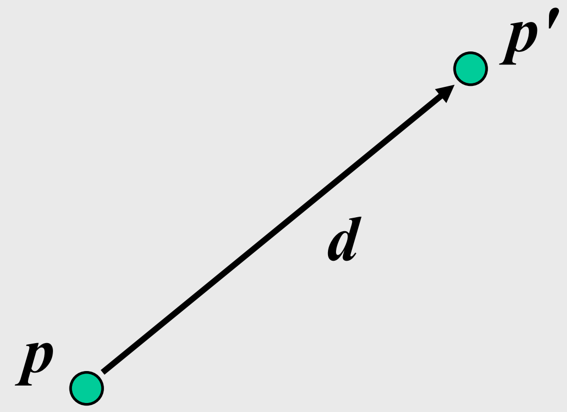
Then $\boldsymbol{p}' = \boldsymbol{T}\boldsymbol{p}$ translates $\boldsymbol{p}$ to $\boldsymbol{p}'$

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ z + dz \\ 1 \end{bmatrix}$$

Translation Matrix

We can express translation using a 4x4 matrix $\boldsymbol{T}$ in homogeneous coordinates:

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



Then $\boldsymbol{p}' = \boldsymbol{T}\boldsymbol{p}$ translates $\boldsymbol{p}$ to $\boldsymbol{p}'$

$$\boldsymbol{p} = [x \;\; y \;\; z \;\; 1]^{\mathrm{T}}$$
$$\boldsymbol{p}' = [x' \; y' \; z' \; 1]^{\mathrm{T}}$$

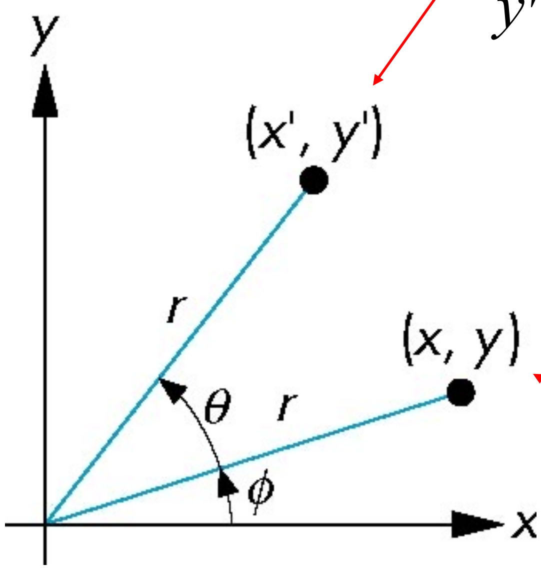
The 4x4 form is better for implementation because

- all affine transformations can be expressed in terms of matrices and,
- multiple transformations can be concatenated together by multiplication

Rotation (2D)

- Consider rotation about the origin by θ degrees
  - radius remains the same, angle increases by θ

$$x' = r\cos(\phi + \theta)$$
$$y' = r\sin(\phi + \theta)$$

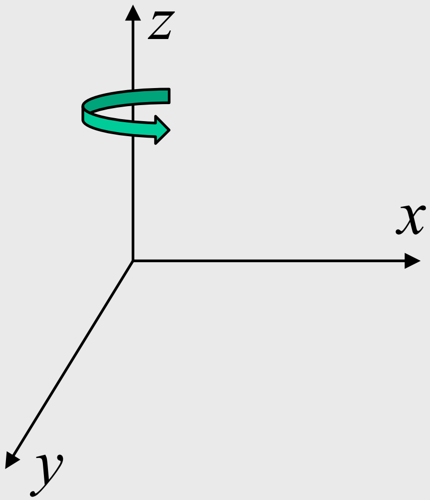


$$\boxed{\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}}$$

$$x = r\cos\phi$$
$$y = r\sin\phi$$

Rotation about the z axis

- Rotation about $z-$axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions on a plane of constant z



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

  - or in homogeneous coordinates

$$\boldsymbol{p}' = \boldsymbol{R}_z(\theta)\,\boldsymbol{p} \qquad \boldsymbol{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about x and y axes

- Same arguments with rotation about z axis
  - For rotation about x-axis, x is unchanged
  - For rotation about y-axis, y is unchanged

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

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

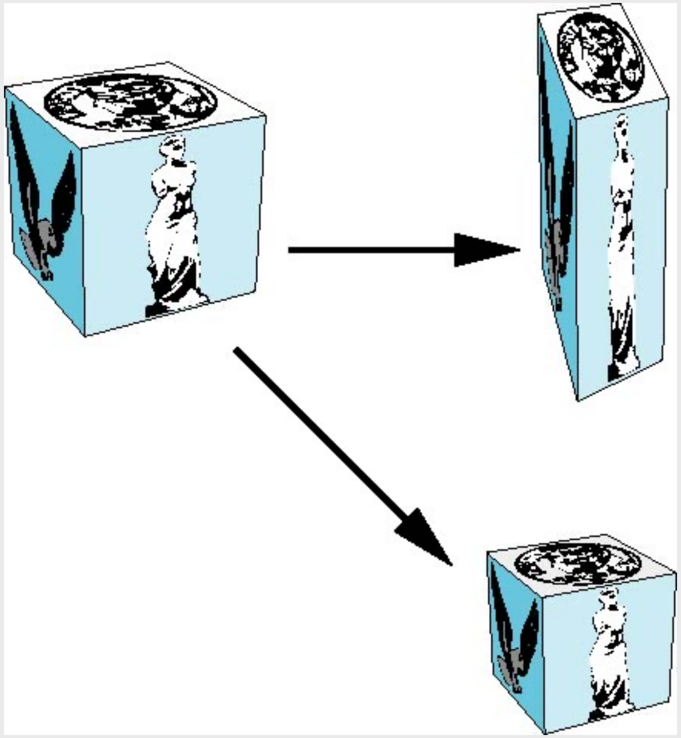
Scaling

Expand or contract along each axis (with fixed point of origin)

