Computer Graphics

Geometry Objects & Transformation

Teacher: A.prof. Chengying Gao(高成英)

E-mail: mcsgcy@mail.sysu.edu.cn

School of Data and Computer Science

Outline

- **Geometry**
- Representation
- Transformation
- Transformation in OpenGL

Basic geometric elements

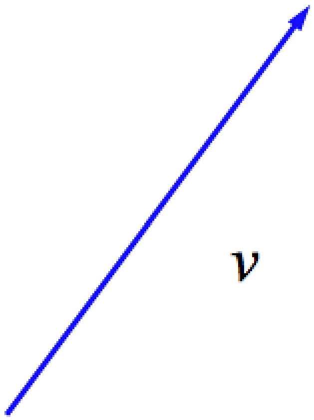
- Geometry study the relationship among objects in N-dimensional space
  - In computer graphic, we mainly focus on objects in 2D & 3D space.
- Hoping to get a minimum set of geometric shapes and we can construct complex object base on it.
- Three basic geometric elements
  - Scalar
  - Vector
  - Point

Scalar

- Scalar can be defined as a member of collection
  - Collection has two operation (addition and multiplication).
  - They comply with some basic arithmetic axioms (associativity law, commutatively law, inverse)
  - real numbers, complex numbers, and rational functions.
- Scalar doesn't have geometric properties

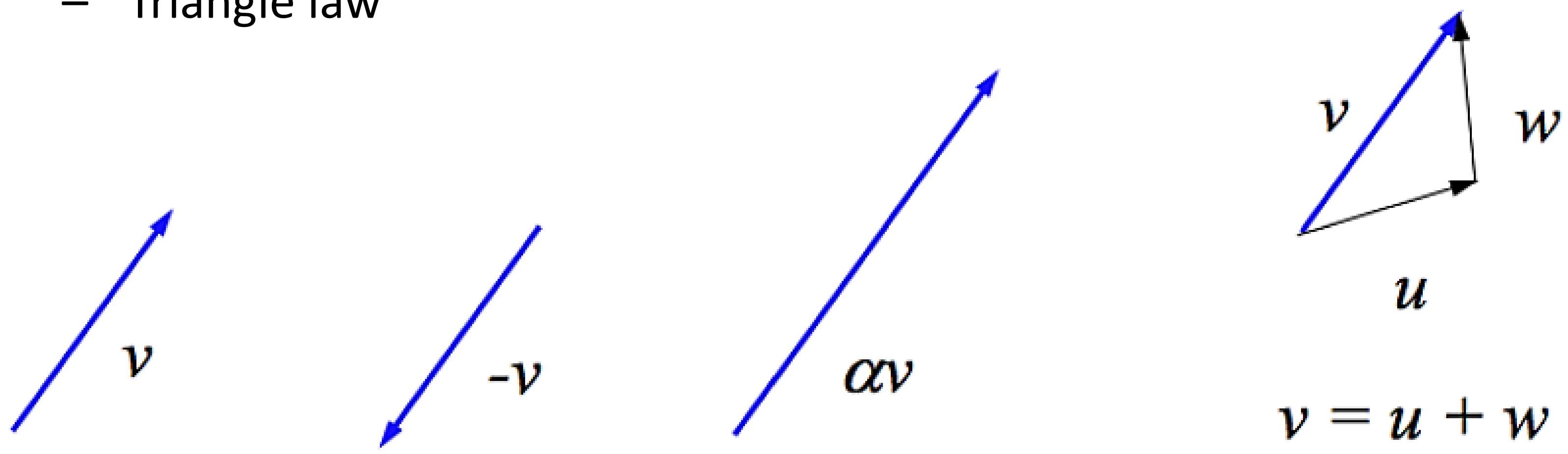
Vector

- Definition: vector is a line having the two properties
  - Direction
  - Length: $|v|$
- Examples:
  - Power
  - Speed
  - Directed line segment

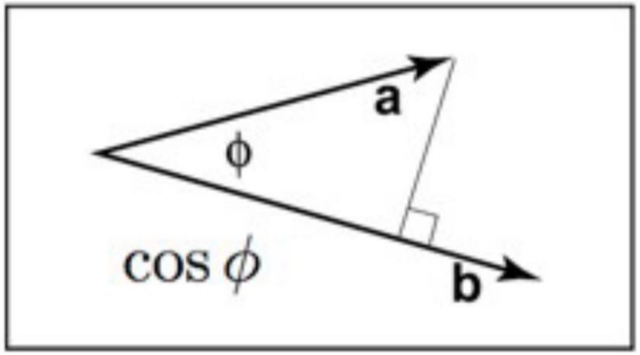


Vector operations

- Each vector has an inverse
  - Same length but different directions
- Each vector can be multiplied by a scalar
- A zero vector
  - Length is 0, direction is uncertain
- Sum of two vectors is a vector
  - Triangle law



Inner product（Dot product）



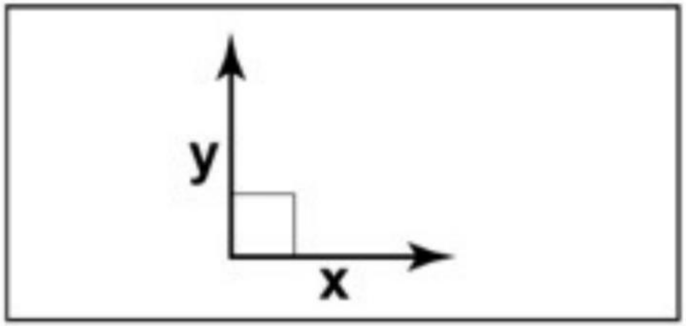
$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos \phi$$

The projection of a onto b

N. B. the projection is 0 if a is perpendicular to b

Computer Graphics 2014, ZJU

Orthonormal Vector



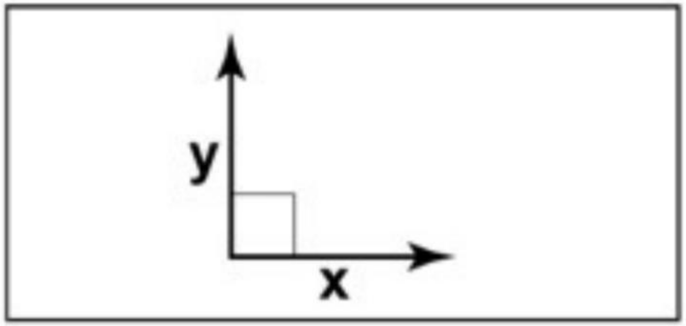
Perpendicular $\mathbf{x} \cdot \mathbf{y} = 0$

Unit length $\mathbf{x} \cdot \mathbf{x} = 1$

$\mathbf{y} \cdot \mathbf{y} = 1$

Computer Graphics 2014, ZJU

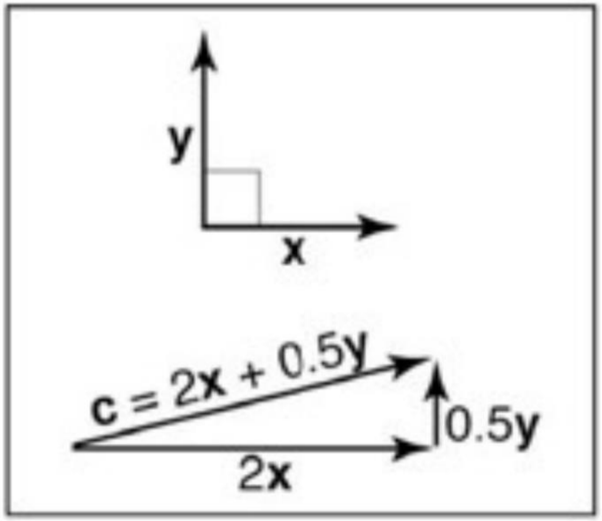
Orthonormal Vector



$$
\begin{aligned}
&\textbf{Perpendicular} \quad & \mathbf{x} \cdot \mathbf{y} &= 0 \\
&\textbf{Unit length} \quad & \mathbf{x} \cdot \mathbf{x} &= 1 \\
& & \mathbf{y} \cdot \mathbf{y} &= 1
\end{aligned}
$$

Computer Graphics 2014, ZJU

Coordinates and Vectors



$$\mathbf{c} = \alpha\mathbf{x} + \beta\mathbf{y}$$

$$\begin{aligned}\alpha &= \mathbf{x}\cdot\mathbf{c} = \alpha\,\mathbf{x}\cdot\mathbf{x} + \beta\,\mathbf{x}\cdot\mathbf{y}\\ \beta &= \mathbf{y}\cdot\mathbf{c} = \alpha\,\mathbf{y}\cdot\mathbf{x} + \beta\,\mathbf{y}\cdot\mathbf{y}\end{aligned}$$

Computer Graphics 2014, ZJU

Dot product between two vectors

$$\mathbf{a} = x_a\mathbf{x} + y_a\mathbf{y}$$

$$\mathbf{b} = x_b\mathbf{x} + y_b\mathbf{y}$$

$$\mathbf{a} \cdot \mathbf{b} = x_a x_b + y_a y_b$$

$$\mathbf{a} \cdot \mathbf{a} = x_a^2 + y_a^2 = |\mathbf{a}|^2$$

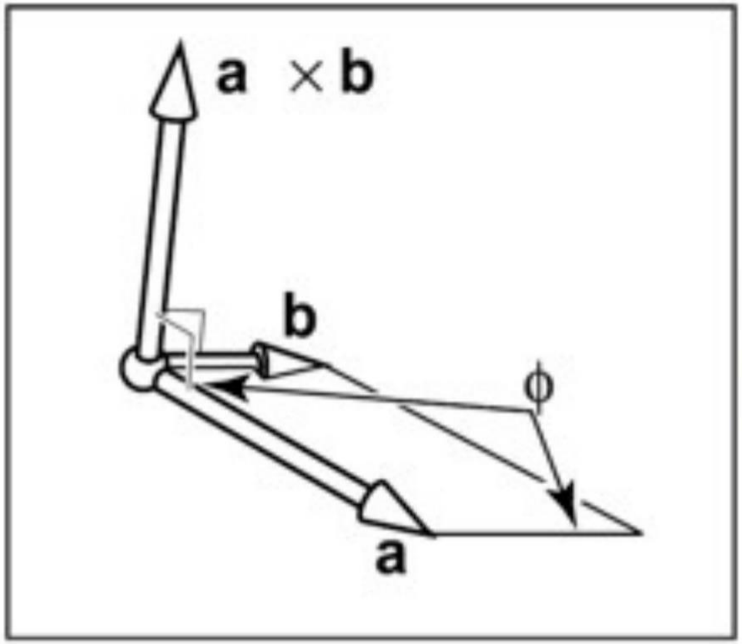
$$|\mathbf{a}| = \sqrt{x_a^2 + y_a^2} = \sqrt{\mathbf{a} \cdot \mathbf{a}}$$

Computer Graphics 2014, ZJU

Dot product: some applications in CG

- Find angle between two vectors (e.g. cosine of angle between light source and surface for shading)
- Finding projection of one vector on another (e.g. coordinates of point in arbitrary coordinate system)
- Advantage: can be computed easily in Cartesian components

Cross Product



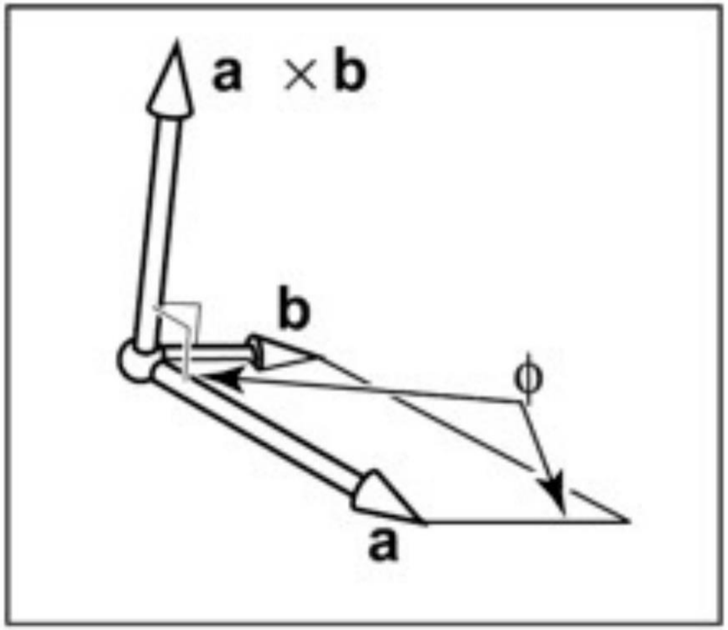
$$\mathbf{c} = \mathbf{a} \times \mathbf{b}$$

$$\begin{aligned} x_c &= y_a z_b - z_a y_b \\ y_c &= z_a x_b - x_a z_b \\ z_c &= x_a y_b - z_a x_b \end{aligned}$$

c perpendicular to both a and b
|c| is equal to the area of quadrilateral a b

Computer Graphics 2014, ZJU

Cross Product



Right-Hand Rule

$$
\begin{aligned}
\mathbf{x} \times \mathbf{y} &= \mathbf{z} \\
\mathbf{y} \times \mathbf{z} &= \mathbf{x} \\
\mathbf{z} \times \mathbf{x} &= \mathbf{y}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{x} \times \mathbf{x} &= 0 \\
\mathbf{y} \times \mathbf{y} &= 0 \\
\mathbf{z} \times \mathbf{z} &= 0
\end{aligned}
$$

Computer Graphics 2014, ZJU

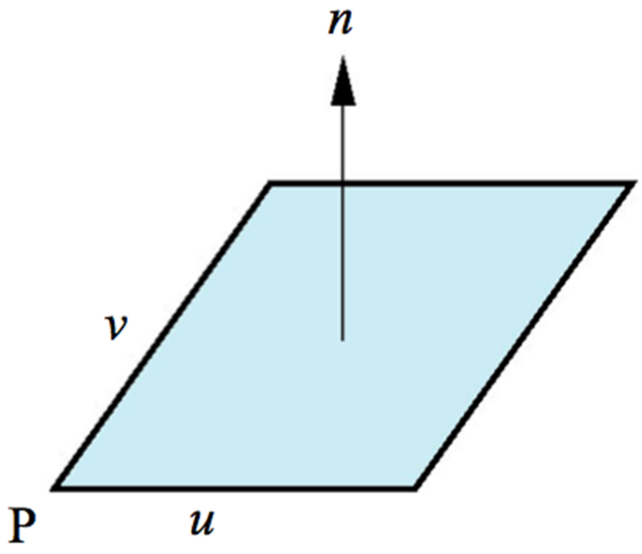
Normals

- Each plane has a vector n perpendicular to itself
- If a plane is determined with a point and two vectors

$$\mathbf{P}(\alpha, \beta) = \mathbf{R} + \alpha \boldsymbol{u} + \beta \boldsymbol{v}$$

- we can get n by the following equation

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



Linear space

- The most important mathematical space is the (linear) vector space.
- Two basic geometric elements:
  - scalar, vector
- Operation
  - Scalar multiplication: $u = \alpha v$
  - Vector addition: $w = u + v$

Linear combination

- Given n vectors v1, v2, …, vn and n scalar a1, a2, …, an, then

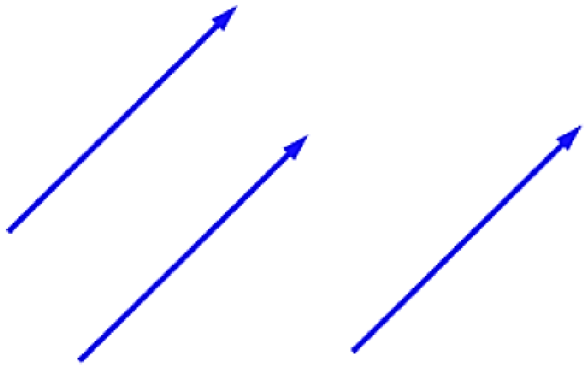
$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

  is also a vector, called the linear combination of this set of vectors.

- Irrelevant with coordiante

Vectors have no positions

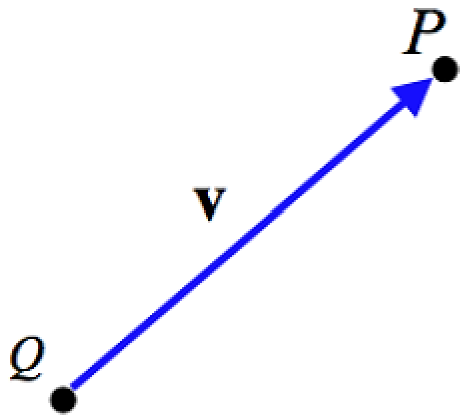
- **The following vectors are equal**
  - As they have same length and direction



- **It is not enough for geometry to only have vector space**
  - We still need points.

Point

- Position in space
  - Use uppercase letters
- Operational between points and vectors
  - Subtraction with two points, we can get a vector
  - Addition with a point and a vector, we get a point



$$\mathbf{v} = P - Q$$
$$P = \mathbf{v} + Q$$

Affine space

- Space constructed by **points** and vectors
- Operational:
  - Vector + Vector = Vector
  - Scalar x Vector = Vector
  - Point + Vector = Point
  - Scalar + Scalar = Scalar

Linear combination of points

- Fixed coordinate system, given two points, what is P1 + P2?
  - P1 is origin, P1 + P2 = P2
  - P1 and P2 are symmetric on origin, P1 + P2 = origin
  - The Positions of P1, P2 are relevant with coordiante
- Combination coefficients have limitations
  - When $\alpha_1 + \alpha_2 + \ldots + \alpha_n = 1$, linear combination of points is a point
  - $\frac{1}{2} \mathbf{P}_1 + \frac{1}{2} \mathbf{P}_2 = \mathbf{P}_1 + \frac{1}{2} (\mathbf{P}_2 - \mathbf{P}_1)$ = point + vector = point

Affine convex combination

- Consider:

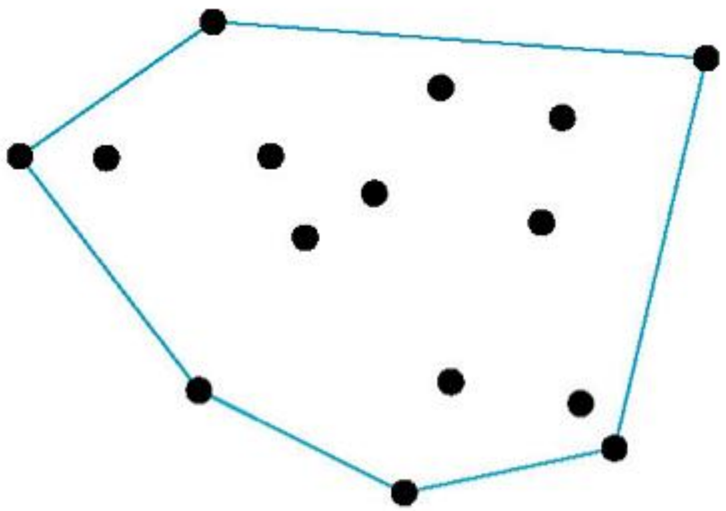
$$P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n$$

When $\alpha_1 + \alpha_2 + \ldots + \alpha_n = 1$, the equation above has meaning and the result is called the affine convex combination for P_1, P_2, …, P_n.

- If $\alpha_i \geq 0$, , we get the convex hull for P_1, P_2, …, P_n

Convex Hull

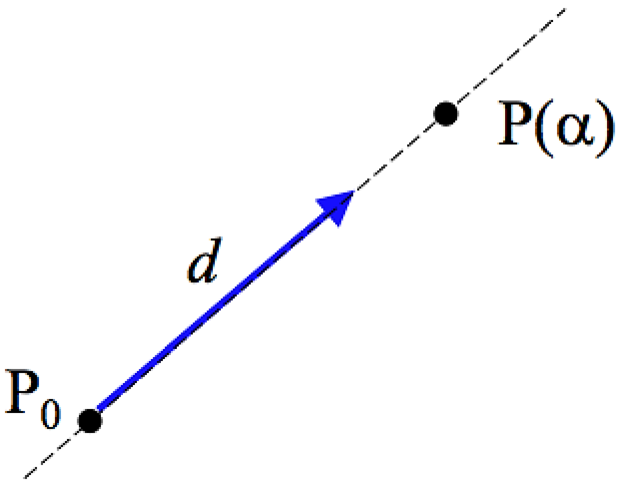
- The minimum convex contains $P_1, P_2, \ldots, P_n$
- Can use the "Shrink" method to get it



Line

- All points comply with the following form

$$P(\alpha) = P_0 + \alpha\, d$$



Parametric form

- **It is the parametric form definition for line**
  - More general and stable
  - Can be used in curves and surfaces
- **Two-dimensional form**
  - Explicit: $y = mx + h$
  - Implicit: $ax + by + c = 0$
  - Parametric: $$x(\alpha) = x_0 + (1 - \alpha)\, x_1$$ $$y(\alpha) = y_0 + (1 - \alpha)\, y_1$$

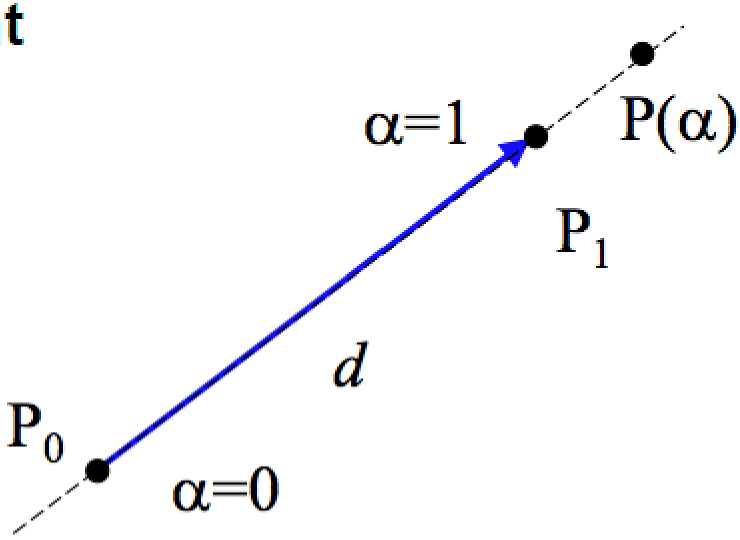
Rays and segments

- If a > 0, P(a) is a ray start from P_0 with direction d

- If use two points to define vector d, then:

$$P(\alpha) = P_0 + \alpha\,(P_1 - P_0) = (1 - \alpha)\,P_0 + \alpha\,P_1$$

- When $0 \le \alpha \le 1$, we get a segment



Linear interpolation

- **Given two points A and B, their affine combination**

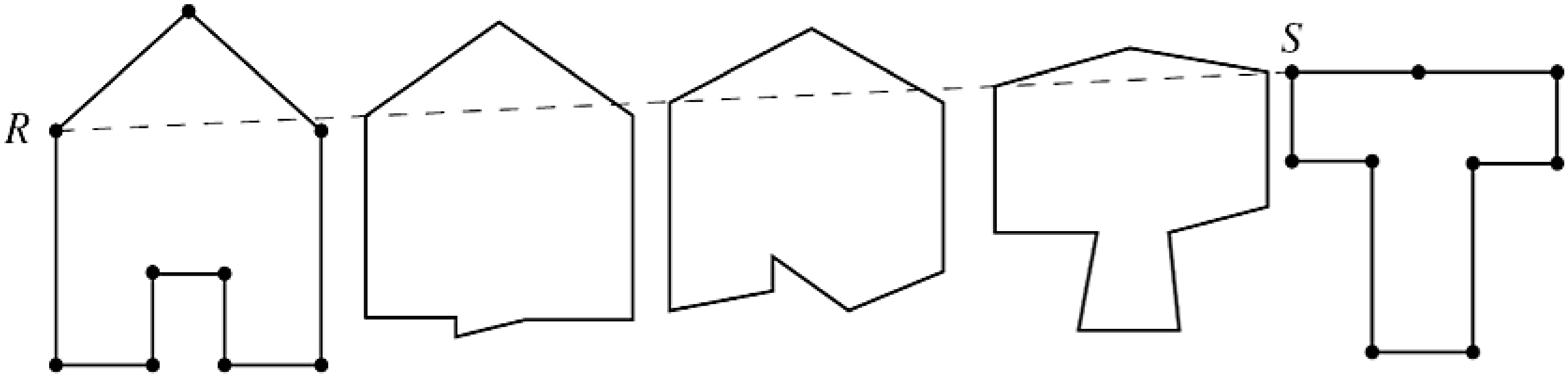
$$\mathbf{P}(t) = (1 - t)\,\mathbf{A} + t\,\mathbf{B}$$

  defines a line pass these two points.

