



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



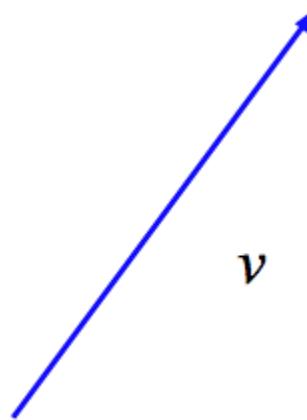
Scalar

- 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 representation

We can represent a **point**, $\mathbf{p} = (x, y)$, in the plane or $\mathbf{p} = (x, y, z)$ in 3D space

- ◆ as column vectors

$$\begin{bmatrix} x \\ y \end{bmatrix} \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- ◆ as row vectors

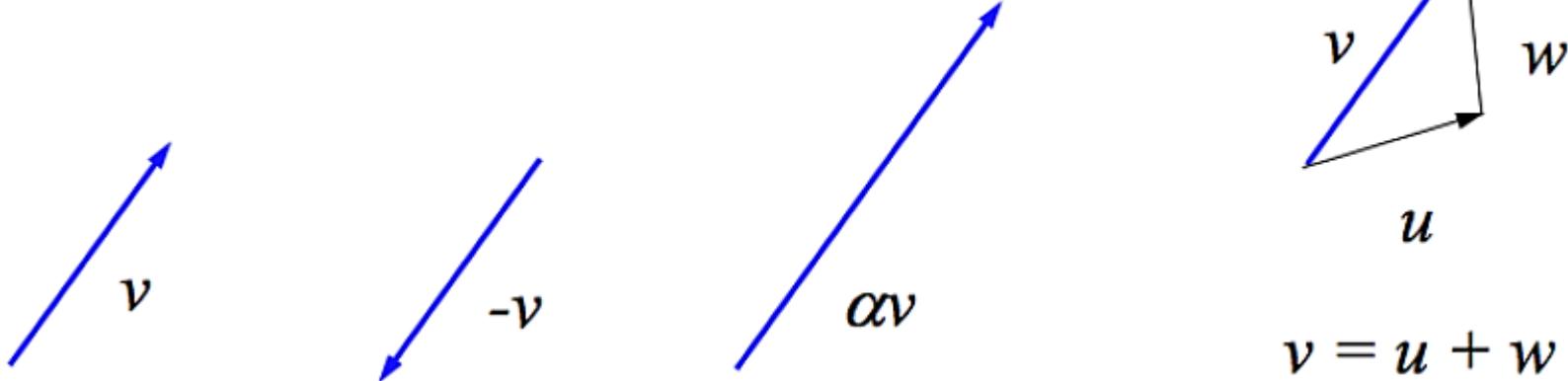
$$[x \ y]$$

$$[x \ y \ z]$$

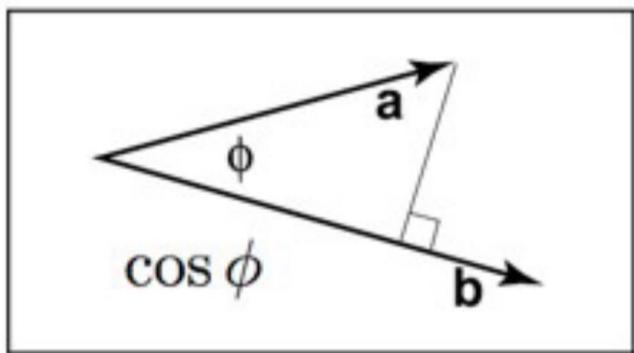


Vector operations

- 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)



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

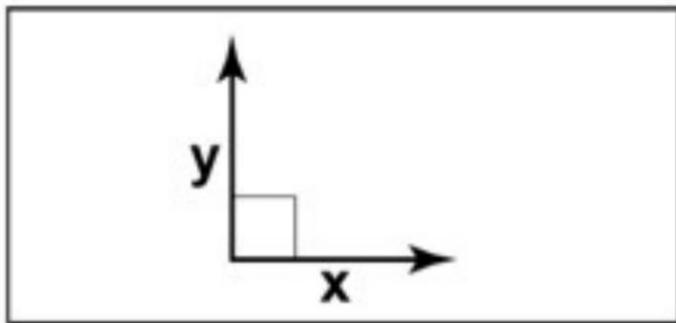
The projection of a onto b

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

Computer Graphics 2014, ZJU



Orthonormal Vector



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

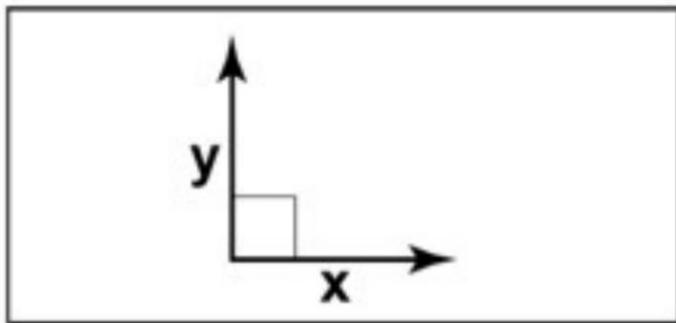
Unit length $\mathbf{x} \cdot \mathbf{x} = 1$

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

Computer Graphics 2014, ZJU



Orthonormal Vector



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

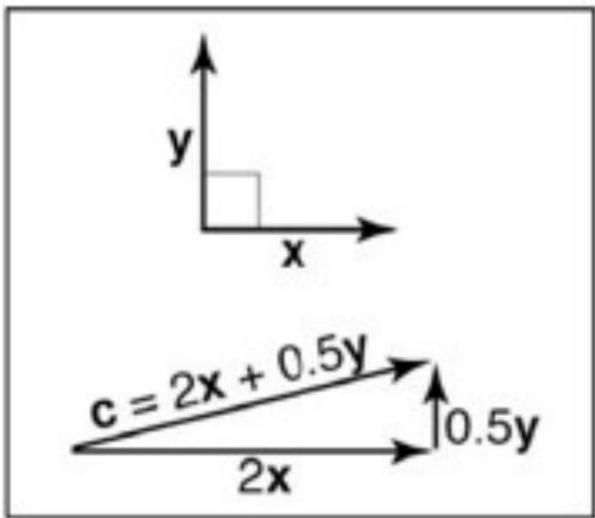
Unit length $\mathbf{x} \cdot \mathbf{x} = 1$

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

Computer Graphics 2014, ZJU



Coordinates and Vectors



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

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

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

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

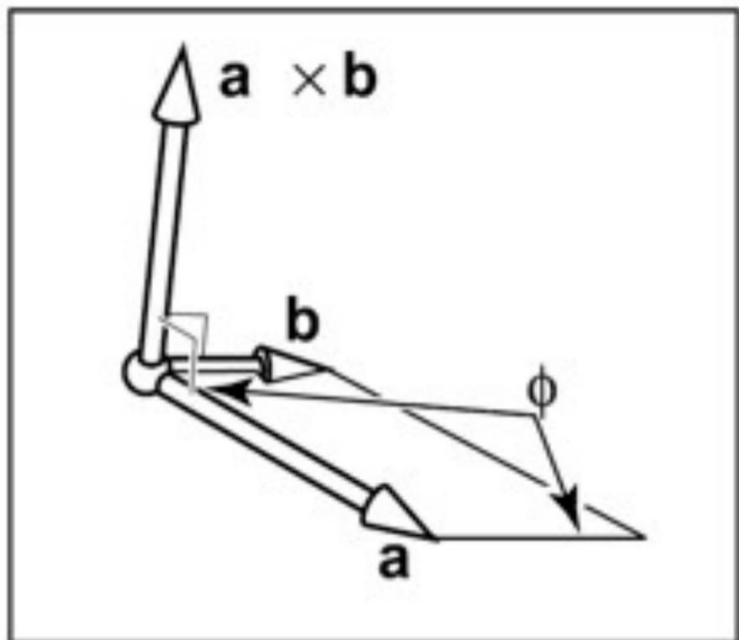


Dot product: some applications in CG

- 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



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

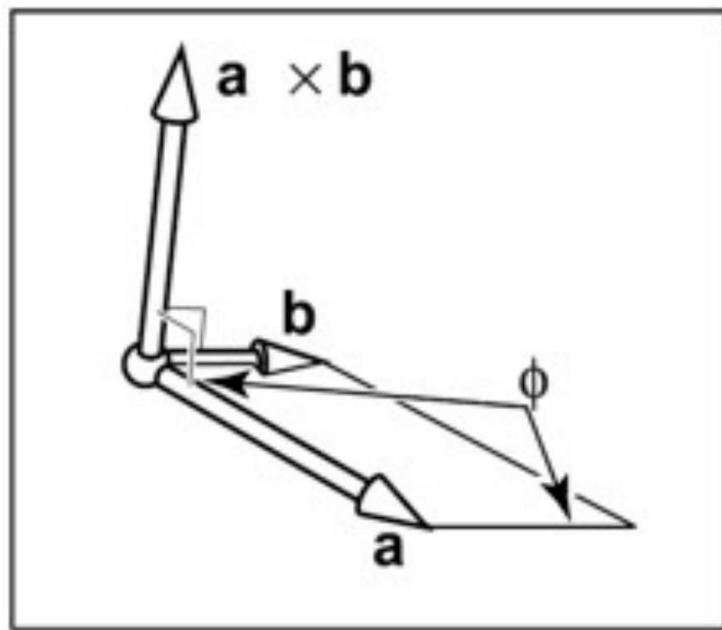
$$\begin{aligned}x_c &= y_a z_b - z_a y_b \\y_c &= z_a x_b - x_a z_b \\z_c &= x_a y_b - y_a x_b\end{aligned}$$