$$x' = s_x x$$
$$y' = s_y y$$
$$z' = s_z z$$

$$\boldsymbol{p}' = \boldsymbol{S}\boldsymbol{p}$$

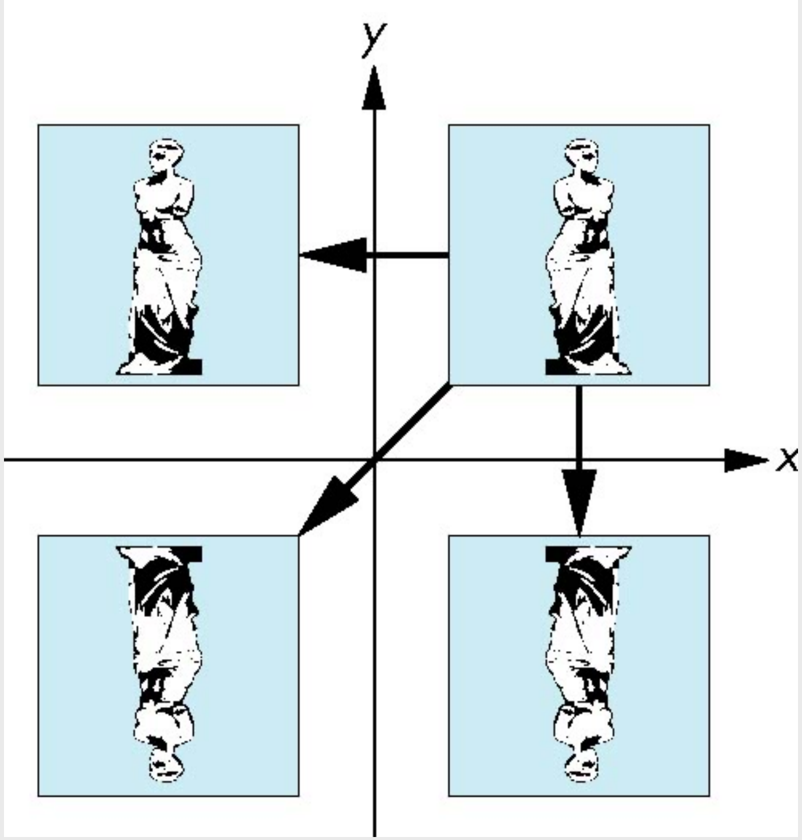
$$\boldsymbol{S} = \boldsymbol{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Reflection

corresponds to scaling with negative factors

$s_x = -1$
$s_y = 1$



original

$s_x = -1$
$s_y = -1$

$s_x = 1$
$s_y = -1$

Inverse Transformations

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations
  - Translation: $\boldsymbol{T}^{-1}(d_x, d_y, d_z) = \boldsymbol{T}(-d_x, -d_y, -d_z)$
  - Rotation: $\boldsymbol{R}^{-1}(\theta) = \boldsymbol{R}(-\theta)$
    - Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$

$$\boldsymbol{R}^{-1}(\theta) = \boldsymbol{R}^{\mathrm{T}}(\theta)$$

  - Scaling: $\boldsymbol{S}^{-1}(s_x, s_y, s_z) = \boldsymbol{S}(1/s_x, 1/s_y, 1/s_z)$

Concatenation

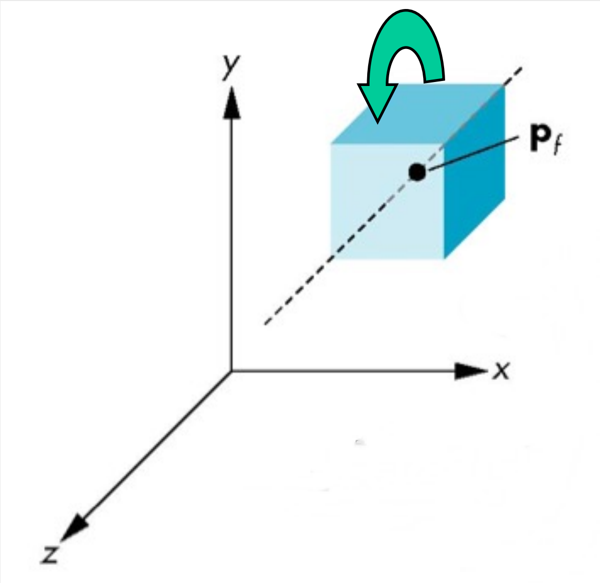
- We can form arbitrary affine transformation matrices by multiplying rotation, translation, and scaling matrices
- Since the same transformation is applied to many vertices, the cost of forming a matrix $\boldsymbol{M}=\boldsymbol{ABC}$ only once is not significant compared to the cost of computing $\boldsymbol{Mp}$ for many vertices $\boldsymbol{p}$
- The difficult part is how to form a desired transformation from the specifications in the application

