Homework Assignment #1

- Will be graded!
- 'Snowflake' 2D animation
  - Understanding polygon draw.
  - Data handling
  - Vertex Shader (transformations)
  - Window Handling & Timer
  - Creativity!!
- Due: **March 30 (Wednesday)** before midnight

Lecture & Lab

Tue/Thur			Wednesday	
Date	**Topic**	**Assignment**		**TA (LAB)**
Mar 8, 10	Introduction and HelloWorld 2D	HW #0	**9**	OpenGL Intro 1 (Simple 2D)
Mar 15, 17	Linear and Affine Transformation	HW #1	14	open lab
Mar 22, 24	Frames in Graphics		**23**	OpenGL Intro 2 (3D & viewing
Mar 29, 31	HelloWorld 3D, Projection	HW #2	30	open lab
Apr 5, 7	Depth	<transformation	6	open lab
Apr 12, 14	From Vertex to Pixels	w/ simple anim>	13	<Election day>
Apr 19	Geometric Modeling,		20	open lab
Apr 20~26	Midterm Exam			
Apr 28, May 3	Color and Shading		**4**	Lighting setup exercise
May 10, 12	Raytracing	HW #3	11	open lab
May 17, 19	Lighting	Shading/Lighting	18	open lab
May 24, 26	Texture Mapping		**25**	Texture mapping exercise
May 31, Jun 2	Sampling	HW #4	1	open lab
Jun 7, 9	Resampling	Texture mapping	8	open lab
Jun 14	Animation			
Jun 15~21	Final Exam			

Lab (E11: 307)

- Must come sessions
  - OpenGL Introduction 1 (3/9)
  - OpenGL Introduction 2 (3/23)
  - Shading/Lighting Session (5/4 tentative)
  - Texture mapping Session (5/25 tentative)

- Other Wednesdays 7~10 PM
  - Open lab
  - TA Help Hour
  - You may come and work on your homework and ask questions to TA

LAB Session Tomorrow

- Strongly Recommended!

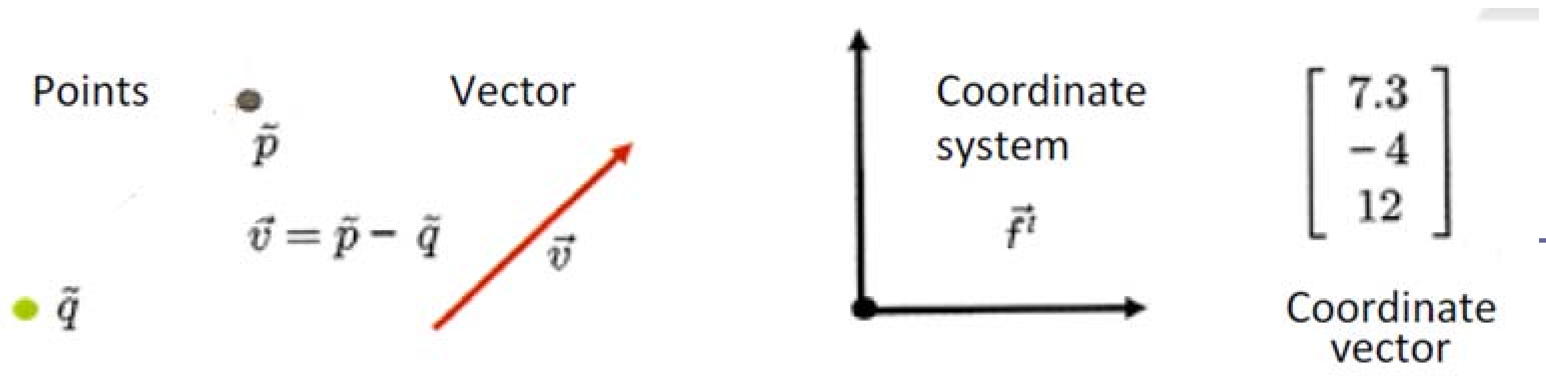
- Bring your notebook computer and power cord (just in case).

Last Week

Chaps 2 & 3: Linear & Affine Transformation

- Double buffering
- 4x4 Matrix (change of frames)
  - Frame defined by the basis vectors and a reference point
  - *Homogenous representations* of a point and a vector
- Affine transformation
  - Rotation, translation, scaling, sheering
  - Object transformation
  - Rigid-body transformation

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



$$\vec{v} = \sum_i^n c_i \vec{b}_i = \begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \vec{\mathbf{b}}^t \mathbf{c}$$

basis

Coordinate vector

- Linear transformation

$$\vec{v} \Rightarrow L(\vec{v}) = L\left(\sum_i c_i \vec{b}_i \right) = \sum_i c_i L(\vec{b}_i)$$

- 3-by-3 matrix:

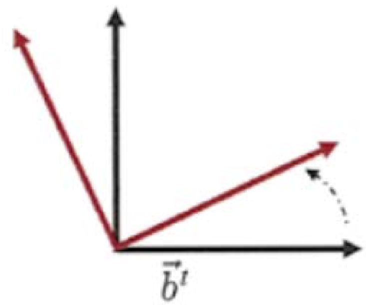
$$\begin{bmatrix} L(\vec{b}_1) & L(\vec{b}_2) & L(\vec{b}_3) \end{bmatrix} = \begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \end{bmatrix} \begin{bmatrix} M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,1} & M_{3,2} & M_{3,3} \end{bmatrix}$$

- Putting all together:

$$\begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \Rightarrow \begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \end{bmatrix} \begin{bmatrix} M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,1} & M_{3,2} & M_{3,3} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

- A matrix to transform one vector to another:

$$\vec{v} = \vec{\mathbf{b}}^t \mathbf{c} \Rightarrow \underset{a}{\boxed{\vec{\mathbf{b}}^t M}}\mathbf{c}$$



- change a basis of a vector $\vec{\mathbf{b}}^t$ to $\vec{\mathbf{a}}^t$

$$\vec{\mathbf{a}}^t = \vec{\mathbf{b}}^t M \,, \qquad \vec{v} = \vec{\mathbf{b}}^t \mathbf{c} = \vec{\mathbf{a}}^t M^{-1} \mathbf{c}\,.$$

- Linear transform of a vector

$$\vec{v} = \vec{\mathbf{b}}^t \mathbf{c} \Rightarrow \vec{\mathbf{b}}^t M \mathbf{c}$$

- Linear transform of a basis

$$\vec{v} = \vec{\mathbf{b}}^t \mathbf{c} = \vec{\mathbf{a}}^t M^{-1} \mathbf{c}\,.$$

- Movement of a point (original $\tilde{o}$ → a point $\tilde{p}$)

$$\tilde{p} = \tilde{o} + \vec{v}.$$

$$\tilde{p} = \tilde{o} + \sum_i c_i \vec{b}_i = \begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 & \tilde{o} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}.$$

- Affine frame (made of three vectors and a point):

$$\tilde{p} = \vec{\mathbf{f}}^t \mathbf{c}.$$

$$\begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 & \tilde{o} \end{bmatrix} = \vec{\mathbf{f}}^t$$

Transforming a point:

$$\tilde{p} = \vec{\mathbf{f}}^t \mathbf{c} \Rightarrow \vec{\mathbf{f}}^t A \mathbf{c}$$

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Linear transformation

- 3-by-3 transform matrix → 4-by-4 affine transform

$$\begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 & \tilde{o} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix} \Rightarrow$$

$$\begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 & \tilde{o} \end{bmatrix} \begin{bmatrix} a & b & c & 0 \\ e & f & g & 0 \\ i & j & k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}.$$

Translation transformation

- translation transformation to points

$$\begin{bmatrix} \vec{b_1} & \vec{b_2} & \vec{b_3} & \tilde{o} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} \vec{b_1} & \vec{b_2} & \vec{b_3} & \tilde{o} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}.$$

Affine transform matrix

Note: $TL \neq LT$

- An affine matrix can be factored into a linear part and a translational part:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & h \\ 0 & 0 & 1 & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & b & c & 0 \\ e & f & g & 0 \\ i & j & k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\begin{bmatrix} l & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} i & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} l & 0 \\ 0 & 1 \end{bmatrix} \qquad A = TL$$

Rigid body transformation

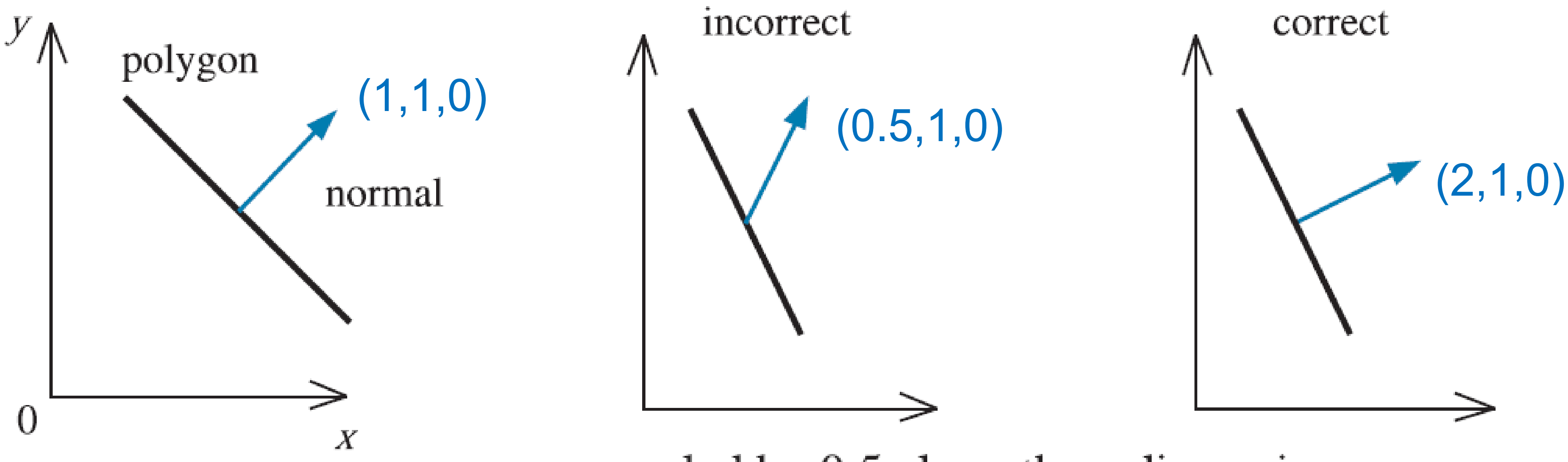
- When the linear transform is a rotation, we call this as rigid body transformation (rotation + translation only).

$$A = TR$$

- A rigid body transformation preserves dot product between vectors, handedness of a basis, and distance between points.
- Its geometric topology is maintained while transforming it.

Normal transforms

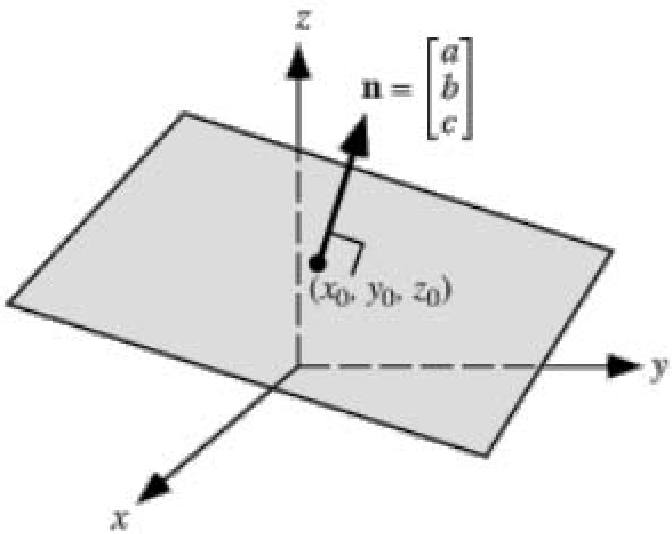
- We can see this in the following diagram, where the normal is incorrect if the same transformation is applied to both the geometry and normals.
- What's wrong with using the model transformation matrix to move our normals into world space?

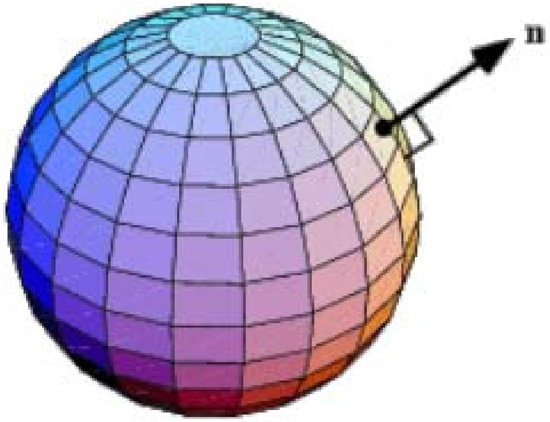


scaled by 0.5 along the *x* dimension

Normals

- Normal: a vector that is orthogonal to the tangent plane of the surfaces at that point.
  - the tangent plane is a plane of vectors that are defined by subtracting (infinitesimally) nearby surface points: $\vec{n} \cdot (\tilde{p}_1 - \tilde{p}_2) = 0$



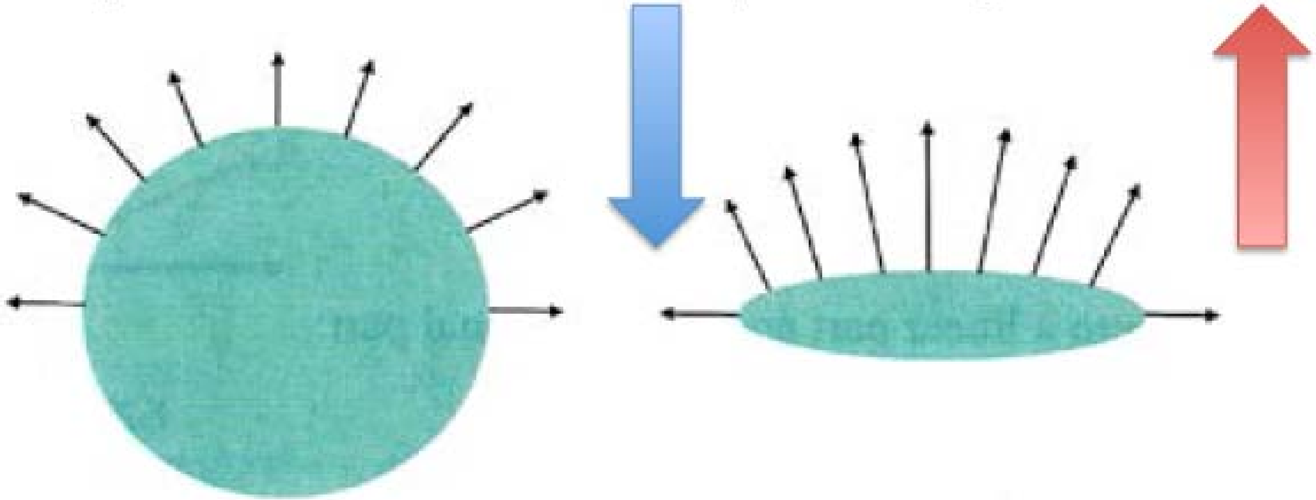


Normals

- We use normals for shading
- how do they transform
- suppose i rotate forward
  - normal gets rotated forward
- suppose squash in the y direction

Changing a shape

- Squashing a sphere makes its normals stretch along the *y* axis instead of squashing.