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

Computer Graphics 2014, ZJU



Cross Product



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

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

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

$$\mathbf{x} \times \mathbf{x} = \mathbf{0}$$

$$\mathbf{y} \times \mathbf{y} = \mathbf{0}$$

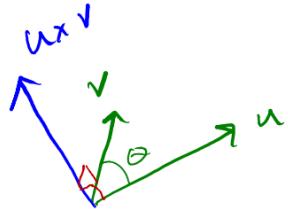
$$\mathbf{z} \times \mathbf{z} = \mathbf{0}$$

Right-Hand Rule

Computer Graphics 2014, ZJU



Vector cross products



$$u, v \perp u \times v$$

N

$$N = (B-A) \times (C-A)$$

$$\hat{N} = \frac{N}{\|N\|}$$

$$u \times v = \det \begin{bmatrix} \hat{x} & \hat{y} & \hat{z} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{bmatrix} = (u_y v_z - u_z v_y) \hat{x} + (u_z v_x - u_x v_z) \hat{y} + (u_x v_y - u_y v_x) \hat{z}$$

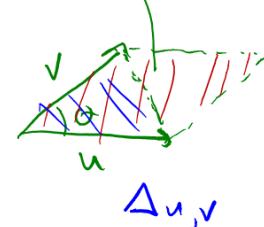
$$(u \times v) \cdot u = 0$$

$$(u \times v) \cdot v = 0$$

$$u \times v = -v \times u$$

$$u \times v = \|u\| \|v\| \sin \theta \hat{N}$$

$$\|u \times v\| = \|u\| \|v\| |\sin \theta|$$



$$u \times v = 0 \Rightarrow u \parallel v$$

$$u \neq 0, v \neq 0$$

$$u = \alpha v$$

$$\|u \times v\| = \text{Area}(\triangle_{u,v})$$

$$\frac{1}{2} \|u \times v\| = \text{Area}(\Delta_{u,v})$$



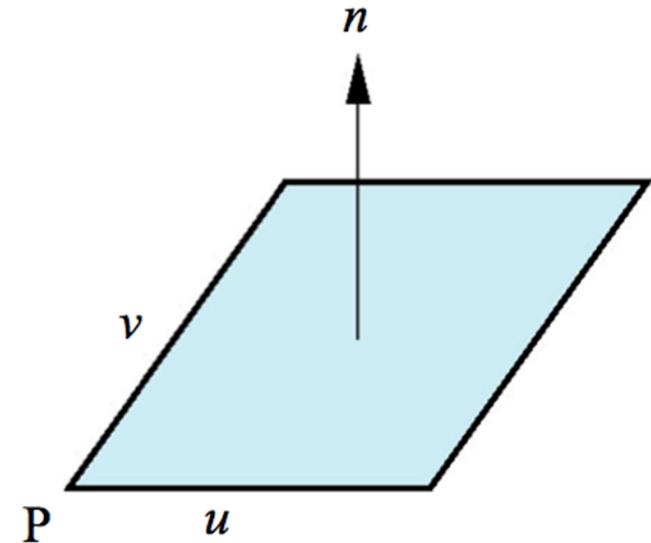
Normals

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

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

- we can get n by the following equation

$$n = u \times v$$



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 v_1, v_2, \dots, v_n and n scalar a_1, a_2, \dots, a_n , then

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

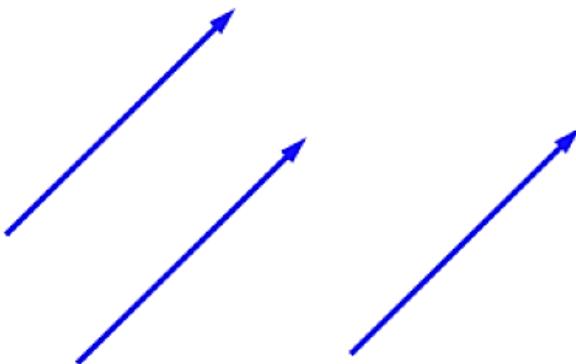
is also a vector, called the linear combination of this set of vectors.

- Irrelevant with coordinate



Vectors have no positions

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

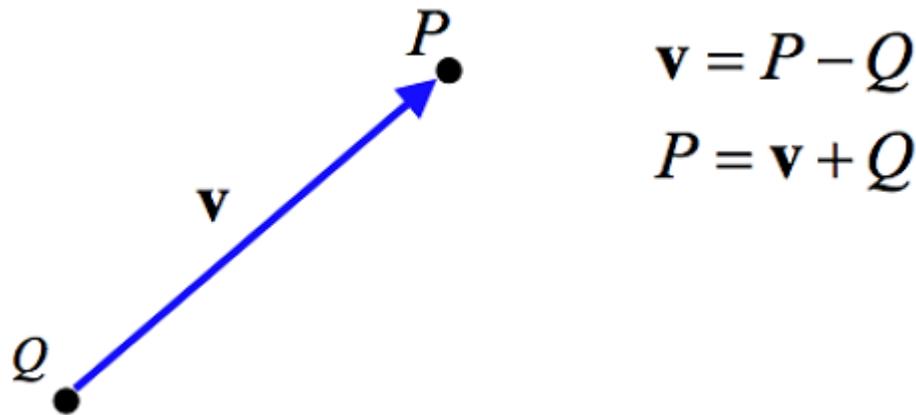


- **It is not enough for geometry to only have vector space**
 - We still need points.



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



Affine space

- Space constructed by **points** and vectors
- Operational:
 - Vector + Vector -> Vector
 - Scalar x Vector -> Vector
 - Point – Point -> Vector
 - Point + Vector -> Point
 - Point + Point -> Chaos



Linear combination of points

- Fixed coordinate system, given two points, what is $P_1 + P_2$?
 - P_1 is origin, $P_1 + P_2 = P_2$
 - P_1 and P_2 are symmetric on origin, $P_1 + P_2 = \text{origin}$
 - The Positions of P_1, P_2 are relevant with coordinate

$\alpha + \beta = 1 \Rightarrow \text{point}$
 $\alpha + \beta = 0 \Rightarrow \text{vector}$
else $\Rightarrow \text{chaos}$

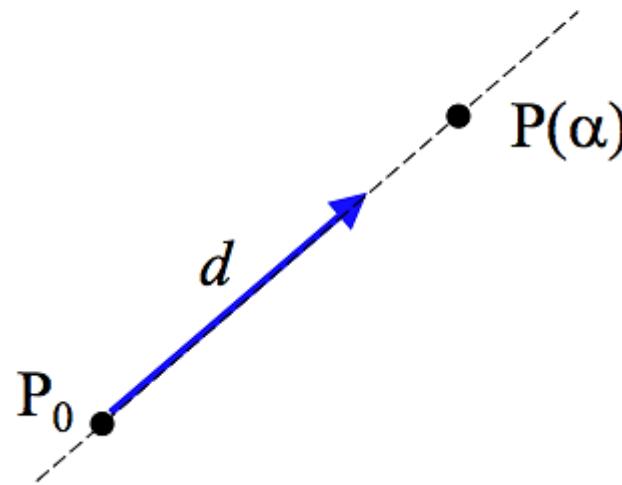
- Combination coefficients **have limitations**
 - When $\alpha_1 + \alpha_2 + \dots + \alpha_n = 1$, linear combination of points is a point
 - $\frac{1}{2} P_1 + \frac{1}{2} P_2 = P_1 + \frac{1}{2}(P_2 - P_1) = \text{point} + \text{vector} = \text{point}$



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: $x(\alpha) = x_0 + (1 - \alpha)x_1$
 $y(\alpha) = y_0 + (1 - \alpha)y_1$