- **Linear interpolation is applied in art and animation**
  - Key Frame

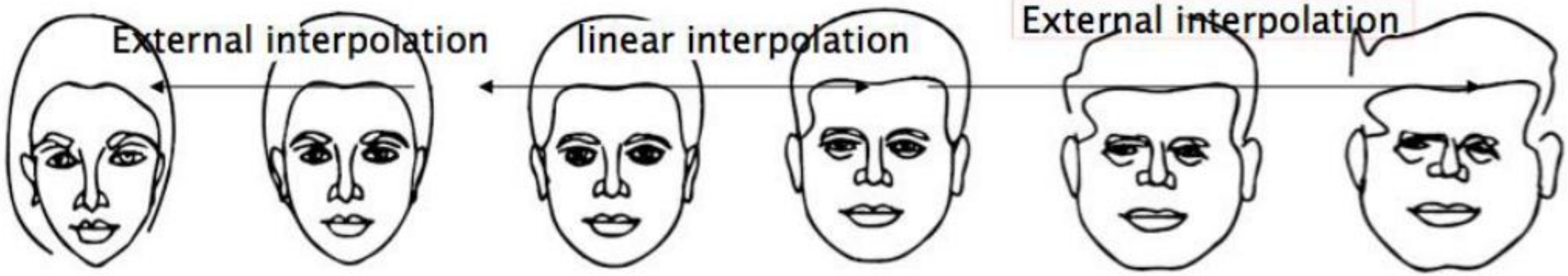
Polygon deformation

- Given two lines with the same number of vertices, we can get a smooth transition from the first to the second polyline



Man to Woman

Celebrity Face

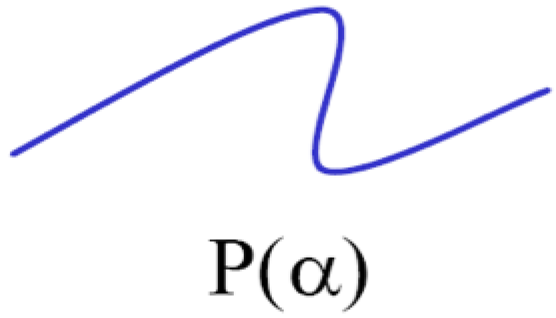


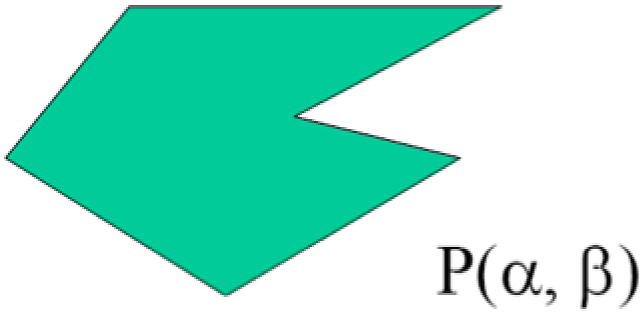
Elizabath Taylor

John F. Kennedy

Curve and Surface

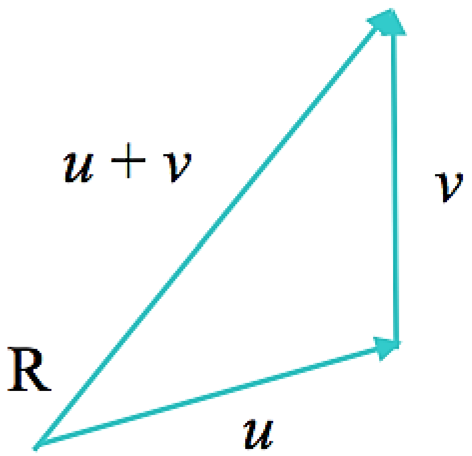
- Curve is single parameter defined geometry with form P(a), the function is non-linear.
- Surface is define with P(a, b), the function is non-linear.
  - linear function is plane & polygon



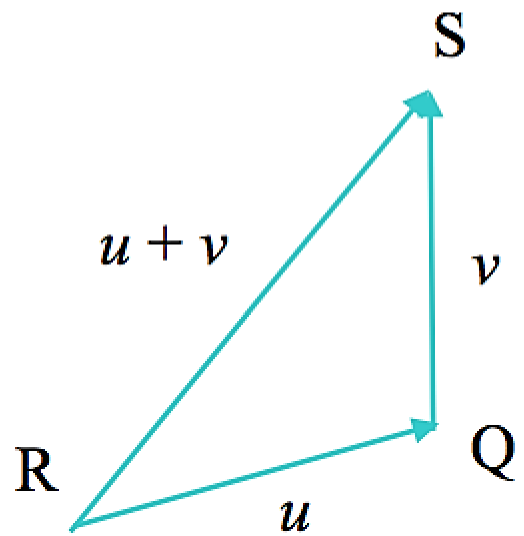


Plane

- A plane is determined by a point with two vectors or three points

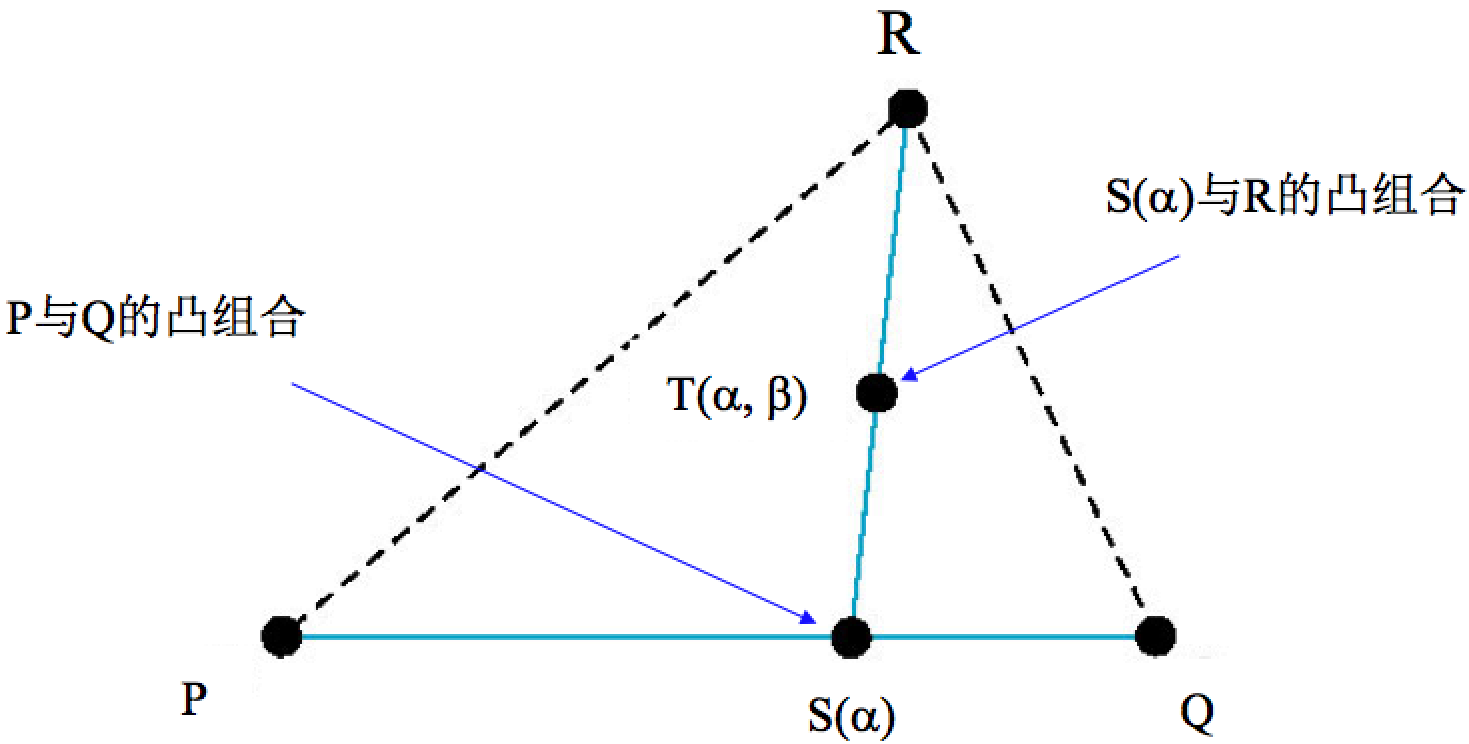


$$P(\alpha, \beta) = R + \alpha u + \beta v$$



$$P(\alpha, \beta) = R + \alpha(Q - R) + \beta(S - Q)$$

Triangle



当0≤α, β≤1时定义在三角形内的点

Outline

- Geometry
- **Representation**
- Transformation
- Transformation in OpenGL

Representation

- Until now we have only discussed the geometric objects, without using any reference frame, for example, the coordinate system
- Requires a reference point and the frame to contact with objects in the physical world
  - Position: Where is a point?(if there is not frame, we can not answer it)
  - World coordinate system

Coordinate

- **Basis for n dimensional vector space v_1, v_2, …, v_n**
- **A vector v can be express in this form**

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

- **Scalar set{a_1, a_2, …, a_n} is called the representation of the given basis**

$$a = [\alpha_1, \alpha_2, \ldots, \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

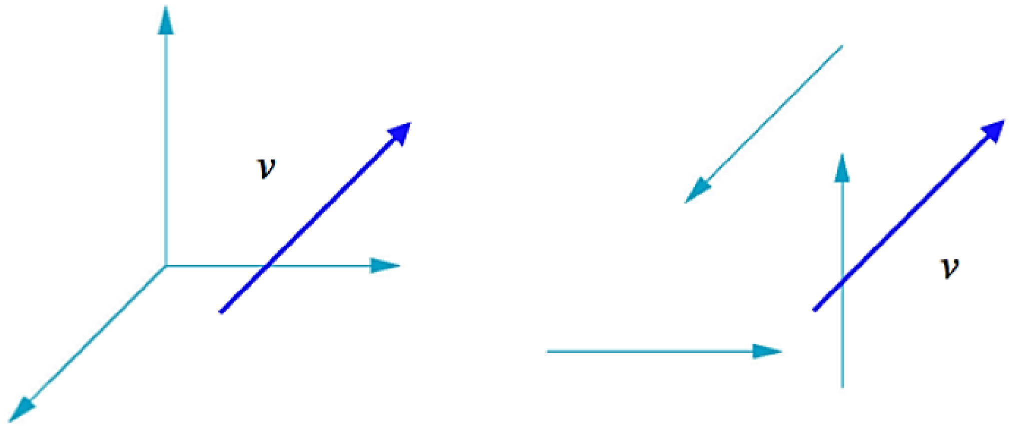
Example

$$v = 2v_1 + 3v_2 - 4v_3$$
$$a = [2, 3, -4]^T$$

- Note that the above statement is relative to a particular basis
- Eg: OpenGL represents a vector with respect to the world coordinate system, it is necessary to transform to the camera coordinate system .

Coordinate

- **Which is right?**

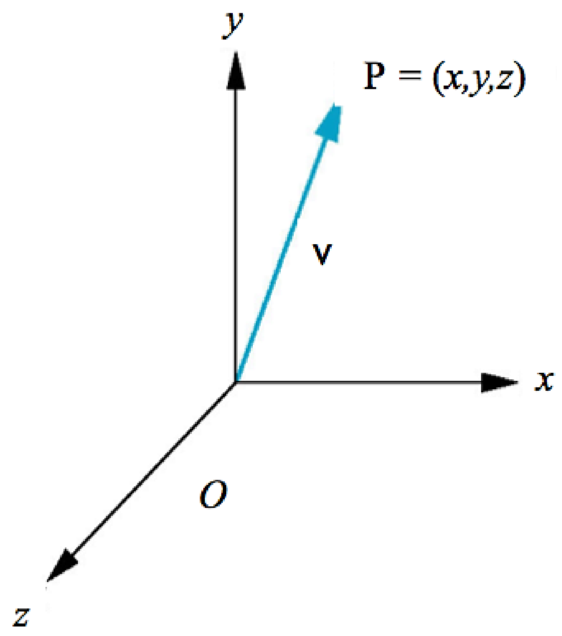


(a) (b)

Both, vectors don't have a fixed position

Frame

- Coordinate system is insufficient to represent points.
- We need an **origin** to construct a frame. The origin and the basis vectors determine a frame (标架).



Representation in frame

- Frame is determined by (O, v_1, v_2, …, v_n)
- Within a given frame, every vector can be written uniquely as:

$$w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = \mathbf{a}^T \mathbf{v},$$

just as in a vector space;

- every point can be written uniquely as

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = P_0 + \mathbf{b}^T \mathbf{v}.$$

Point and Vector confusion

- **Consider point and vector**

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$
$$\mathbf{P} = O + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$$

- **They have similar representation, so it is easy to confusion them**

$$v = [\alpha_1, \alpha_2, \ldots, \alpha_n]^T,$$
$$\mathbf{P} = [\beta_1, \beta_2, \ldots, \beta_n]^T,$$

Unified representation

- If 0·P = 0(zero vector), 1 · P = P, then

$$
\begin{aligned}
v &= \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \\
&= [v_1, v_2, \ldots, v_n, O][\alpha_1, \alpha_2, \ldots, \alpha_n, 0]^T \\
\mathbf{P} &= O + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n \\
&= [v_1, v_2, \ldots, v_n, O][\beta_1, \beta_2, \ldots, \beta_n, 1]^T
\end{aligned}
$$

- N+1 dimensional homogeneous coordinate representation

$$
\begin{aligned}
v &= [\alpha_1, \alpha_2, \ldots, \alpha_n, 0]^T \\
\mathbf{P} &= [\beta_1, \beta_2, \ldots, \beta_n, 1]^T
\end{aligned}
$$

Homogeneous coordinate

- **General form for 4-dimension homogeneous coordinate:**

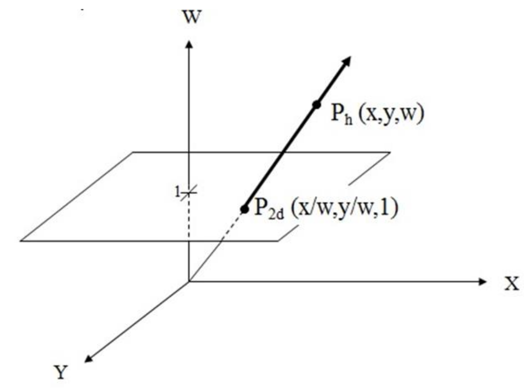
$$\mathbf{P} = [x, y, z, w]^T,$$

  - **When w is not 0, we can get 3-dimension point's coordinate by the following:**

$$x \leftarrow x/w, \quad y \leftarrow y/w, \quad z \leftarrow z/w$$

  - **When w is 0, P is a vector**

- **Note: In homogenous coordinate, a straight line through the origin is mapping to a point in three-dimensional space**



Homogeneous coordinate and CG

- Homogeneous coordinates is the key to all computer graphics systems
  - All standard transform (rotate, zoom) can be applied to 4×4 matrix multiplication
  - Hardware pipeline system can be applied to the four-dimensional representation
  - For the orthogonal projection, you can ensure vector by $w = 0$, ensure point by $w = 1$
  - For perspective projection, the need for special treatment: perspective division

Coordinate transformation

- **Consider the same vector with two different basis:**

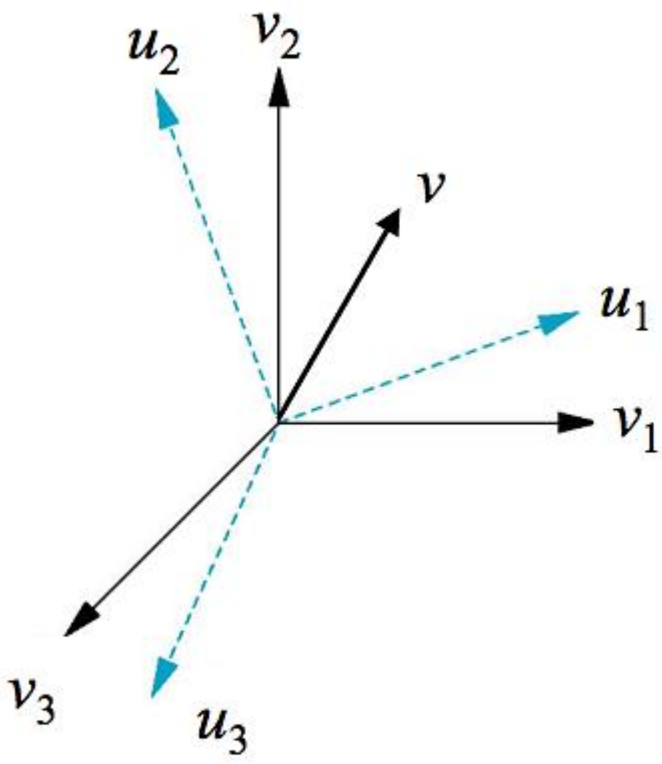
$$a = [\alpha_1, \alpha_2, \alpha_3]^T$$
$$b = [\beta_1, \beta_2, \beta_3]^T$$

- **Among them**

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [v_1, v_2, v_3]\ [\alpha_1, \alpha_2, \alpha_3]^T$$
$$= \beta_1 u_1 + \beta_2 u_2 + \beta_3 u_3 = [u_1, u_2, u_3]\ [\beta_1, \beta_2, \beta_3]^T$$

Use 1st Basis to represent 2nd

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



Matrix form

- **All coefficients define a 3 × 3 matrix**

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

- **We can connect the two basis by**

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

Changing the frame

- Perform similar operation to homogeneous coordinate
- Consider frame

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

- Any vector or point can be represented by one of them

One Frame represent another

- **Similar to the changes in basis, we have**

$$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$$
$$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$$
$$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$$
$$Q_0 = \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + P_0$$

- **These equations can be written in the form**

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = \mathrm{M} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix},$$

One Frame represent another

- where now M is the 4 × 4 matrix

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

- M is called the matrix representation of the change of frames.

One Frame represent another

- We can also use M to compute the changes in the representations directly.
- Suppose that a and b are the homogeneous coordinate representations either of two points or of two vectors in the two frames. Then

$$\mathbf{b}^T \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = \mathbf{b}^T \mathbf{M} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} = \mathbf{a}^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}.$$

- Hence: $\mathbf{a} = \mathbf{M}^T \mathbf{b}.$

One Frame represent another

- **When we work with representations, as is usually the case, we are interested in M^T, which is of the form**

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

and is determined by 12 coefficients(4 coefficients is fixed).

Transform representation

- Any point or vector has the same form in two frames
  - 1^{st} frame: $a = [\alpha_1, \alpha_2, \alpha_3, \alpha_4]^T$
  - 2^{nd} frame: $b = [\beta_1, \beta_2, \beta_3, \beta_4]^T$

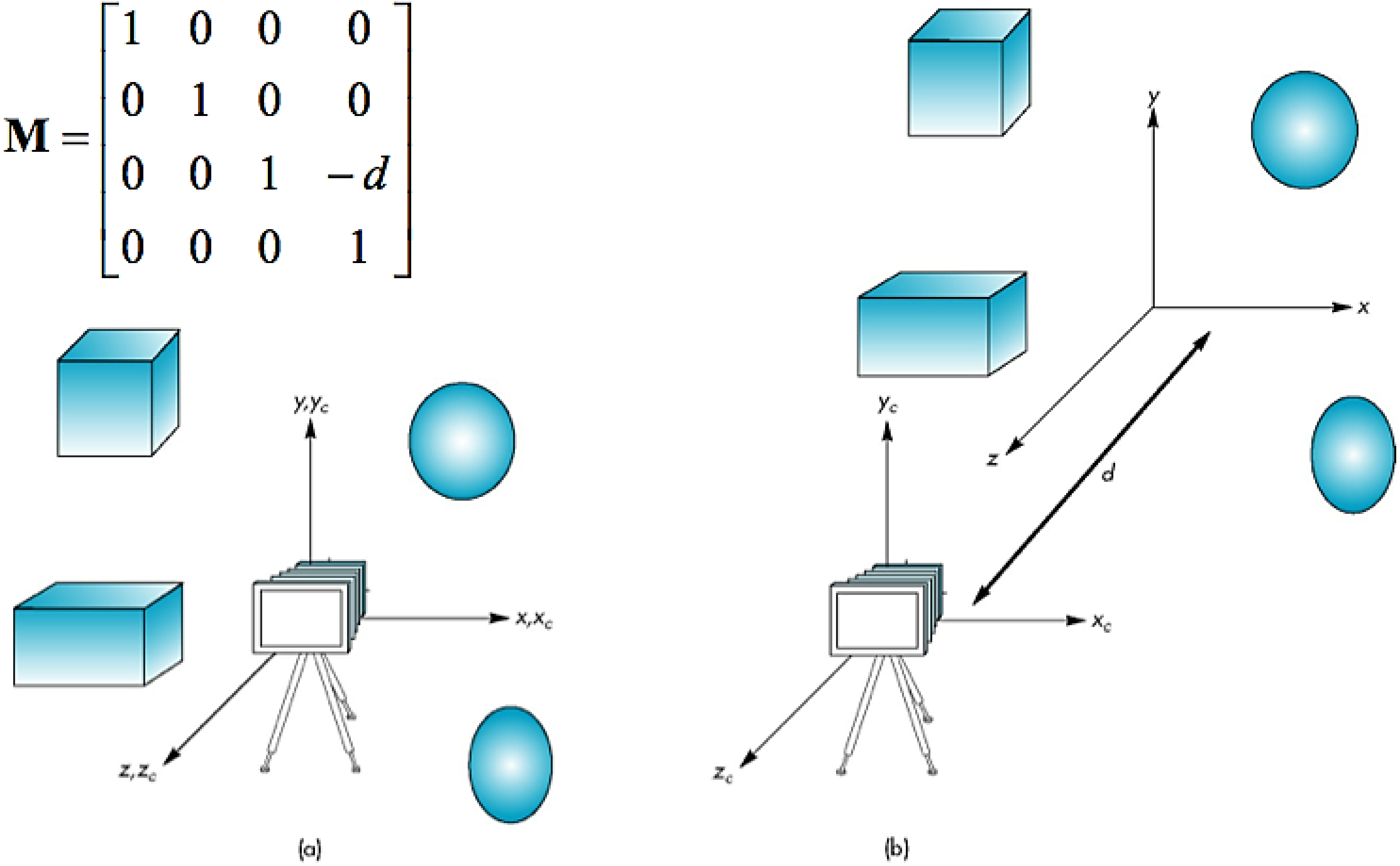
When represents a point $\alpha_4 = \beta_4 = 1$,When represents a vector $\alpha_4 = \beta_4 = 0$, and $a = M^T b$,The size of matrix M is 4x4, which defines a affine transformation with homogeneous coordinate.

Advantages of affine transformation

- All of the affine transformation remain **linearity**
- The most important is that all affine transformations can be represented as matrix multiplications in homogeneous coordinates.
  - The uniform representation of all affine transformations makes carrying out successive transformations far easier than in three-dimensional space.
  - modern hardware implements homogeneous coordinate operations directly, using parallelism to achieve high-speed calculations.