- normal gets higher in the y direction
- what is the rule?

$$\begin{bmatrix} nx \\ ny \\ nz \end{bmatrix} \neq \begin{bmatrix} nx' \\ ny' \\ nz' \end{bmatrix}.$$

Transforming normals

- Since the normal $\vec{n}$ and very close points $\tilde{p}_1$ and $\tilde{p}_2$ are on a surface: $\vec{n}\cdot(\tilde{p}_1-\tilde{p}_2)=0$

$$\begin{bmatrix} nx & ny & nz & * \end{bmatrix}\left(\begin{bmatrix} x1 \\ y1 \\ z1 \\ 1 \end{bmatrix}-\begin{bmatrix} x0 \\ y0 \\ z0 \\ 1 \end{bmatrix}\right)=0 .$$

- After applying an affine transform A,

the normal of the transformed geometry

$$\left(\begin{bmatrix} nx & ny & nz & * \end{bmatrix}A^{-1}\right)\left(A\left(\begin{bmatrix} x1 \\ y1 \\ z1 \\ 1 \end{bmatrix}-\begin{bmatrix} x0 \\ y0 \\ z0 \\ 1 \end{bmatrix}\right)\right)=0 .$$

Transforming normals

- Transformed normals:

$$\begin{bmatrix} nx' & ny' & nz' \end{bmatrix} = \begin{bmatrix} nx & ny & nz \end{bmatrix} I^{-1}.$$

- Transposing this expression:

$$\begin{bmatrix} nx' \\ ny' \\ nz' \end{bmatrix} = I^{-t} \begin{bmatrix} nx \\ ny \\ nz \end{bmatrix}.$$

Transforming normals

- Remember l is a rotation matrix (orthonormal), thus its inverse transpose is the same as the original: $l^{-t} = l$.

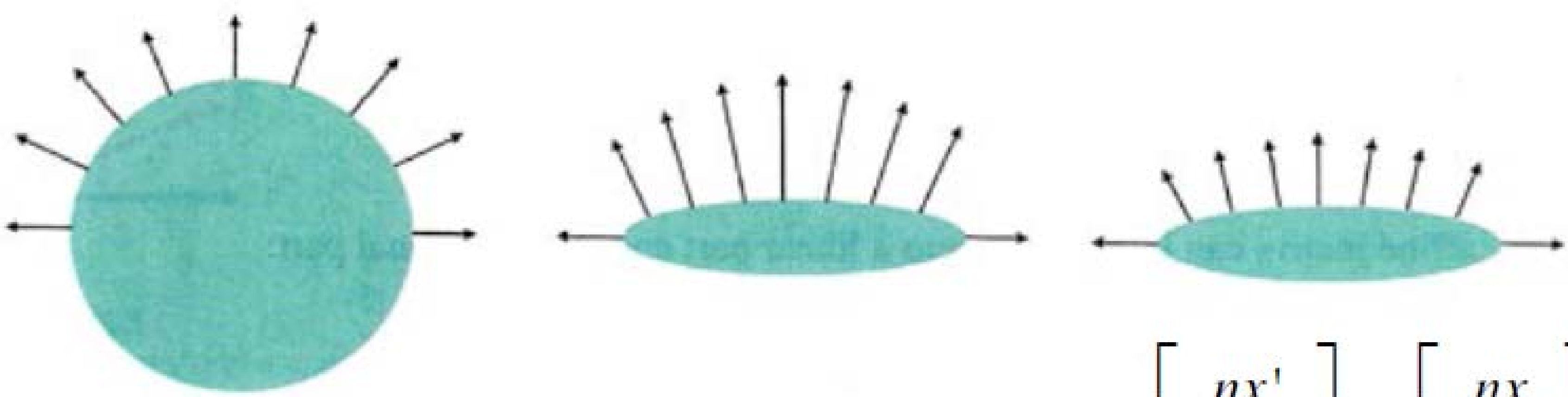
$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$LL^t = I \; (L^t = L^{-1}), \det L = 1$$

- inverse transpose
  - so inverse transpose/transpose inverse is the rule
  - for rotation, transpose = inverse
  - for scale, transpose = nothing

Transforming normals

- Renormalize to correct unit normals of squashed shape:



$$l^{-1}\begin{bmatrix} nx' \\ ny' \\ nz' \end{bmatrix} \Rightarrow \begin{bmatrix} nx \\ ny \\ nz \end{bmatrix}.$$

Normal transforms

- The correct way to calculate the normal
  - apply the same rotation transformation, (inverse of a rotation matrix is its transpose,)
  - but invert the scale (its inverse inverts the scale factors, and transposition has no effect)
  - translation can safely be ignored as it will not affect the normal vector
- So, transpose of the inverse of the model transformation matrix!

- But, the inverse of a matrix is not always guaranteed to exist.
  - inverse of a matrix, A
    = the adjoint of A over the determinant of A,
  - the adjoint of a matrix is guaranteed to exist
- → We can use the adjoint instead of the inverse, and then re-normalize the vector.

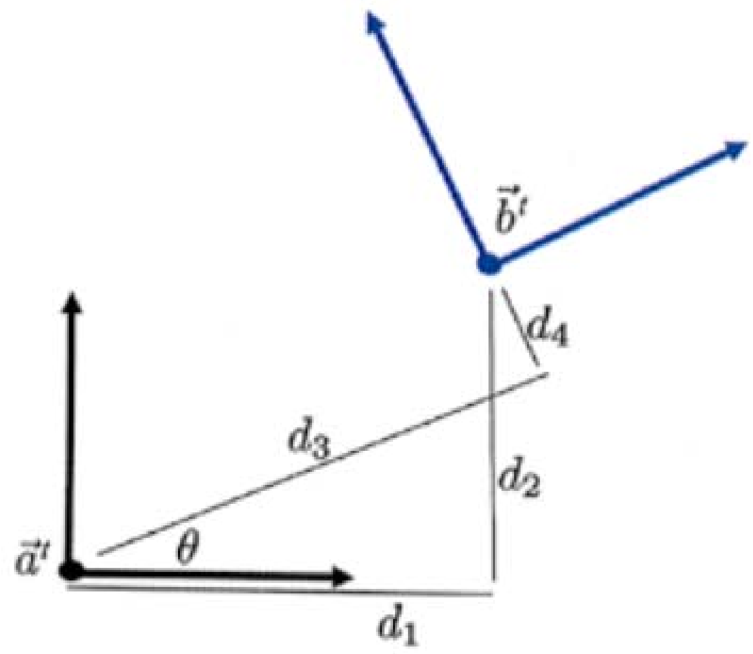
Chapter 4. Respect

- Frame is important …

Chapter 5. Frames in Graphics

Chapter 4

RESPECT



Scaling a point over frame

- We are transforming a point $\tilde{p}$ in a frame $\vec{\mathbf{f}}^t$

$$\tilde{p} = \vec{\mathbf{f}}^t \mathbf{c}$$

- With a matrix

$$\mathbf{S} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

the stretches by factor of two in first axis of $\vec{\mathbf{f}}^t$

- Performing a transform: $\vec{\mathbf{f}}^t \mathbf{c} \Rightarrow \vec{\mathbf{f}}^t S \mathbf{c}$

- Suppose another frame: $\vec{\mathbf{a}}^t = \vec{\mathbf{f}}^t A$

Scaling a point over frame

- We could express the point with a new coordinate vector

$$\tilde{p} = \vec{\mathbf{f}}^t \mathbf{c} = \vec{\mathbf{a}}^t \mathbf{d} \qquad \vec{\mathbf{a}}^t = \vec{\mathbf{f}}^t A$$

$$\vec{\mathbf{f}}^t \mathbf{c} = \vec{\mathbf{f}}^t A \mathbf{d} \qquad \vec{\mathbf{f}}^t = \vec{\mathbf{a}}^t A^{-1}$$

$$\mathbf{d} = A^{-1} \mathbf{c}$$

- Now S transforms the point $\tilde{p}$ with respect to $\vec{\mathbf{a}}^t$

$$\vec{\mathbf{a}}^t \mathbf{d} \Rightarrow \vec{\mathbf{a}}^t S \mathbf{d}$$

Left-of rule

- Point is transformed with respect to the the frame that appears immediately to the left of the transformation matrix in the expression.
- We read

$$\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t S$$

$\vec{\mathbf{f}}^t$ is transformed by S with respect to $\vec{\mathbf{f}}^t$

- We read

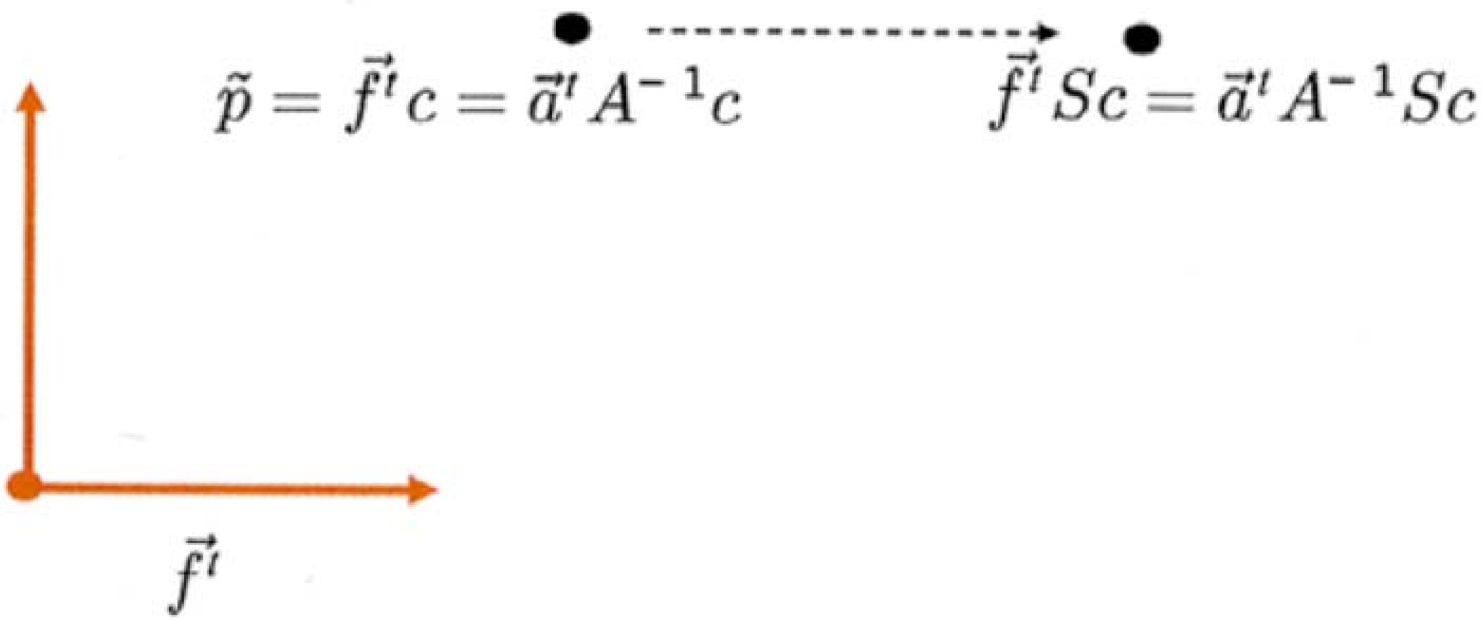
$$\vec{\mathbf{f}}^t = \vec{\mathbf{a}}^t A^{-1} \Rightarrow \vec{\mathbf{a}}^t S A^{-1}$$

$\vec{\mathbf{f}}^t$ is transformed by S with respect to $\vec{\mathbf{a}}^t$

Scaling a point over frame

$$\tilde{p} = \vec{\mathbf{f}}^t \mathbf{c} \Rightarrow \vec{\mathbf{f}}^t S \mathbf{c}$$

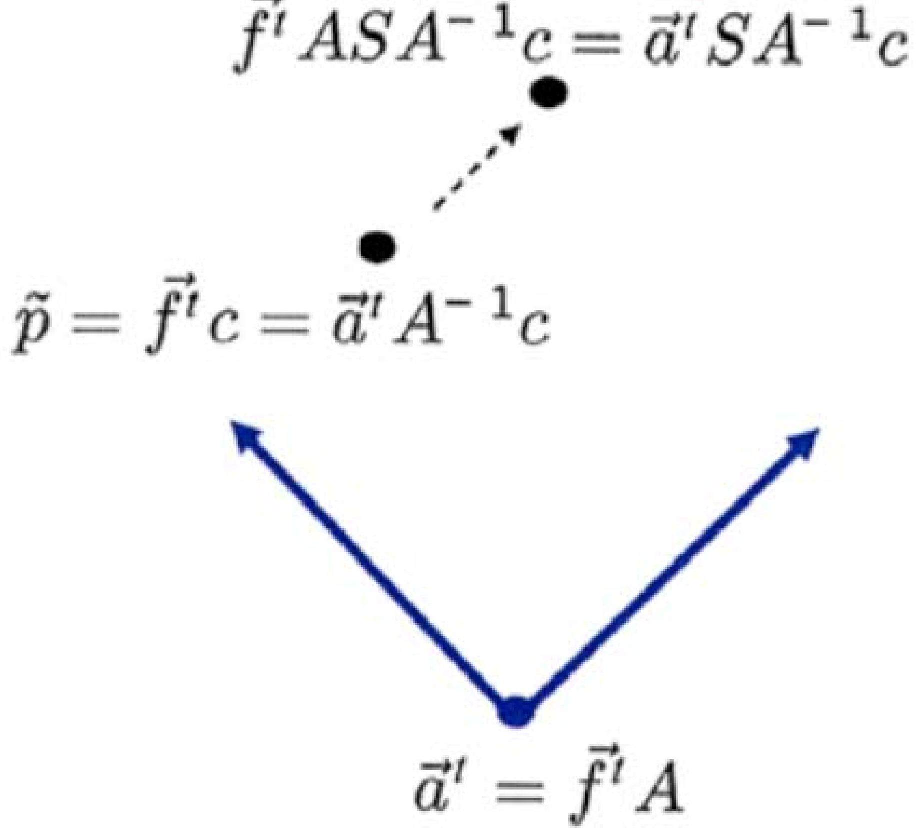
$\tilde{p}$ is transformed by S with respect to $\vec{\mathbf{f}}^t$



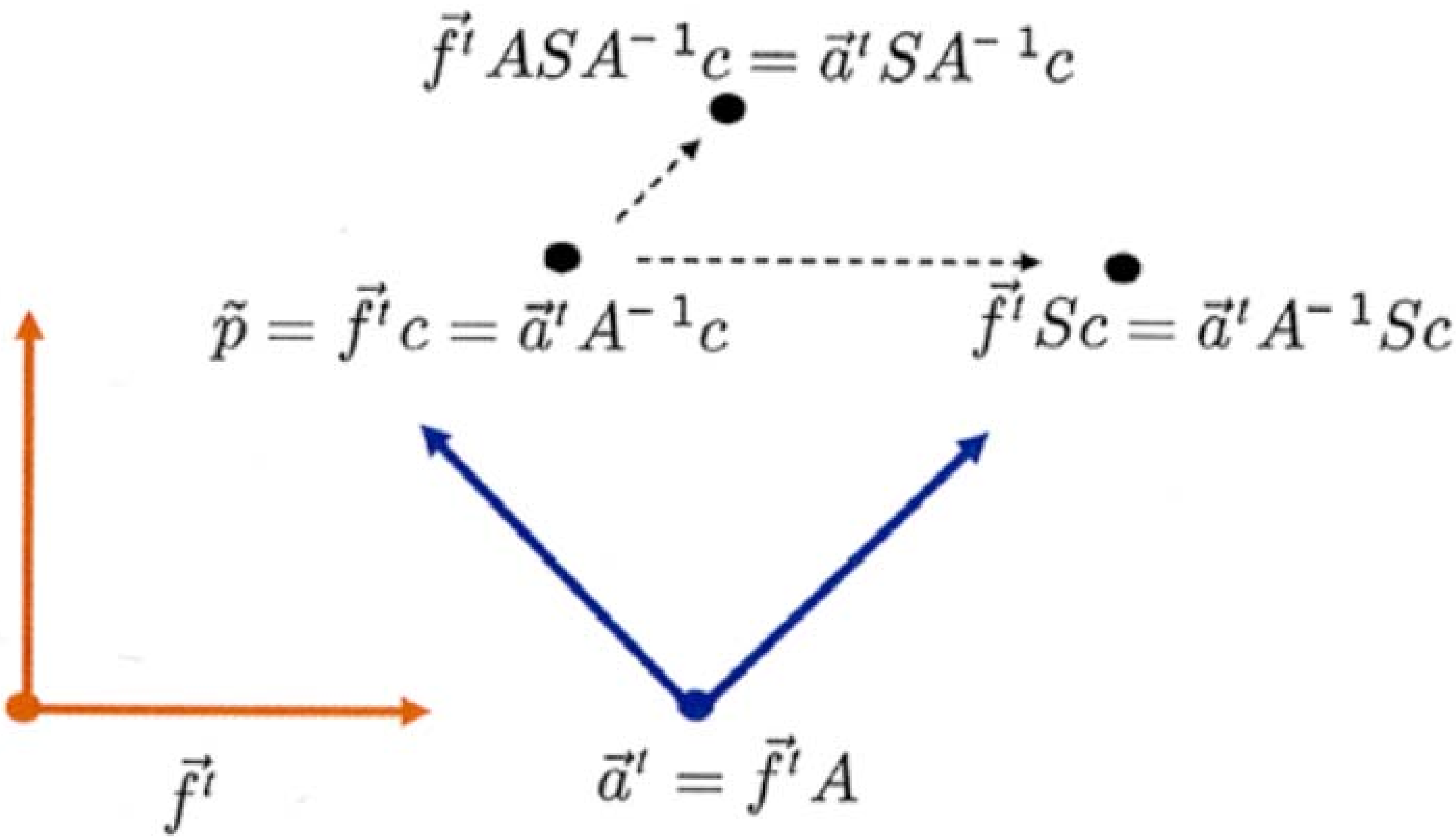
Scaling a point over frame

$\tilde{p} = \vec{\mathbf{a}}^t A^{-1}\mathbf{c} \Rightarrow \vec{\mathbf{a}}^t S A^{-1}\mathbf{c}$