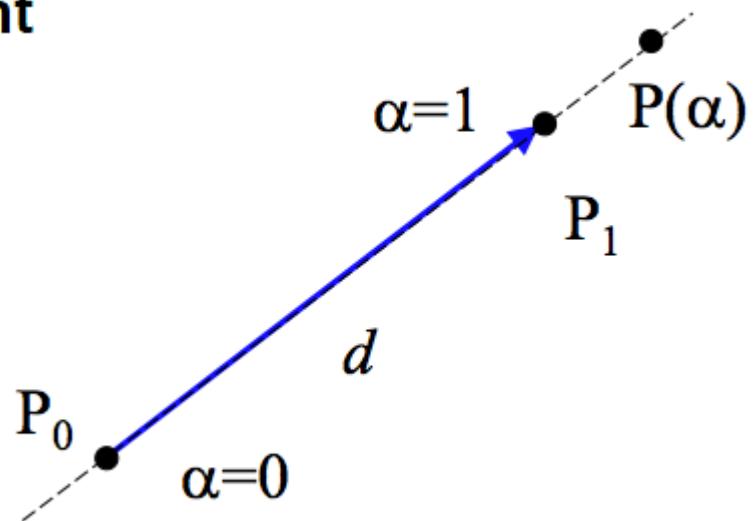


Rays and segments

- 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 $0 \leq \alpha \leq 1$, we get a segment



Linear interpolation

- Given two points A and B, their affine combination

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

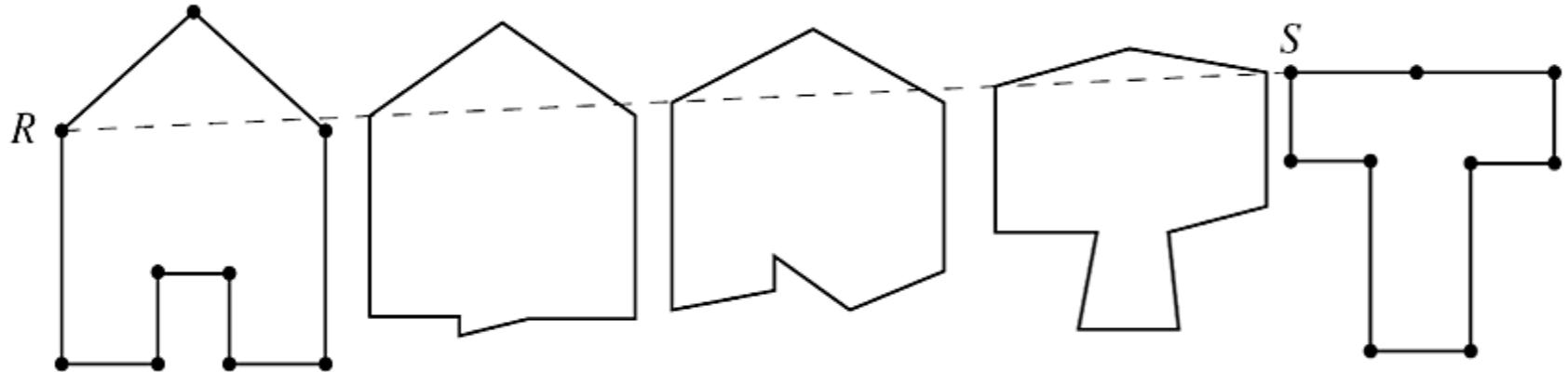
defines a line pass these two points.

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

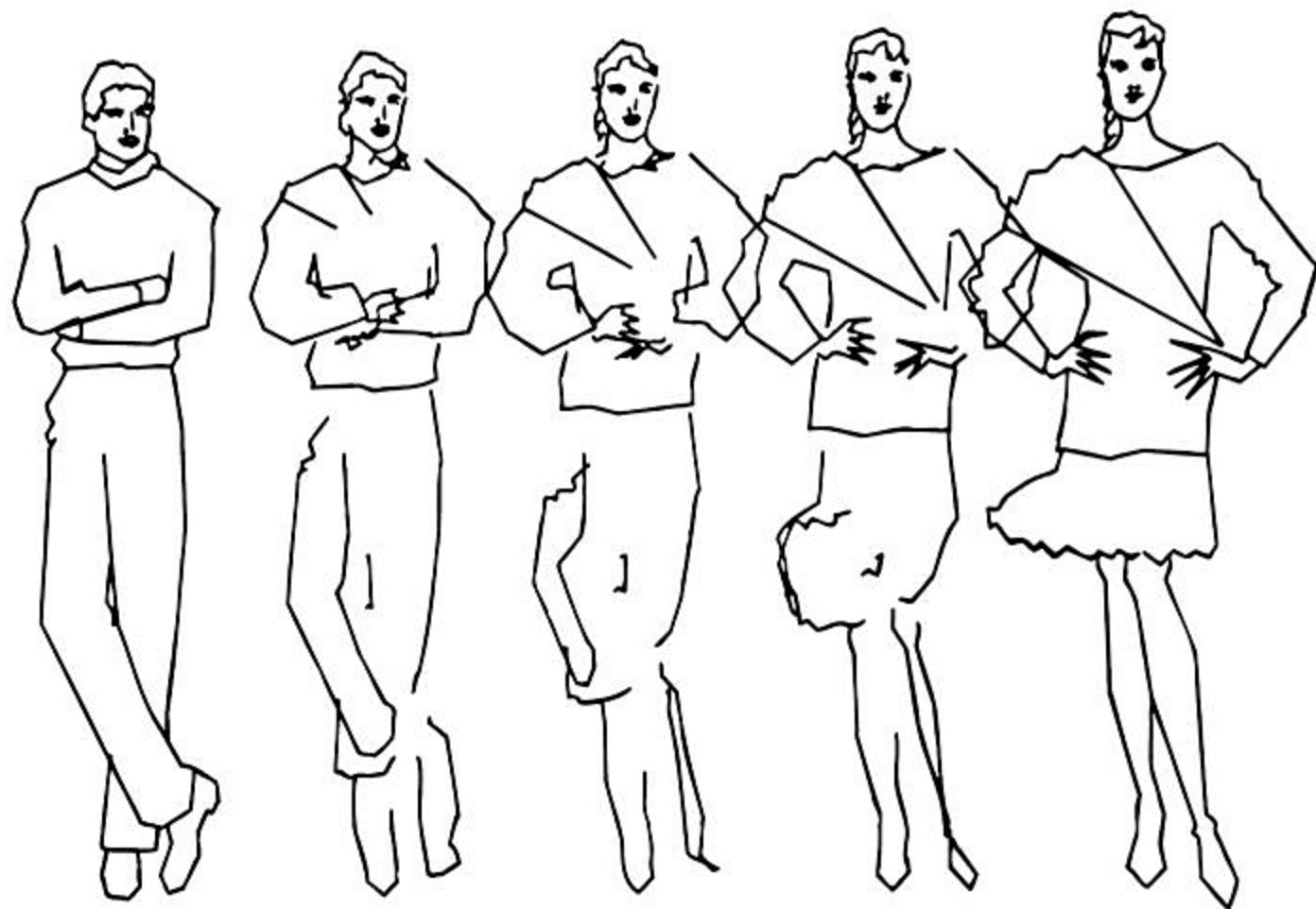


Polygon deformation

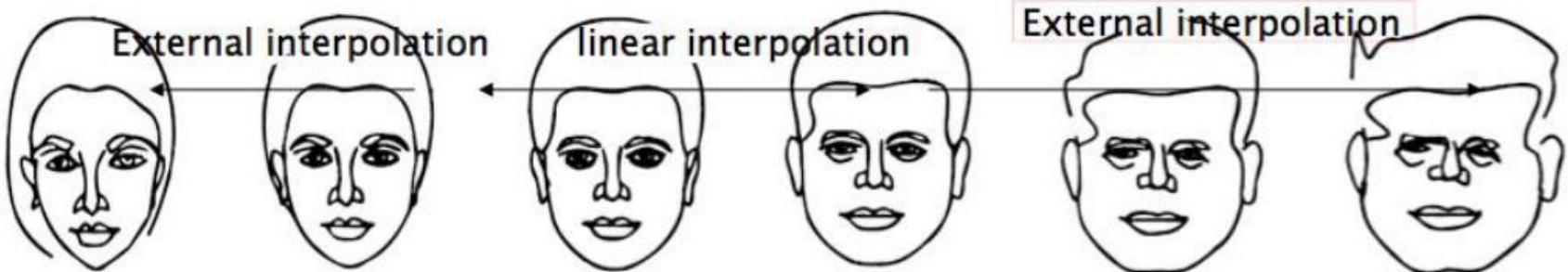
- Given two lines with the same number of vertices, we can get a smooth transition from the first to the second polyline