Order of Transformations

- Note that the matrix on the right is the first applied
- Mathematically, the following are equivalent

$$p' = ABCp = A(B(Cp))$$

Rotation around a Fixed Point other than the Origin



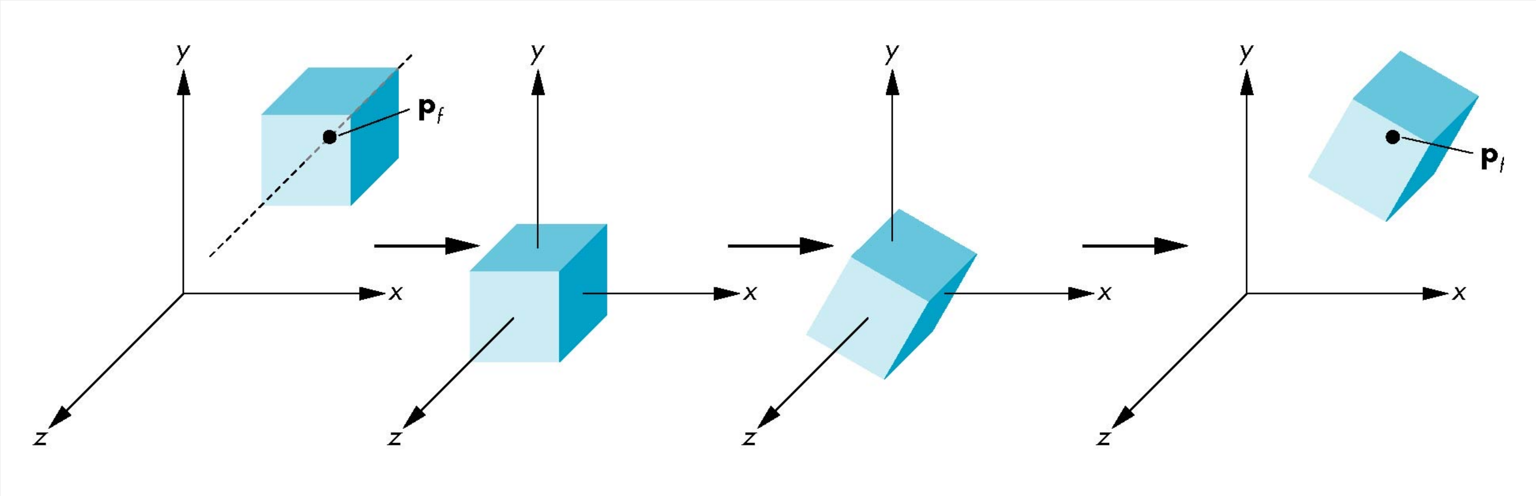
$$M = ?$$

Rotation around a Fixed Point other than the Origin

1. Move fixed point to origin (along with the cube)
2. Rotate
3. Move fixed point back

But how to compose $\boldsymbol{R}(\theta)$ if rotation is around an arbitrary axis?

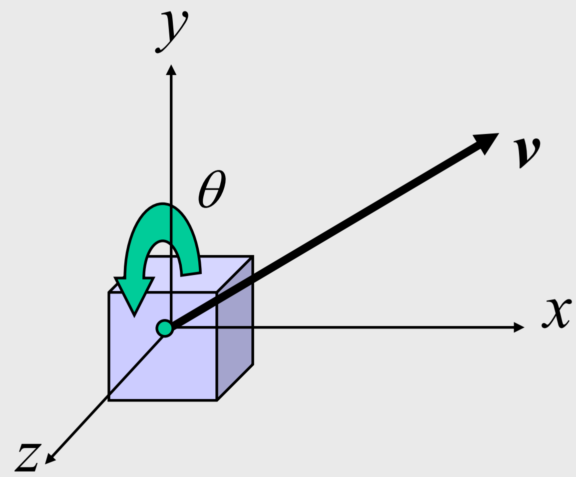
$$\boldsymbol{M} = \boldsymbol{T}(\boldsymbol{p}_f)\, \boldsymbol{R}(\theta)\, \boldsymbol{T}(-\boldsymbol{p}_f)$$