$\tilde{p}$ is transformed by S with respect to $\vec{\mathbf{a}}^t$



Scaling a point over frame



Rotating a point over frame

- The same reasoning to transformations of frames themselves:

$$\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t R$$

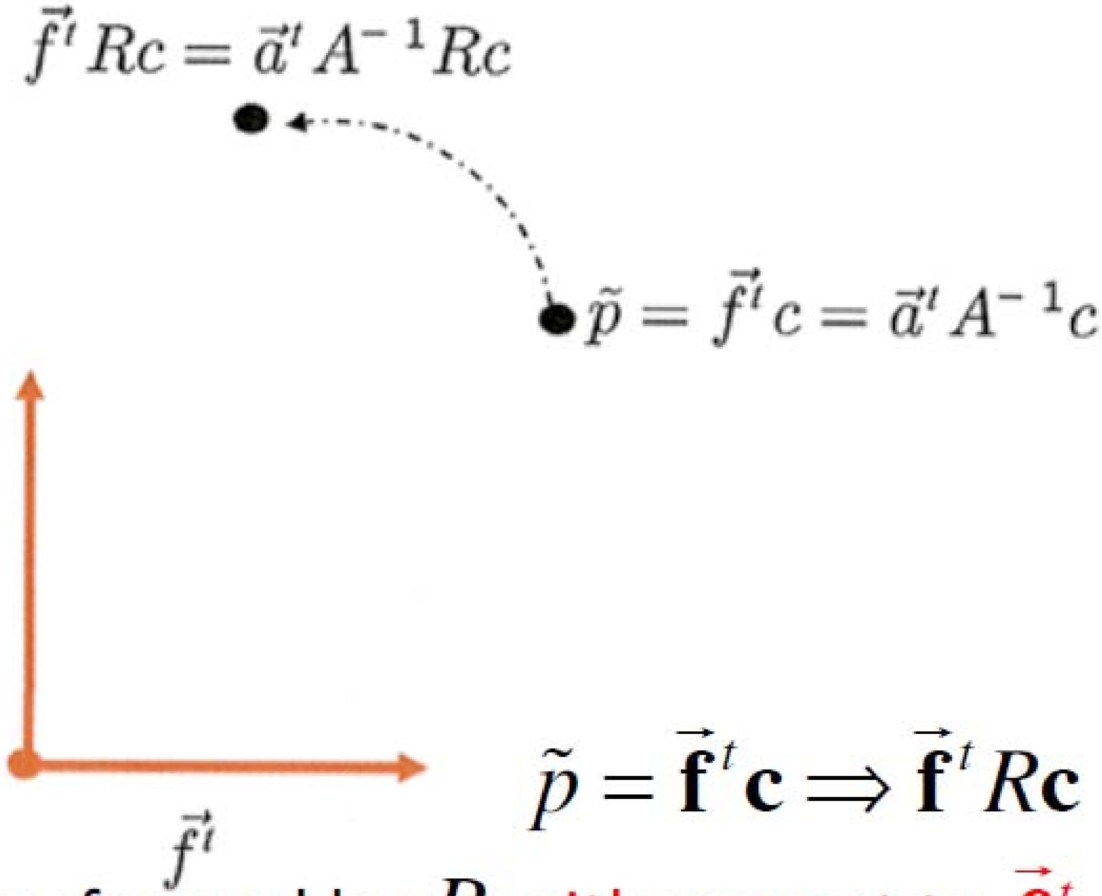
$\vec{\mathbf{f}}^t$ is transformed by R with respect to $\vec{\mathbf{f}}^t$

- In another frame:

$$\vec{\mathbf{f}}^t = \vec{\mathbf{a}}^t A^{-1} \Rightarrow \vec{\mathbf{a}}^t R A^{-1}$$

$\vec{\mathbf{f}}^t$ is transformed by R with respect to $\vec{\mathbf{a}}^t$

Rotating a point over frame



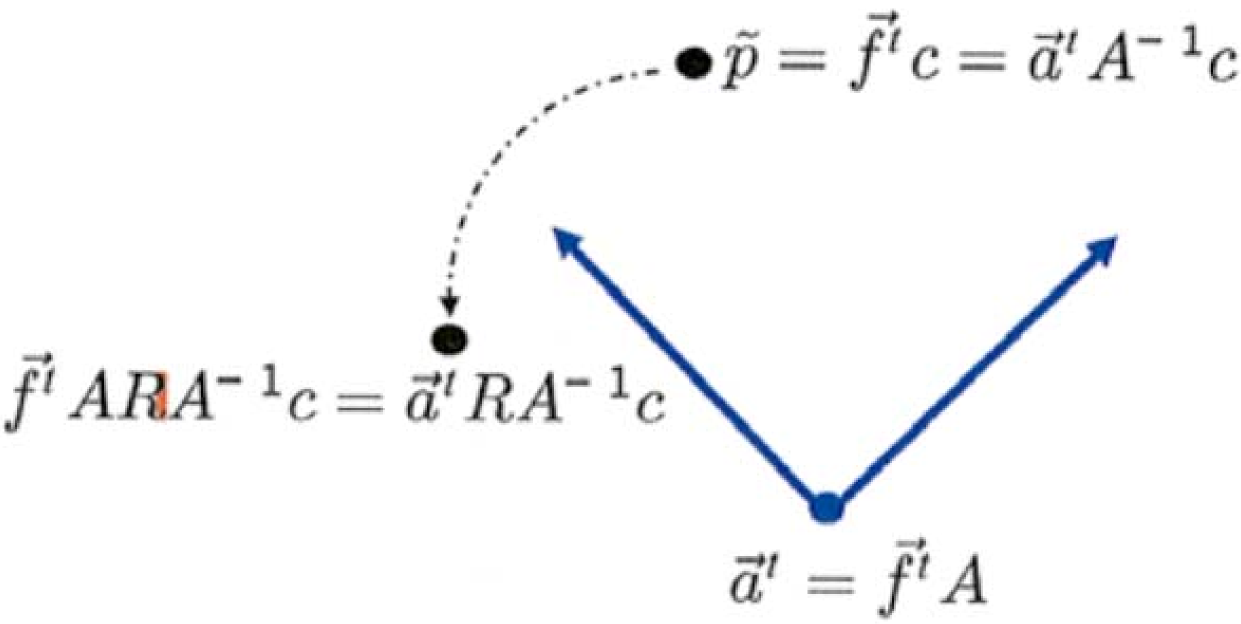
$$\tilde{p} = \vec{\mathbf{f}}^t \mathbf{c} \Rightarrow \vec{\mathbf{f}}^t R\mathbf{c}$$

$\tilde{p}$ is transformed by R with respect to $\vec{\mathbf{f}}^t$

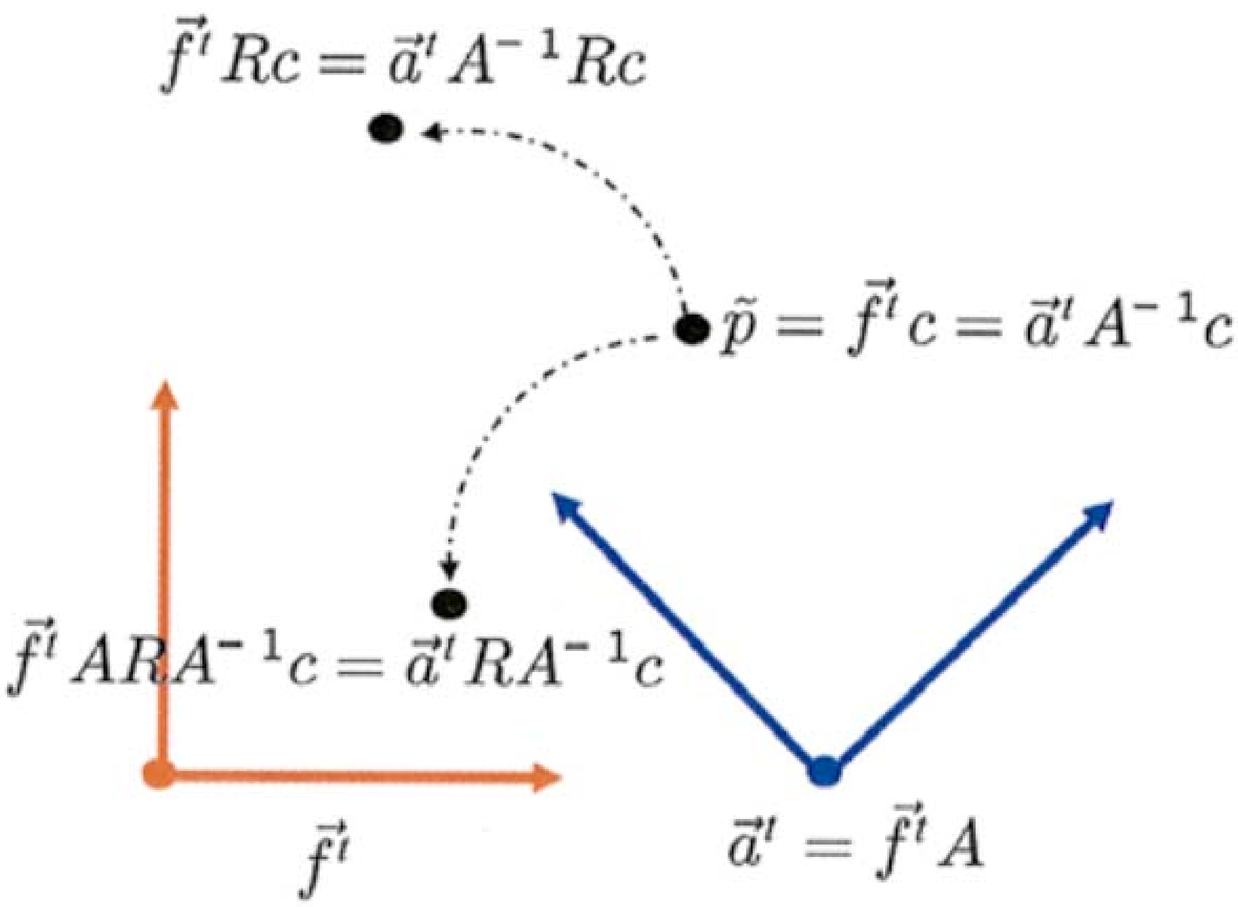
Rotating a point over frame

$$\tilde{p} = \vec{\mathbf{a}}^t A^{-1}\mathbf{c} \Rightarrow \vec{\mathbf{a}}^t R A^{-1}\mathbf{c}$$

$\tilde{p}$ is transformed by R with respect to $\vec{\mathbf{a}}^t$

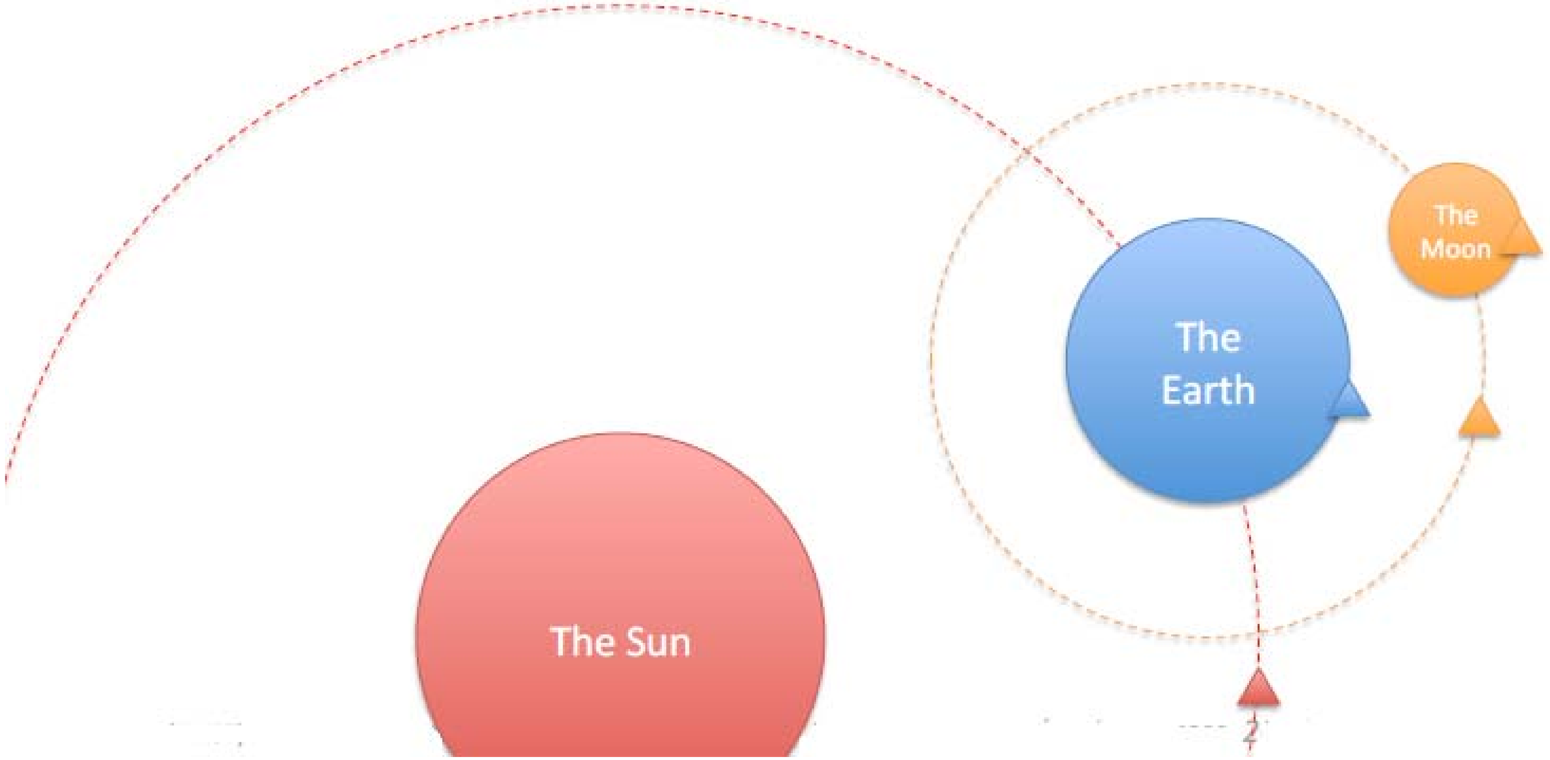


Rotating a point over frame



Auxiliary Frame

- You want to build the solar system
  - The Moon rotates around the Earth's frame
  - The Earth rotates around the Sun's frame



Transforms using an Auxiliary Frame

- Sometimes we need to transform a frame $\vec{\mathbf{f}}^t$ in some specific way, represented by a matrix M, with respect to some auxiliary frame $\vec{\mathbf{a}}^t$

$$\vec{\mathbf{a}}^t \Rightarrow \vec{\mathbf{f}}^t A$$

- The transform frame can then be expressed as

$$\vec{\mathbf{f}}^t$$

$$= \vec{\mathbf{a}}^t A^{-1}$$

$$\Rightarrow \vec{\mathbf{a}}^t M A^{-1}$$

$$= \vec{\mathbf{f}}^t A M A^{-1}$$

Multiple Transformations

- Rotation and translation with frame

  $\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t TR$

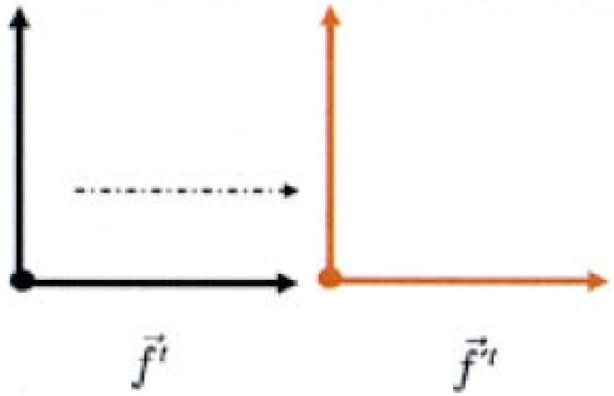
- in general, matrix multiplication is not commutative!!!