Man to Woman



Celebrity Face



Elizabeth Taylor



John F. Kennedy

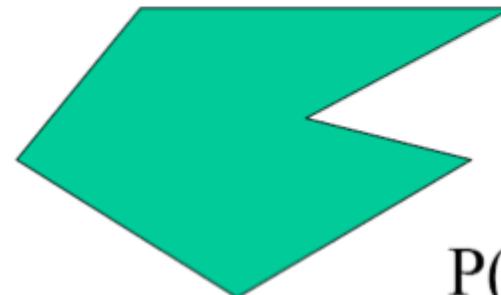


Curve and Surface

- 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



$P(\alpha)$

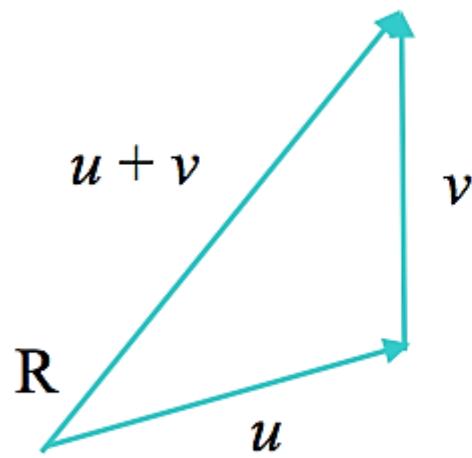


$P(\alpha, \beta)$

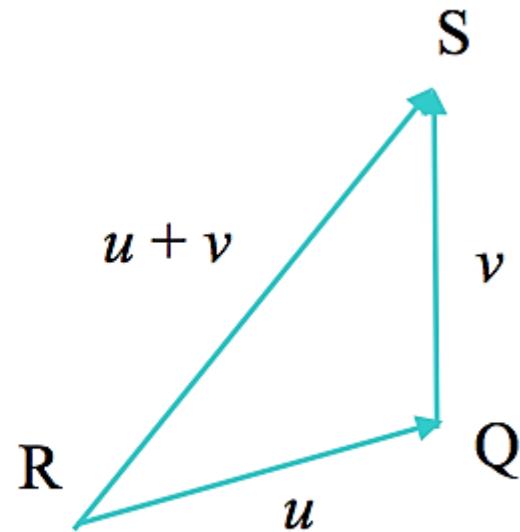


Plane

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



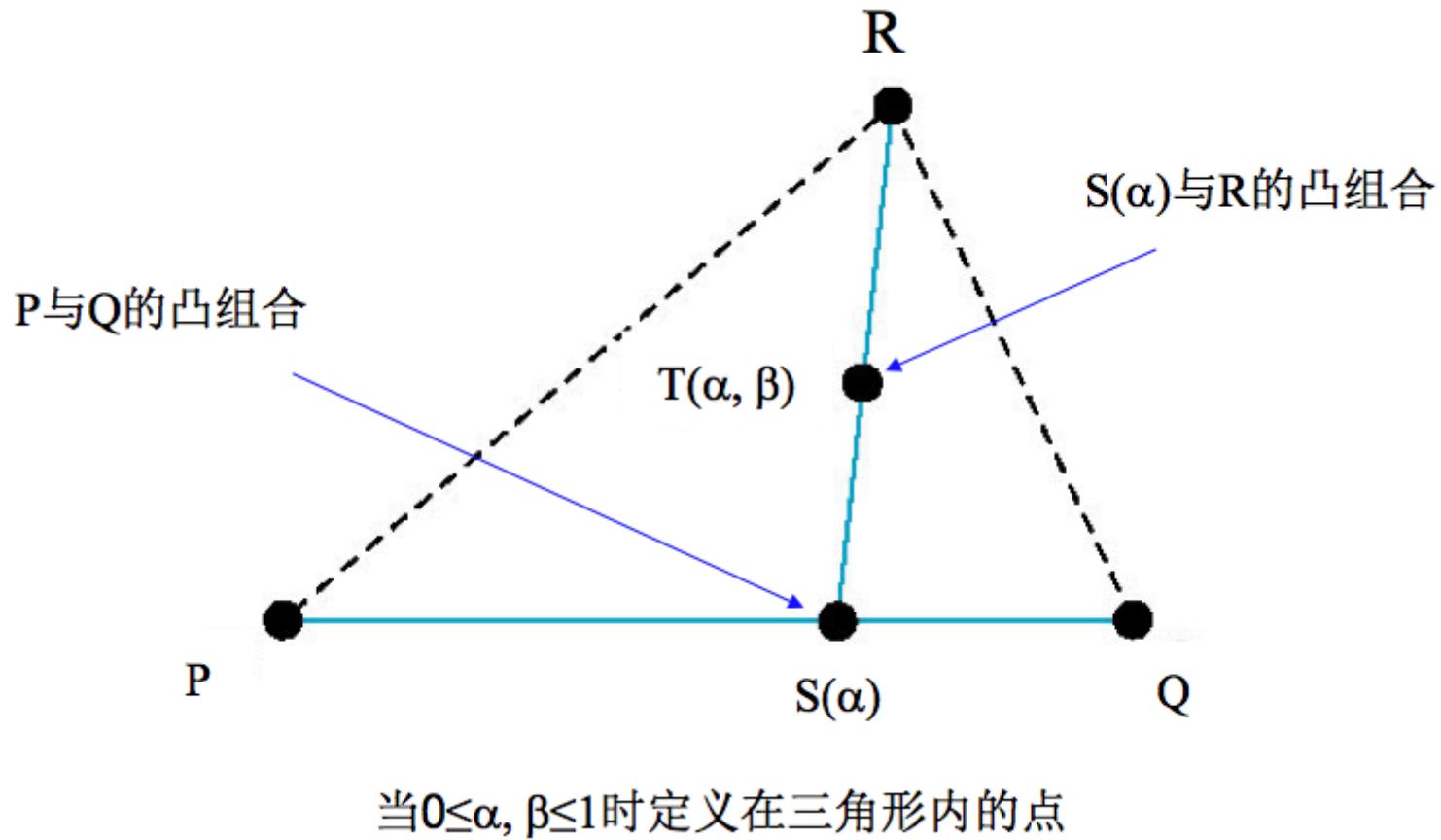
$$P(\alpha, \beta) = R + \alpha u + \beta v$$



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



Triangle



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 coordinate 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, \dots, v_n
- A vector v can be express in this form

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

- Scalar set $\{a_1, a_2, \dots, a_n\}$ is called the representation of the given basis

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



Example

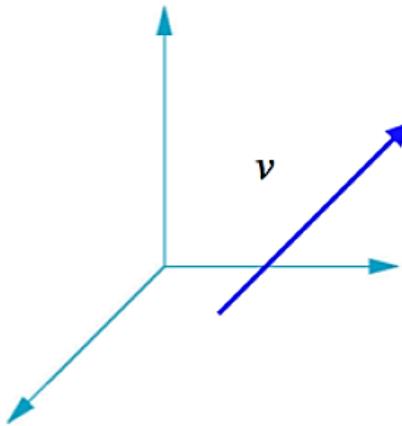
$$\mathbf{v} = 2\mathbf{v}_1 + 3\mathbf{v}_2 - 4\mathbf{v}_3$$
$$\mathbf{a} = [2, 3, -4]^T$$

- 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 .

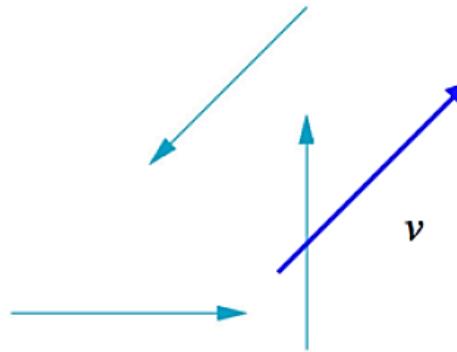


Coordinate

- Which is right?



(a)



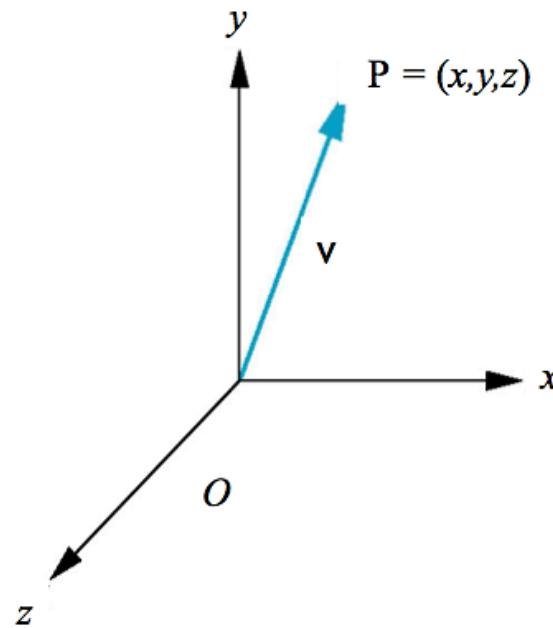
(b)

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 $(O, v_1, v_2, \dots, v_n)$**
- **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

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

$$P = O + \beta_1 v_1 + \beta_2 v_2 + \dots + \beta_n v_n$$

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

$$v = [\alpha_1, \alpha_2, \dots, \alpha_n]^T,$$

$$P = [\beta_1, \beta_2, \dots, \beta_n]^T,$$



Unified representation

- If $0 \cdot P = 0$ (zero vector), $1 \cdot P = P$, then

$$\begin{aligned}v &= \alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n \\&= [v_1, v_2, \dots, v_n, 0] [\alpha_1, \alpha_2, \dots, \alpha_n, 0]^T\end{aligned}$$

$$\begin{aligned}P &= O + \beta_1 v_1 + \beta_2 v_2 + \dots + \beta_n v_n \\&= [v_1, v_2, \dots, v_n, 0] [\beta_1, \beta_2, \dots, \beta_n, 1]^T\end{aligned}$$