Rotation Around the Origin & an Arbitrary Axis

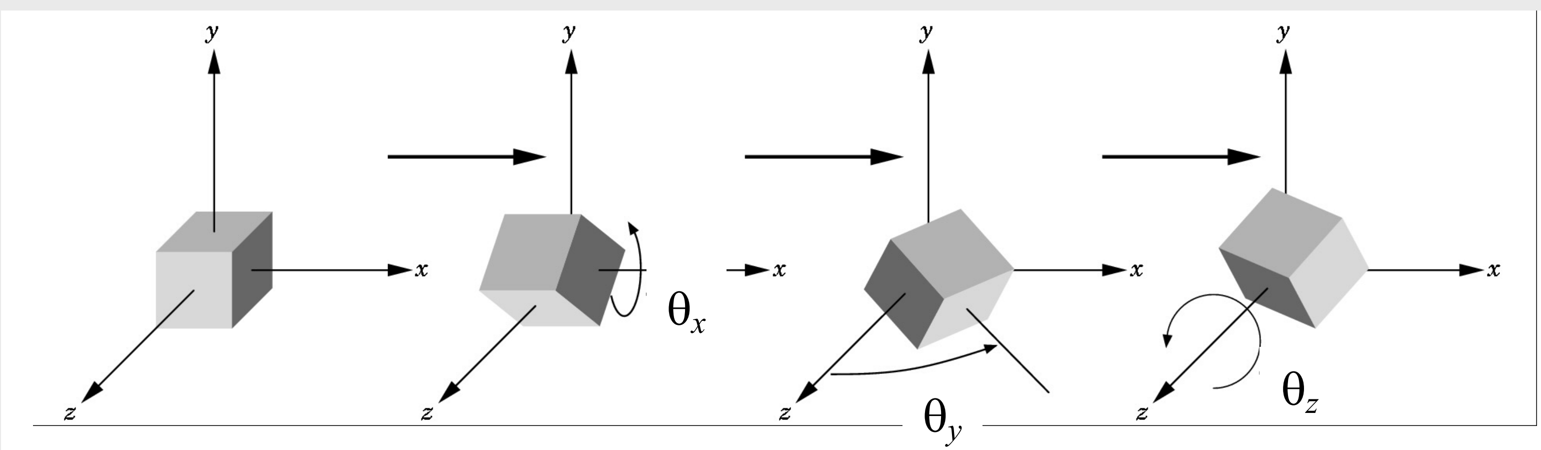
A rotation by θ about an arbitrary axis and the origin can always be decomposed into a concatenation of rotations about x, y, and z axes:

$$\boldsymbol{R}(\theta) = \boldsymbol{R}_z(\theta_z)\, \boldsymbol{R}_y(\theta_y)\, \boldsymbol{R}_x(\theta_x)$$



θ_x, θ_y, θ_z are called Euler angles.

- Note that rotations do not commute.
- We could use rotations in another order but with different angles, to get the same effect.

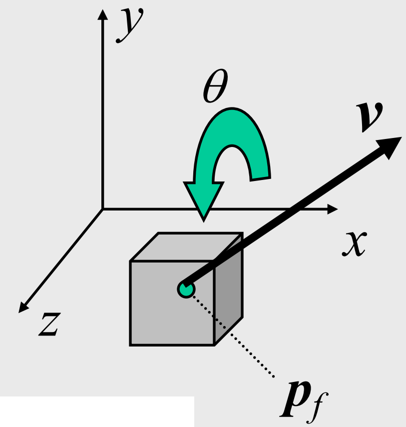


Would need to compute the corresponding Euler angles; instead we'll use the formulation in the next slide to get the most general rotation

General Rotation around an Arbitrary Axis and Point

$$\boldsymbol{M} = \boldsymbol{T}(\boldsymbol{p}_f)\, \boldsymbol{R}(\theta)\, \boldsymbol{T}(-\boldsymbol{p}_f)$$

$$\boldsymbol{M} = \boldsymbol{T}(\boldsymbol{p}_f)\, \boldsymbol{R}_x(-\theta_x)\, \boldsymbol{R}_y(-\theta_y)\, \boldsymbol{R}_z(\theta)\, \boldsymbol{R}_y(\theta_y)\, \boldsymbol{R}_x(\theta_x)\, \boldsymbol{T}(-\boldsymbol{p}_f)$$



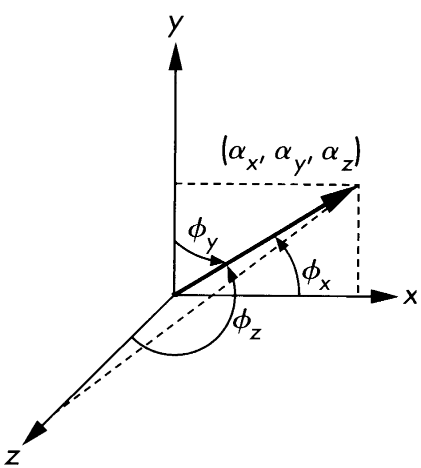


Figure 4.57 Direction angles.

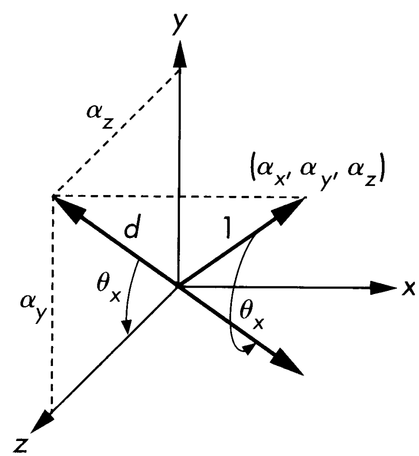


Figure 4.58 Computation of the x rotation.

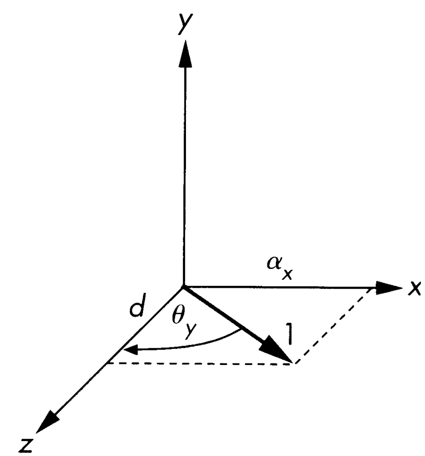


Figure 4.59 Computation of the y rotation.

Remark: Capability of rotation about two axes is sufficient to get any desired orientation.