  $\vec{\mathbf{f}}^t TR \neq \vec{\mathbf{f}}^t RT$

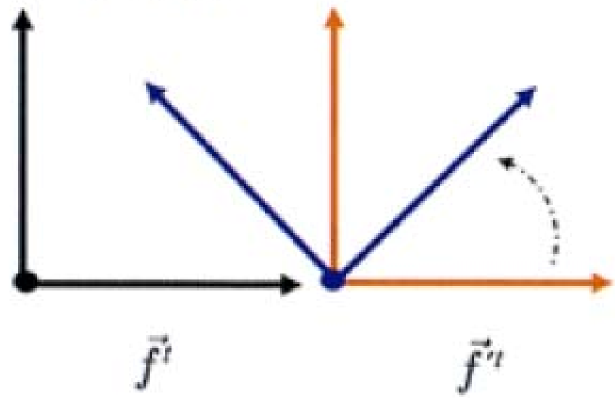
- There are two different ways to apply multiple transformations
  - Local transformation
  - Global transformation

Local Transformations

- Local transformations $\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t TR$



(a) Local translation



(a) Local rotation

- In the first step,

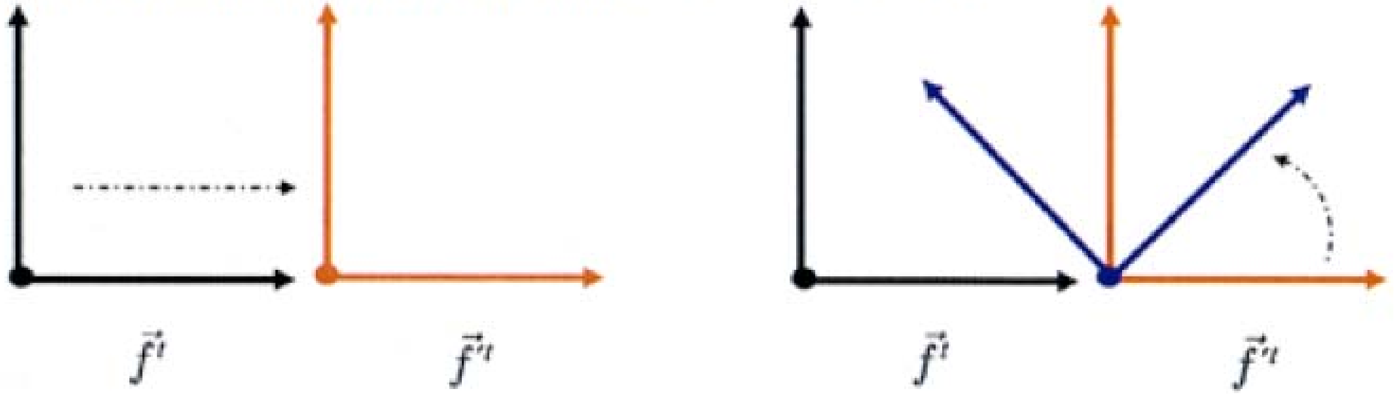
$\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t T = \vec{\mathbf{f}}'^t$

$\vec{\mathbf{f}}^t$ is transformed by T with respect to $\vec{\mathbf{f}}^t$

as the resulting frame: $\vec{\mathbf{f}}'^t$

Local Transformations

- Local transformations $\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t TR$



(a) Local translation (a) Local rotation

- In the second step,

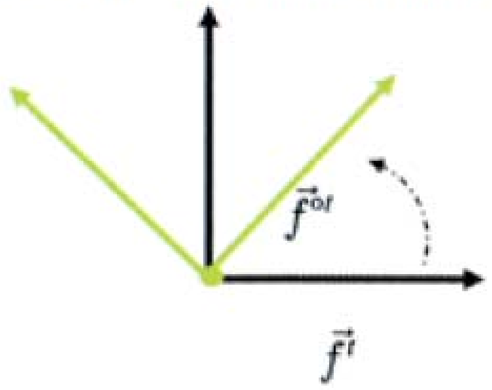
$\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t TR,$

$\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}'^t R.$

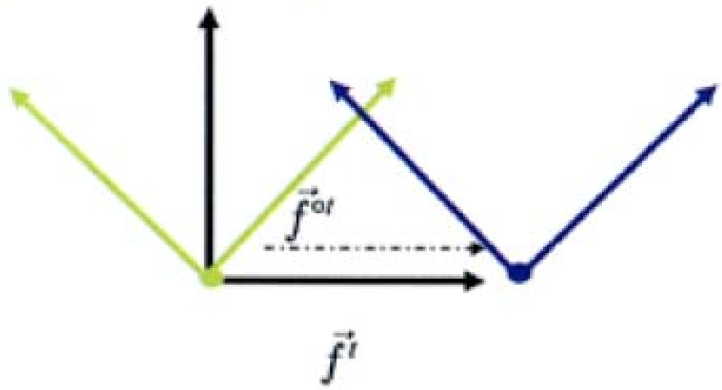
$\vec{\mathbf{f}}^t$ is transformed by R with respect to $\vec{\mathbf{f}}'^t$

Global Transformations

- Global transformations $\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t TR$



(c) Global rotation



(c) Global translation

- In the first step (in the reverse order)

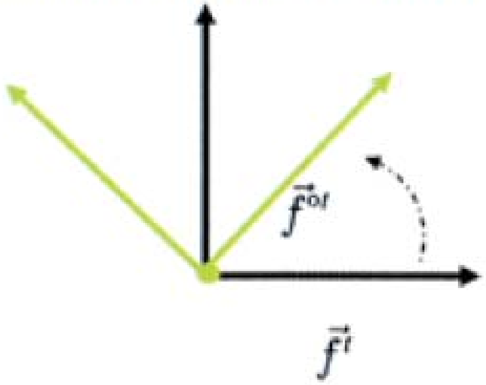
$\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t R = \vec{\mathbf{f}}^{\circ t}$

$\vec{\mathbf{f}}^t$ is transformed by R with respect to $\vec{\mathbf{f}}^t$

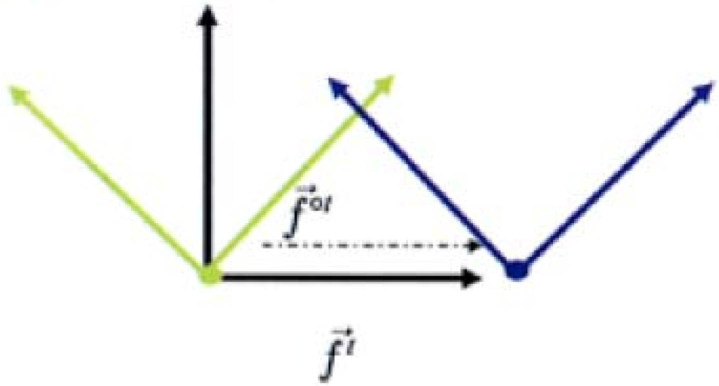
as the resulting frame: $\vec{\mathbf{f}}^{\circ t}$

Global Transformations

- Global transformations $\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t TR$



(c) Global rotation



(c) Global translation

- In the second step

$$\vec{\mathbf{f}}^{\circ t} = \vec{\mathbf{f}}^t R \Rightarrow \vec{\mathbf{f}}^t TR$$

$\vec{\mathbf{f}}^{\circ t}$ is transformed by T with respect to $\vec{\mathbf{f}}^t$

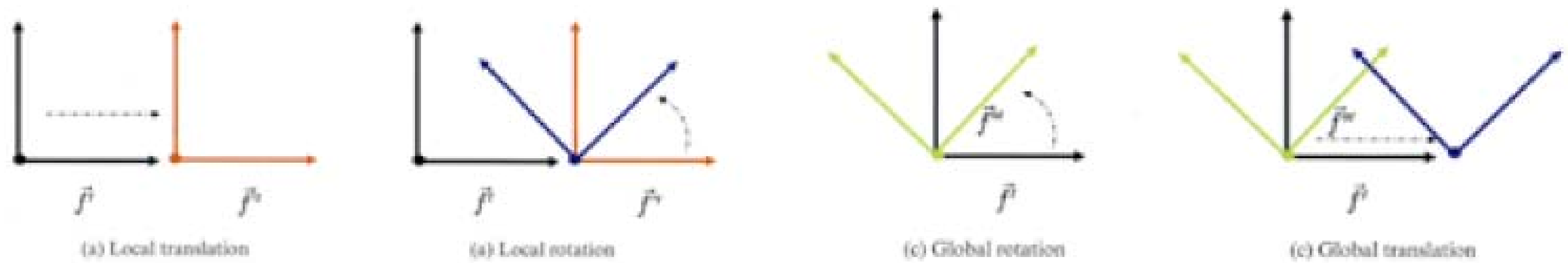
Two interpretations of transformations

- Two different ways for multiple transformations:
  1. (Local transformations) Translate with respect to $\vec{\mathbf{f}}^t$ then rotate with respect to the intermediate frame $\vec{\mathbf{f}}'^t$
  2. (Global transformations) Rotate with respect to $\vec{\mathbf{f}}^t$ then translate with respect to the original frame $\vec{\mathbf{f}}^t$

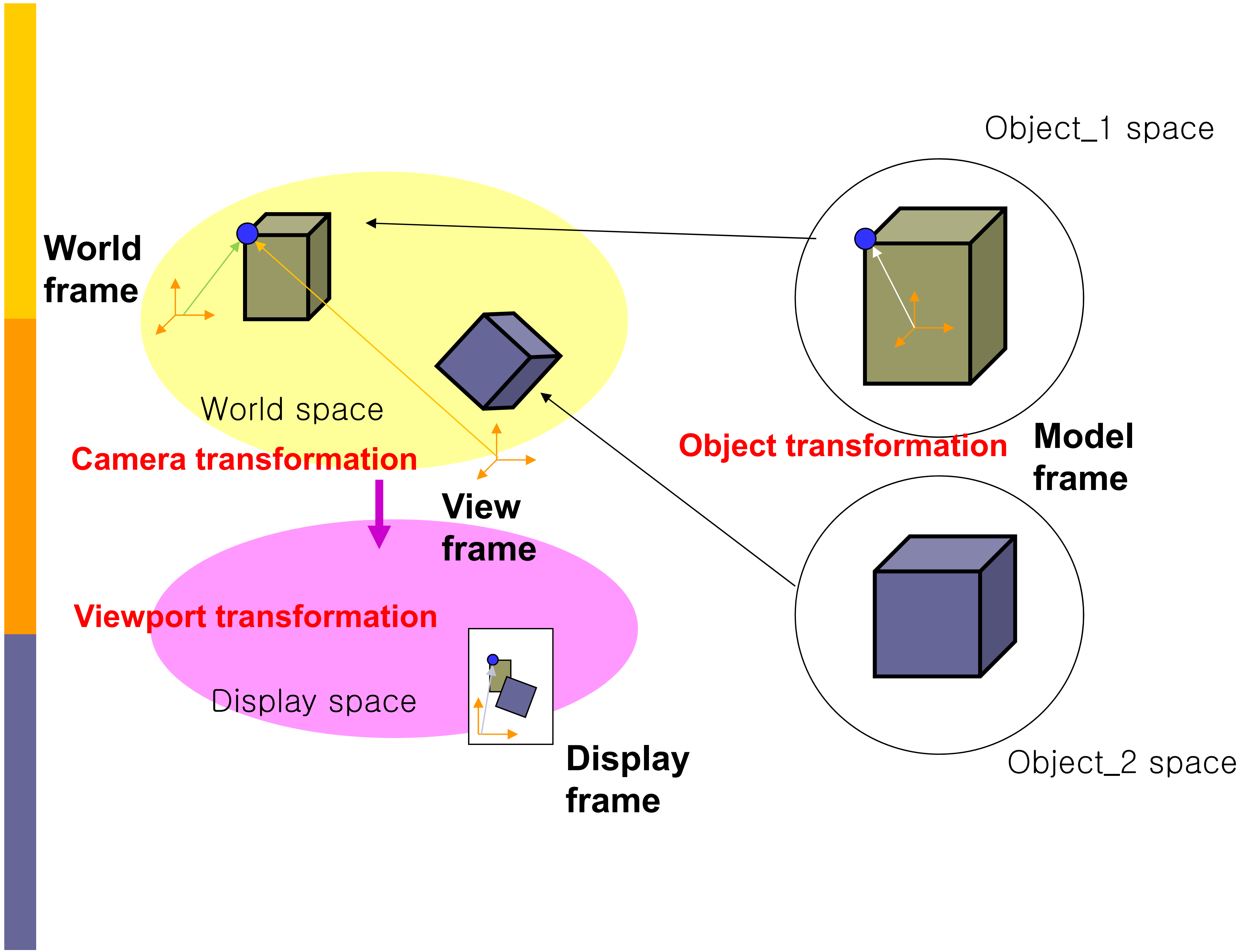
$$\vec{\mathbf{f}}^t \Rightarrow \vec{\mathbf{f}}^t TR$$

Local transformations

Global transformations



Chapter 5. Frames in Graphics


Object_1 space
World frame
World space
Camera transformation
Object transformation
Model frame
View frame
Viewport transformation
Display space
Display frame
Object_2 space

Transformation in OpenGL

- A simplified view of the OpenGL *fixed* pipeline.

vertices → **Model-view** → **Projection** → vertices

  - Each vertex passes through 2 transformations that are defined by the *current* model view and projection *matrices*, which are part of the OpenGL state.
  - Initially both are set to 4x4 identity matrices.