- N+1 dimensional **homogeneous** coordinate representation

$$v = [\alpha_1, \alpha_2, \dots, \alpha_n, 0]^T$$

$$P = [\beta_1, \beta_2, \dots, \beta_n, 1]^T$$



Homogeneous coordinate

- General form for 4-dimension homogeneous coordinate:

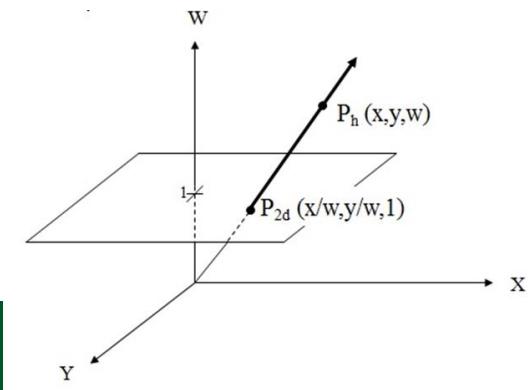
$$\mathbf{P} = [x, y, z, w]^T,$$

- 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, \mathbf{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:

$$\mathbf{a} = [\alpha_1, \alpha_2, \alpha_3]^T$$

$$\mathbf{b} = [\beta_1, \beta_2, \beta_3]^T$$

- Among them

$$\begin{aligned}\mathbf{v} &= \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \alpha_3 \mathbf{v}_3 = [\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] [\alpha_1, \alpha_2, \alpha_3]^T \\ &= \beta_1 \mathbf{u}_1 + \beta_2 \mathbf{u}_2 + \beta_3 \mathbf{u}_3 = [\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3] [\beta_1, \beta_2, \beta_3]^T\end{aligned}$$

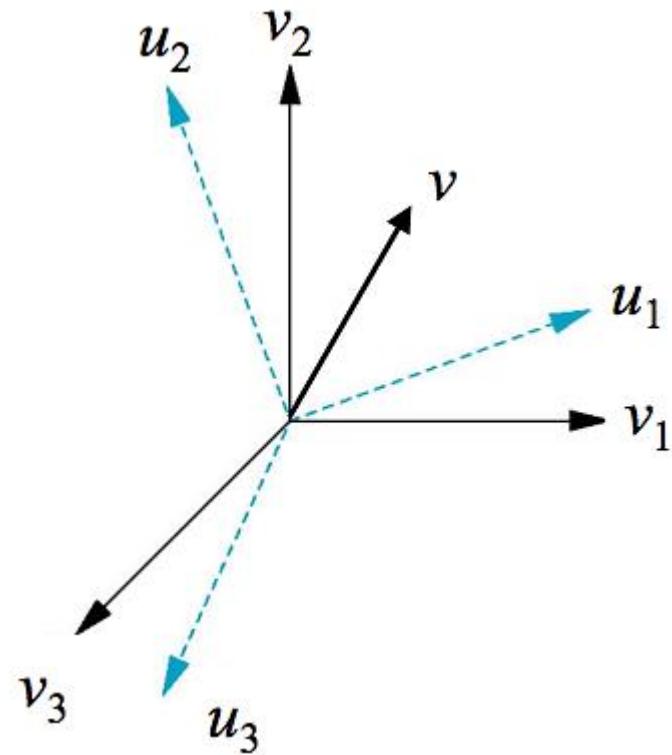


Use 1st Basis to represent 2nd

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

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

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



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

$$\begin{aligned} & (\mathbf{P}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3) \\ & (\mathbf{Q}_0, \mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3) \end{aligned}$$

- 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**

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



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

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

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

- 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
 - 1st frame: $a = [\alpha_1, \alpha_2, \alpha_3, \alpha_4]^T$
 - 2nd 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.



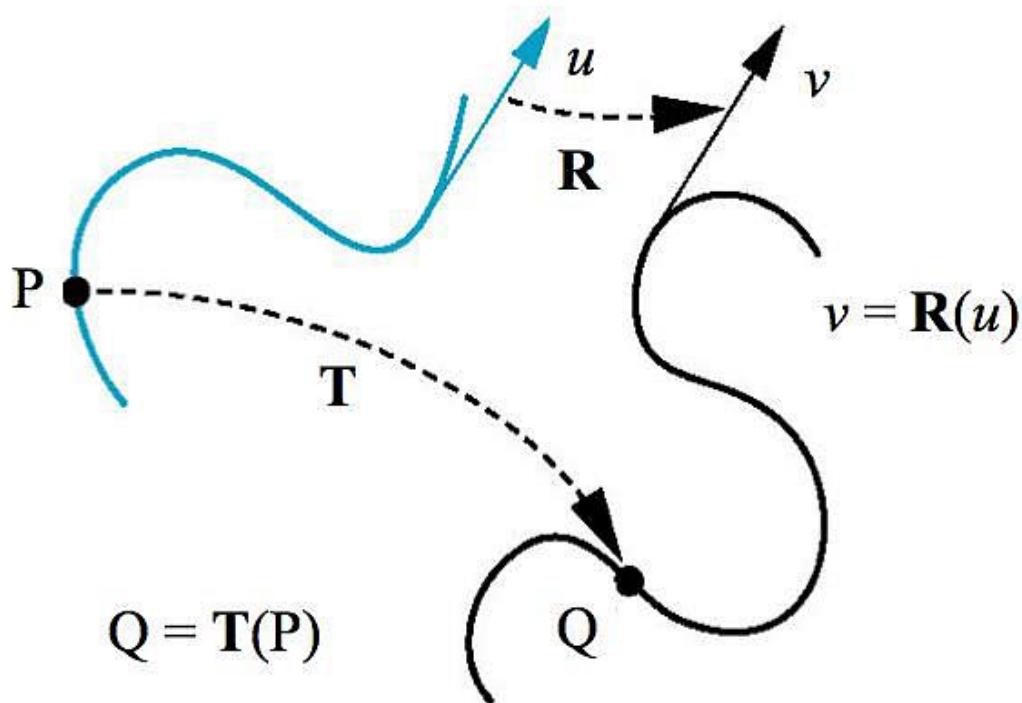
Outline

- Geometry
- Representation
- Transformation
- Transformation in OpenGL



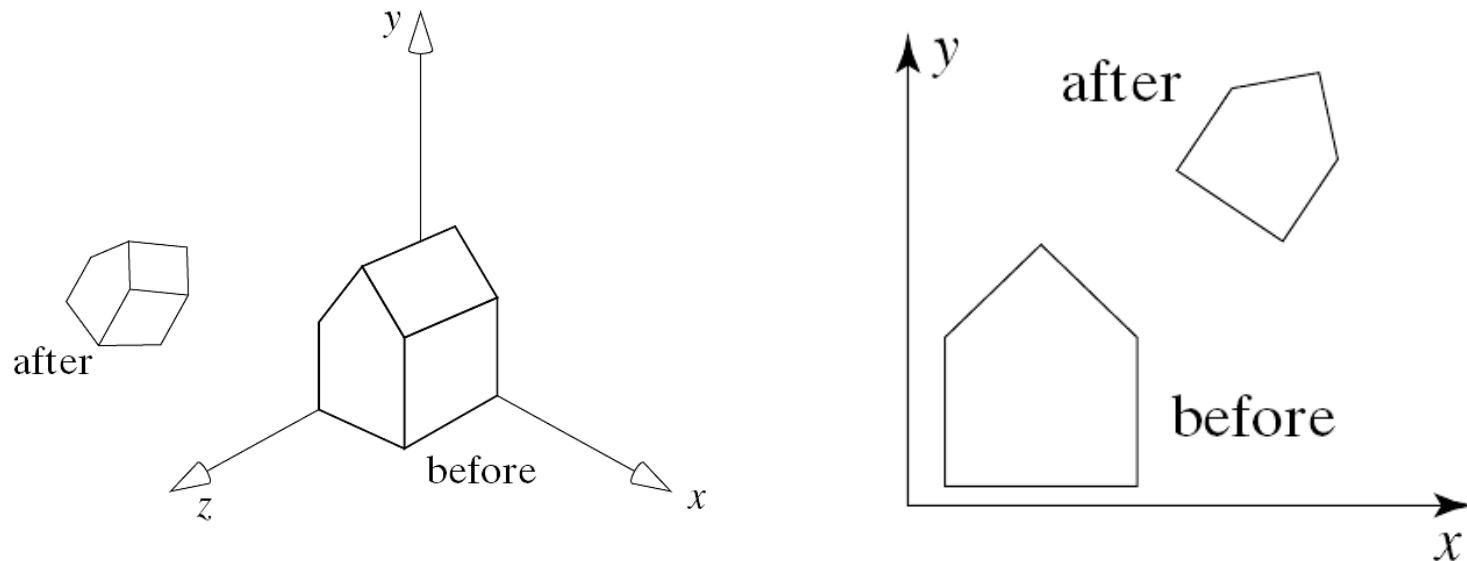
General transformation

- The so-called transformation is to map points in one space to other points in another: $(x', y', z') = \underline{f}(x, y, z)$.