Movement of the camera

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

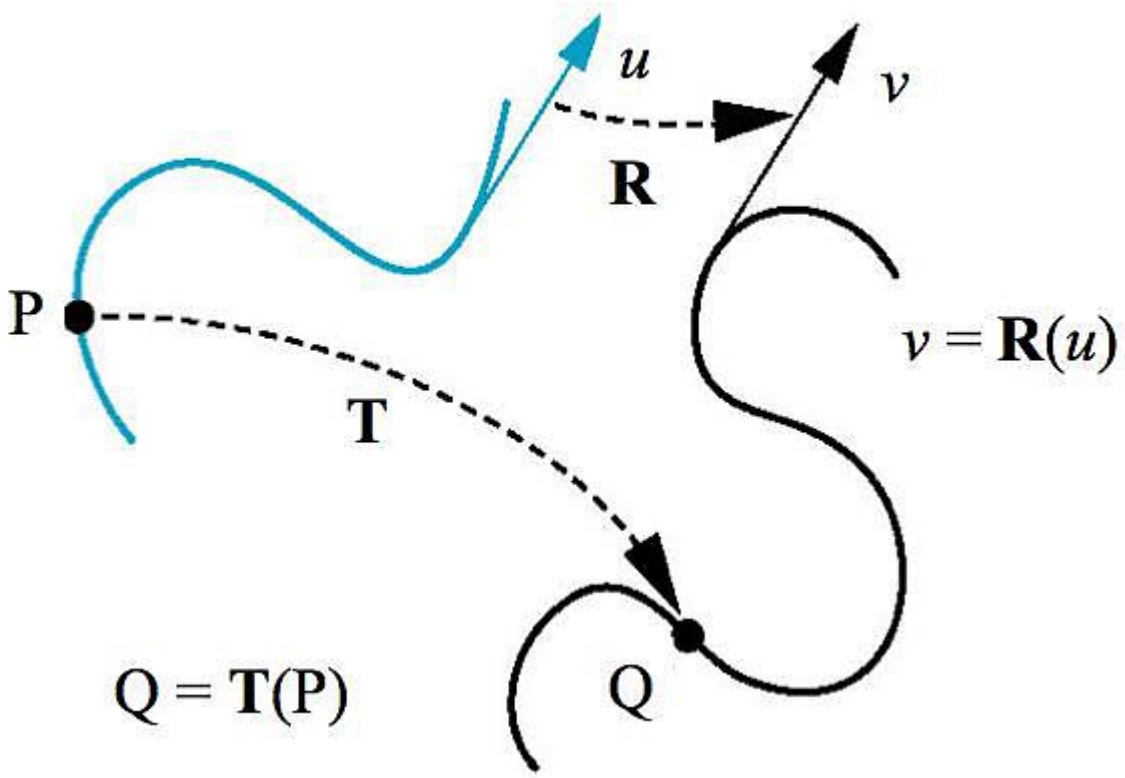


Outline

- Geometry
- Representation
- **Transformation**
- Transformation in OpenGL

General transformation

- The so-called transformation is to map points to other points, the vectors are mapped to other vectors



Linear Transformations

- Combinations of
  - shear
  - scale
  - rotate
  - reflect

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$x' = ax + by$$
$$y' = cx + dy$$

- Properties (why?)
  - satisfies T(sx+ty) = s T(x) + t T(y)
  - origin maps to origin
  - Straight lines map to straight lines
  - parallel lines remain parallel
  - closed under composition

Affine Transformations

- Combinations of
  - linear transformations
  - translations

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

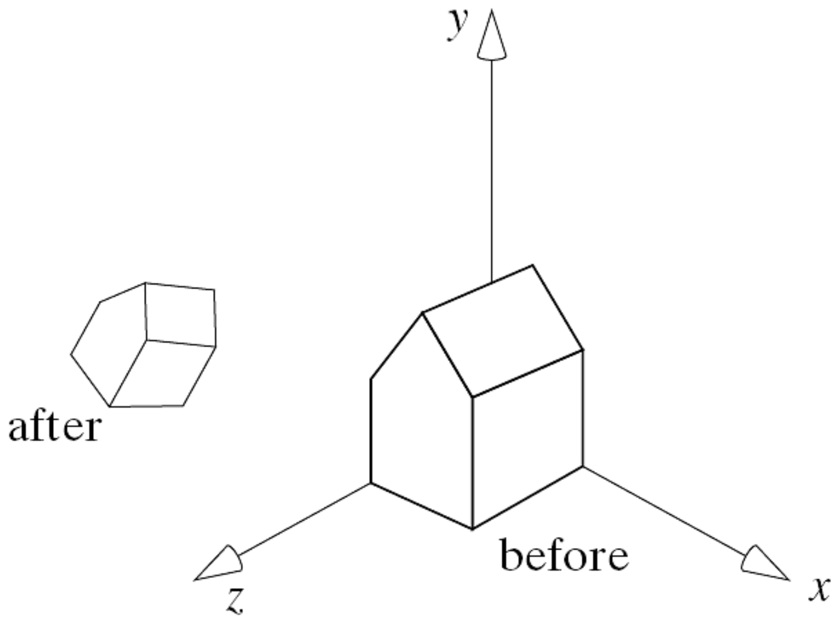
- Properties (why?)
  - origin does not necessarily map to origin
  - lines map to lines
  - parallel lines remain parallel
  - ratios are preserved
  - closed under composition

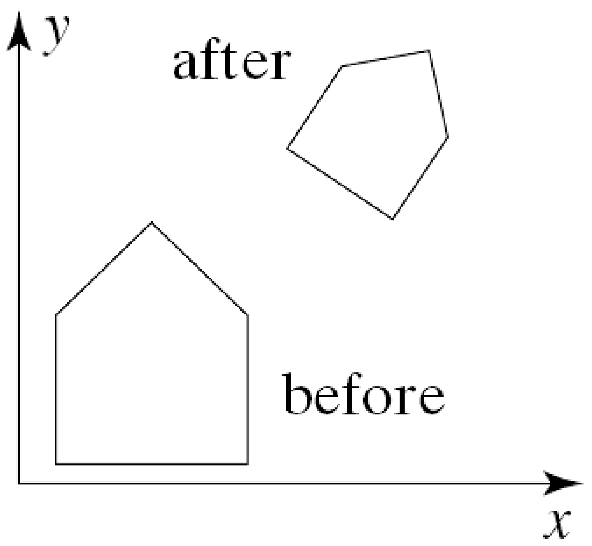
Affine transformation

- Maintaining collinearity
- Many important physical feature of transformation
  - Rigid transformation: rotation, translation（Only alter position and Orientation）
  - Other affine transformations（Scaling, shear） will alter object’s shape.
- In CG world，we just need to change the line of the two endpoints, and the system automatically after the conversion to draw the line between the two endpoints.

Why we need transformation？

- Procedures to compute new positions of objects
- Used to modify objects or to transform (map) from one coordinate system to another coordinate system

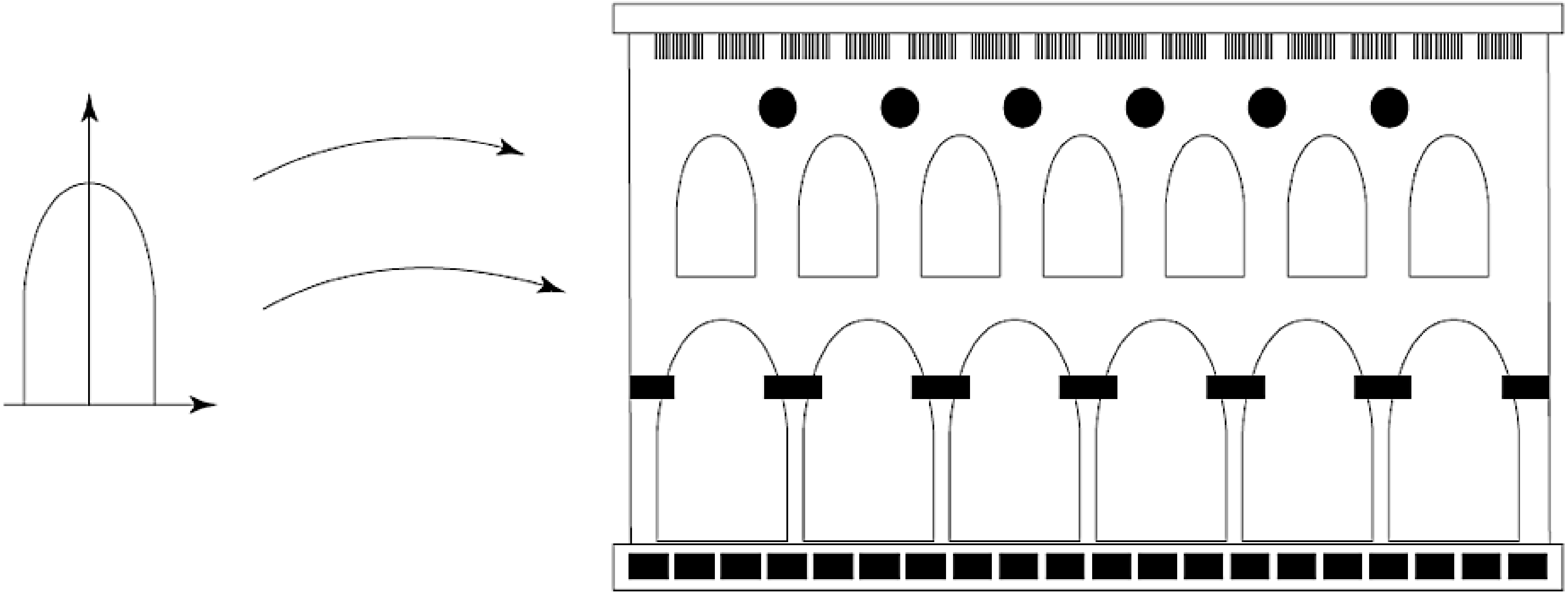




As all objects are eventually represented using points, it is enough to know how to transform points.

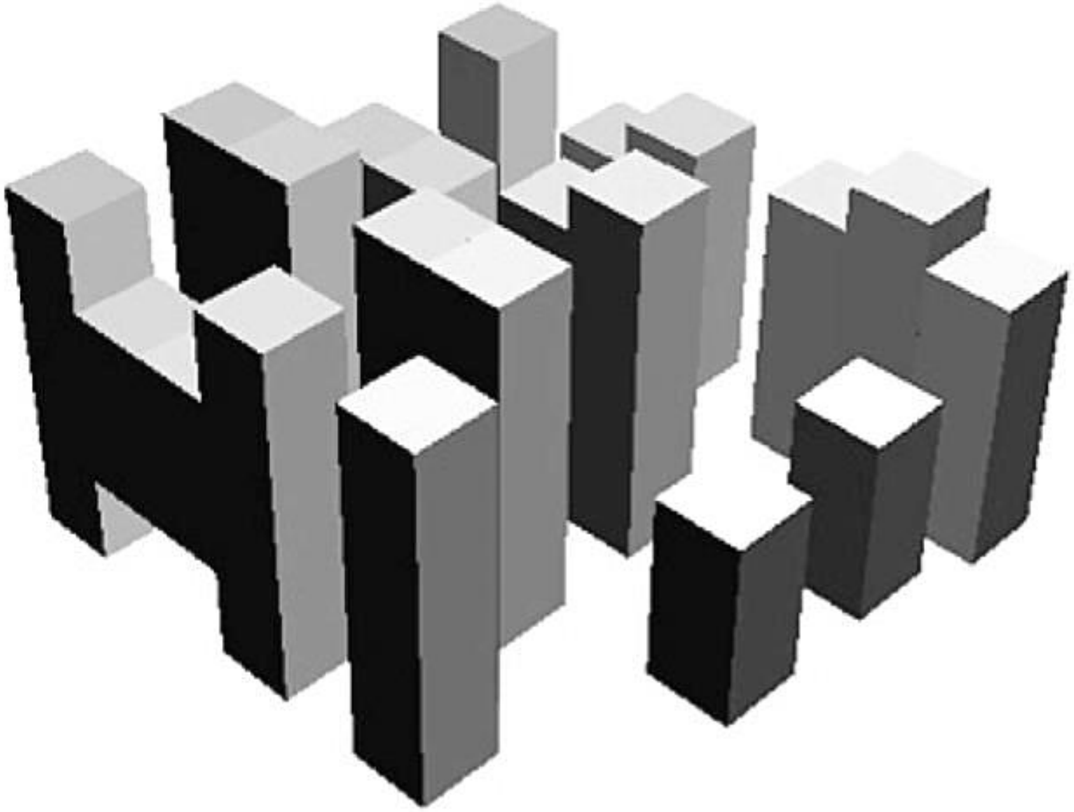
Function 1

- Construct scenes



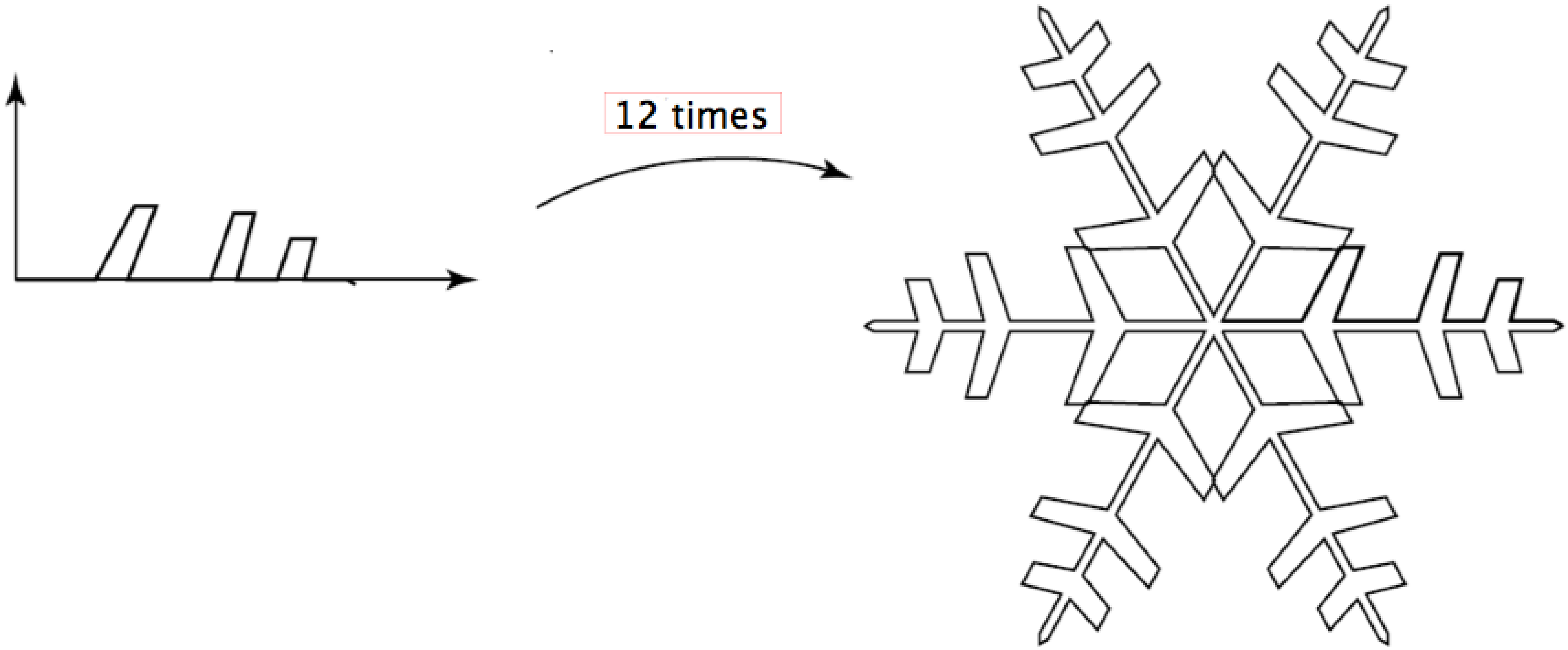
Function 1

- Construct 3D scene



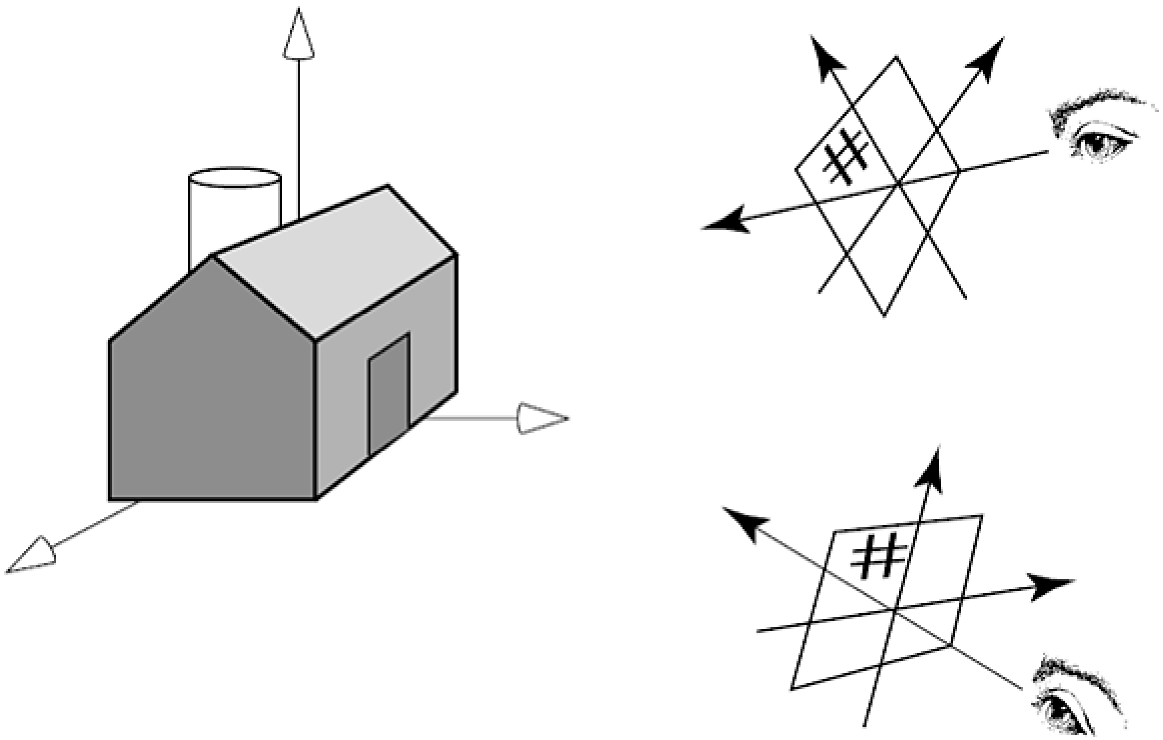
Function 1

- Snowflake structure



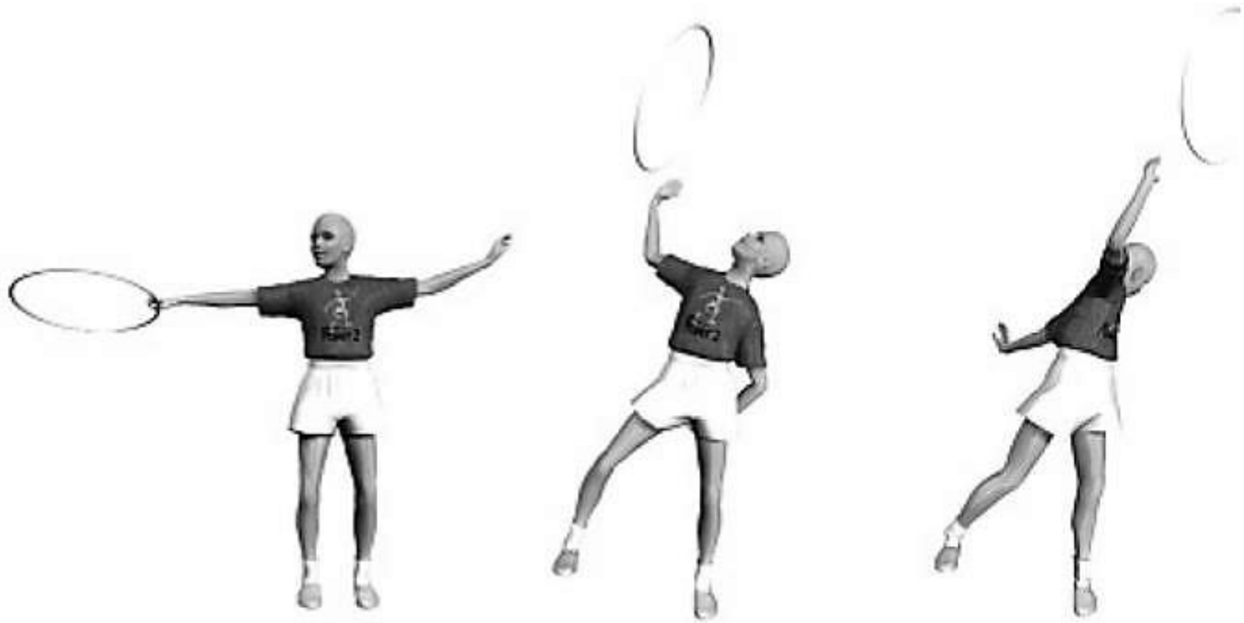
Function 2

- The designer may want to view object from different angles of the same scene, then he can:
  - the object is fixed, the position of the camera is transformed

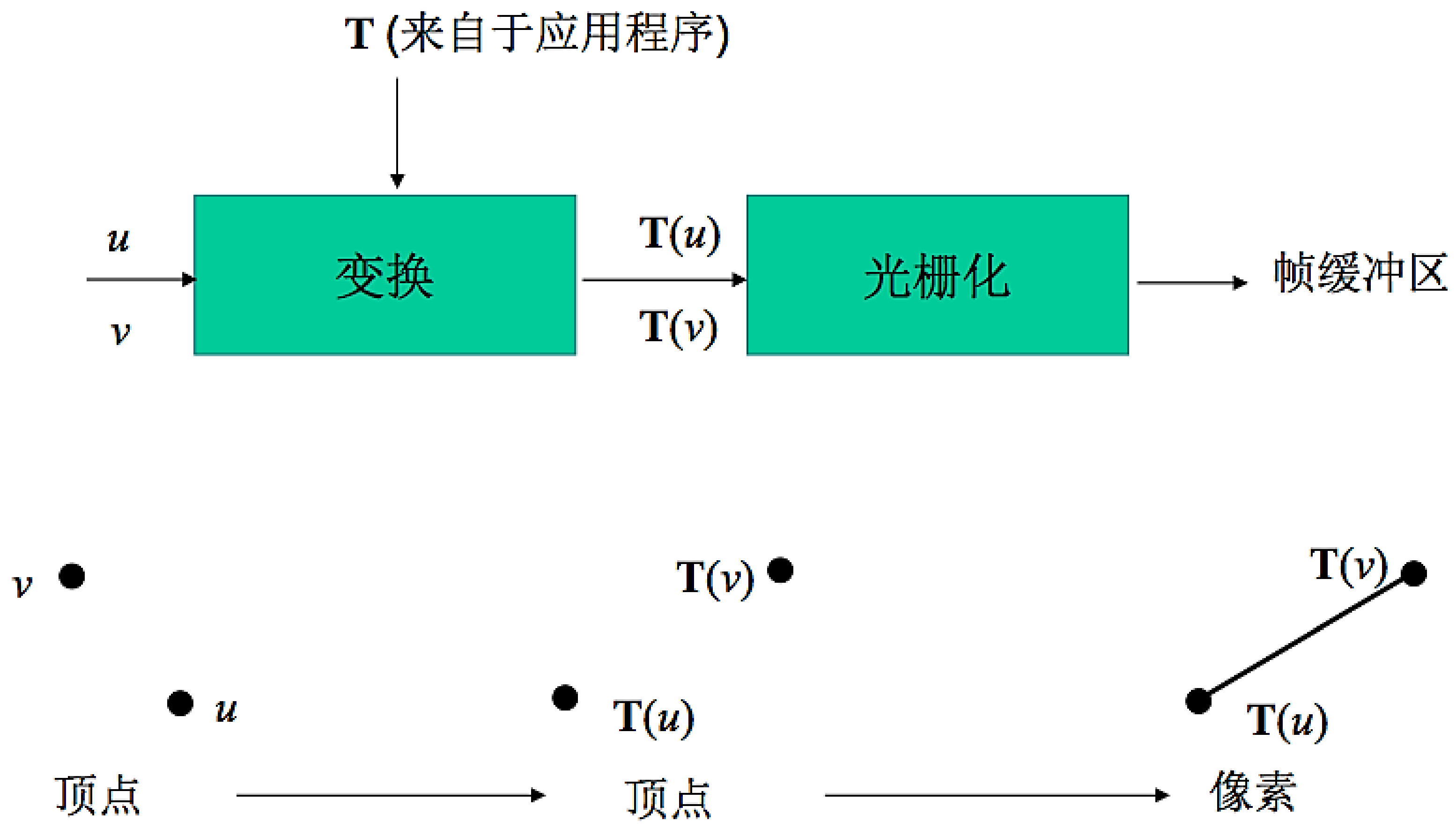


Function 3

- In computer animation, in the adjacent frames, the position of several objects move relative to each other.
  - This is done by translating and rotating the local coordinate system.