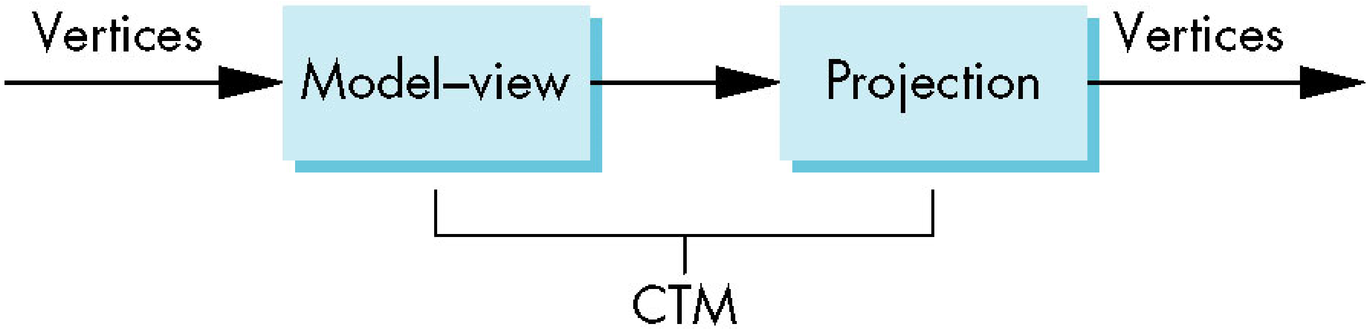
- Model-view matrix
  - is used to position objects relative to a camera.
- Projection matrix
  - forms the image through projection

- → We use same concept (thinking as in two stages, i.e., product of two matrices) in our vertex shader.

Transformation Matrix

- Current Transformation Matrix

- In (old) OpenGL
  - The matrix that is applied to all primitives is the product of the model-view matrix and the projection matrix.



Transformation Matrix

Old OpenGL functions

- $C \leftarrow I$
- $C \leftarrow C\,T$
- $C \leftarrow C\,S$
- $C \leftarrow C\,R$
- $C \leftarrow M$

- glLoadIdentity():
- glTranslatef(Tx, Ty, Tz);

  components of the displacement vector
- glScalef(Sx, Sy, Sz);

  scale factors along the coordinate axes
- glRotatef(angle, Vx, Vy, Vz);

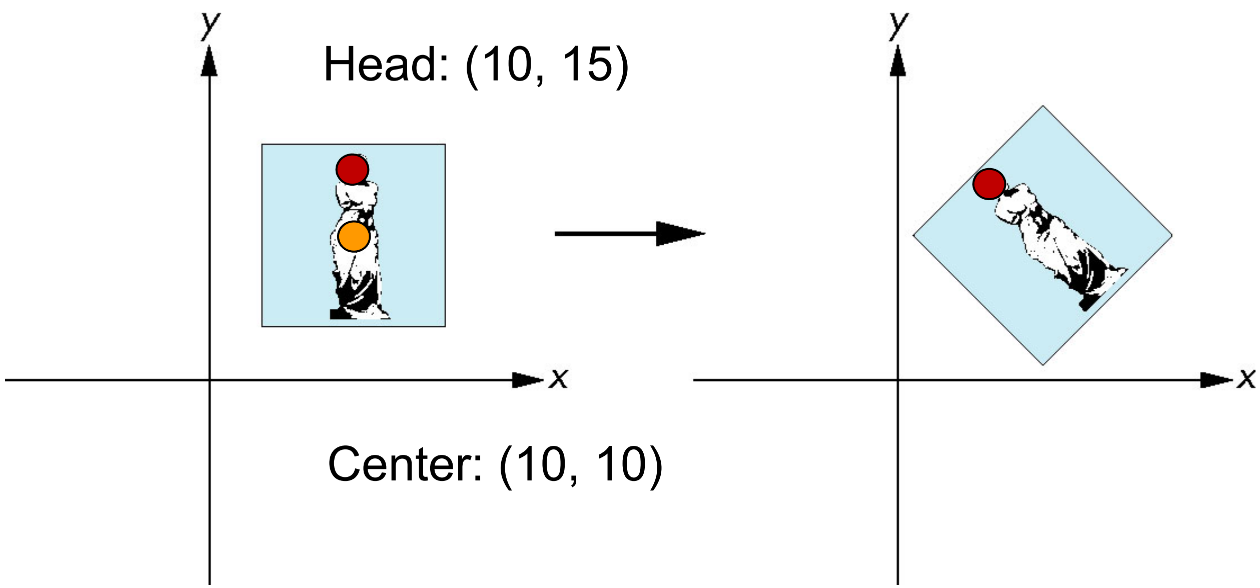
  angle in degrees,

  the component of the rotation vector
- glLoadMatrixf(ptr_to_matrix);

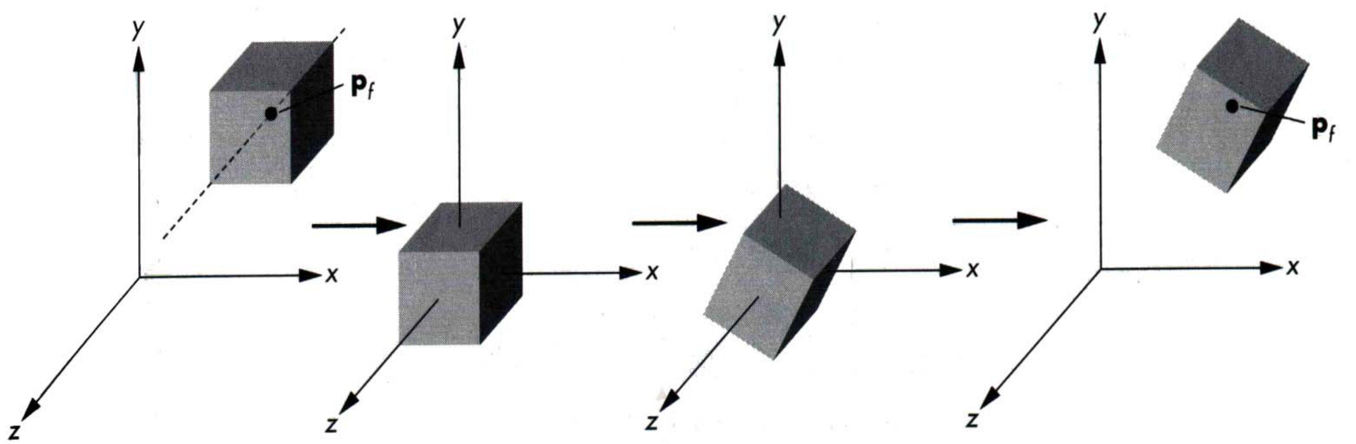
We will create these matrices on our own.

Quiz

1) Give a matrix M to transform the square rotated about its center = (10,10), 45 degree counter-clock wise.
2) Compute the new position of the head marked in red (10, 15) using your matrix M.



Example



$C = I$

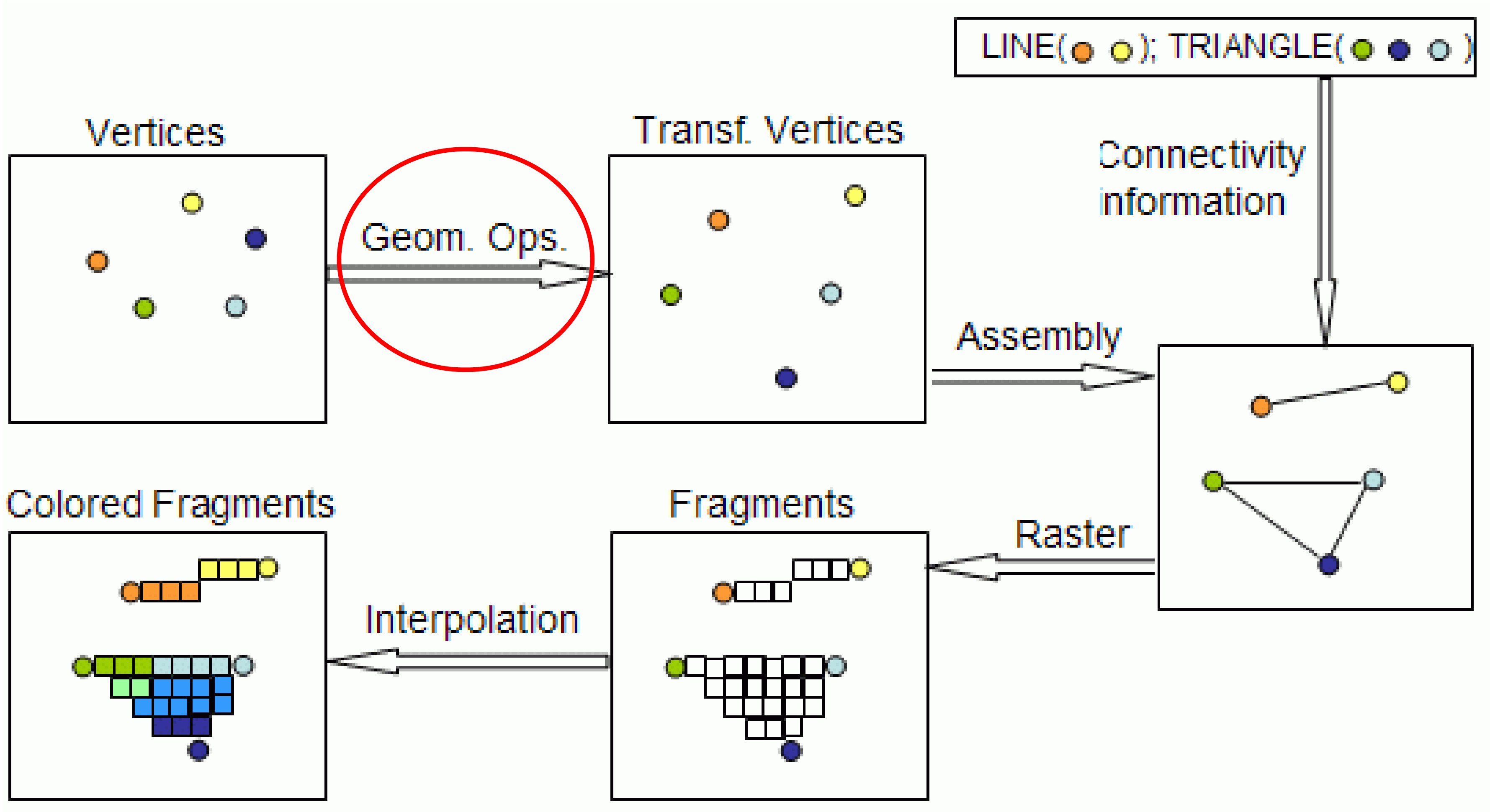
$C = T(P_f)$

$C = C\,R(\theta) = T(P_f)\,R(\theta)$

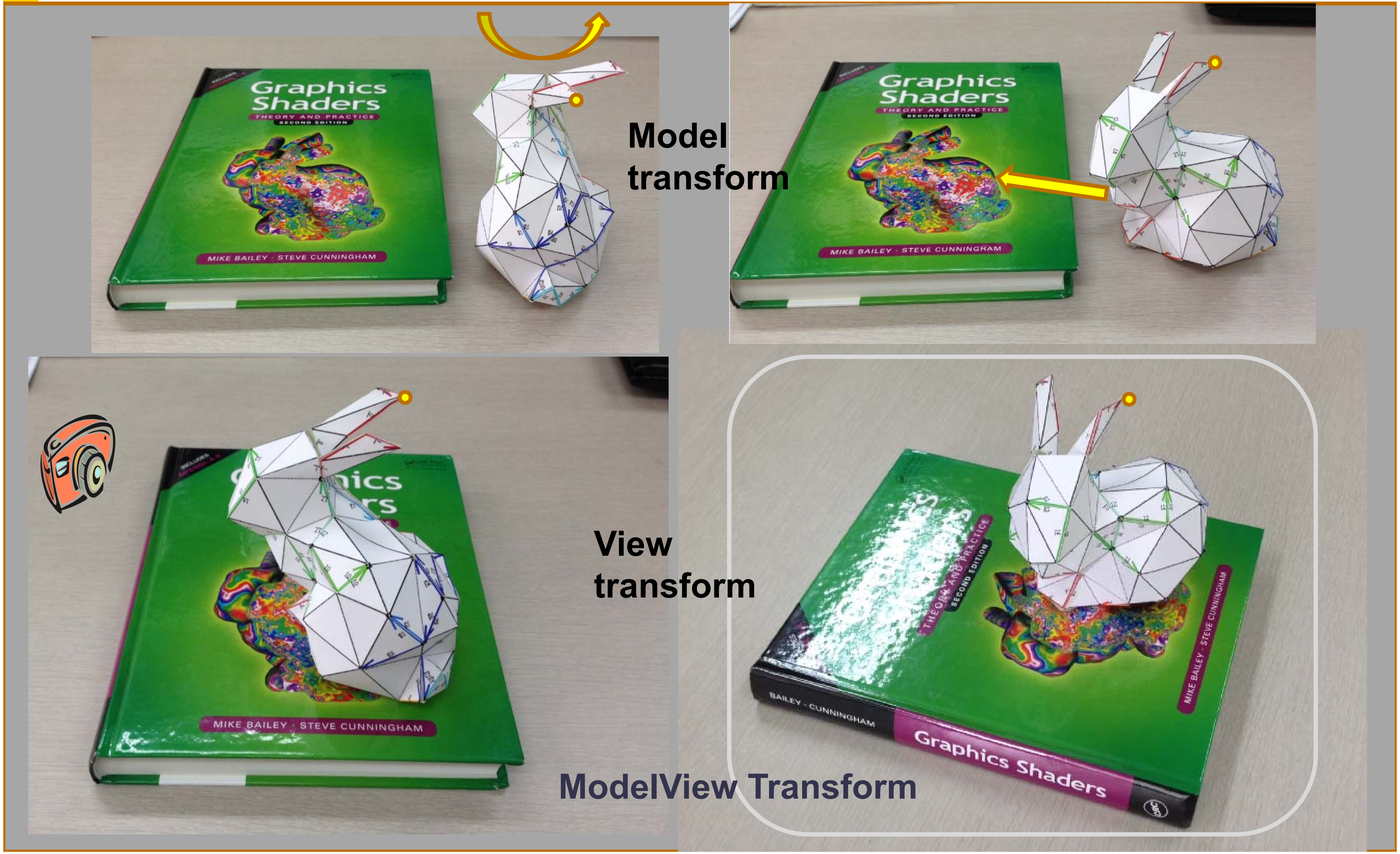
$C = C\,T(-P_f)$

$= T(P_f)\,R(\theta)\,T(-P_f)$

The transformation specified last is the one applied first!