Read Section 3.10.4 in textbook

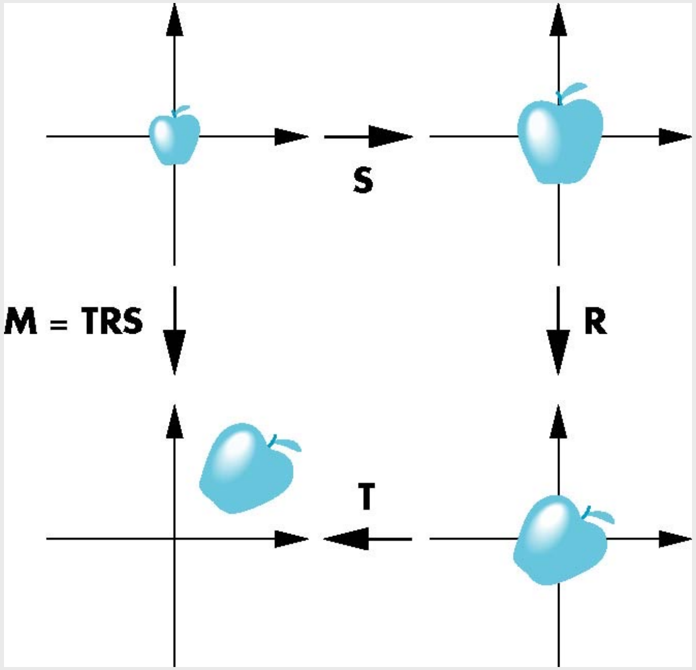
Instancing

- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an *instance transformation* to its vertices to

  Scale

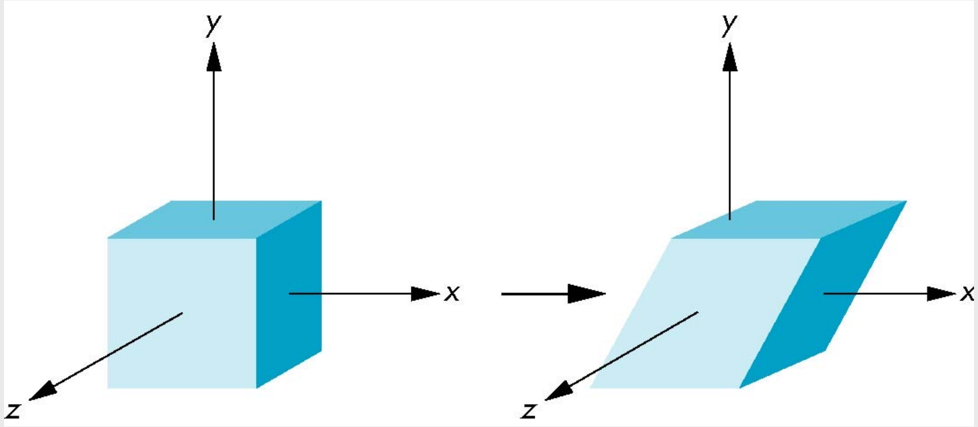
  Orient (Rotate)

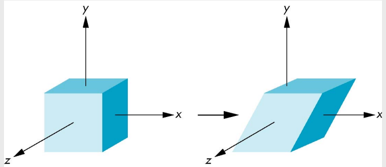
  Locate (Translate)



Shear

- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions

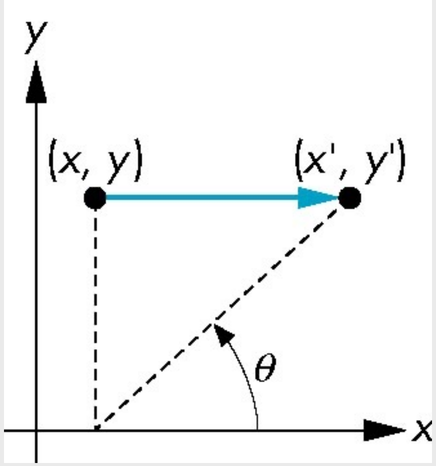




Shear Matrix

Consider simple shear along x-axis

$$x' = x + y\cot\theta$$
$$y' = y$$
$$z' = z$$



$$\boldsymbol{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

θ determines amount of shear

Objectives

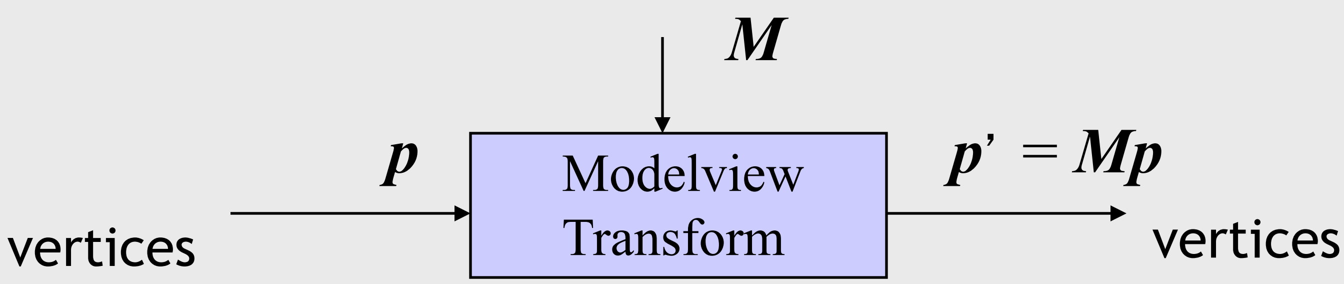
- Learn how to carry out transformations in OpenGL
  - Rotation
  - Translation
  - Scaling
- Introduce OpenGL transformation matrices
  - Model-view
  - Projection (Later)