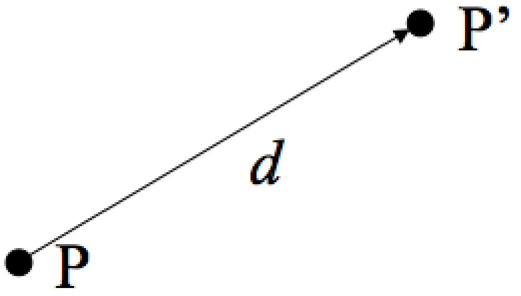


Pipeline



Translation

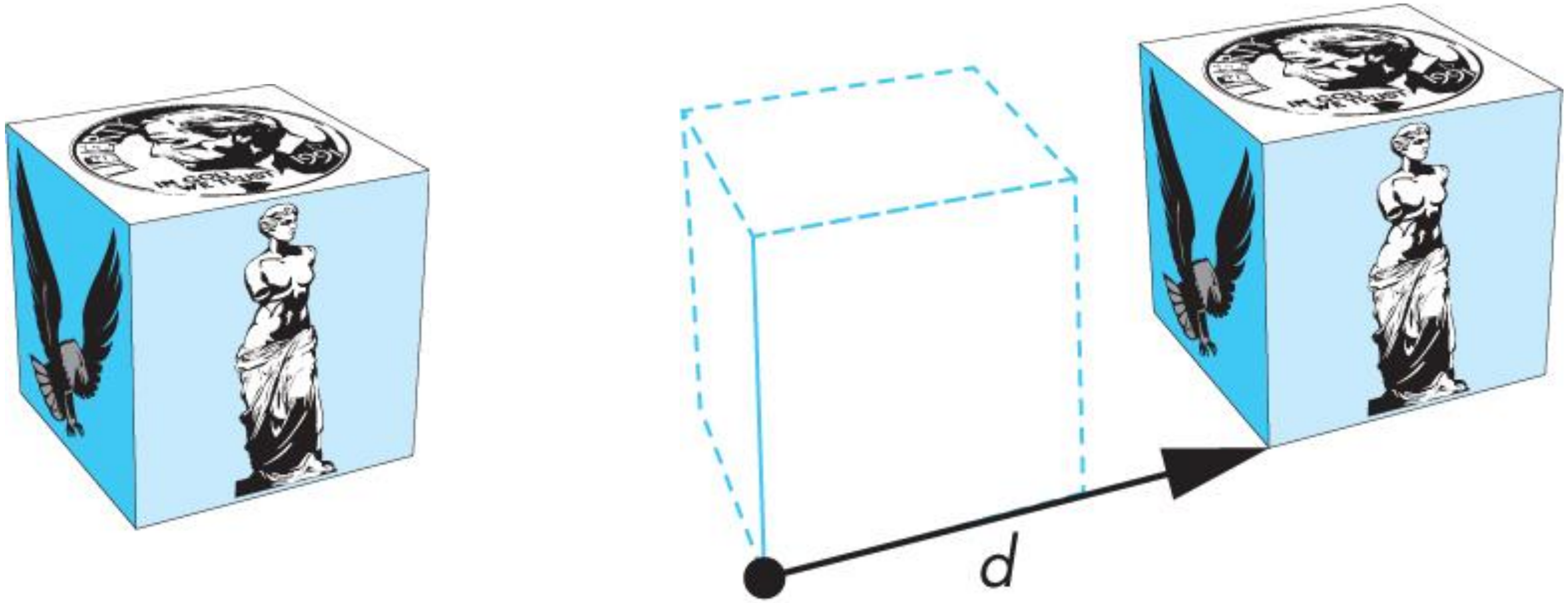
- Put a point to a new position



- Determined by a vector d
  - Three free degrees
  - P' = P + d

Translation of objects

- Translate all points of an object along the same vector.



Before After

Representation of Translation

- Homogeneous coordinates in a frame

$$p = [x, y, z, 1]^T$$
$$p' = [x', y', z', 1]^T$$
$$d = [d_x, d_y, d_z, 0]^T$$

- Then p' = p + d or

$$x' = x + d_x,$$
$$y' = y + d_y,$$
$$z' = z + d_z.$$

注意：这个表达式是四维的，而且表示的是：
点 = 点 + 向量

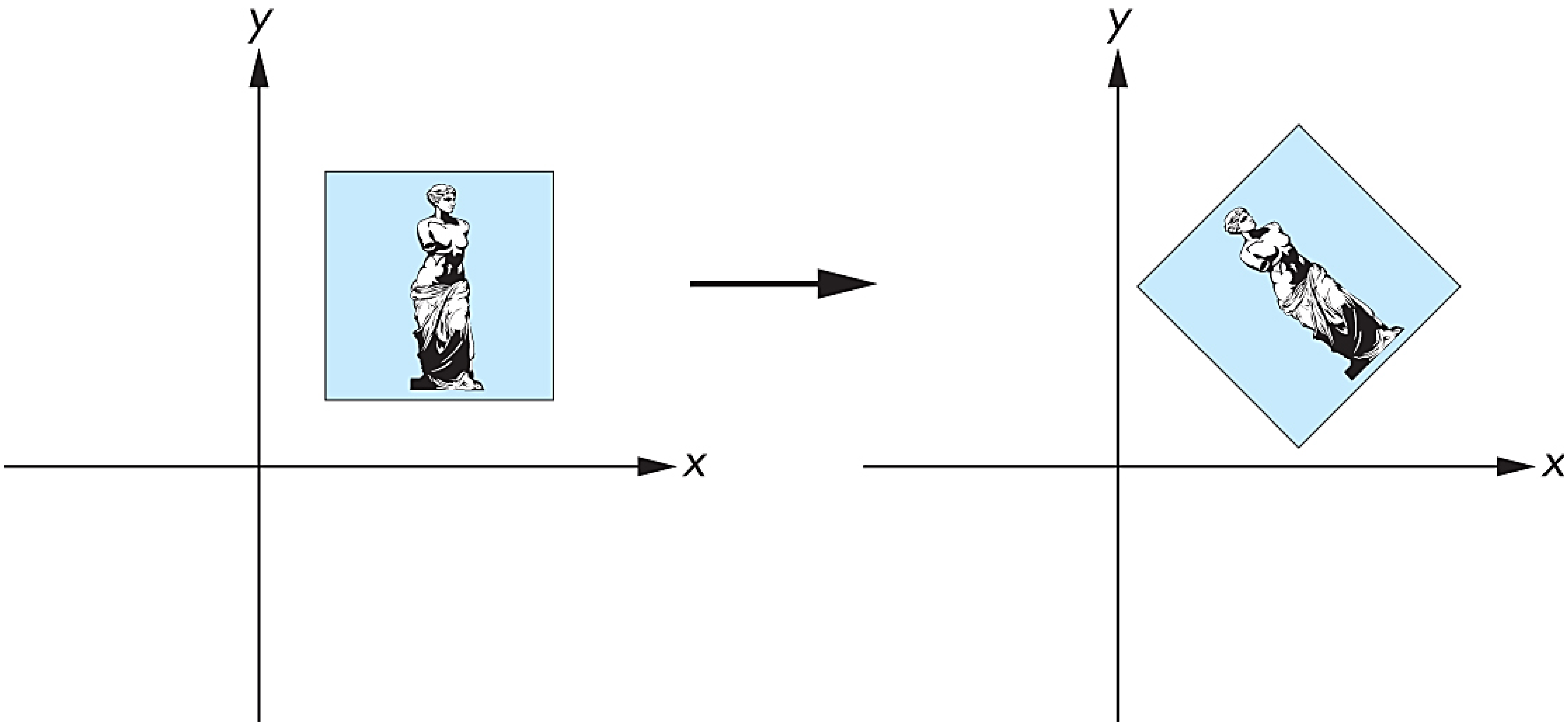
Translation matrix

- Using a 4 × 4 homogeneous coordinates matrix T to represents the translation
- p' = Tp

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

- This form is more easily achieved, because all the affine transformation can be used in this kind of form

2D rotation



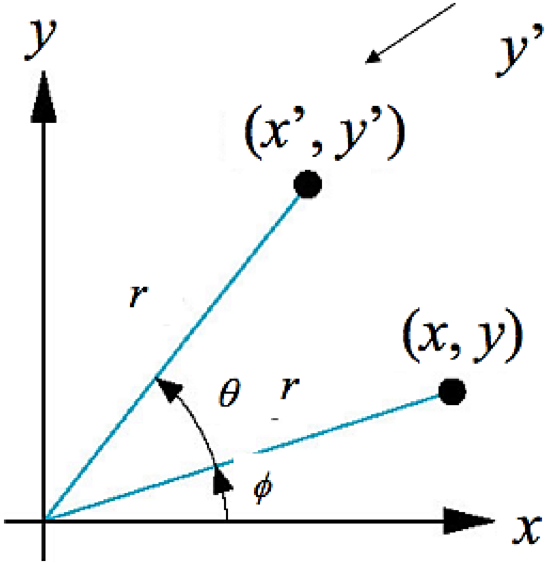
2D rotation

- Consider θ degrees rotation about the origin

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

cos(A+B) = cosAcosB-sinAsinB

sin(A+B) = sinAcosB+cosAsinB



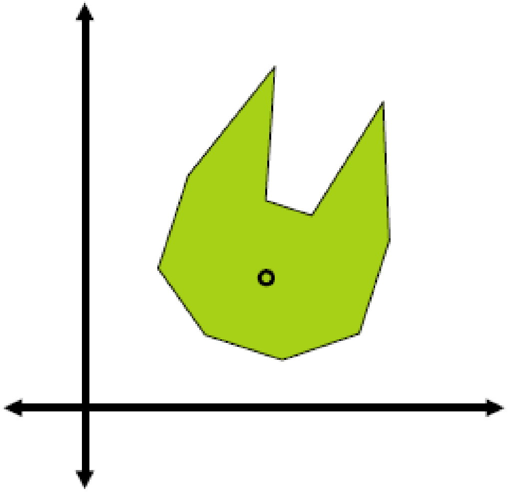
$$\boxed{\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}}$$

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

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

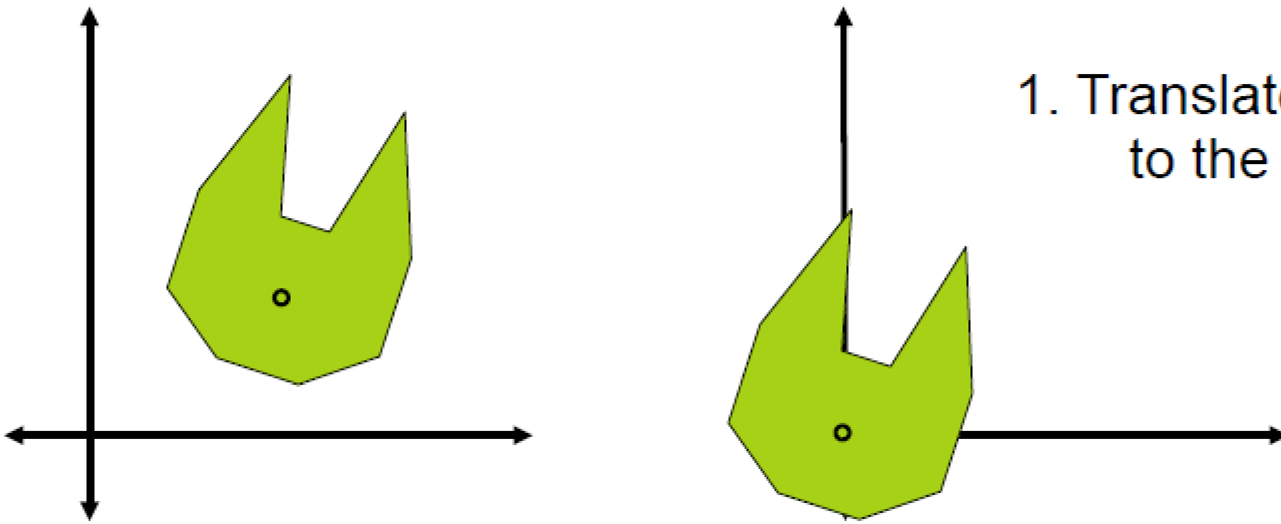
Simple Rotate

To rotate the cat's head about its nose



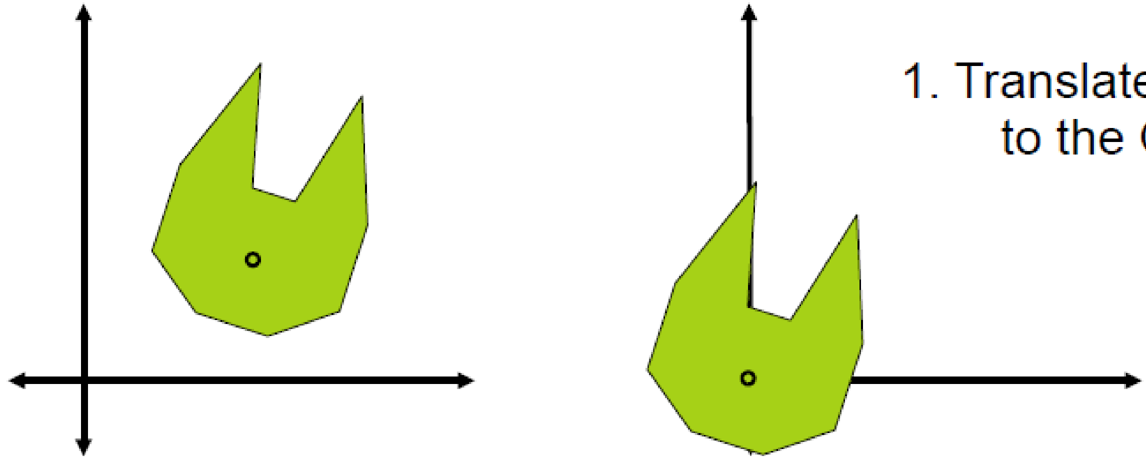
Simple Rotate

To rotate the cat's head about its nose

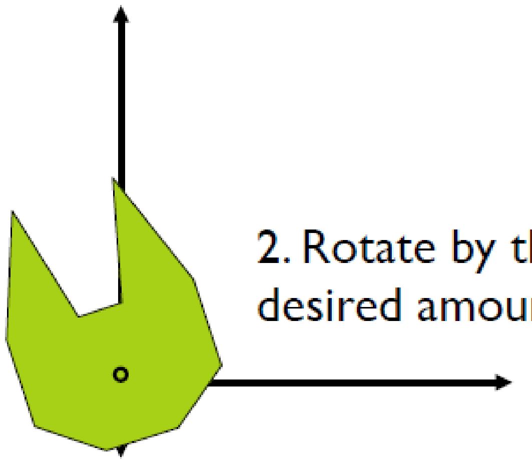


1. Translate the Nose to the Origin

Simple Rotate

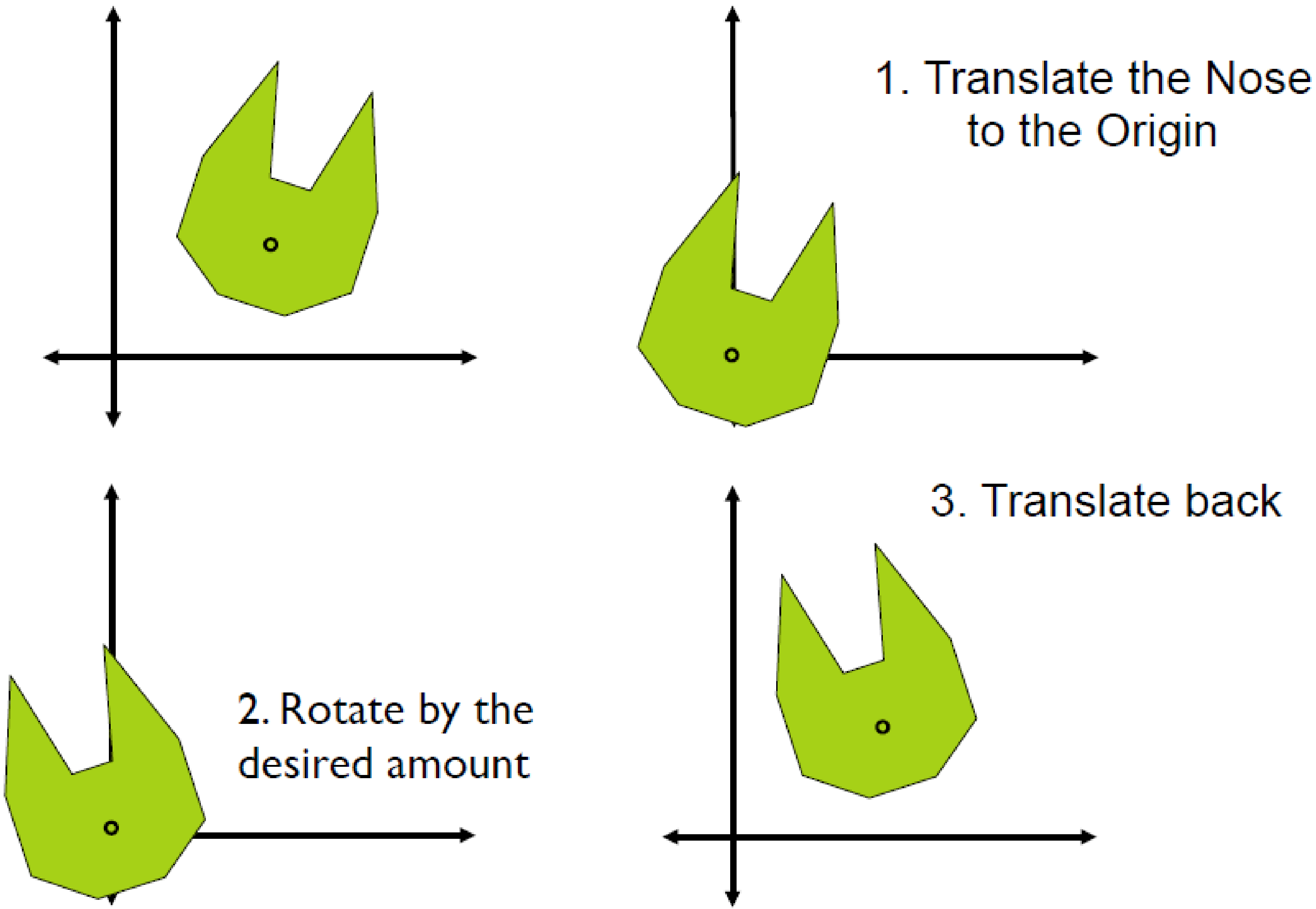


1. Translate the Nose to the Origin



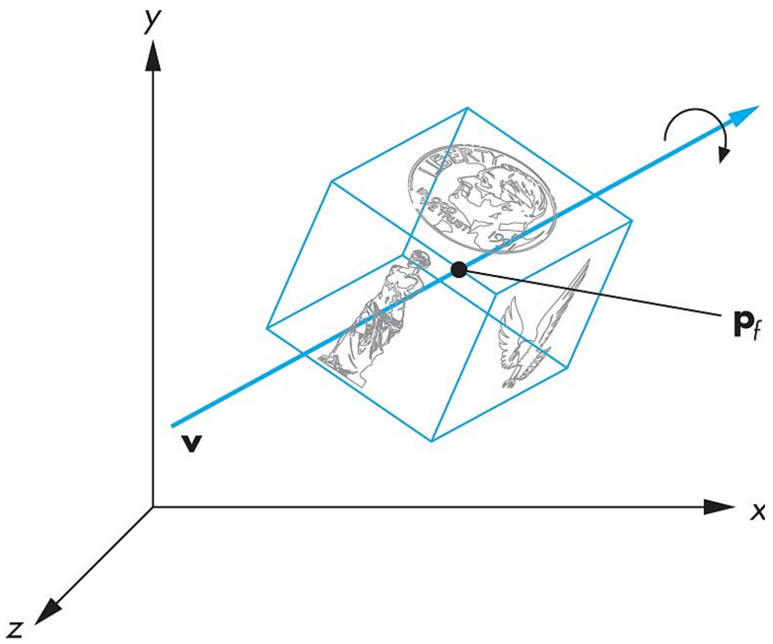
2. Rotate by the desired amount

Simple Rotate



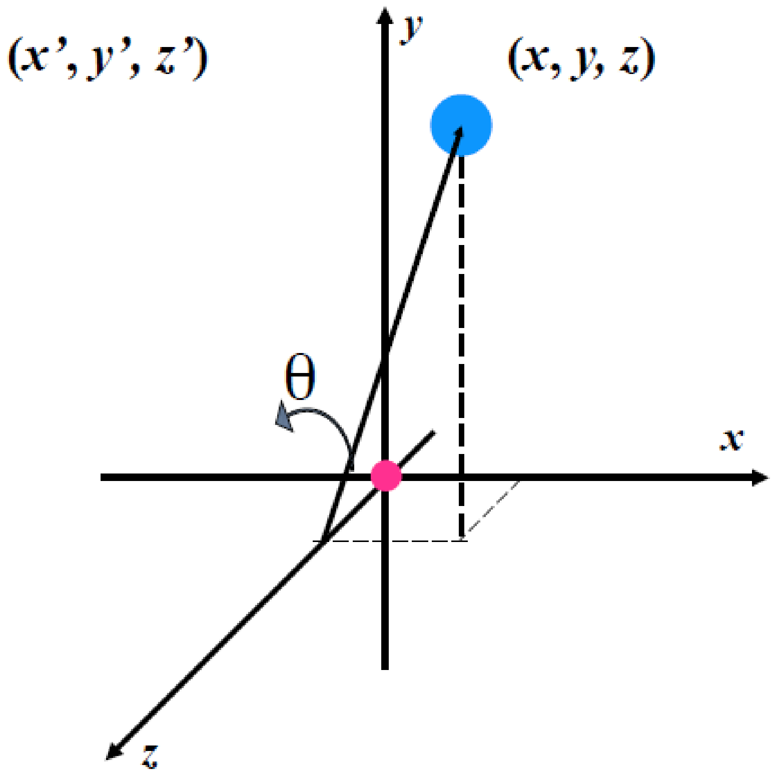
3D rotation

- Several special conditions:
  - Respectively rotatable around the x, y, z-axis
  - Rotate along the general axis through the origin
  - Rotate along a general axis except the origin



3D rotation around Z-axis

Rotation (around Z axis)

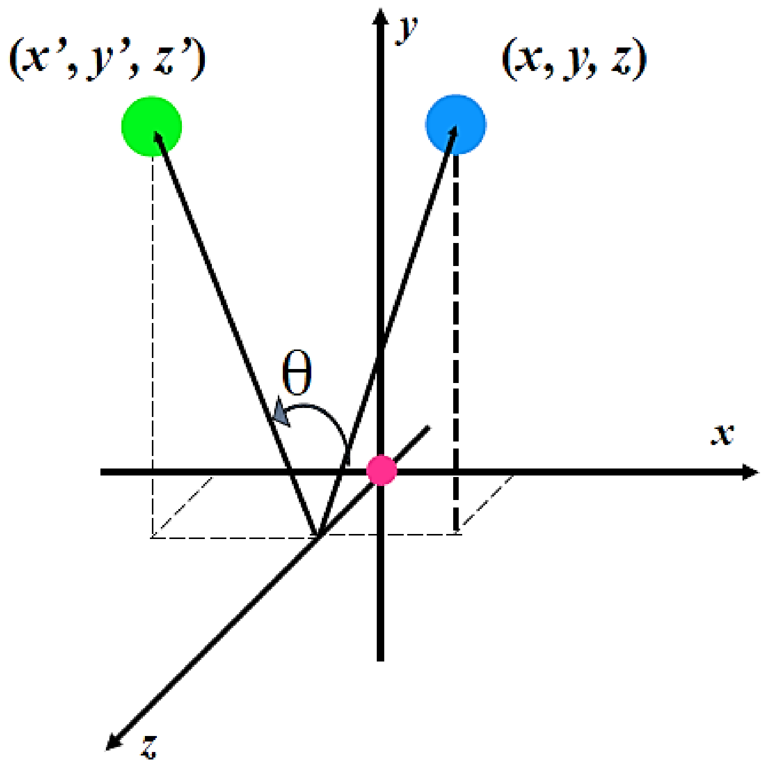


$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D rotation around Z-axis

Rotation (around Z axis)