LINE( ); TRIANGLE( )
Vertices
Geom. Ops.
Transf. Vertices
Connectivity information
Assembly
Colored Fragments
Fragments
Raster
Interpolation

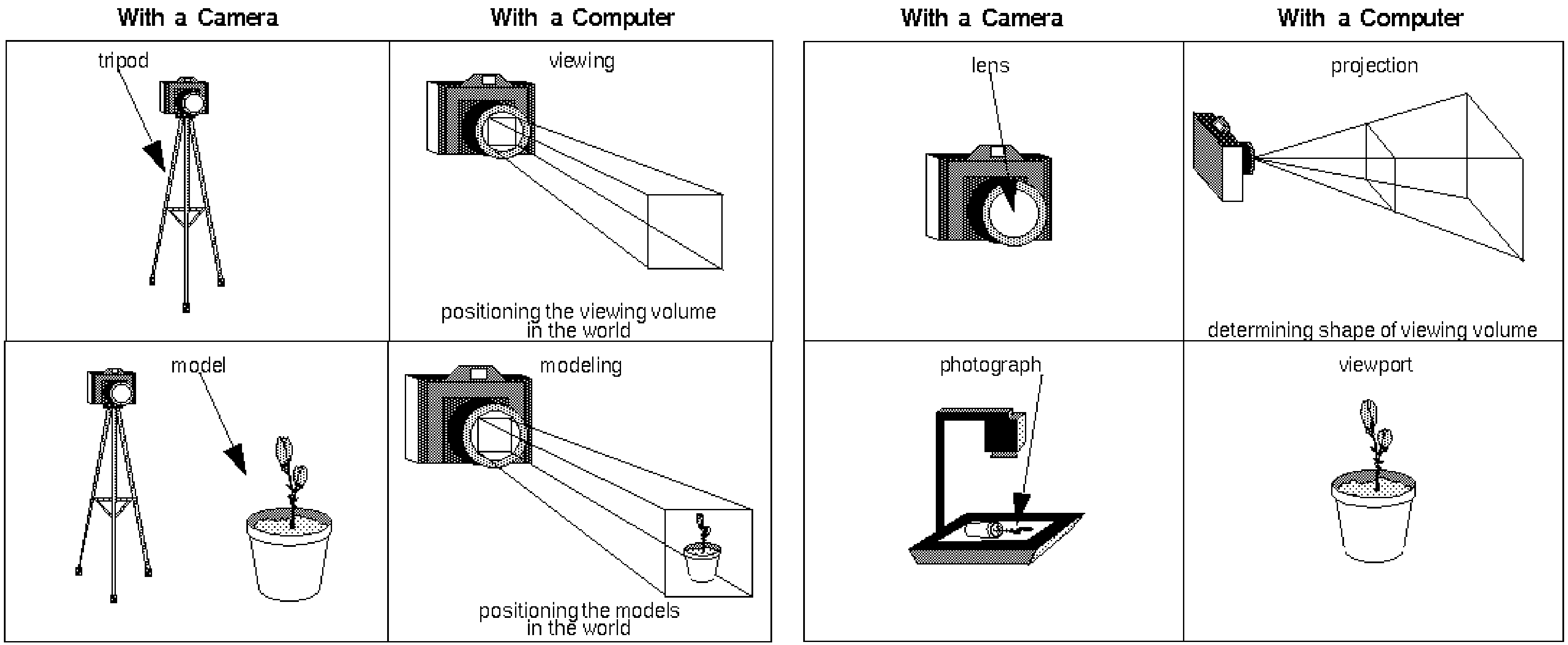
Model + View + Projection Transform



Model-view Transform

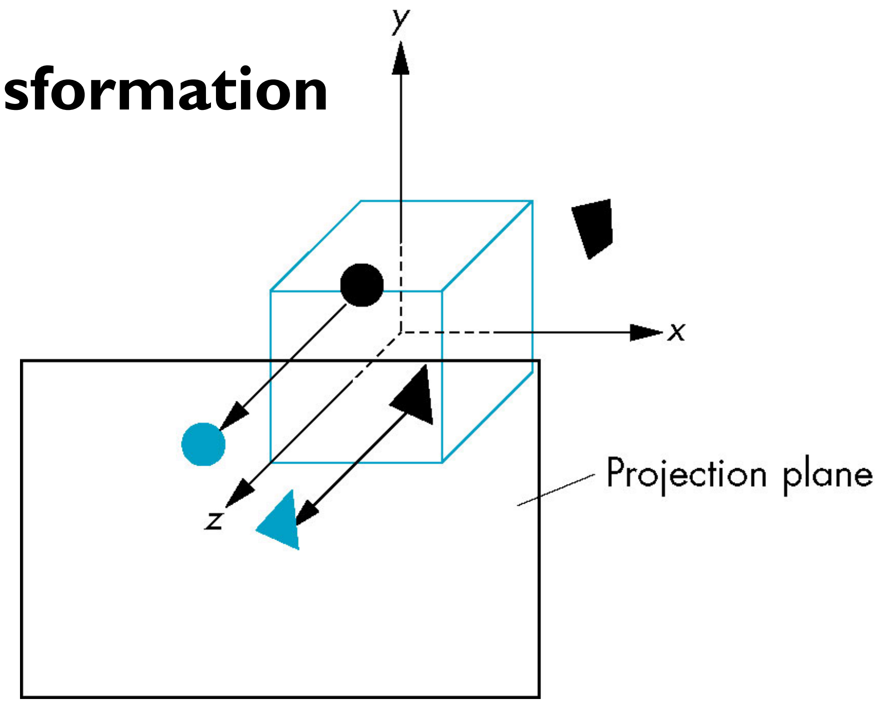
- Model (object) transform
  - For each object
- View transform
  - Camera location and orientation
  - To whole scene (all objects)

Camera Analogy



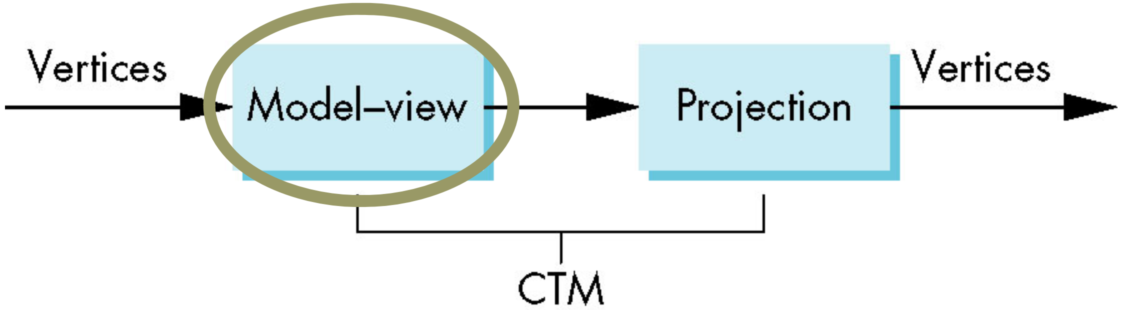
Viewing with a Computer

- Viewing consists of
  1. **Position the camera**
     (model-view transformation)
  2. **Apply the projection transformation**
     (orthographic or perspective)



Positioning of the Camera

- The matrix that is applied to all primitives is the product of the model-view matrix and the projection matrix.

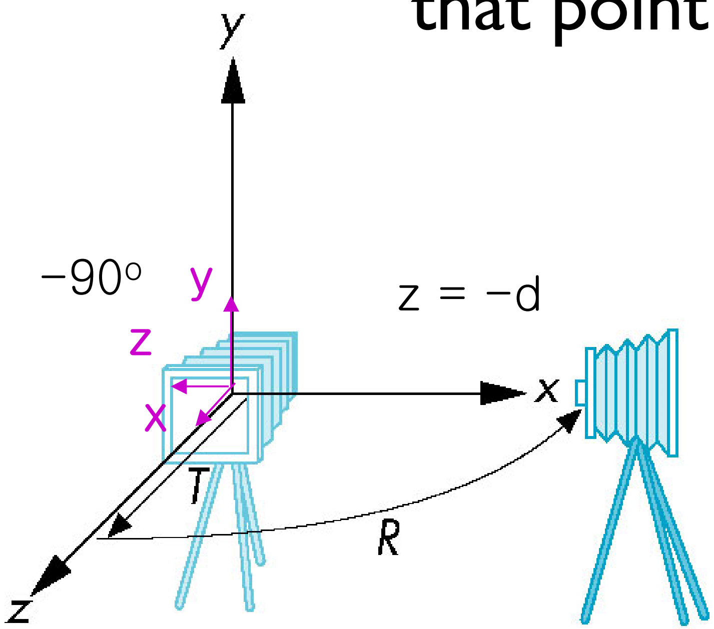


- **Model-view matrix**
  - **To position objects in space**
  - **To convert from the reference frame used for modeling to the frame of the camera**

Positioning of the Camera Frame

- Example:

  want the image of the faces of the object that point in the positive x-direction



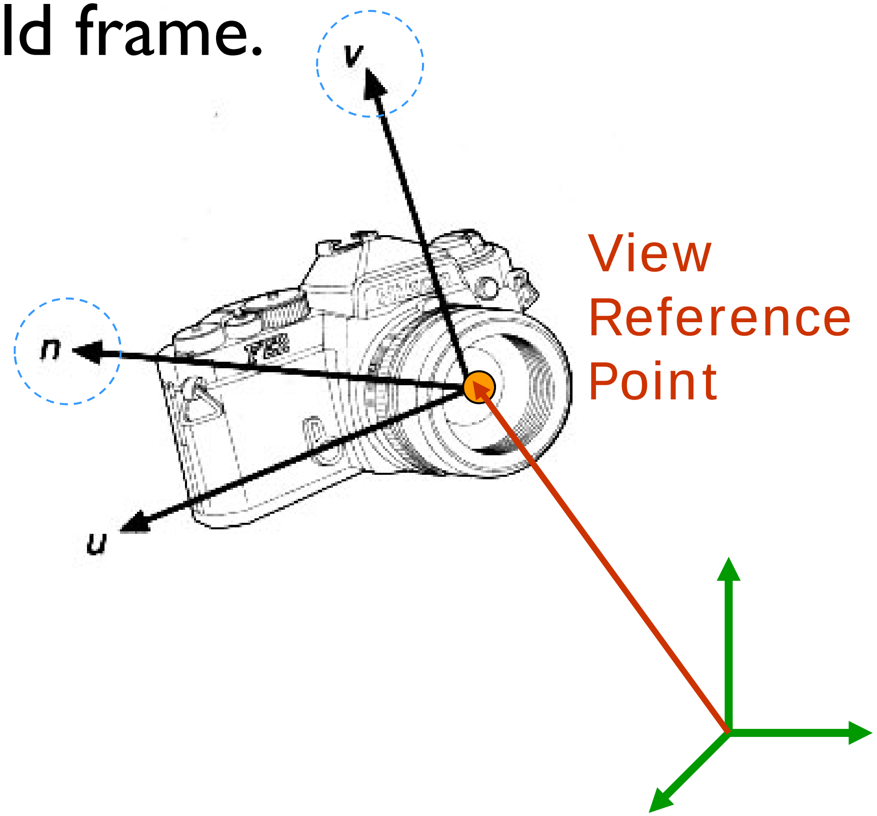

```
mat4 model_view;

model_view =
   Translate(0.0, 0.0, -d)
     * Rotate (-90.0, 0.0, 1.0, 0.0);
```

Direct Camera Placement

- Describe the camera's position and orientation in the world frame.

- **Define the viewing coordinate system, u-v-n.**

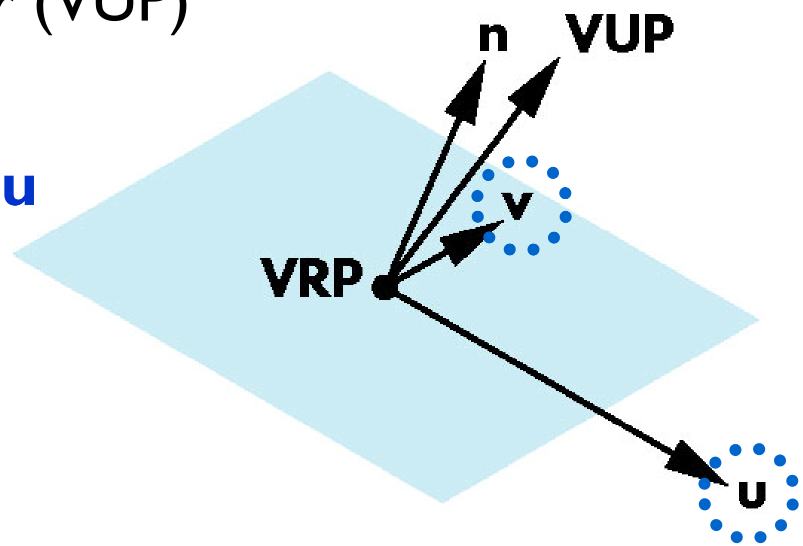
User specifies a view-reference point,
a view-up vector and
a view-plane normal.



Direct Camera Placement

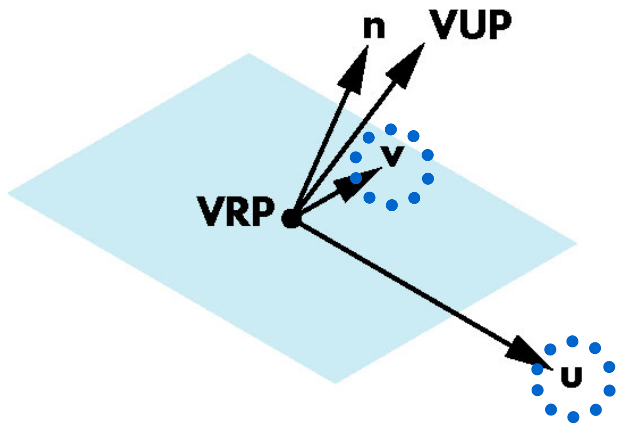
- Describe the camera's position and orientation in the world frame.
  - Specify the **view reference point** (VRP)
  - Specify the **view plane normal, n**
  - Specify the **view-up vector** (VUP)
  - Compute the **up-vector**, **v**
  - Compute the **side vector**, **u**

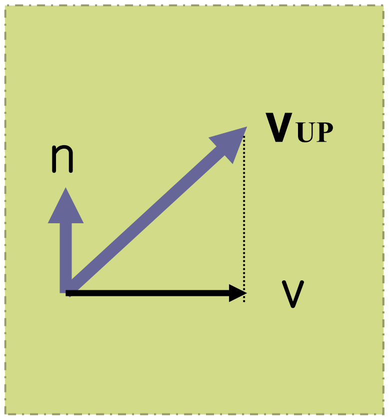
  - Defines the viewing coordinate system, u-v-n.



Direct Camera Placement

- Viewing-coordinate system.
  - **VRP = p = $(x,y,z,1)^T$**
  - **n = $(n_x, n_y, n_z, 0)^T$**
  - **$v_{UP} = (v_{up\_x}, v_{up\_y}, v_{up\_z}, 0)^T$**
  - Compute the up-vector, **v**
    - **$n \cdot v = 0$**
    - **$v = v_{UP} - \{(v_{UP} \cdot n) / (n \cdot n)\}n$**
  - Compute the side vector, **u**
    - **$u = v \times n$**
  - Normalize **u,v,n**
  - **u' – v' – n' system**

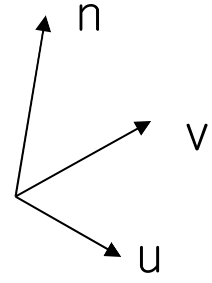


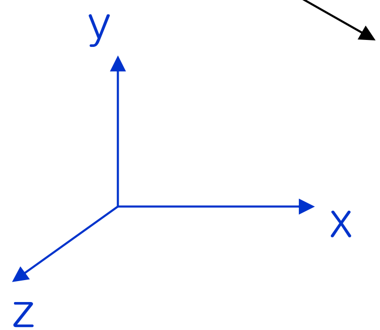


Direct Camera Placement

- View-orientation matrix
  - Orients a vector in the u'-v'-n' with respect to the original system
  - The rotation matrix
  - We want to represent the vector in the original system with respect to the camera system.
    - u – v – n system

$$M = \begin{bmatrix} u'_x & v'_x & n'_x & 0 \\ u'_y & v'_y & n'_y & 0 \\ u'_z & v'_z & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

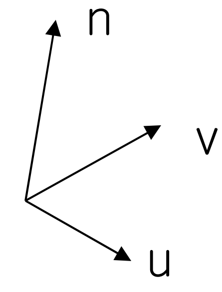


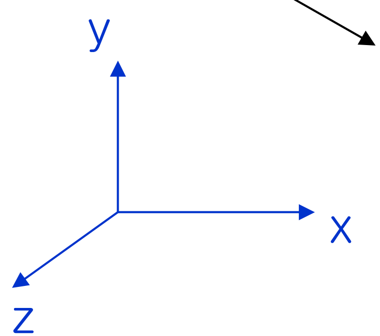


Direct Camera Placement

- View-orientation matrix
  - Orients a vector in the u'-v'-n' with respect to the original system

$$\mathbf{M} = \begin{bmatrix} u'_x & v'_x & n'_x & 0 \\ u'_y & v'_y & n'_y & 0 \\ u'_z & v'_z & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$





  - The rotation matrix
- We want $\mathbf{M}^{-1}$
  - Because M is a rotation matrix, $M^{-1} = M^T = R$

$$\mathbf{M}^T = \begin{bmatrix} u'_x & u'_y & u'_z & 0 \\ v'_x & v'_y & v'_z & 0 \\ n'_x & n'_y & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