Pre-OpenGL Matrices

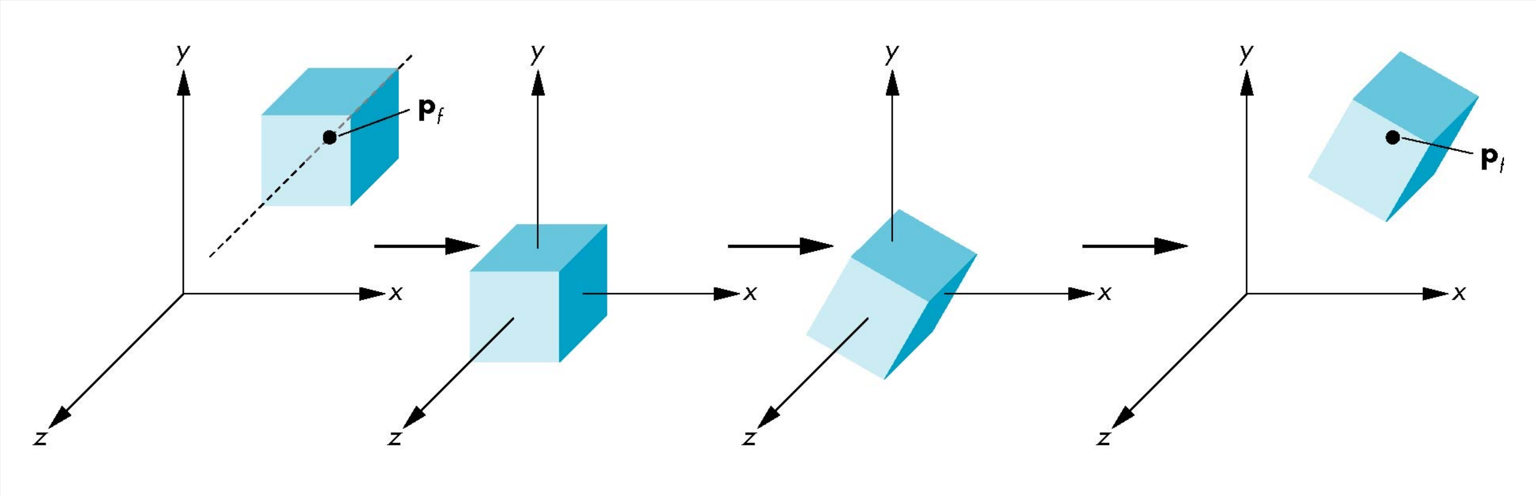
- In OpenGL, matrices were part of the state
- Multiple types
  - Model-View (**`GL_MODELVIEW`**)
  - Projection (**`GL_PROJECTION`**)
  - ….
- Single set of functions for manipulation
- Select which to manipulate by
  - **`glMatrixMode(GL_MODELVIEW);`**
  - **`glMatrixMode(GL_PROJECTION);`**
- All removed as of OpenGL 3.1

Modelview Matrix

- Modelview matrix $\boldsymbol{M}$ is a 4x4 homogeneous coordinate matrix
- Defined usually in the application as part of the state
- Applied (in the shader) to all vertices that pass down the pipeline



Rotation about a Fixed Point



$$M = T^{-1}RT$$

- Involves at least three 4x4 matrix multiplications
  - (may also need to compose R)
- Built only once (possibly in the application)
- Note that the last matrix operation specified (the rightmost) is the first transformation which effects vertices.

Rotation, Translation, Scaling

Create an identity matrix:

```
mat4 m = Identity();
```

You need to implement this general rotation function yourself; not defined in mat.h header file.

mat.h includes RotateX(), RotateY() and RotateZ() functions

Multiply on the right (whenever a transformation is needed):

```
mat4 r = Rotate(theta, vx, vy, vz)
m = m*r;
```

`theta` specifies angle in degrees (counter-clockwise)
`(vx, vy, vz)` defines axis of rotation

Do the same with translation and scaling:

```
mat4 s = Scale(sx, sy, sz)
mat4 t = Translate(dx, dy, dz);
m = m*s*t;
```