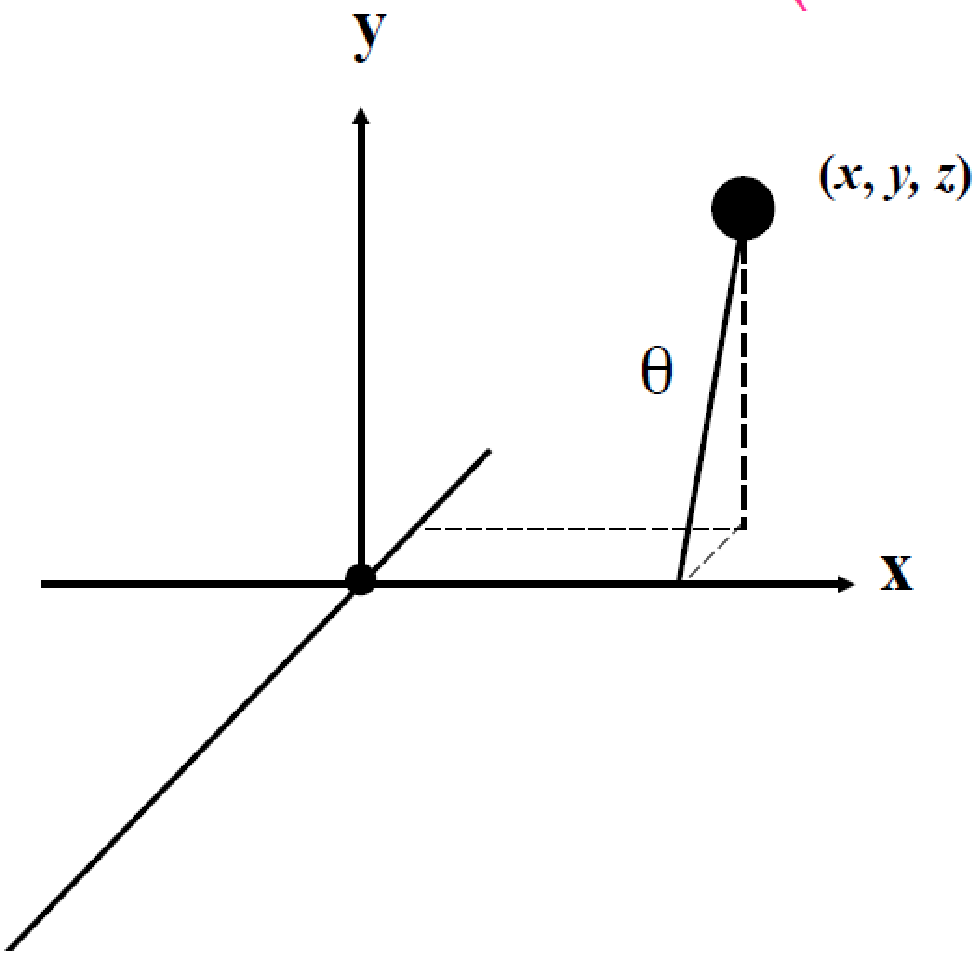


$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D rotation around X-axis

Rotation (around X axis)

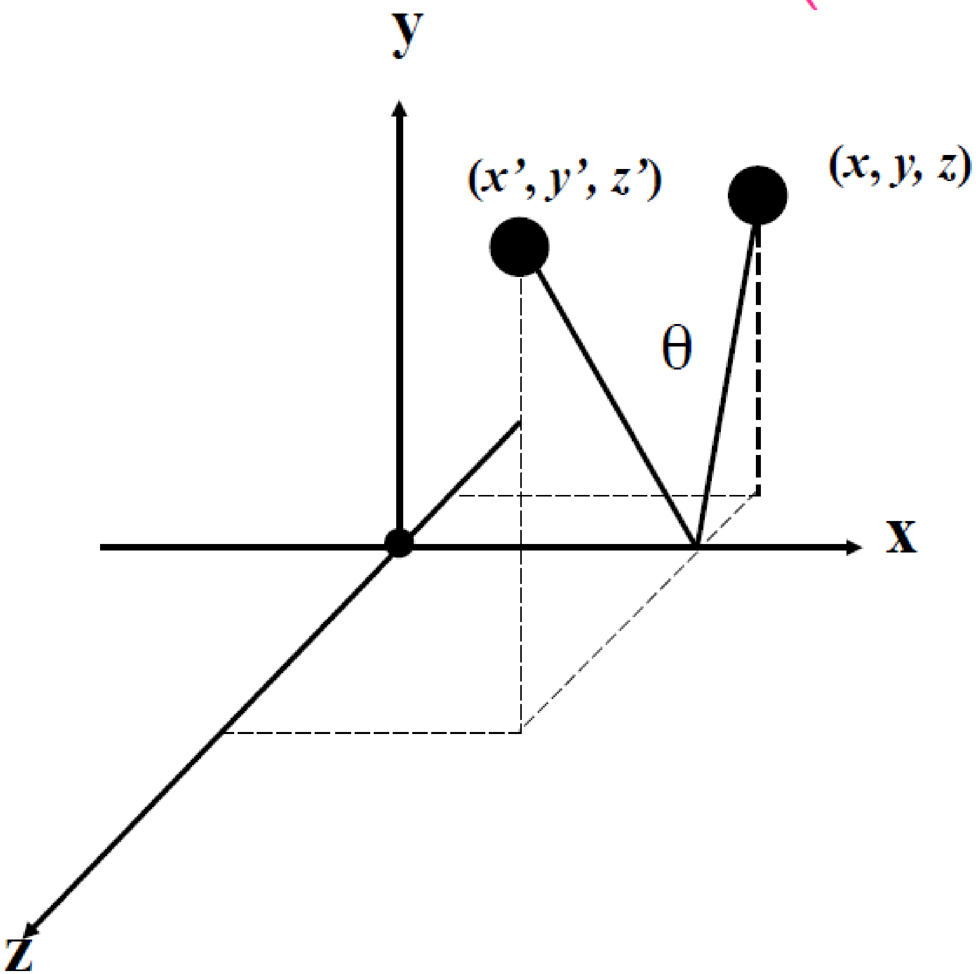


$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D rotation around X-axis

Rotation (around X axis)



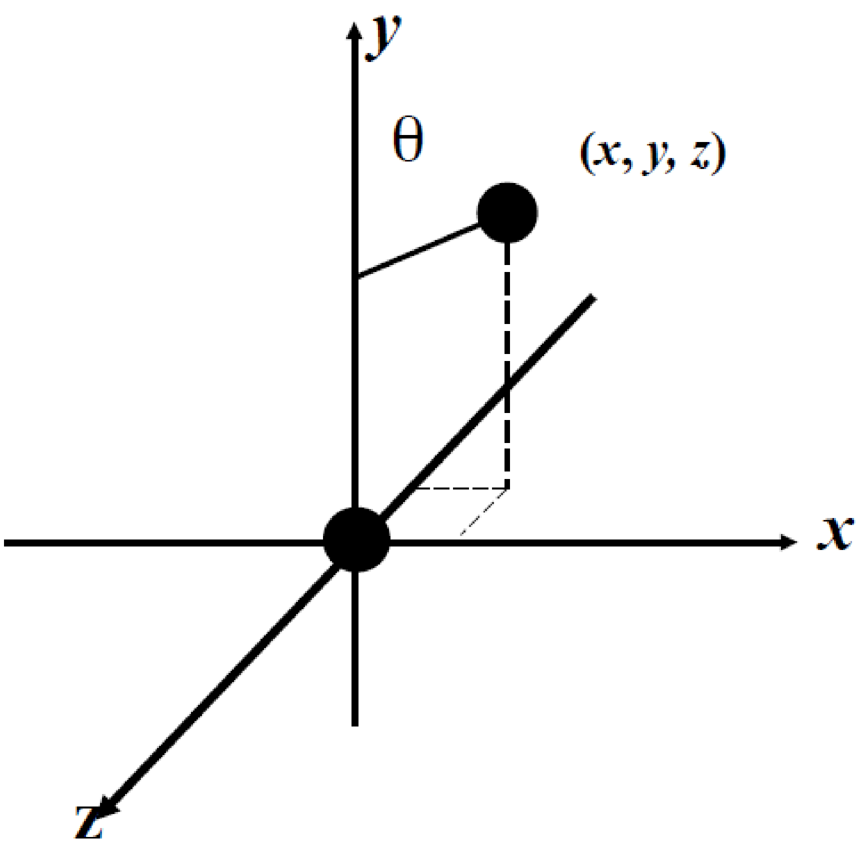
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Computer Graphics 2014, ZJU

3D rotation around Y-axis

- Rotation (around Y axis)

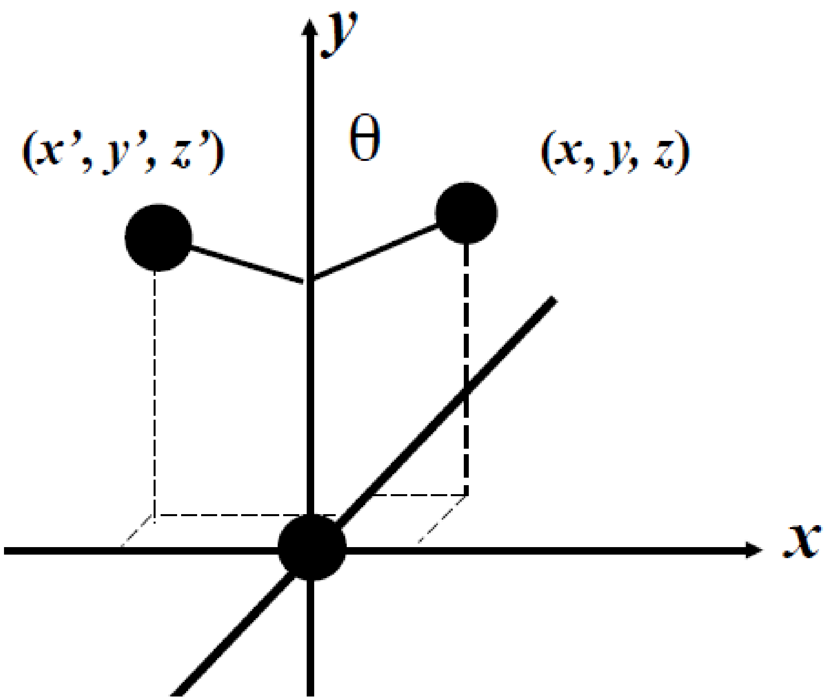


$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

question : why?

Computer Graphics 2014, ZJU

3D rotation around Y-axis



$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

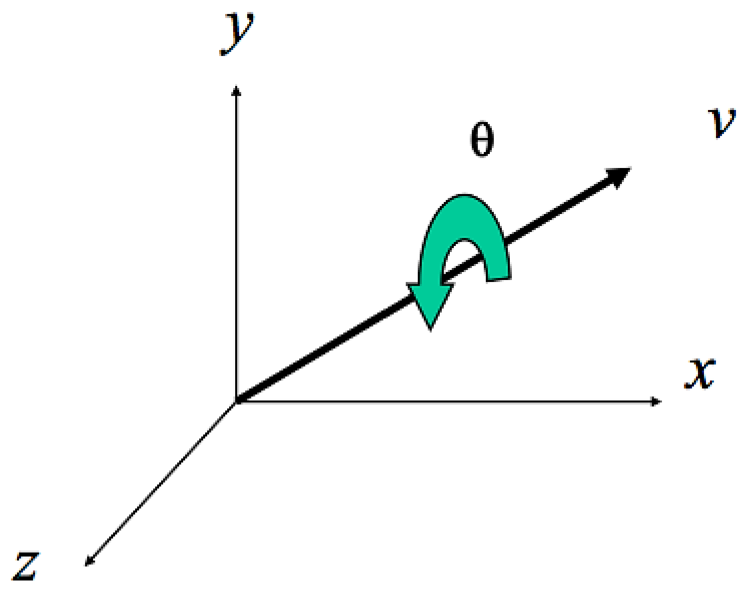
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotate along the general axis through the origin

- Can be decomposed as the combination of rotation on x, y, z axis

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

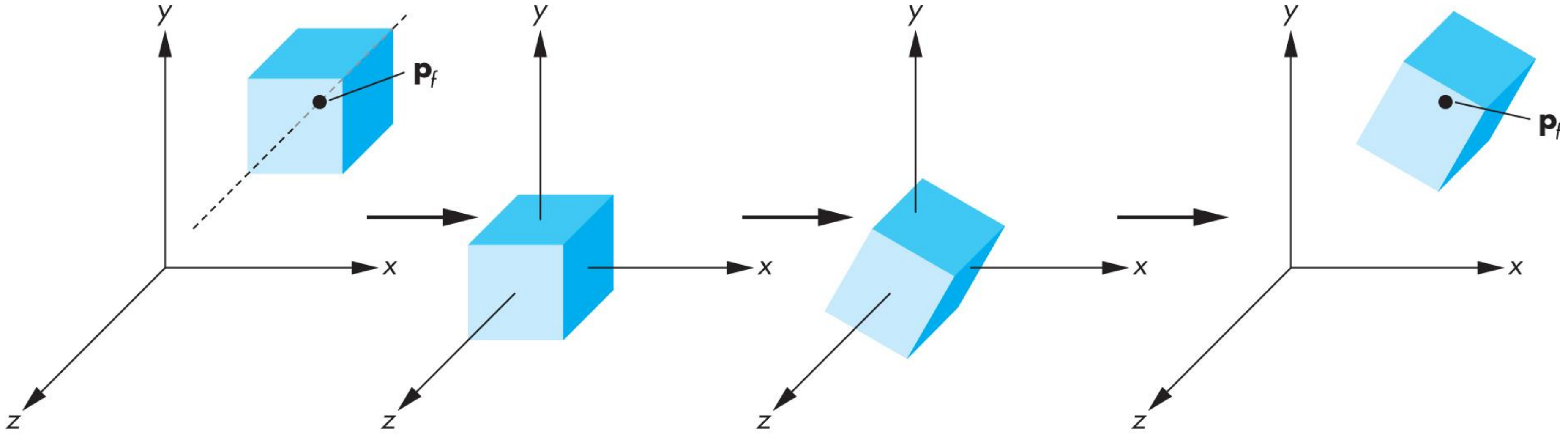
- Note that the rotation order can not be exchanged



Rotate around a fixed point except origin

- Move the fixed point to origin
- Rotate
- Move the fixed point back to its initial place

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

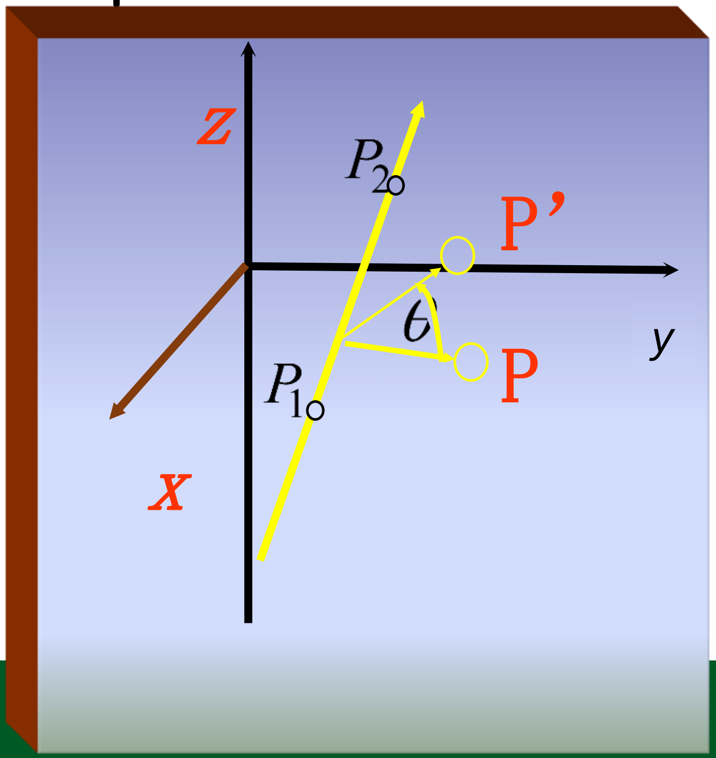


Rotation surrounding a general axis

- Given axis defined by two points(已知旋转轴):

$$P_1 = (x_1, y_1, z_1) \qquad P_2 = (x_2, y_2, z_2)$$

- P rotates to P’ with respect to the axis by θ
- We derive the rotation matrix by composition

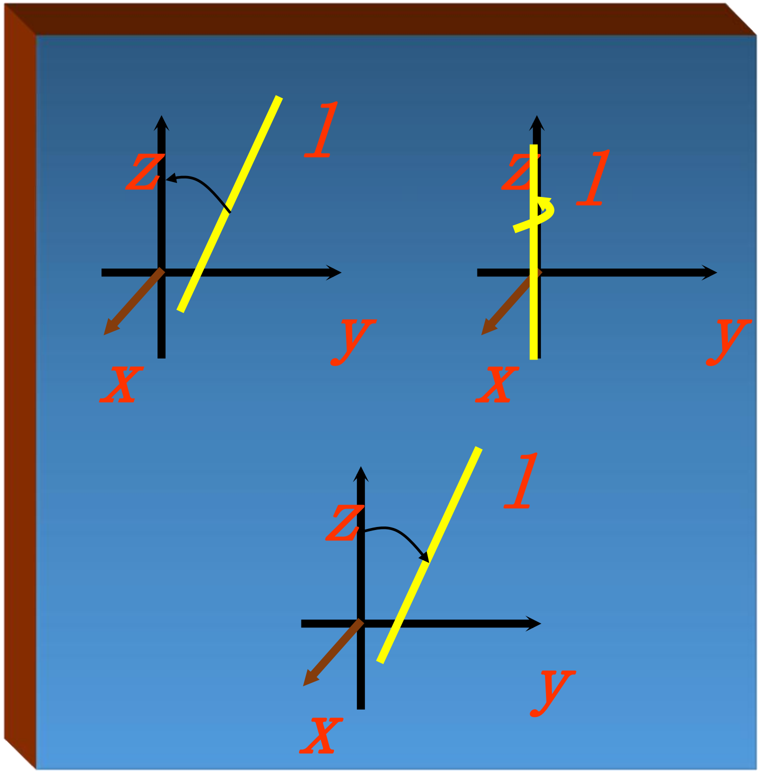


Three steps (三步骤)

(1) Transform l such that it overlaps with z-axis

(2) Rotate surrounding z-axis by θ

(3) Reverse transform



Step 1

(1) Transform l such that it overlaps with z-axis: can be decomposed three step again

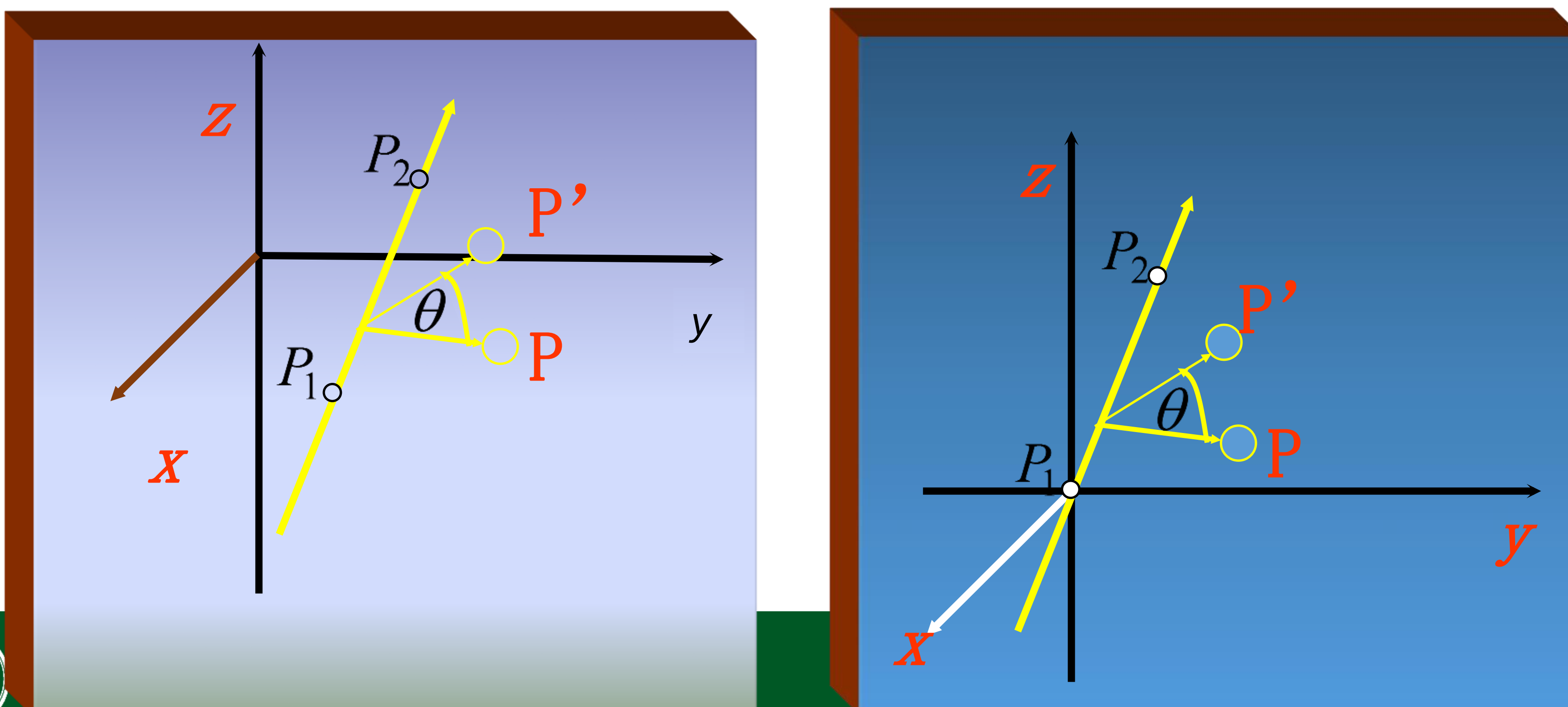
(1a) Translate such that l passes through the origin

(1b) Rotate surrounding x-axis such that l locate on the ZOX plane

(1c) Rotate around y-axis such that l locate on the ZOX plane

(1a) Translation to P_1

$$T(-x_1,-y_1,-z_1) = \begin{pmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



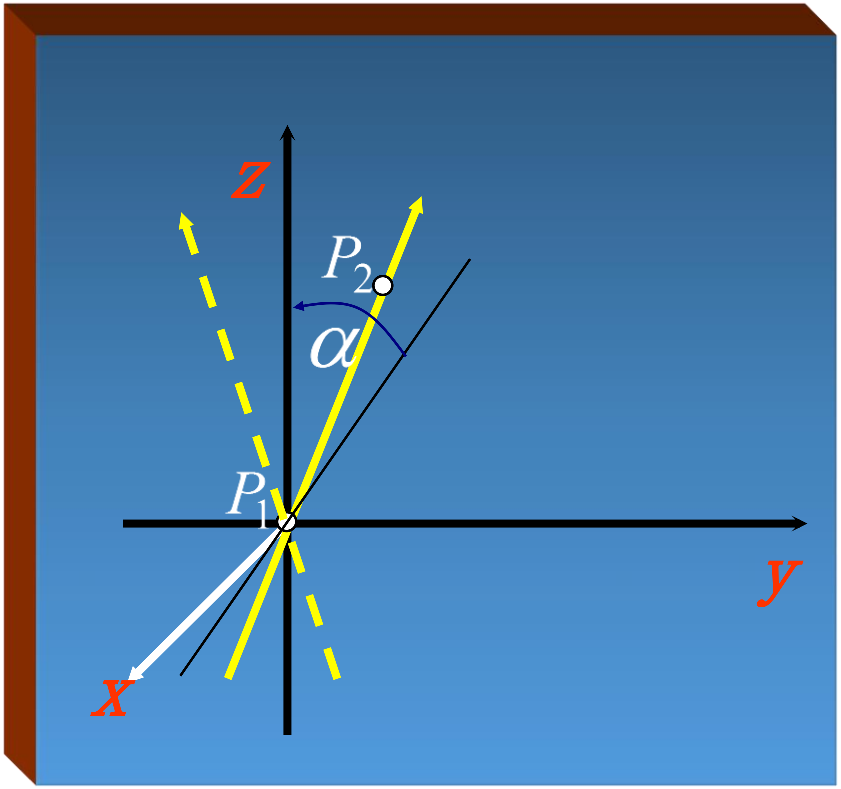
1b) Rotate by surrounding X-axis

$$R_x(\alpha)=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & \cos\alpha & -\sin\alpha & 0\\ 0 & \sin\alpha & \cos\alpha & 0\\ 0 & 0 & 0 & 1\end{pmatrix}.$$