World, object and eye frames

- World frame (world coordinates)
  - a basic right-handed orthonormal frame $\vec{\mathbf{w}}^t$
  - we never alter this frame
  - other frames can be described wrt the world frame
- Object frame (object coordinates)
  - model the geometry of the object using vertex coordinates
  - not need to be aware of the global placement
  - a right-handed orthonormal frame of object $\vec{\mathbf{o}}^t$
- Eye frame (camera coordinates): later on

World vs. object frame

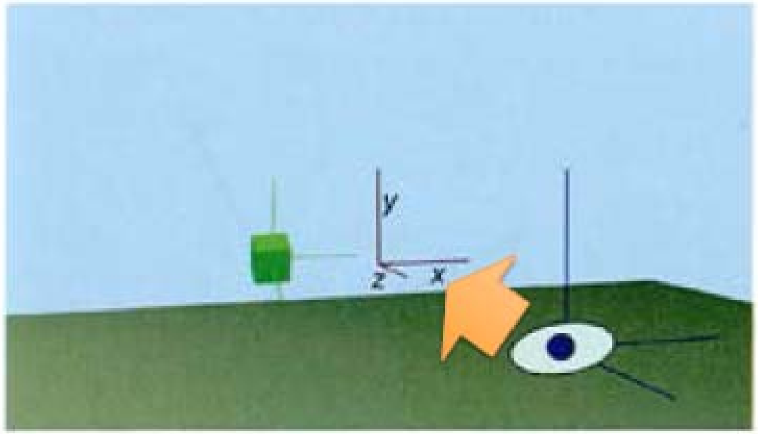
- The relationship between the world frame and object frame:
  - affine 4-by-4 matrix O (rigid body transformation: rotation + translation only)

$$\vec{\mathbf{o}}^t = \vec{\mathbf{w}}^t O$$

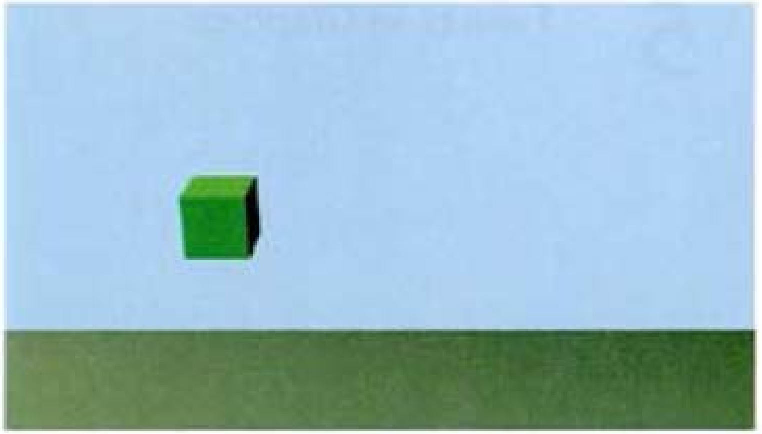
- The meaning of O is the relationship between the world frame to the object's coordinate system.
- To move the object frame $\vec{\mathbf{o}}^t$ itself, we change the matrix O.

The eye's view

- The world frame is in red
- The object frame is in green
- The eye frame is in blue
  - The eye is looking down its negative z toward the object.



(a) The frames

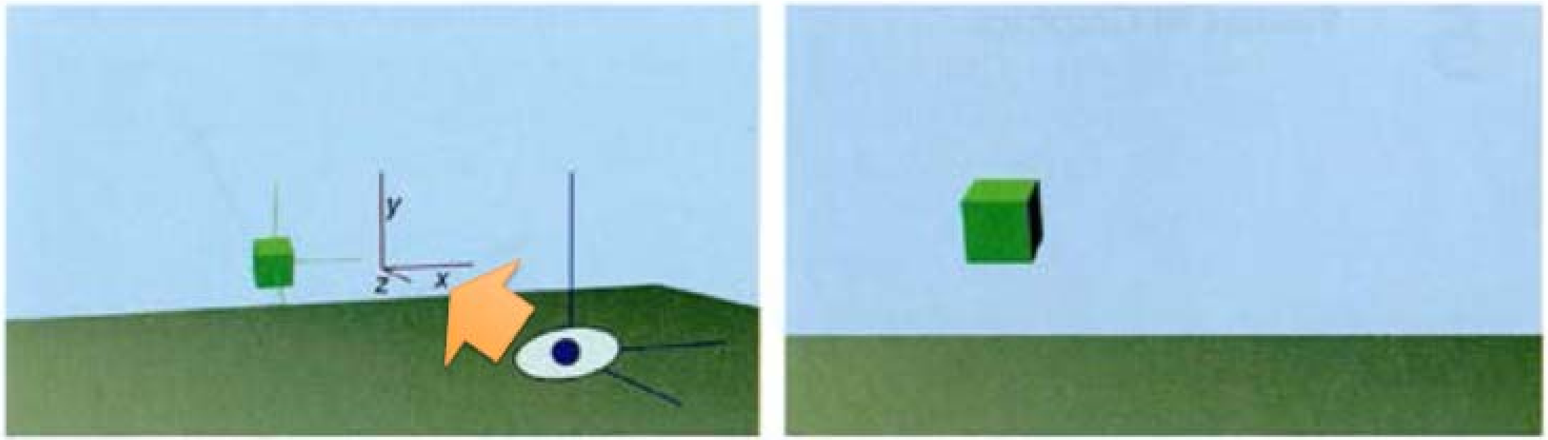


(b) The eye's view

The eye frame

- Eye frame (camera coordinates)
  - a right-handed orthonormal frame $\vec{\mathbf{e}}^t$
  - the eye looks down its negative z axis to make a picture

$$\vec{\mathbf{e}}^t = \vec{\mathbf{w}}^t E$$



(a) The frames (b) The eye's view

Extrinsic transformation of the eye

- we explicitly store the matrix E $\qquad \vec{\mathbf{e}}^t = \vec{\mathbf{w}}^t E$

  $$\tilde{p} = \vec{\mathbf{o}}^t \mathbf{c} = \vec{\mathbf{w}}^t O\mathbf{c} = \vec{\mathbf{e}}^t E^{-1} O\mathbf{c}$$

  – Object coordinates: $\mathbf{c}$

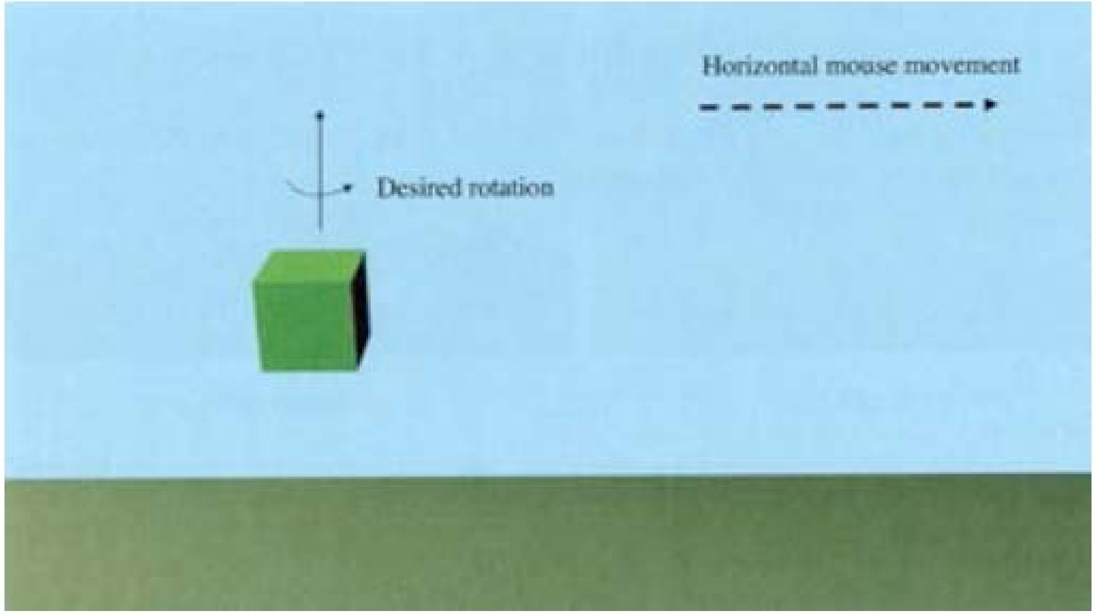
  – World coordinates: $O\mathbf{c}$

  – Eye coordinates: $E^{-1}O\mathbf{c}$

  – Calculating the eye coordinates of every vertexes:

  $$\begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = E^{-1}O \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

Moving an Object



- We want the object to rotate around its own center about the viewer's *y* axis, when we move the mouse to the right.
- How we could do this?

Moving an Object

- Basic idea: set a frame $\vec{\mathbf{a}}^t = \vec{\mathbf{w}}^t A$

  $$\vec{\mathbf{o}}^t$$

  $$= \vec{\mathbf{w}}^t O$$

  $$= \vec{\mathbf{a}}^t A^{-1} O$$

  $$\Rightarrow \vec{\mathbf{a}}^t M A^{-1} O$$

  $$= \vec{\mathbf{w}}^t A M A^{-1} O.$$

- What is the best frame $\vec{\mathbf{a}}^t$ to do this?

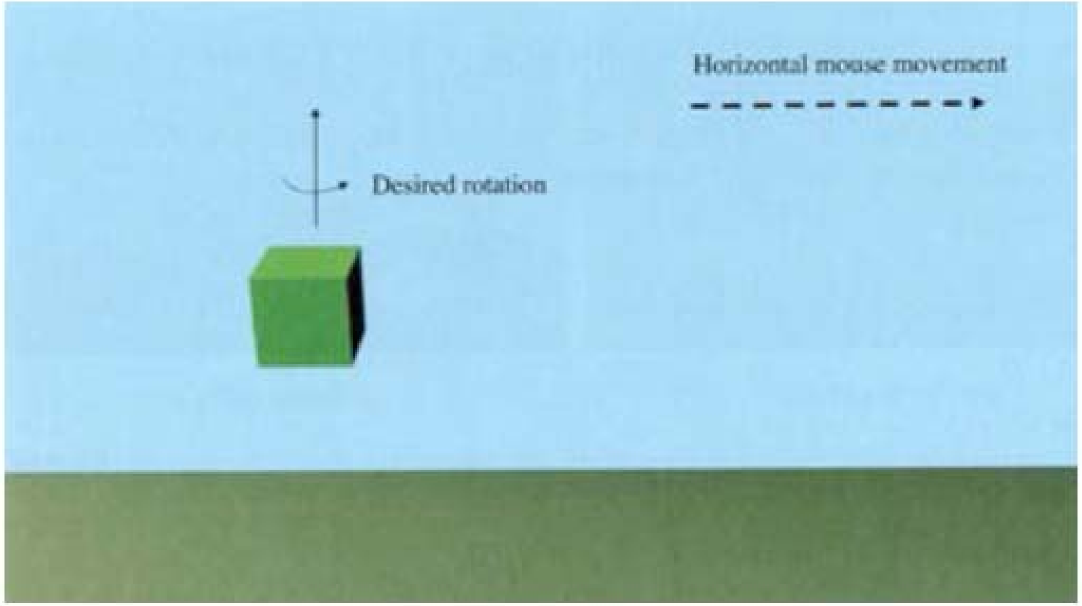
Moving an Object

- What if we choose $\vec{\mathbf{o}}^t$
- we transform this object with respect to $\vec{\mathbf{o}}^t$ rather than with respect to our observation through the window.

Moving an Object

- What if we choose $\vec{\mathbf{o}}^t$
- we transform this object with respect to $\vec{\mathbf{o}}^t$ rather than with respect to our observation through the window.
- What if we transform $\vec{\mathbf{o}}^t$ with respect to $\vec{\mathbf{e}}^t$
- we will rotate around the origin of the eye's frame $\vec{\mathbf{e}}^t$ (it appears to orbit around the eye).
- Then what frame it should be?

Moving an Object



- We actually want two different operations
  1. to transform (rotate) the object at its origin
  2. but the rotation axis should be the *y* axis of the eye.

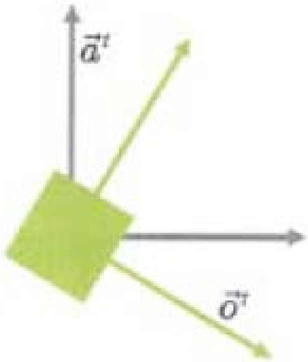
How to move an Object

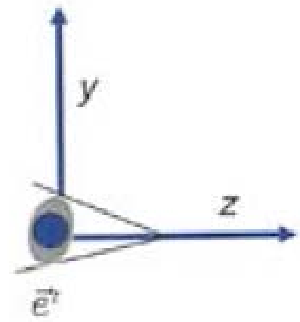
- Recalling the Affine transform.: $A = TR$
- The object's Affine transform.: $O = (O)_T (O)_R$
  (we want the object's rotation about the object's origin)
- The eye's Affine transform.: $E = (E)_T (E)_R$
  (we want the object's rotation about the eye's y axis)
- The desired auxiliary frame $\vec{\mathbf{a}}^t$
  (imagine in a inverse way):

$$\vec{\mathbf{a}}^t = \vec{\mathbf{w}}^t (O)_T (E)_R$$

$$A = (O)_T (E)_R$$

From the left, we translate the world frame t the center of the object's frame, and then rotating the object's frame about that point to align with the directions of the eye.





Moving the eye

- We use the same auxiliary coordinate system.
- But in this case, the eye would orbit around the center of the object.
- Apply an affine transform directly to the eye's own frame (turning one's head, first-person motion)

$$\vec{\mathbf{e}}^t = \vec{\mathbf{w}}^t E,$$

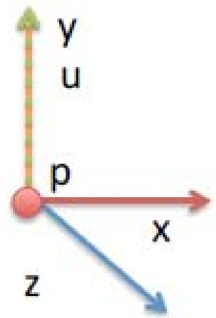
$$E \leftarrow EM$$

The eye matrix (camera transform)

- Specifying the eye matrix $\vec{\mathbf{e}}^t = \vec{\mathbf{w}}^t E$ by:
  - the eye point $\tilde{p}$
  - the view point (where the eye looks at) $\tilde{q}$
  - the up vector $\vec{u}$

NB P. 35 contains errors (see errata)!!!