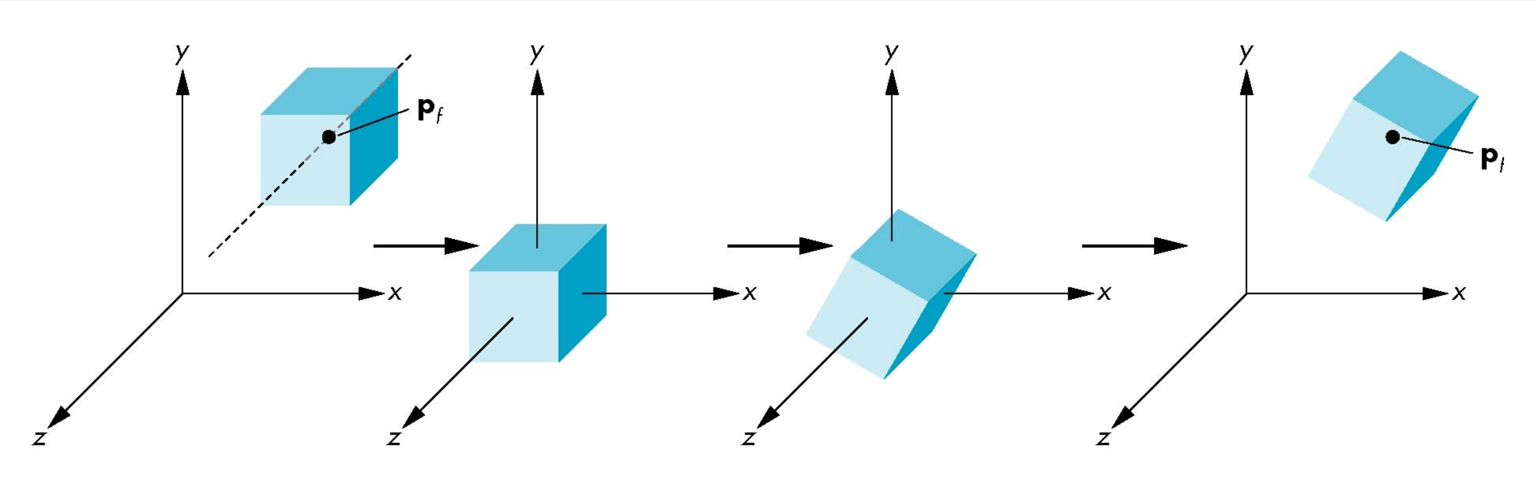
Example

- Rotation about z-axis by 30 degrees around a fixed point of (1.0, 2.0, 3.0)

```
m = Translate(1.0, 2.0, 3.0)*
    Rotate(30.0, 0.0, 0.0, 1.0)*
    Translate(-1.0, -2.0, -3.0);
```

you can use `RotateZ(30)`

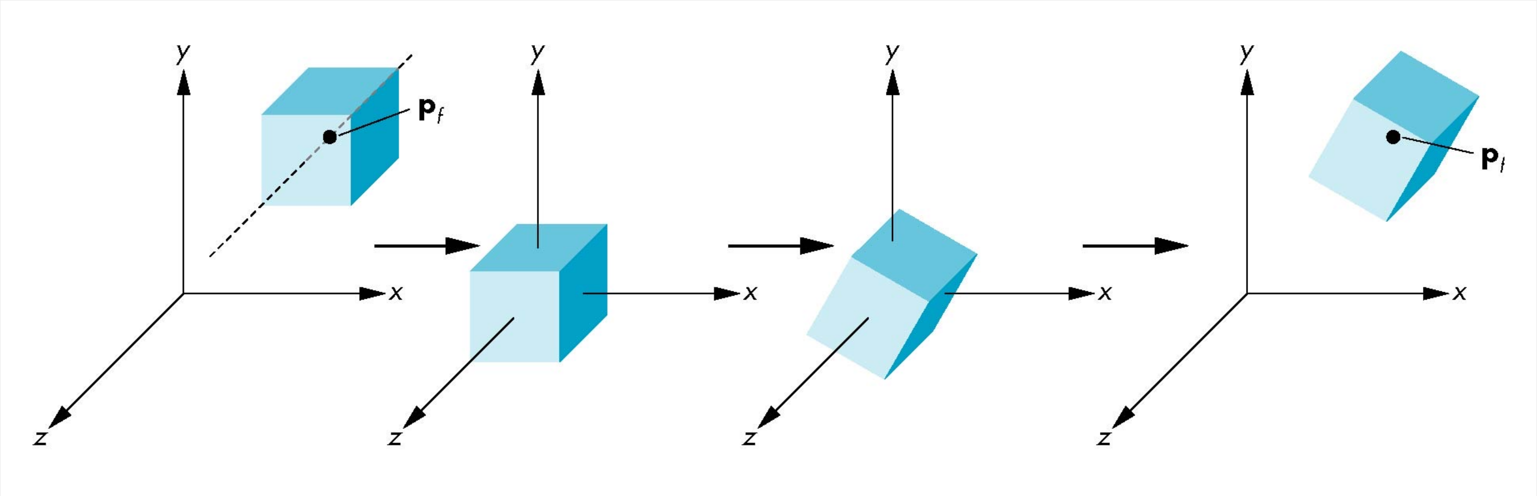
- Remember that last matrix specified in the program is the first applied



Rotation around a Fixed Point other than the Origin

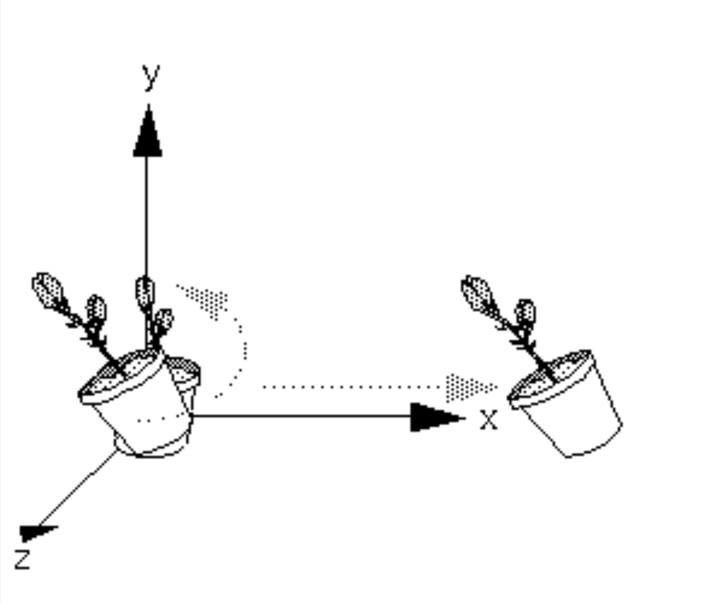
1. Move fixed point to origin
2. Rotate
3. Move fixed point back

$$\boldsymbol{M} = \boldsymbol{T}(\boldsymbol{p}_f)\, \boldsymbol{R}(\theta)\, \boldsymbol{T}(-\boldsymbol{p}_f)$$