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

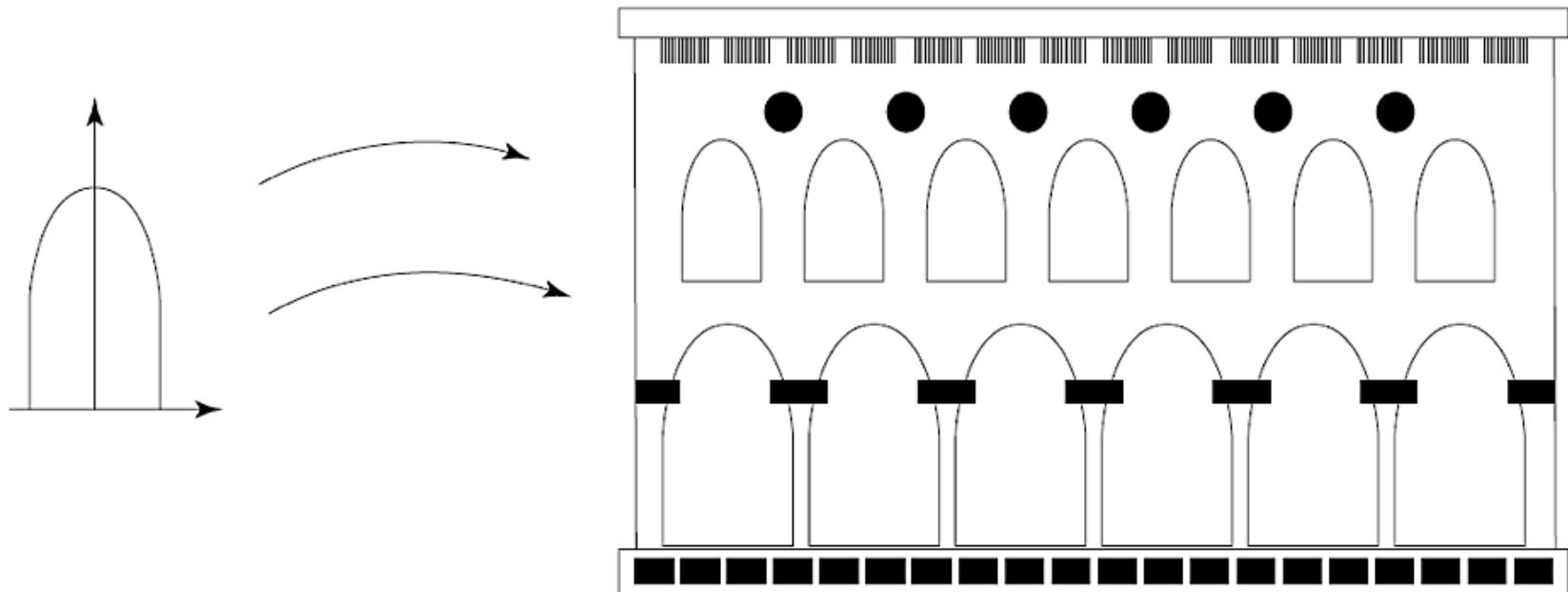


As all objects are eventually represented using points, it is enough to know how to transform points.