NB matrix sent to the vertex shader is E^{-1}*O



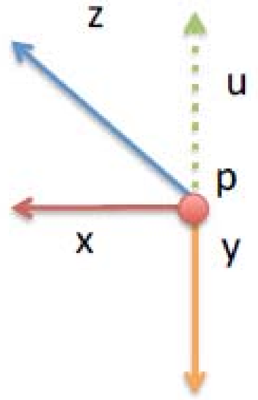
$$\mathbf{z} = normalize(p - q)$$
$$\mathbf{x} = normalize(\mathbf{u} \times \mathbf{z})$$
$$\mathbf{y} = \mathbf{z} \times \mathbf{x}$$
$$normalize(\mathbf{c}) = \mathbf{c} / \sqrt{c_1^2 + c_2^2 + c_3^2}$$

$$E = \begin{bmatrix} x_1 & y_1 & z_1 & p_1 \\ x_2 & y_2 & z_2 & p_2 \\ x_3 & y_3 & z_3 & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The view matrix (gluLookAt)

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = E^{-1} O \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

- Specifying the view matrix $V = E^{-1}$
  - the eye point $\tilde{p}$
  - the view point (where the eye looks at) $\tilde{q}$
  - the up vector $\vec{u}$



$$\mathbf{z} = normalize(q - p)$$

$$\mathbf{x} = normalize(\mathbf{u} \times \mathbf{z})$$

$$\mathbf{y} = \mathbf{x} \times \mathbf{z}$$

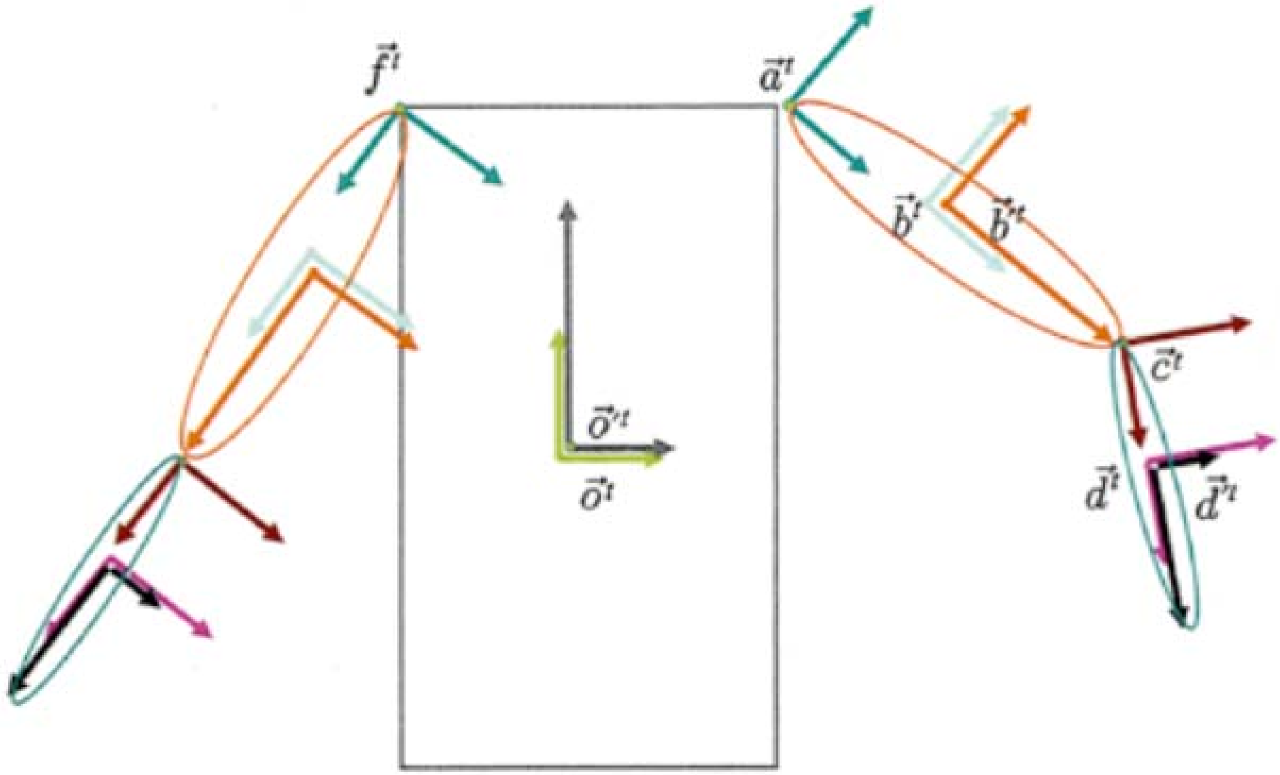
$$normalize(\mathbf{c}) =$$

$$\mathbf{c} / \sqrt{c_1^2 + c_2^2 + c_3^2}$$

NB gluLookAt is not the part of modern OpenGL!

Hierarchy frames

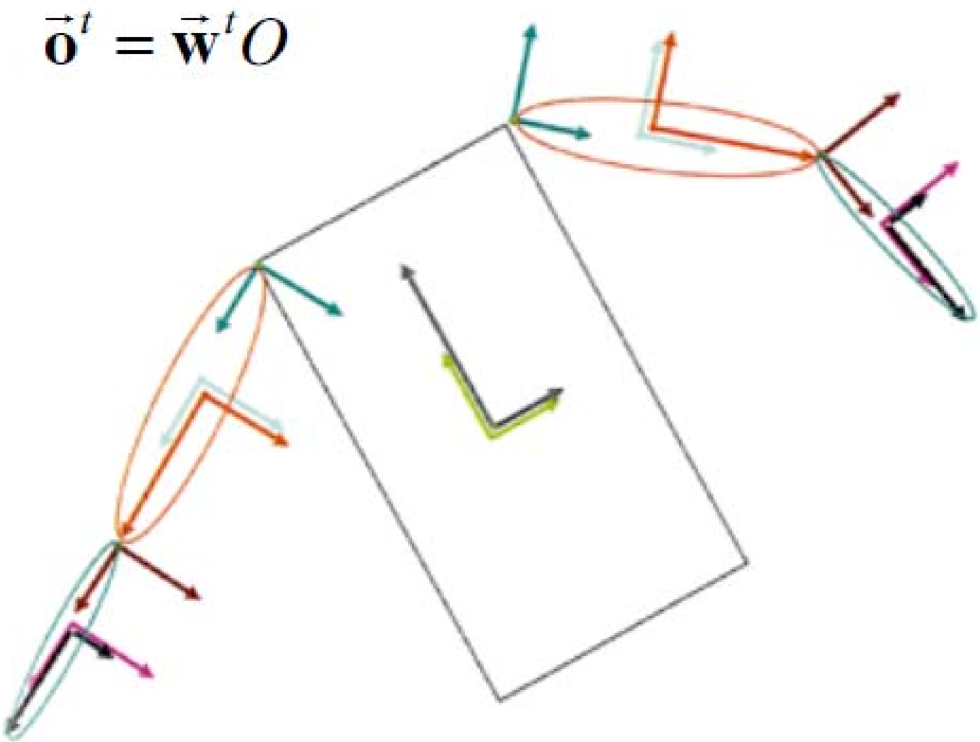
- An object can be treated as being assembled by some fixe and movable subobjects.



$$\vec{\mathbf{o}}^t = \vec{\mathbf{w}}^t O$$
$$\vec{\mathbf{o}}'^t = \vec{\mathbf{o}}^t O'$$
$$\vec{\mathbf{a}}^t = \vec{\mathbf{o}}^t A$$
$$\vec{\mathbf{b}}^t = \vec{\mathbf{a}}^t B$$
$$\vec{\mathbf{b}}'^t = \vec{\mathbf{b}}^t B'$$
$$\vec{\mathbf{c}}^t = \vec{\mathbf{b}}^t C$$
$$\vec{\mathbf{d}}^t = \vec{\mathbf{c}}^t D$$
$$\vec{\mathbf{d}}'^t = \vec{\mathbf{d}}^t D'$$
$$\vec{\mathbf{f}}^t = \vec{\mathbf{o}}^t F$$

Moving the entire robot

- We just update its *O* matrix to the object frame, instead of relating it to the world frame



$$\vec{\mathbf{o}}^t = \vec{\mathbf{w}}^t O$$

$$\vec{\mathbf{a}}^t = \vec{\mathbf{w}}^t OA$$

$$\vec{\mathbf{b}}^t = \vec{\mathbf{w}}^t OAB$$

$$\vec{\mathbf{b}}''^t = \vec{\mathbf{w}}^t OABB'$$

$$\vec{\mathbf{c}}^t = \vec{\mathbf{w}}^t OABC$$

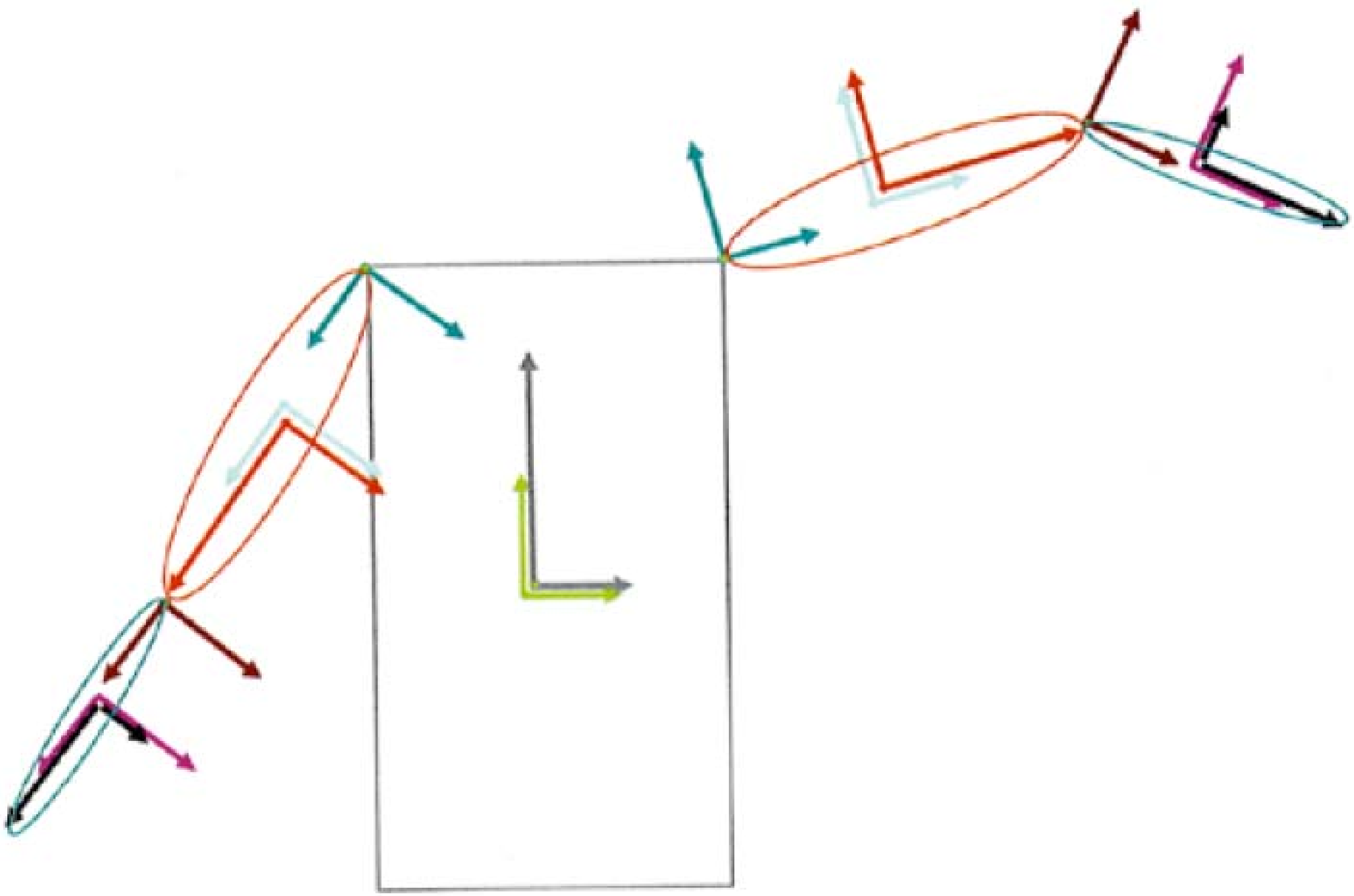
$$\vec{\mathbf{d}}^t = \vec{\mathbf{w}}^t OABCD$$

$$\vec{\mathbf{d}}''^t = \vec{\mathbf{w}}^t OABCDD'$$

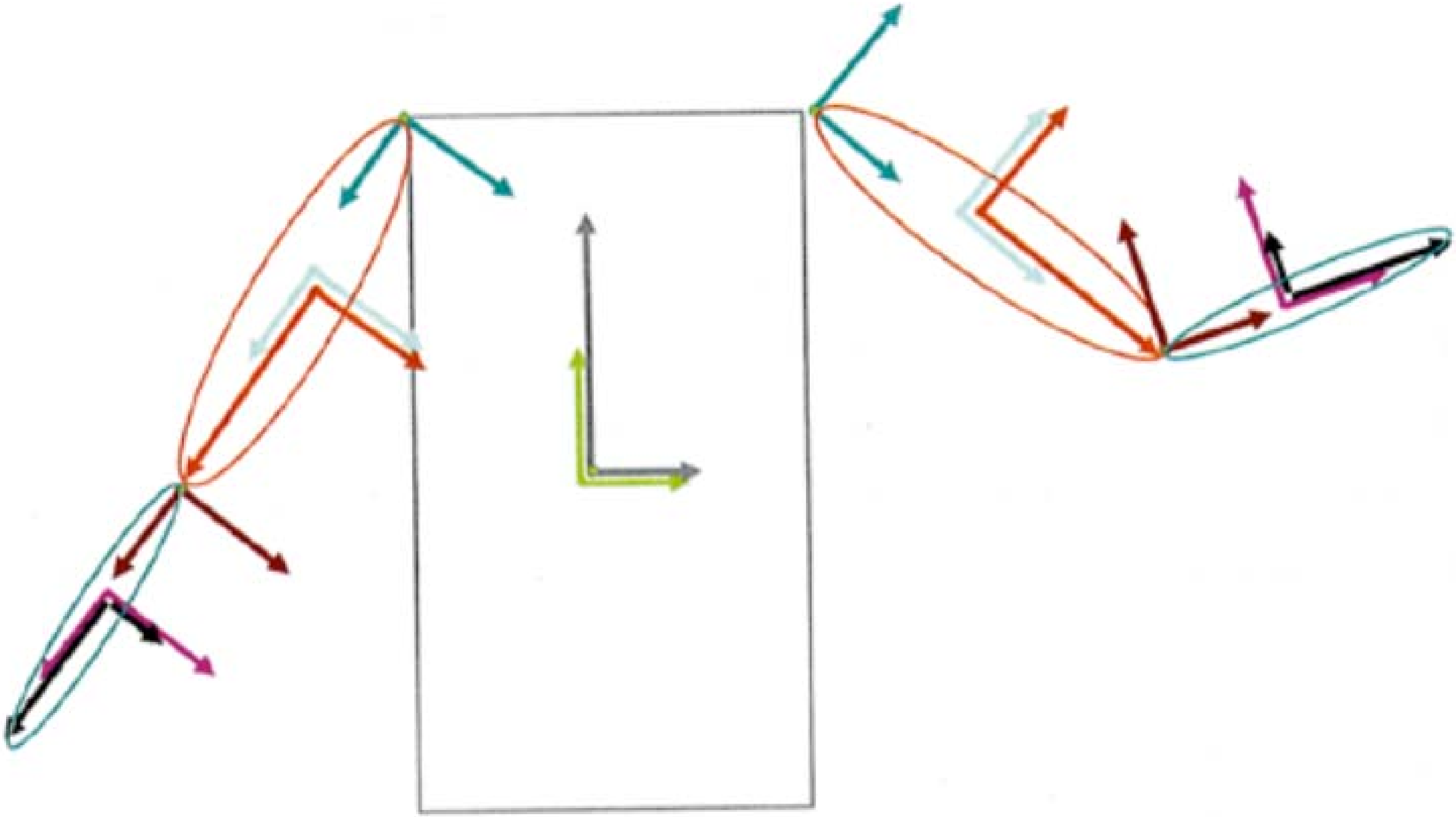
Matrix stack

- Matrix stack data structure can be used to keep track of the matrix
- push(M)
  - creates a new 'topmost' matrix
  - a copy of the previous topmost matrix
  - M. multiplies this new top matrix
- pop()
  - removes the topmost layer of the stack
- descending
  - descend down to a subobject, when a push operation is done
  - this matrix is popped off the stack when returning from this descent to the parent

Moving limbs



Moving limbs



Scene graph pseudocode

```
...
matrixStack.initialize(inv(E));
matrixStack.push(O);
        matrixStack.push(O');
                draw(matrixStack.top(), cube); \\ body
        matrixStack.pop(); \\ O'

        matrixStack.push(A); \\ grouping
                matrixStack.push(B);
                        matrixStack.push(B');
                                draw(matrixStack.top(), sphere); \\ upper arm
                        matrixstack.pop(); \\ B'

                        matrixStack.push(C);
                                matrixStack.push(C');
                                        draw(matrixStack.top(), sphere); \\ lower arm
                                matrixStack.pop(); \\ C'
                        matrixStack.pop(); \\ C
                matrixStack.pop(); \\ B
        matrixStack.pop(); \\ A
\\ current top matrix is inv(E)*O

\\ we can now draw another arm
        matrixStack.push(F);
```

Chapter 6