α 角应当是l在YOZ平面的投影与z轴的夹角。因此：

$$\cos\alpha=\frac{z_2-z_1}{\sqrt{(y_2-y_1)^2+(z_2-z_1)^2}}$$

$$\sin\alpha=\frac{y_2-y_1}{\sqrt{(y_2-y_1)^2+(z_2-z_1)^2}}$$



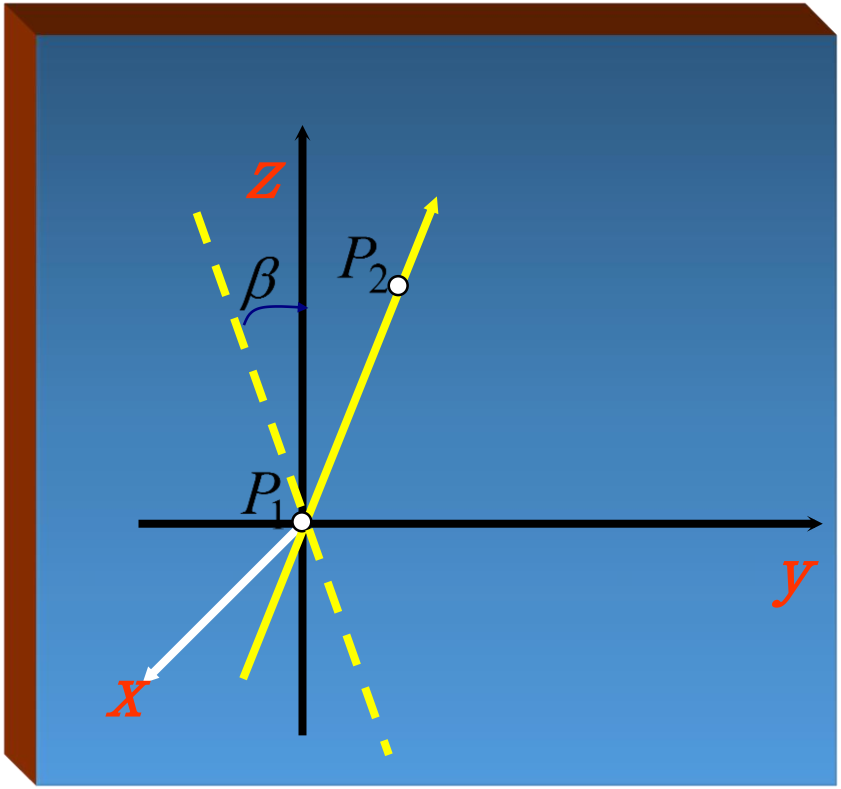
1c) Rotate by surrounding y-axis

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

这个$-\beta$角应当是l旋转到ZOX平面后与Z轴的夹角。因此：

$$cos\beta=\frac{\sqrt{(y_2-y_1)^2+(z_2-z_1)^2}}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}}$$

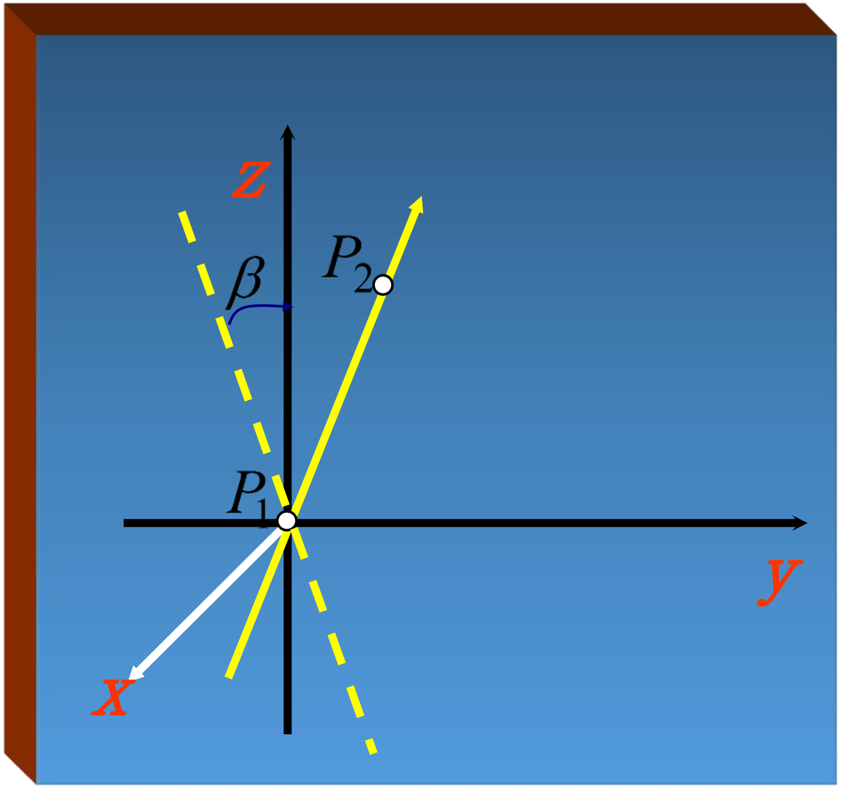
$$sin\beta=\frac{x_2-x_1}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}}$$



Step 2:

Rotate by θ in terms of z-axis

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



Step 3:

Use reverse transformation to derive final transformation

$$\mathrm{R}(\theta) = \underline{\mathrm{T}(x_1, y_1, z_1)} \bullet \underline{R_x^{-1}(\alpha)} \bullet \underline{R_y^{-1}(\beta)} \bullet R_z(\theta)$$
$$\bullet \underline{R_y(\beta)} \bullet \underline{R_x(\alpha)} \bullet \underline{\mathrm{T}(-x_1, -y_1, -z_1)}$$

Scaling

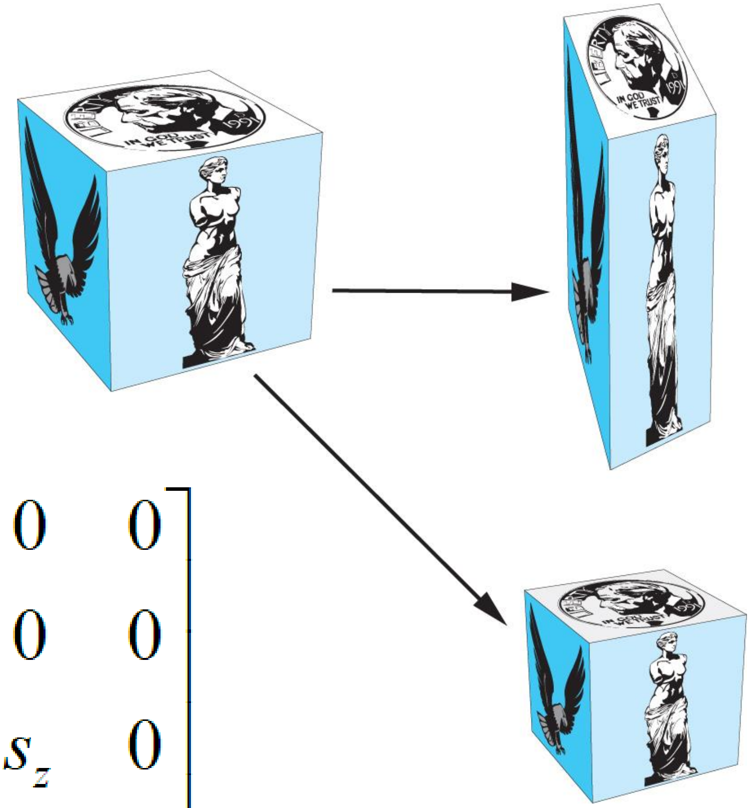
- Scale along each coordinate (origin is fixed point)

$$x' = s_x x$$

$$y' = s_y y$$

$$z' = s_z z$$

$$p' = Sp$$



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

Matrix Notations for Transformations

- Point P (x,y,z) is written as the column vector P_h
- A transformation is represented by a 4x4 matrix M
- The transformation is performed by matrix multiplication

$$Q_h = M * P_h$$

Matrix Representations and Homogeneous Coordinates

- Each of the transformations defined above can be represented by a 4x4 matrix
- Composition of transformations is represented by product of matrices
- So composition of transformations is also represented by 4x4 matrix

Inverses in 3D!

Transformation	Matrix Inverse
Scaling	$\begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Rotation	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\psi & sin\psi & 0 \\ 0 & -sin\psi & cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\phi & sin\phi & 0 & 0 \\ -sin\phi & cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Translation	$\begin{bmatrix} 1 & 0 & 0 & -dx \\ 0 & 1 & 0 & -dy \\ 0 & 0 & 1 & -dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Composite transformation

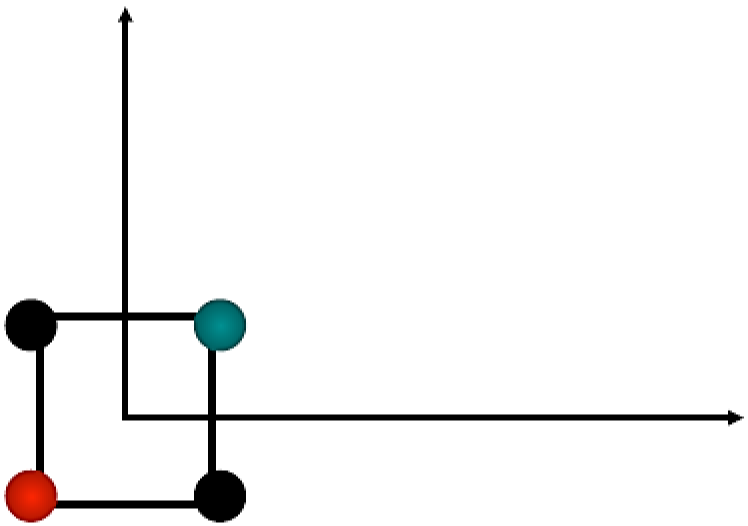
- Often want to combine transforms（E.g. first scale by 2, then rotate by 45 degrees
  - Advantage of matrix formulation: All still a matrix
  - Because many vertices have the same transformation, the price to construct matrix M = ABCD is small
- The difficulty is how to construct a transformation matrix to meet the requirements in accordance with the requirements of the application

Matrix Composition

- matrices are convenient, efficient way to represent series of transformations
  - hardware matrix multiply
  - From the mathematical point of view, the following representation is equivalent： matrix multiplication is associative
    - **p’** = (T*(R*(S***p**)))
    - **p’** = (T*R*S)***p**
- procedure
  - correctly order your matrices!
  - multiply matrices together
  - result is one matrix, multiply vertices by this matrix
  - all vertices easily transformed with one matrix multiply

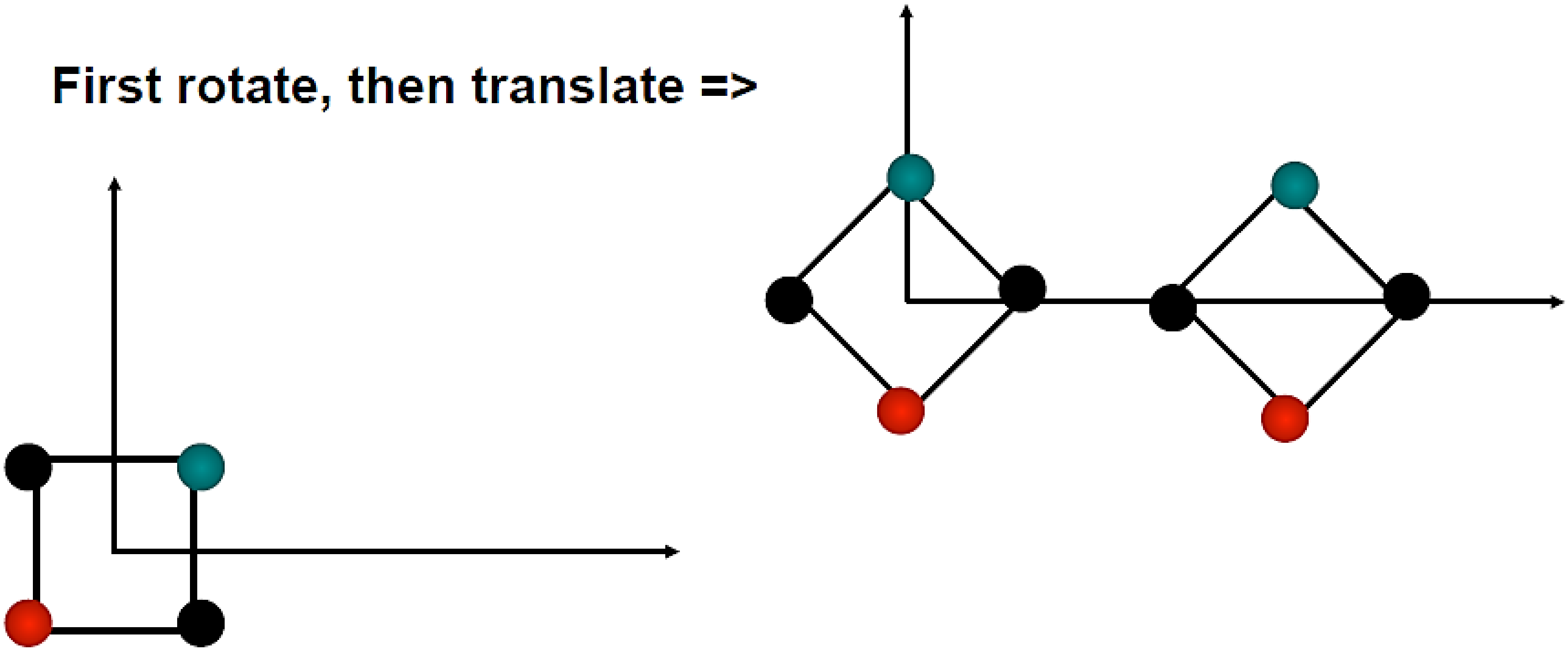
Matrix Multiplication is Not Commutative (不可交换)

Transformation sequence is not commutative



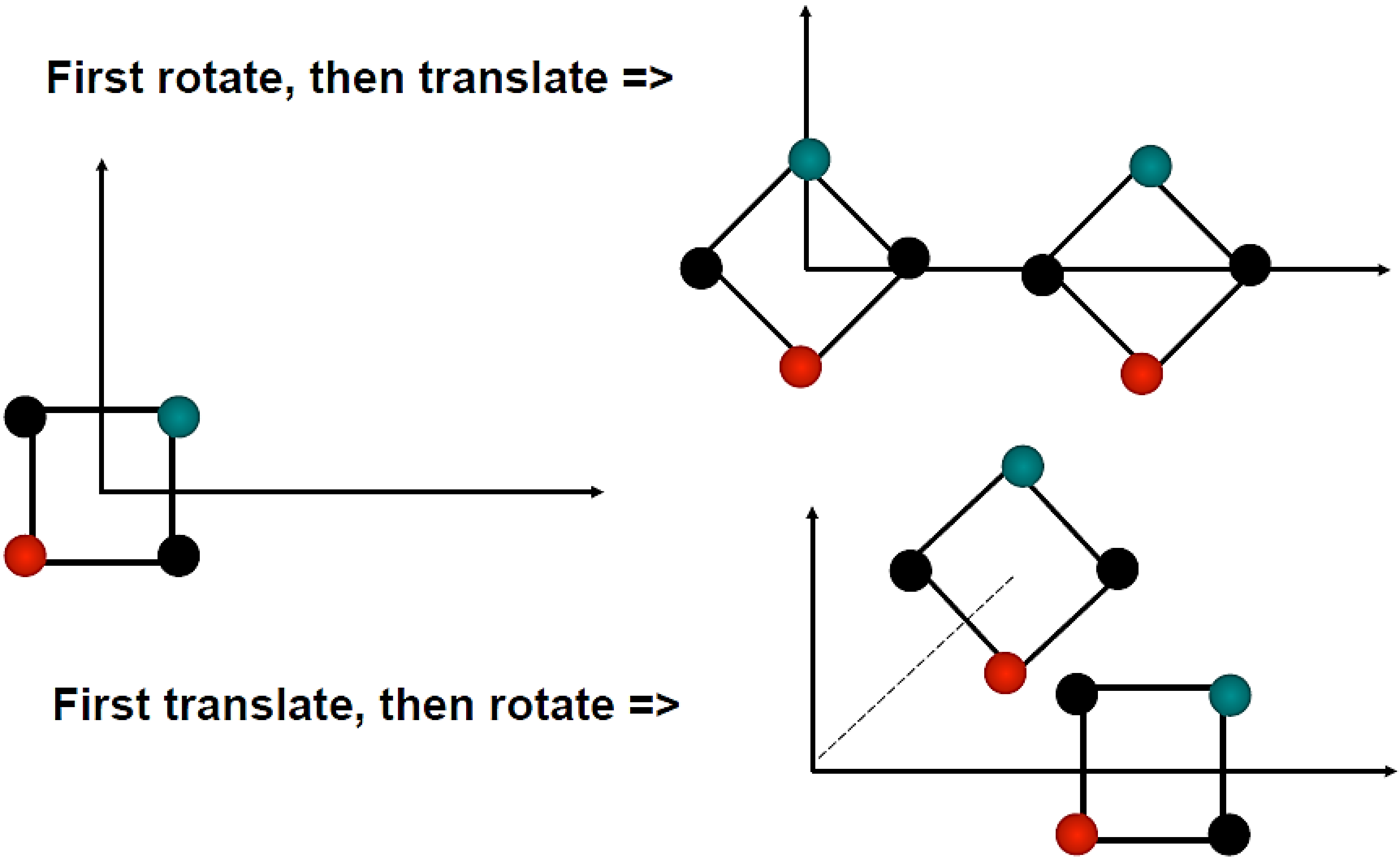
Matrix Multiplication is Not Commutative

First rotate, then translate =>



Matrix Multiplication is Not Commutative

First rotate, then translate =>



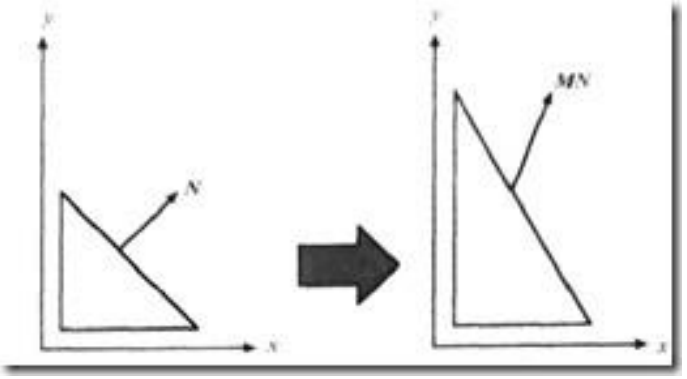
First translate, then rotate =>

Properties of Transformations

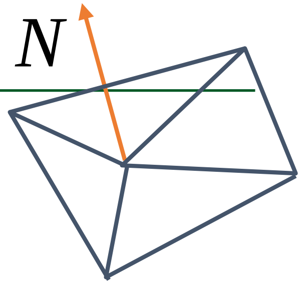
Type / Preserves	Rigid Body: Rotation & translation	Linear: General 3x3 matrix	Affine: Linear + translation	Projective: 4x4 matrix with last row $\neq (0,0,0,1)$
Lengths	Yes	No	No	No
Angles	Yes	No	No	No
Parallelness	Yes	Yes	Yes	No
Straight lines	Yes	Yes	Yes	Yes

Transforming Geometric Objects

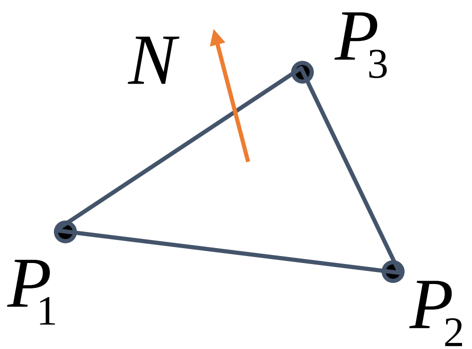
- lines, polygons made up of vertices
  - transform the vertices
  - interpolate between
- does this work for everything? no!
  - normals are trickier



Computing Normals



- **normal**
  - direction specifying orientation of polygon
    - W = 0 means direction with homogeneous coords
    - W = 1 for points of object vertices
  - used for lighting
    - must be normalized to unit length
  - can compute if not supplied with object



$$N = (P_2 - P_1) \times (P_3 - P_1)$$

Assume vertices ordered CCW when viewed from visible side of polygon

Transforming Normals

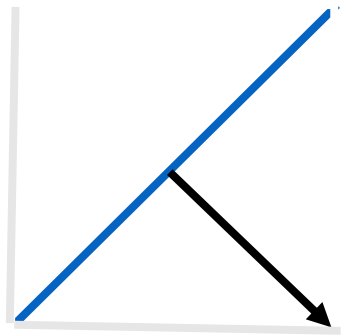
$$\begin{bmatrix} x' \\ y' \\ z' \\ 0 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & T_x \\ m_{21} & m_{22} & m_{23} & T_y \\ m_{31} & m_{32} & m_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

- so if points transformed by matrix M, can we just transform normal vector by M too?
  - translations OK: w = 0 means unaffected
  - rotations OK
  - uniform scaling OK

- these all maintain direction

Transforming Normals

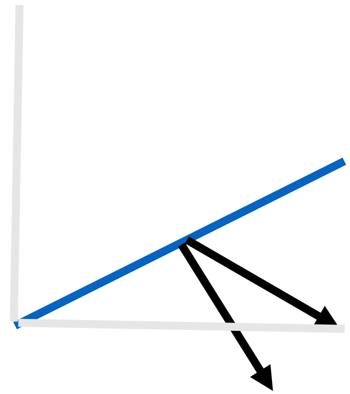
- **nonuniform scaling does not work**
- x- y =0 plane
  - line x = y
  - normal: [1,-1,0]
    - direction of line x = -y
    - (ignore normalization for now)



Transforming Normals

- apply nonuniform scale: stretch along x by 2
  - new plane x = 2y
- normal is direction of line x = -2y or x+2y=0
- transformed normal: [2,-1,0]

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



  - not perpendicular to plane!
  - should be direction of 2x = -y

Planes and Normals

- plane is all points perpendicular to normal
  - $N \bullet P = 0$ (with dot product)
  - $N^T \bullet P = 0$ (matrix multiply requires transpose)

$$N = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}, P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- Implicit form: plane = $ax + by + cz + d$

Finding Correct Normal Transform

- transform a plane

$$P \qquad\qquad P' = MP$$
$$N \longrightarrow N' = QN$$

given M,
what should Q be?

$$N'^T P' = 0$$

stay perpendicular

$$(QN)^T (MP) = 0$$

substitute from above

$$N^T Q^T M P = 0$$

$$(\mathbf{AB})^{\mathbf{T}} = \mathbf{B}^{\mathbf{T}}\mathbf{A}^{\mathbf{T}}$$

$$Q^T M = I$$

$$\mathbf{N}^{\mathbf{T}}\mathbf{P} = 0 \text{ if } \mathbf{Q}^{\mathbf{T}}\mathbf{M} = \mathbf{I}$$

$$\boxed{\mathbf{Q} = \left(\mathbf{M}^{-1}\right)^{\mathbf{T}}}$$

thus the normal to any surface can be transformed by the inverse transpose of the modelling transformation

Outline

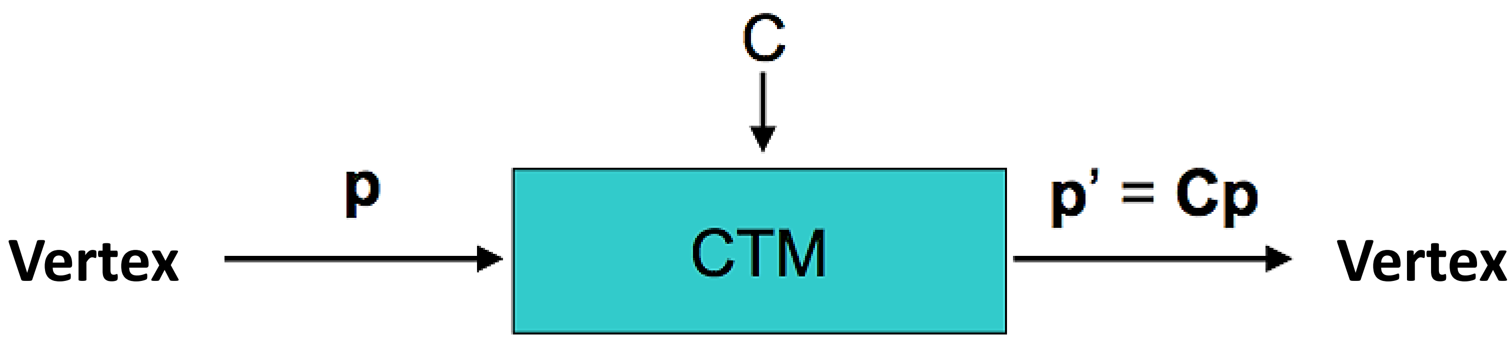
- Geometry
- Representation
- Transformation
- **Transformation in OpenGL**

Programing Transformations

- In OpenGL, the transformation matrices are part of the state, they must be defined *prior to* any vertices to which they are to apply.
- In modeling, we often have objects specified in their own coordinate systems and must use transformations to bring the objects into the scene.
- OpenGL provides *matrix stacks* for each type of supported matrix (model-view, projection, texture) to store matrices.

Current Transformation Matrix (CTM)

- CTM is a 4x4 homogenous coordinate matrix. It is also part of the states. It will be altered by a set of functions and applied to all vertex through pipeline.
- CTM is determined via application.