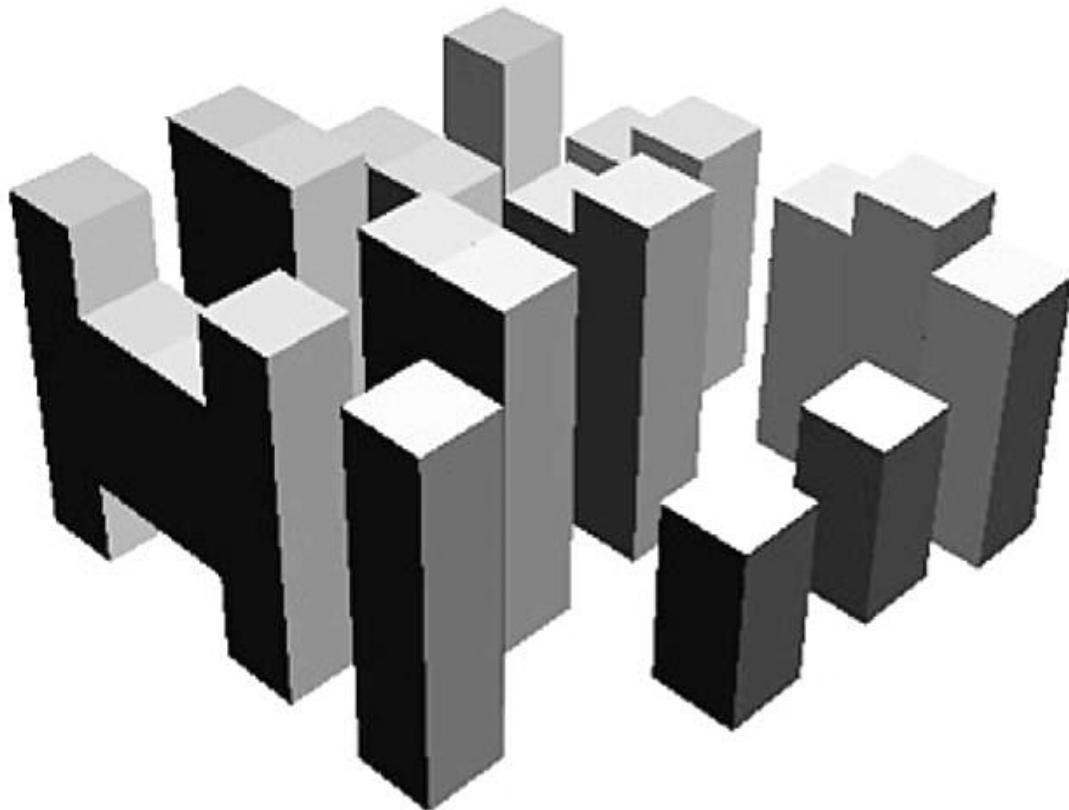
Function 1

- Construct scenes



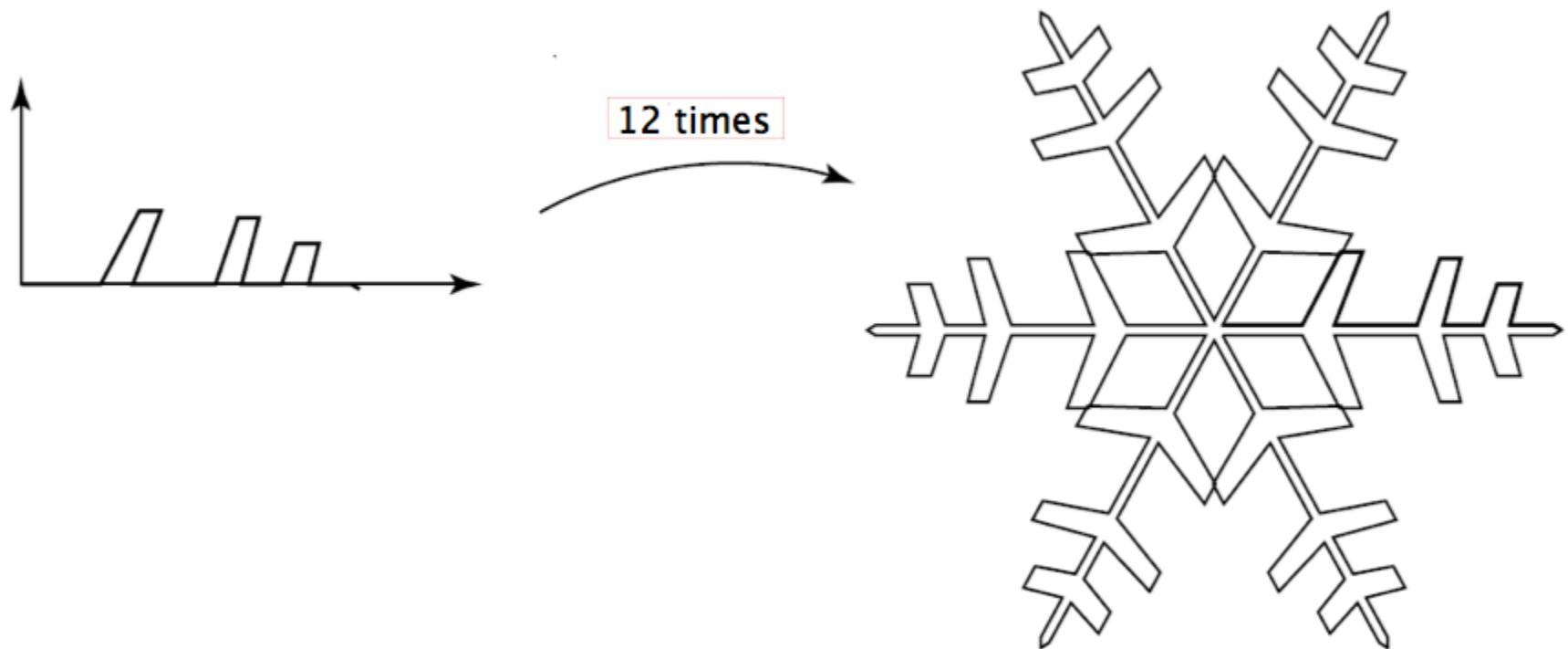
Function 1

- Construct 3D scene



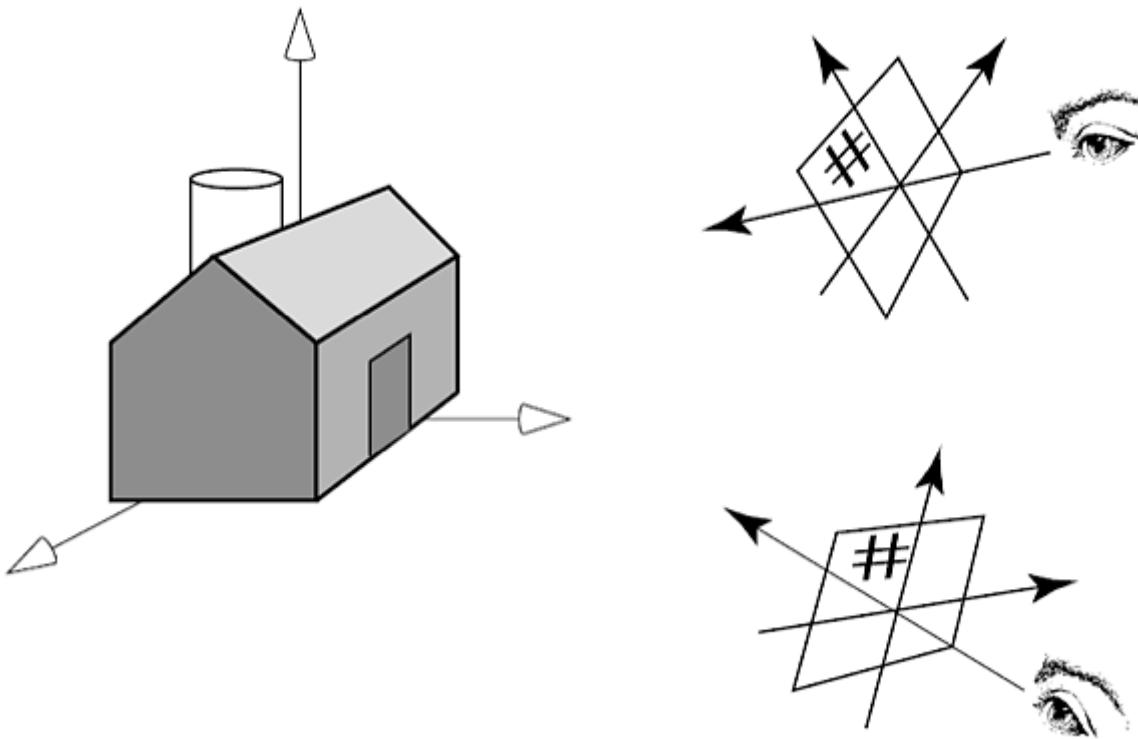
Function 1

- Snowflake structure



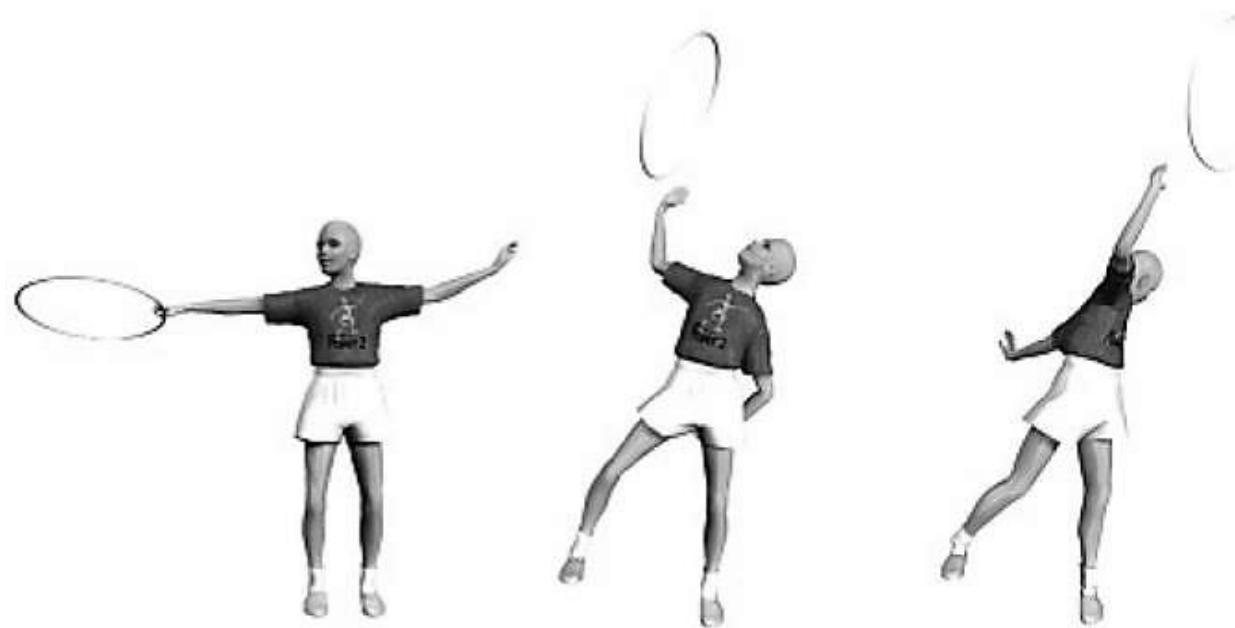
Function 2

- 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.



General transformation

- These transformations can very simple, such as scaling each coordinate, or complex, such as non-linear twists and bends.
- We'll focus on transformations that can be represented easily with **matrix** operations.



Linear Transformations

- Combinations of
 - shear
 - scale
 - rotate
 - reflect
- 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



Scaling

Suppose we set $b=c=0$, but let a and d take on any *positive* value:

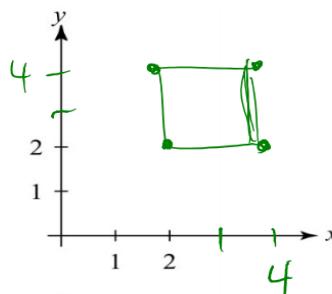
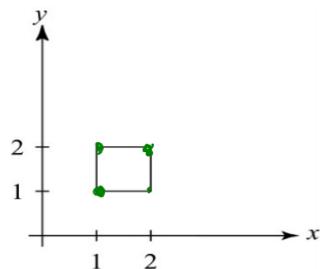
- ◆ Gives a **scaling** matrix:

$$\begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- ◆ Provides **differential (non-uniform) scaling** in x and y :

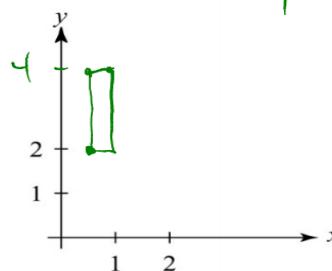
$$x' = ax$$

$$y' = dy$$



$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

$$x' = 2x$$
$$y' = 2y$$



$$\begin{bmatrix} 1/2 & 0 \\ 0 & 2 \end{bmatrix}$$

$$x' = \frac{1}{2}x$$
$$y' = 2y$$



Mirror & Reflection

Suppose we keep $b=c=0$, but let either a or d go negative.