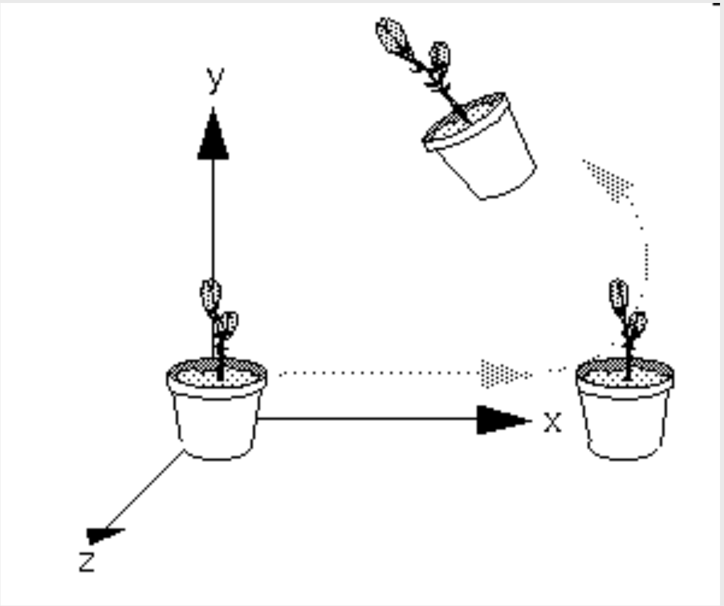


Order of Transformations

```
Translate(5.0, 0.0, 0.0)*
Rotate(45.0, 0.0, 0.0, 1.0)
```




```
Rotate(45.0, 0.0, 0.0, 1.0)*
Translate(5.0, 0.0, 0.0)
```



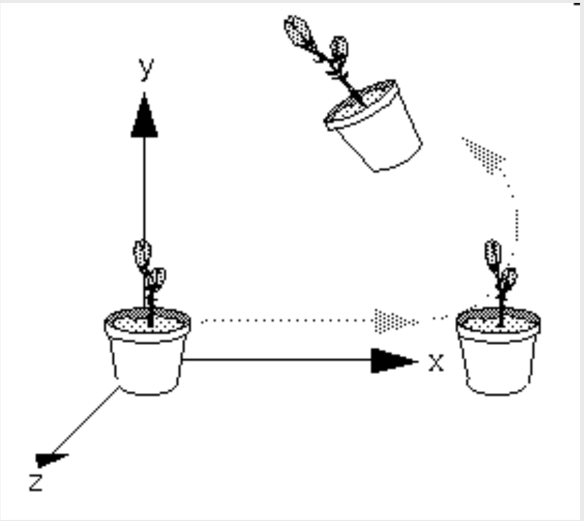
Thinking of Transformations

You can think of transformations in two different ways:

Think in terms of a local **world** coordinate system; first rotate then translate.

```
Rotate(45.0, 0.0, 0.0, 1.0)*
Translate(5.0, 0.0, 0.0)
```

Think in terms of a grand, fixed, **camera** coordinate system; first translate then rotate.



Manipulating Model-View Matrix

An example modified code fragment for rotation around fixed camera frame axes: (from display of the spin cube program):

```
model_view =  RotateX( theta[Xaxis] ) *
              RotateY( theta[Yaxis] ) *
              RotateZ( theta[Zaxis] ) * model_view;
```

Note that here **theta[0]**, **theta[1]**, and **theta[2]** are incremental rotation angles, and that the associated callback function sets one of them to a nonzero value depending on which axis to rotate:

```
void update()
{
  theta[0] = theta[1] = theta[2] = 0.0;
  theta[axis] = 2.0;
}
```

Where to form matrices? Application or Shader

- We can form modelview matrix in application and send to shader and let shader do the rotation
- Or, we could send the angle and axis to the shader and let the shader form the modelview matrix and then do the rotation

Modelview

```
void display( void )
{
….
    model_view =  RotateX( Theta[Xaxis])*
          RotateY( Theta[Yaxis] )*
          RotateZ( Theta[Zaxis] )* model_view;

    glUniformMatrix4fv( ModelView, 1, GL_TRUE,
model_view );


….

}
```

Application code

```
void main()
{
    gl_Position = Projection*Modelview*
          vPosition;
    …
}
```

Vertex shader

Using Model-View and Projection Matrices

- In OpenGL, the model-view matrix is used
  - to build and manipulate models of objects
  - to position the camera
    - can be done by rotations and translations but is often easier to use a **`LookAt()`** function such as the one in **`mat.h`**
- The projection matrix is used to define the view volume and to select the projection type
- Although these matrices are no longer part of the OpenGL state, we usually create them in our own applications
- Next lecture we will see how to create projection matrices