Change the CTM

- Specify CTM mode : glMatrixMode (mode);

  mode = (GL_MODELVIEW | GL_PROJECTION | GL_TEXTURE)

- Load CTM : glLoadIdentity (void); glLoadMatrix{fd} (*m);

  m = 1D array of 16 elements arranged by the columns

- Multiply CTM : glMultMatrix{fd} (*m);

- Modify CTM : (multiplies CTM with appropriate transformation matrix)

  glTranslate {fd} (x, y, z);

  glScale {fd} (x, y, z);

  glRotate {fd} (angle, x, y, z);

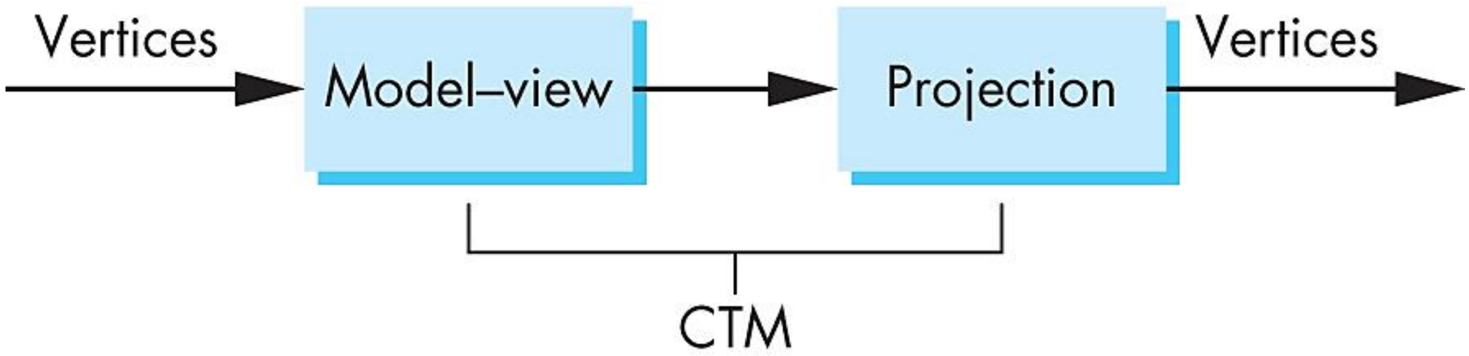
  rotate counterclockwise around ray (0,0,0) to (x, y, z)

Rotation around a fixed point

- Start from Identity: C ← I
- Move the fixed point to origin: C ← CT
- Rotate: C ← CR
- Move the point back: C ← CT^{-1}
- Result: $C = TRT^{-1}$
- Every transformation corresponds to a function of OpenGL.

CTM in OpenGL

- There is a model-view matrix and a projection matrix in the pipeline of OpenGL.
- The combination of these two matrices is CTM in OpenGL.



Using OpenGL Matrices

- Use the following function to specify which matrix you are changing:
  - glMatrixMode(whichMatrix): whichMatrix = GL_PROJECTION | GL_MODELVIEW
- To guarantee a “fresh start”, use glLoadIdentity();
  - Loads the identity matrix into the active matrix

Using OpenGL Matrices

- **To load a user-defined matrix into the current matrix:**
  - glLoadMatrix{fd}(TYPE *m)
- **To multiply the current matrix by a user defined matrix**
  - glMultMatrix{fd}(TYPE *m)
- **SUGGESTION: To avoid row-/column-major confusion, specify matrices as m[16] instead of m[4][4]**

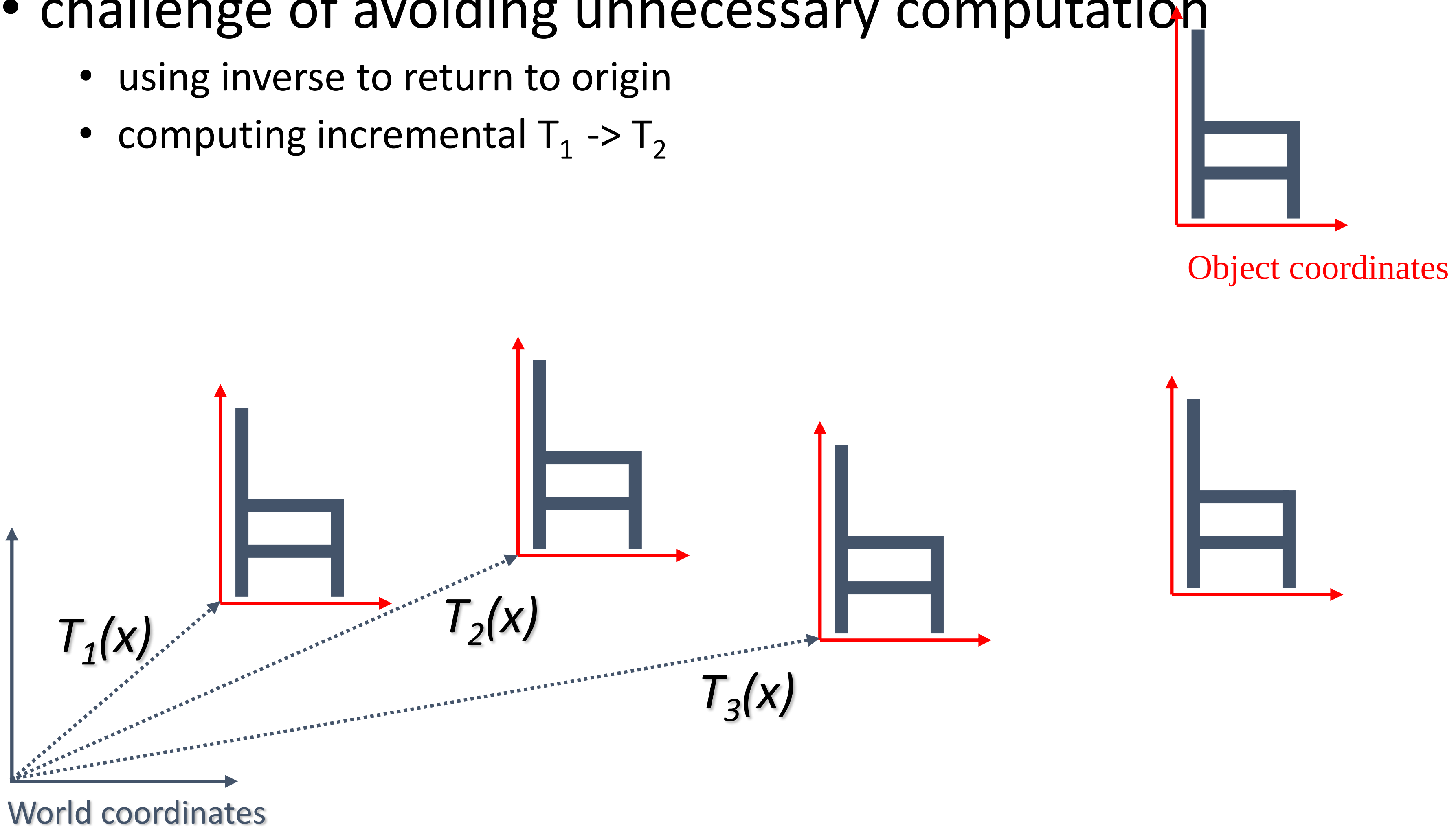
command	result
glLoadMatrixf(A)	stack = [A]
glPushMatrix()	stack = [A, A]
glMultMatrixf(B)	stack = [AB, A]
glPopMatrix()	stack = [A]

Matrix Stacks

- In many cases we need to preserve the transformation matrix in order to use them later
  - Traversing the hierarchical data structure
  - When execute the display list, avoid to change the state
- advantages
  - no need to compute inverse matrices all the time
  - avoids incremental changes to coordinate systems
    - accumulation of numerical errors
- practical issues
  - in graphics hardware, depth of matrix stacks is limited
    - (typically 16 for model/view and about 4 for projective matrix)

Matrix Stacks

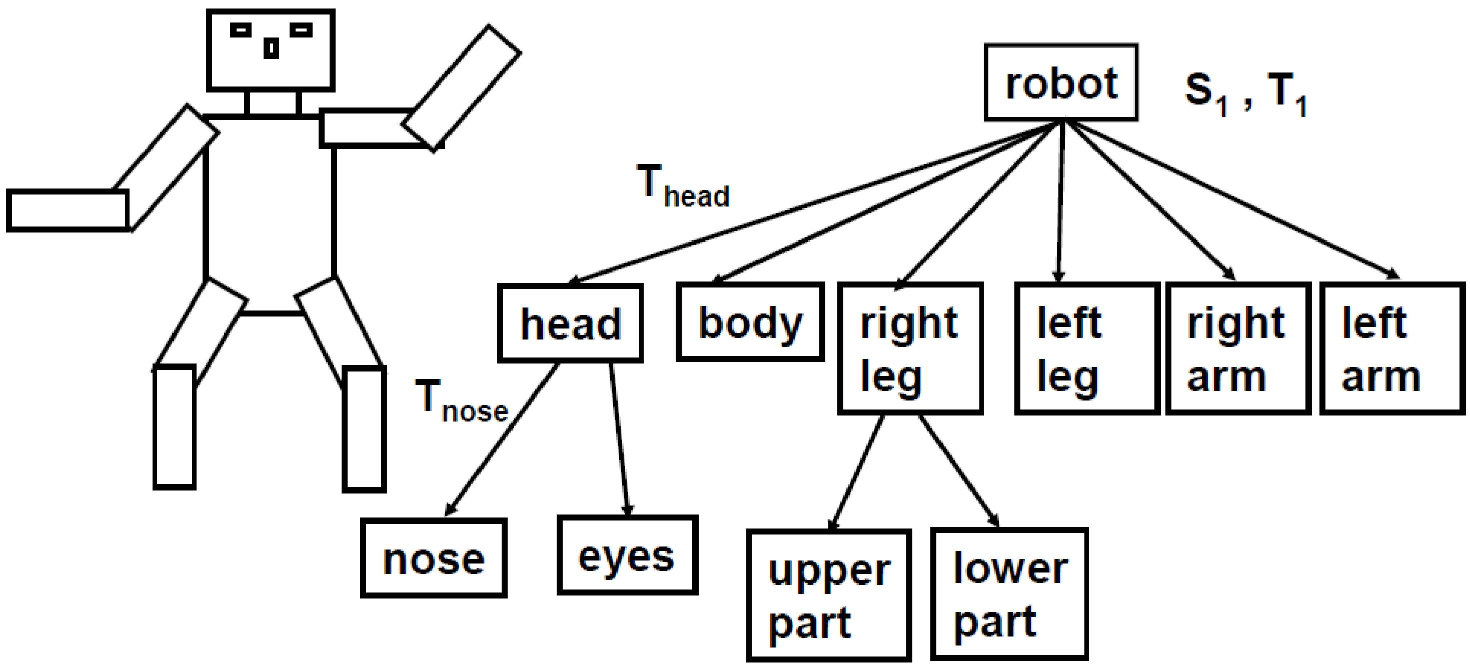
- challenge of avoiding unnecessary computation
  - using inverse to return to origin
  - computing incremental T_1 -> T_2



Matrix Stacks

- Hierarchical representation of an object is a tree.
- The non-leaf nodes are groups of objects.
- The leaf nodes are primitives (e.g. polygons)
- Transformations are assigned to each node, and represent the relative transform of the group or primitive with respect to the parent group
- As the tree is traversed, the transformations are combined into one

Matrix Stacks



Matrix Stacks

To keep track of the current transformation,

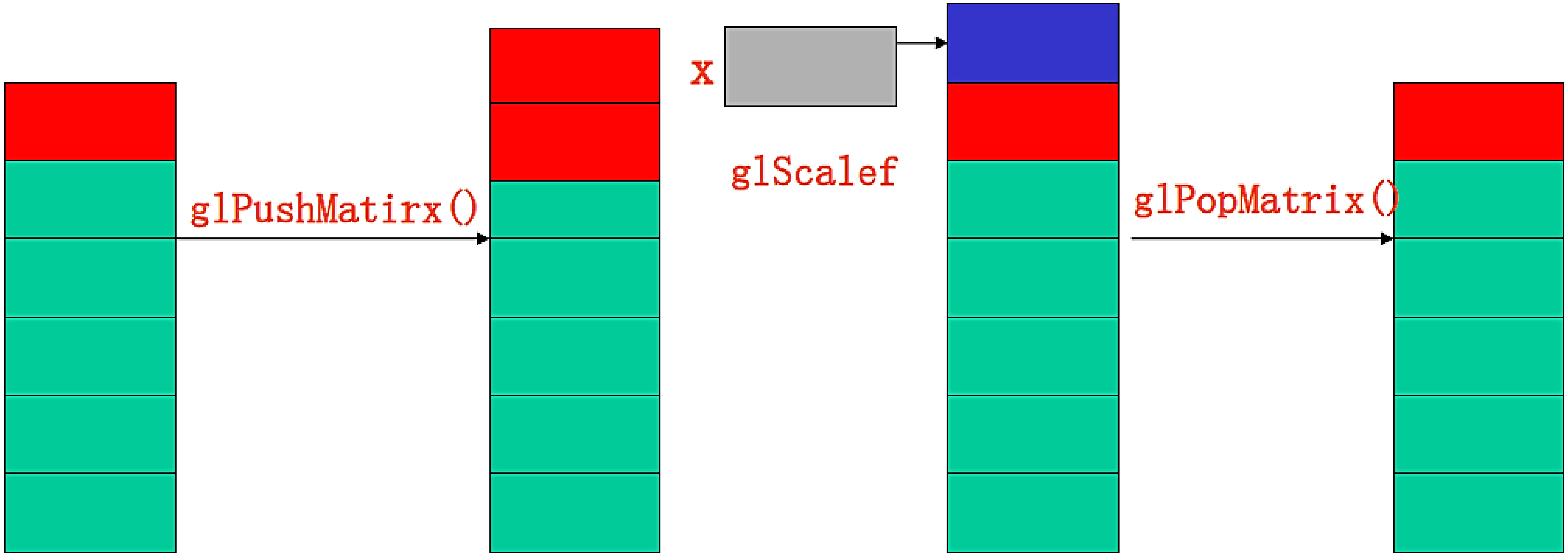
the transformation stack is maintained.

Basic operations on the stack:

- push: create a copy of the matrix on the top and put it on the top
- pop: remove the matrix on the top
- multiply: multiply the top by the given matrix
- load: replace the top matrix with a given matrix

Matrix in OpenGL

- **Mantain matrix stack**
  - glPushMatrix() : used to save current stack
  - glPopMatrix() : used to restore previous stack

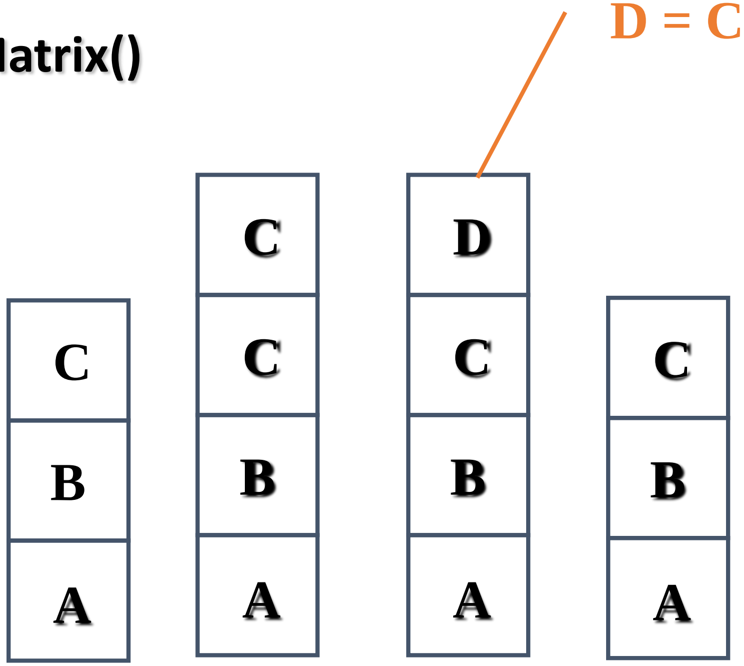


Matrix Stacks

glPushMatrix()

glPopMatrix()

D = C scale(2,2,2) trans(1,0,0)



DrawSquare()

glPushMatrix()

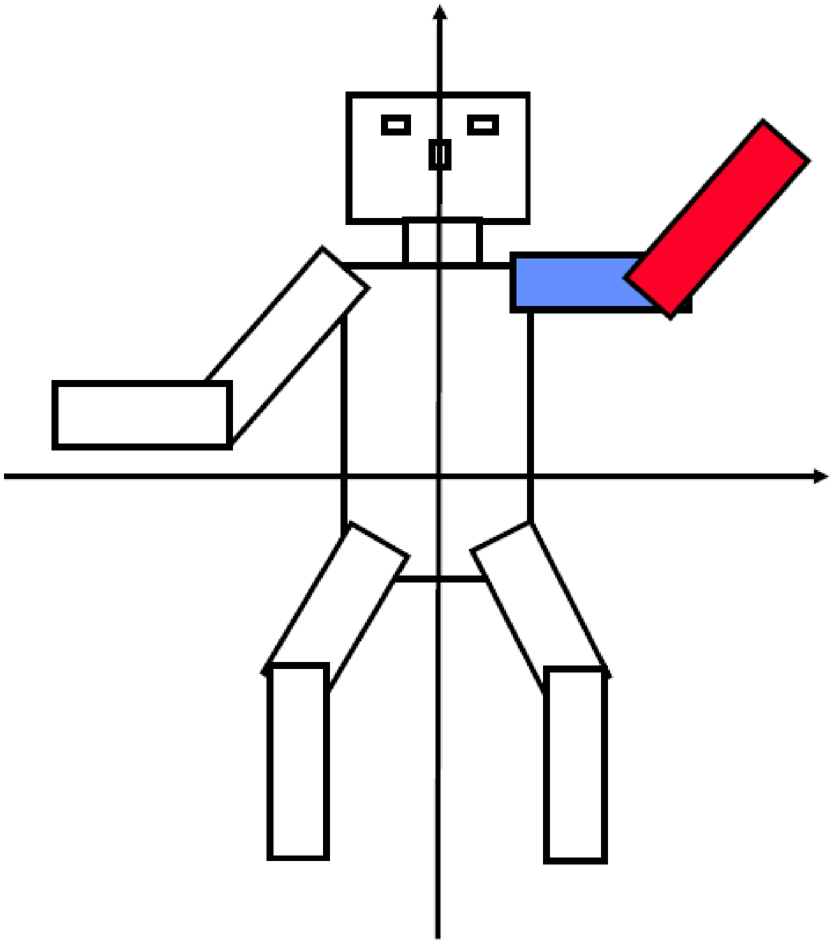
glScale3f(2,2,2)

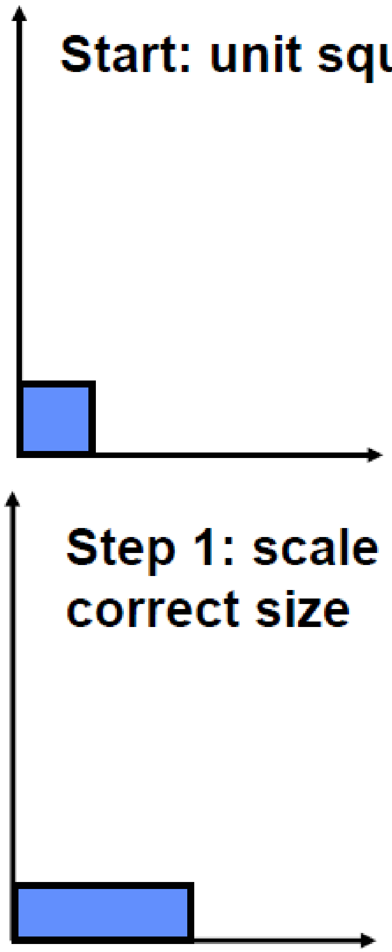
glTranslate3f(1,0,0)

DrawSquare()

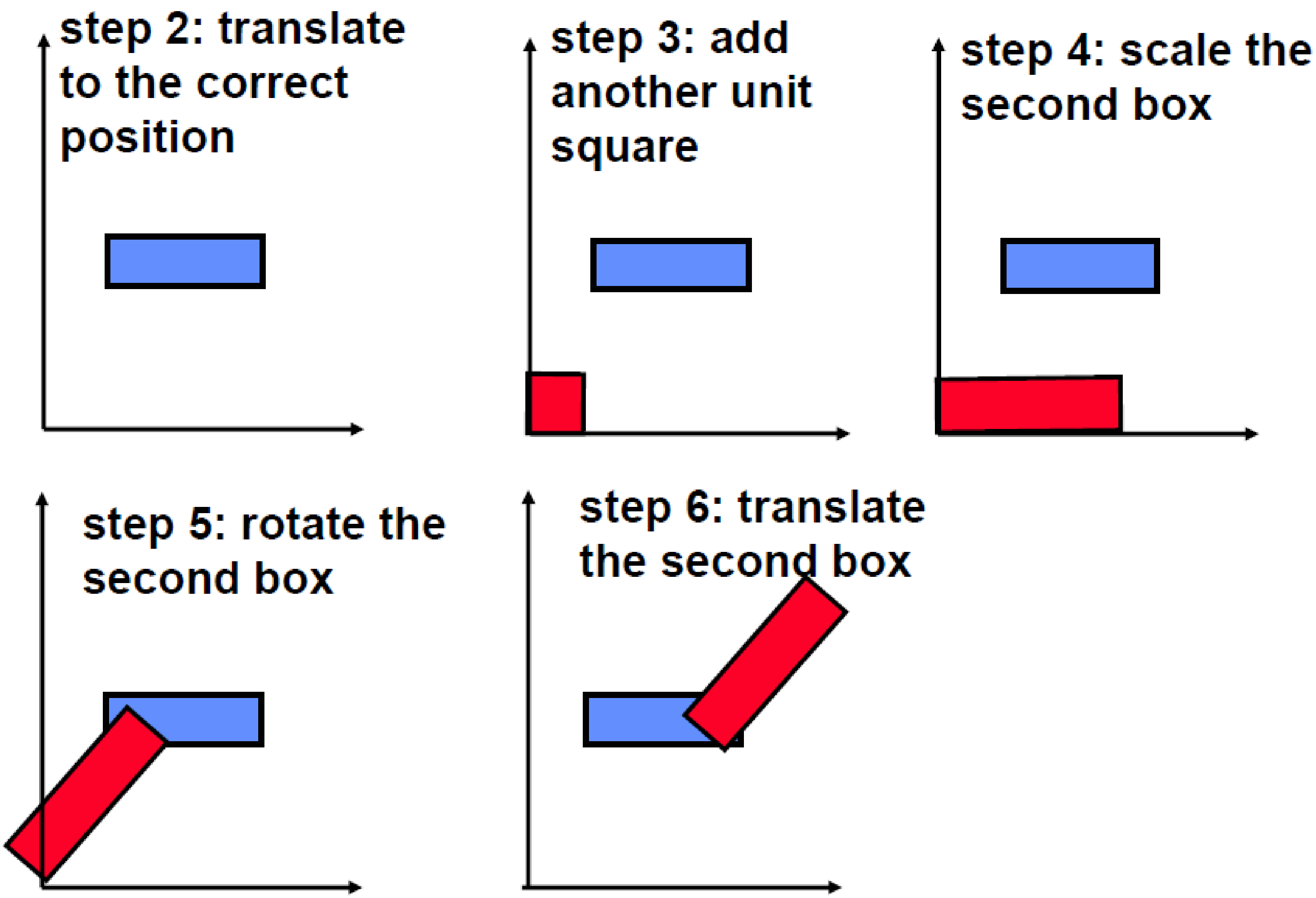
glPopMatrix()

Building the arm





Building the arm



Hierarchical Transformations

- Positioning each part of a complex object separately is difficult
- If we want to move whole complex objects consisting of many parts or complex parts of an object (for example, the arm of a robot) then we would have to modify transformations for each part
- solution: build objects hierarchically

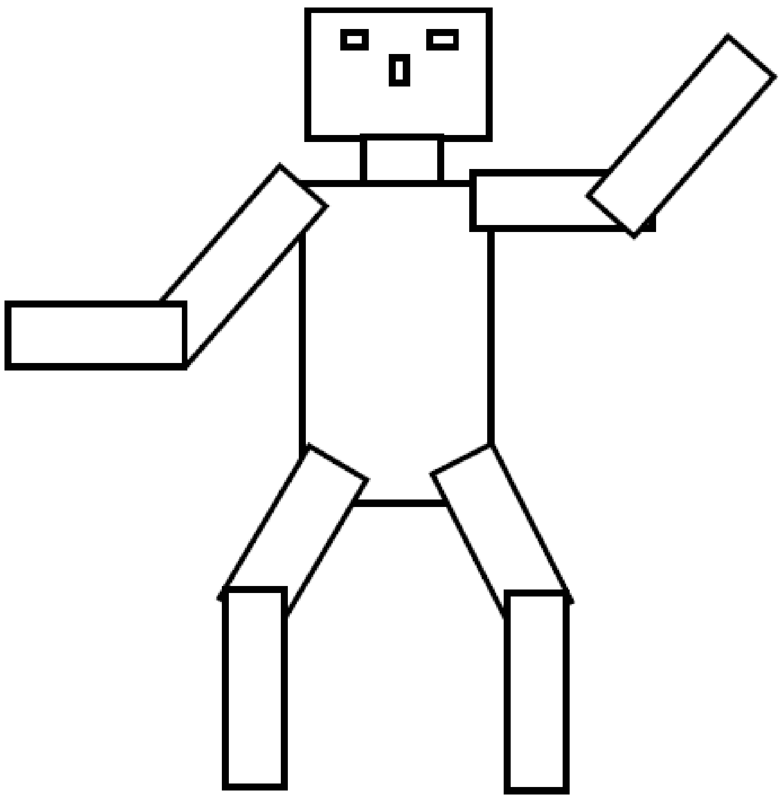
Complex models

-can be built in a simple, modular fashion

-can be stored efficiently

-can be updated simply

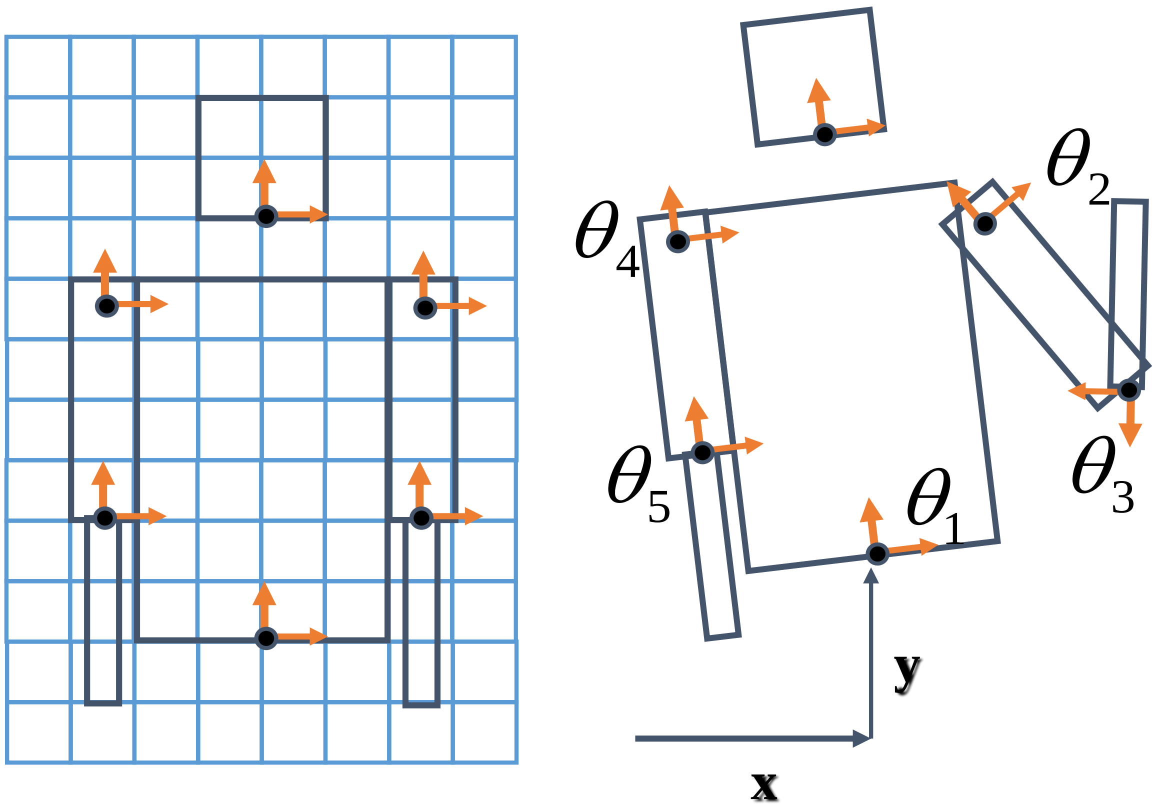
Hierarchical Transformations



Idea: group parts hierarchically, associate transforms with each group.

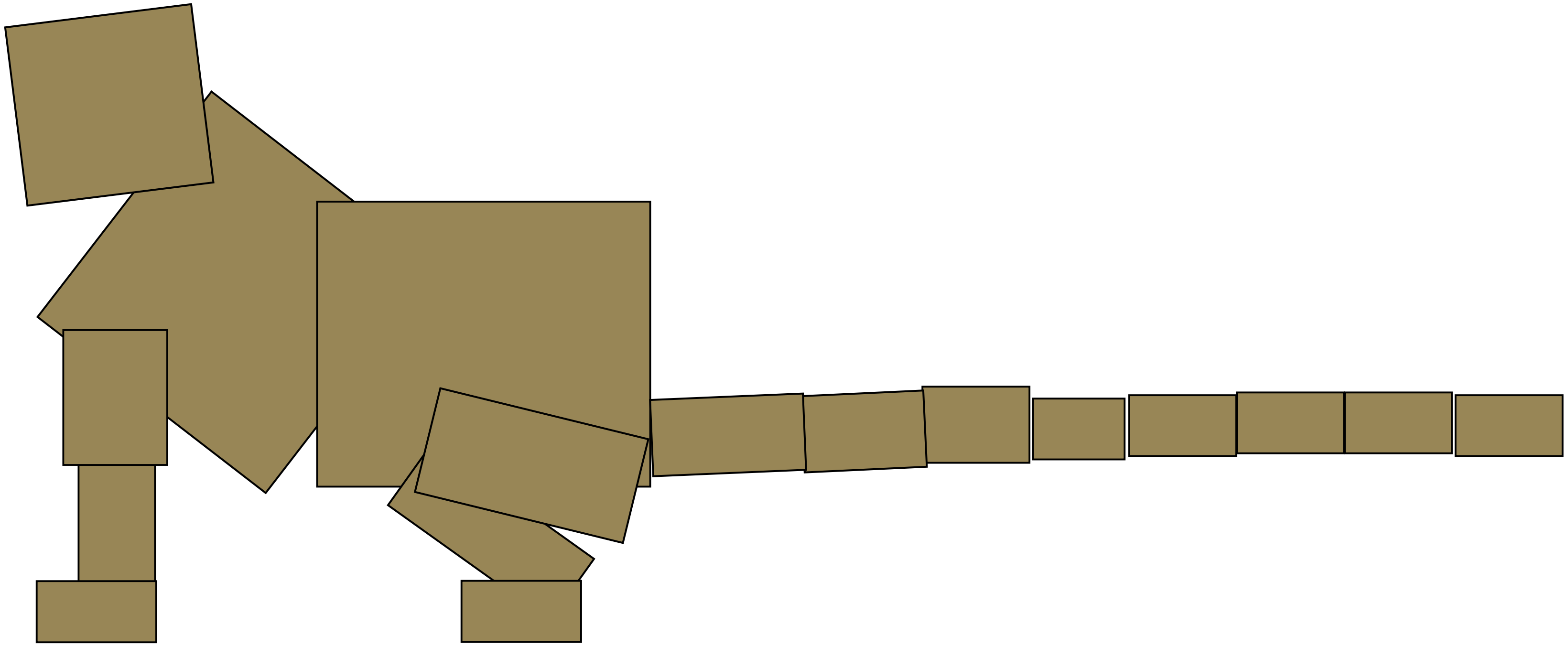
whole robot = head + body + legs + arms
leg = upper part + lower part
head = neck + eyes + ...

Transformation Hierarchy Example

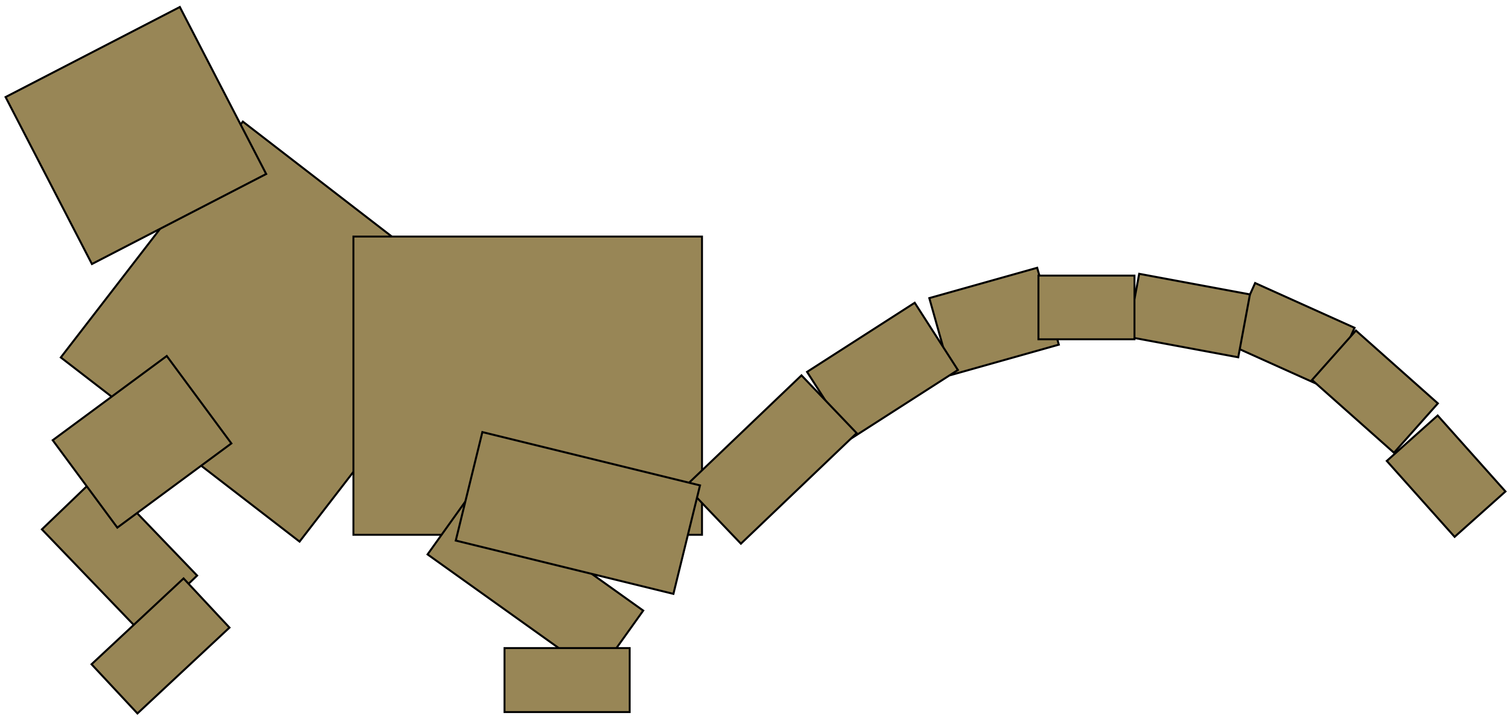



```
glTranslate3f(x,y,0);
glRotatef(θ1,0,0,1);
DrawBody();
glPushMatrix();
  glTranslate3f(0,7,0);
  DrawHead();
glPopMatrix();
glPushMatrix();
  glTranslate(2.5,5.5,0);
  glRotatef(θ2,0,0,1);
  DrawUArm();
  glTranslate(0,-3.5,0);
  glRotatef(θ3,0,0,1);
  DrawLArm();
glPopMatrix();
... (draw other arm)
```

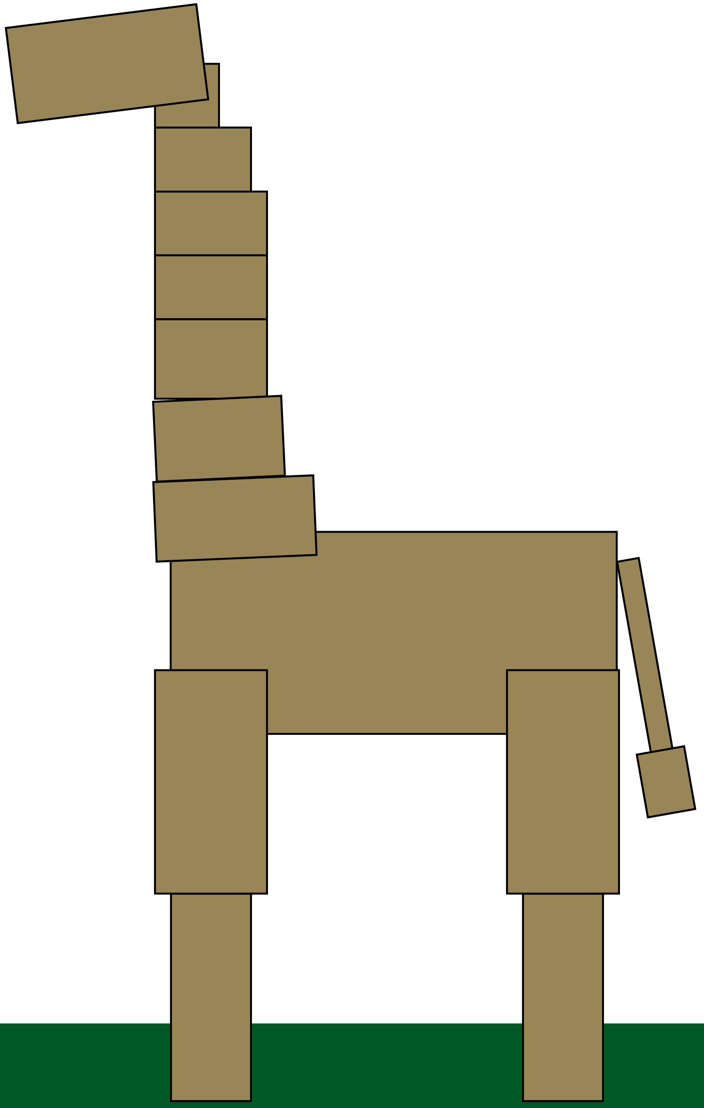
Monkeys!



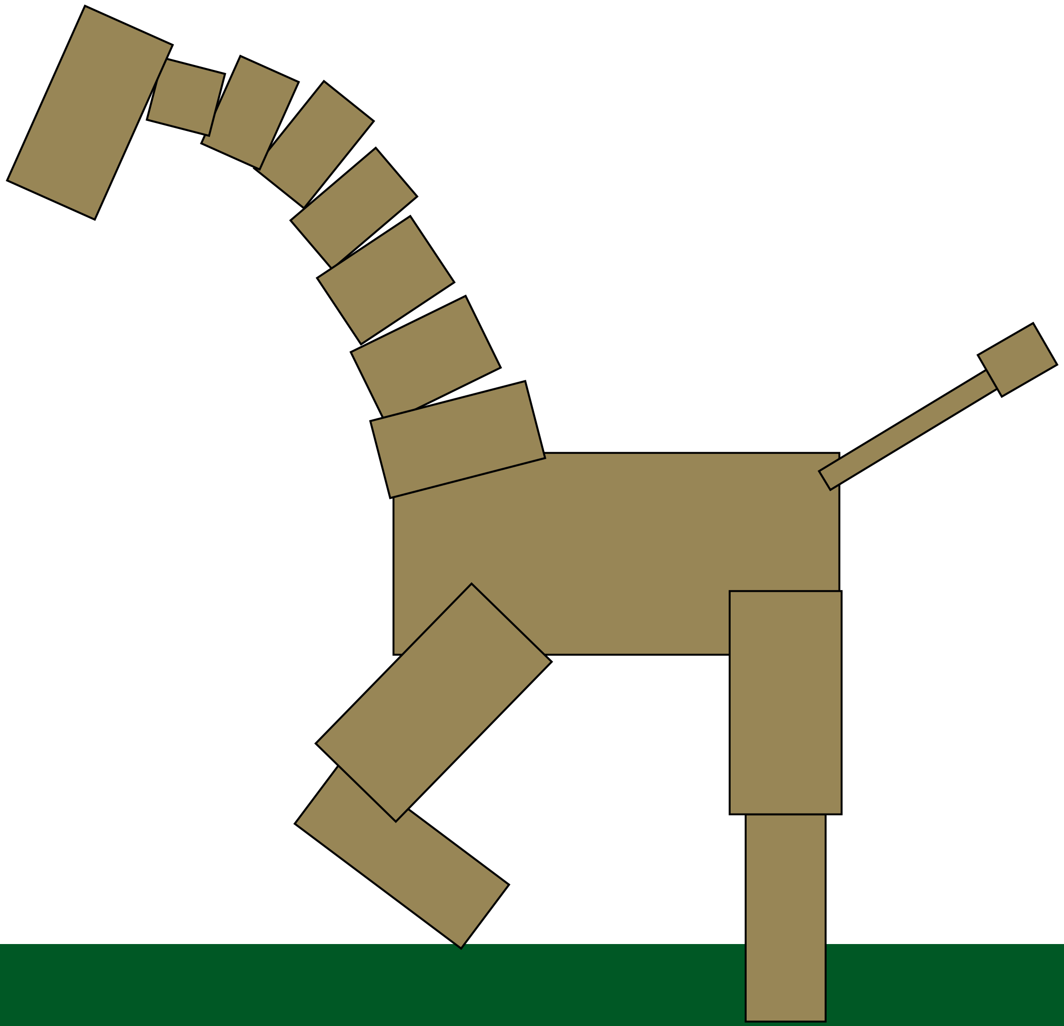
Monkeys!



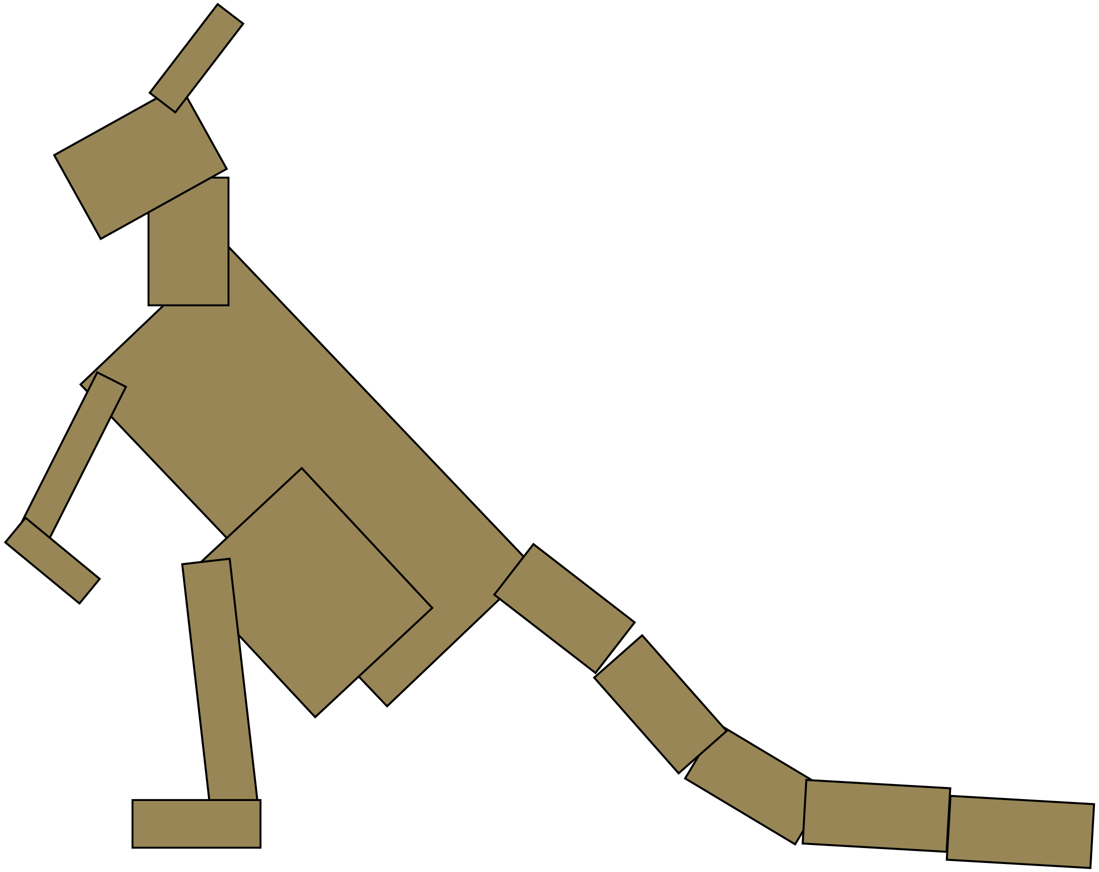
Giraffes!



Giraffes!



Kangaroos!



Quaternions(四元数)

- Quaternions were invented by Hamilton, W. R. , a Ireland mathematicians
- Quaternions are an extension of complex numbers that provide an alternative method for describing and manipulating rotations.
- Less intuitive than our original approach, quaternions provide advantages for animation and hardware implementation of rotation.

Quaternions

- In three dimensions, the problem is more difficult because to specify a rotation about the origin we need to specify both a direction (a vector) and the amount of rotation (a scalar)
- One solution is to use a representation that consists of both a vector and a scalar. Usually, this representation is written as the quaternion

$$a = (q_0, q_1, q_2, q_3) = (q_0, \mathbf{q}),$$

where q = (q1, q2, q3). The operations among quaternions are based on the use of three “complex” numbers i, j, and k with the properties

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1.$$

These numbers are analogous to the unit vectors in three dimensions, and we can write q as：

$$\mathbf{q} = q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}.$$

Operational Rule of Quaternions

- the quaternion a、 b are given by ：

$$a = (q_0, q_1, q_2, q_3) = (q_0, \mathbf{q}), \qquad b = (p_0, \mathbf{p}),$$

- quaternion addition and multiplication

$$a + b = (p_0 + q_0, \mathbf{p} + \mathbf{q}),$$

$$ab = (p_0 q_0 - \mathbf{q} \cdot \mathbf{p}, q_0 \mathbf{p} + p_0 \mathbf{q} + \mathbf{q} \times \mathbf{p}).$$

- a magnitude for quaternions in the normal manner as

$$|a|^2 = q_0^2 + q_1^2 + q_2^2 + q_3^2 = q_0^2 + \mathbf{q} \cdot \mathbf{q}.$$

- the inverse of a quaternion

$$a^{-1} = \frac{1}{|a|^2}(q_0, -\mathbf{q}).$$

Quaternions and Rotation

- Suppose that we use the vector part of a quaternion to represent a point in space

$$p = (0, \mathrm{p}).$$

Thus, the components of p = (x, y, z) give the location of the point.

- Consider the quaternion：

$$r = \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\mathrm{v}\right),$$

where v has unit length. We can then show that the quaternion r is a unit quaternion (|r| = 1), and therefore

$$r^{-1} = \left(\cos\frac{\theta}{2}, -\sin\frac{\theta}{2}\mathrm{v}\right).$$

Quaternions and Rotation

- If we consider the quaternion product of the quaternion p that represents a point with r, we obtain the new quaternion

$$p' = rpr^{-1}.$$

This quaternion has the form $(0, \mathbf{p}')$, where

$$\mathbf{p}' = \cos^2\frac{\theta}{2}\mathbf{p} + \sin^2\frac{\theta}{2}(\mathbf{p}\cdot\mathbf{v})\mathbf{v} + 2\sin\frac{\theta}{2}\cos\frac{\theta}{2}(\mathbf{v}\times\mathbf{p}) - \sin\frac{\theta}{2}(\mathbf{v}\times\mathbf{p})\times\mathbf{v}$$

and thus p’ is the representation of a point. What is less obvious is that p’ is the result of rotating the point p by θ degrees about the vector v.

Quaternions and Rotation

- Because we get the same result, the quaternion product formed from r and p is an alternate to transformation matrices as a representation of rotation with a fixed point of the origin about an arbitrary axis.
- If we count operations, quaternions are faster and have been built into both hardware and software implementations.
- In addition to the efficiency of using quaternions instead of rotation matrices, quaternions can be interpolated to obtain smooth sequences of rotations for animation.

Interface

- A major problem of interactive computer graphics is how to use the equipment of the two-dimensional (such as a mouse) to control three-dimensional objects.
- Alternative ways
  - Virtual track ball(虚拟跟踪球)
  - three–dimensional input device : spaceball（空间球）
  - Using Areas of the Screen: According to the different state of the mouse button, the use of the distance to the center of control angle, position, and zooming