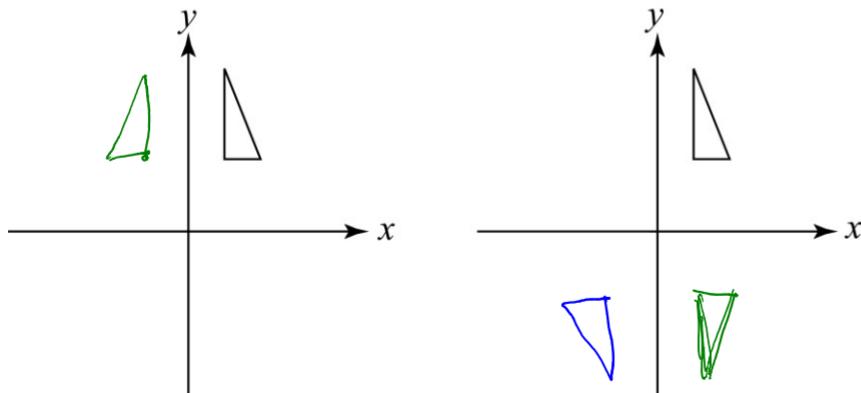
Examples:

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

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

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

rotation by 180°



Shear

Now let's leave $a=d=1$ and experiment with $b\dots$

The matrix

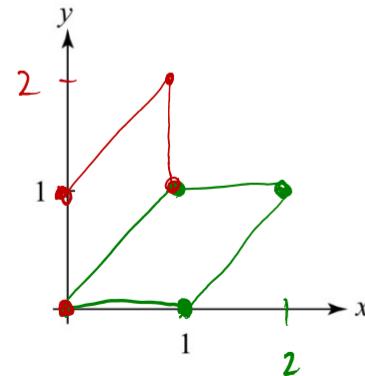
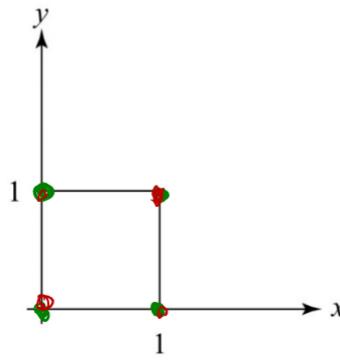
$$\begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix}$$

gives:

$$x' = x + by$$

$$y' = y$$

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ x+y \end{bmatrix}$$

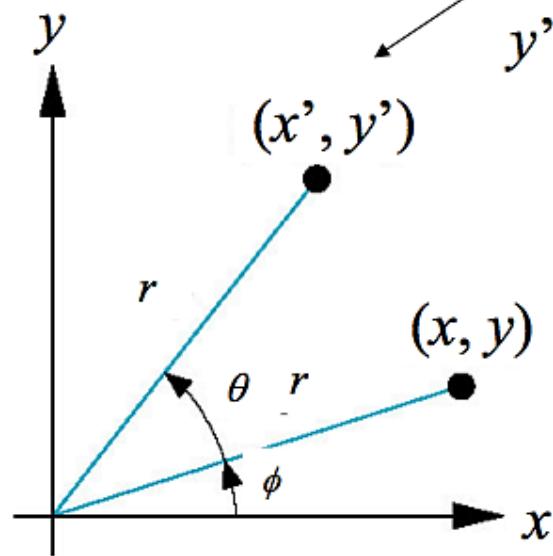


$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x+y \\ y \end{bmatrix}$$



2D rotation

- Consider θ degrees rotation about the origin



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

$$\cos(A+B) = \cos A \cos B - \sin A \sin B$$
$$\sin(A+B) = \sin A \cos B + \cos A \sin B$$

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

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

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



Limitations of 2 x 2 Matrix

A 2 x 2 linear transformation matrix allows

- ◆ Scaling
- ◆ Rotation
- ◆ Reflection
- ◆ Shearing

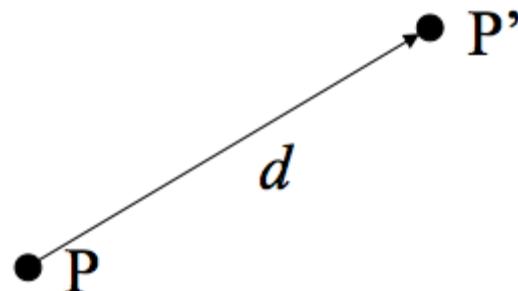
Q: What important operation does that leave out?

translation



Translation

- Put a point to a new position



- Determined by a vector d
 - $P' = P + d$



Homogenous coordinates

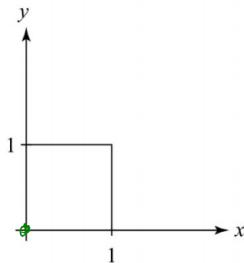
Idea is to loft the problem up into 3-space, adding a third component to every point:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

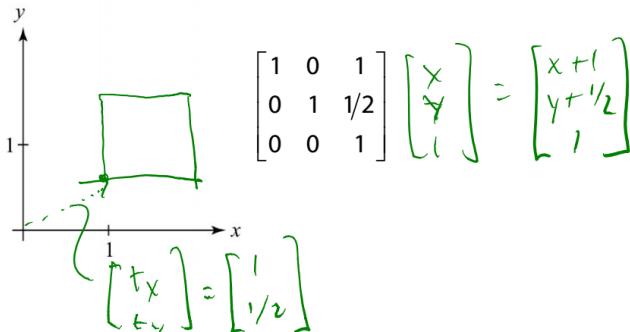
Adding the third "w" component puts us in **homogenous coordinates**.

And then transform with a 3×3 matrix:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = T(\mathbf{t}) \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$



... gives **translation!**



Affine Transformations

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

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



Anatomy of an affine matrix

In matrix form, 2D affine transformations always look like this:

$$M = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} A & \mathbf{t} \\ 0 & 0 & 1 \end{bmatrix}$$

2D affine transformations always have a bottom row of [0 0 1].

An “affine point” is a “linear point” with an added w -coordinate which is always 1:

$$\mathbf{p}_{\text{lin}} = \begin{bmatrix} x \\ y \end{bmatrix}$$
$$\mathbf{p}_{\text{aff}} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\mathbf{p}_{\text{aff}} = \begin{bmatrix} \mathbf{p}_{\text{lin}} \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Applying an affine transformation gives another affine point:

$$M\mathbf{p}_{\text{aff}} = \begin{bmatrix} A\mathbf{p}_{\text{lin}} + \mathbf{t} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + tx \\ cx + dy + ty \\ 1 \end{bmatrix}$$
$$\begin{bmatrix} A \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \\ 1 \end{bmatrix}$$
$$\begin{bmatrix} A\mathbf{p}_{\text{lin}} + \mathbf{t} \\ 1 \end{bmatrix}$$



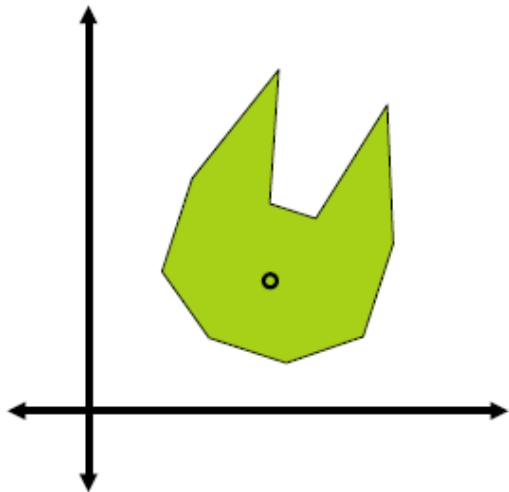
Affine transformation

- 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.



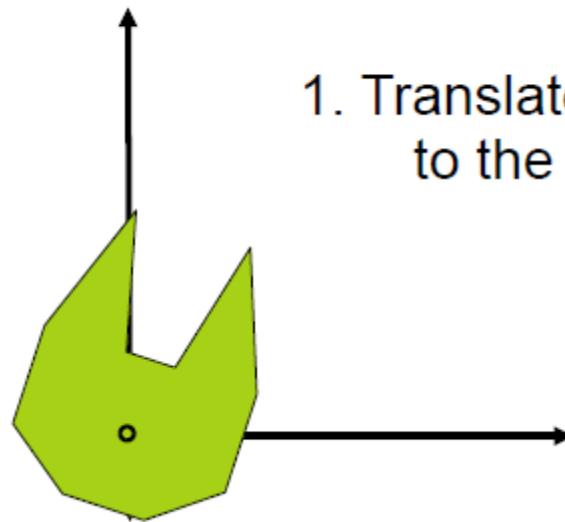
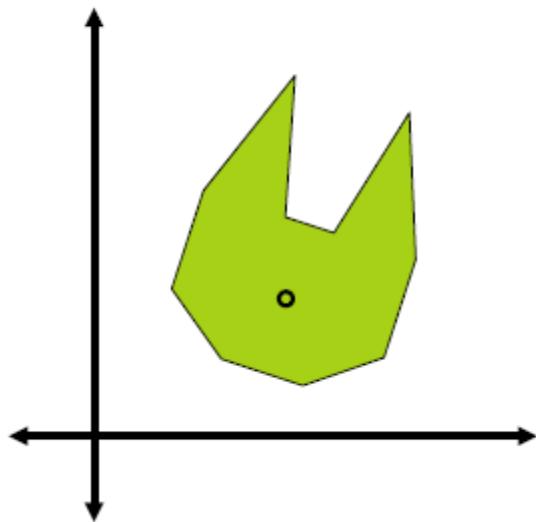
Simple Rotate

To rotate the cat's head about its nose



Simple Rotate

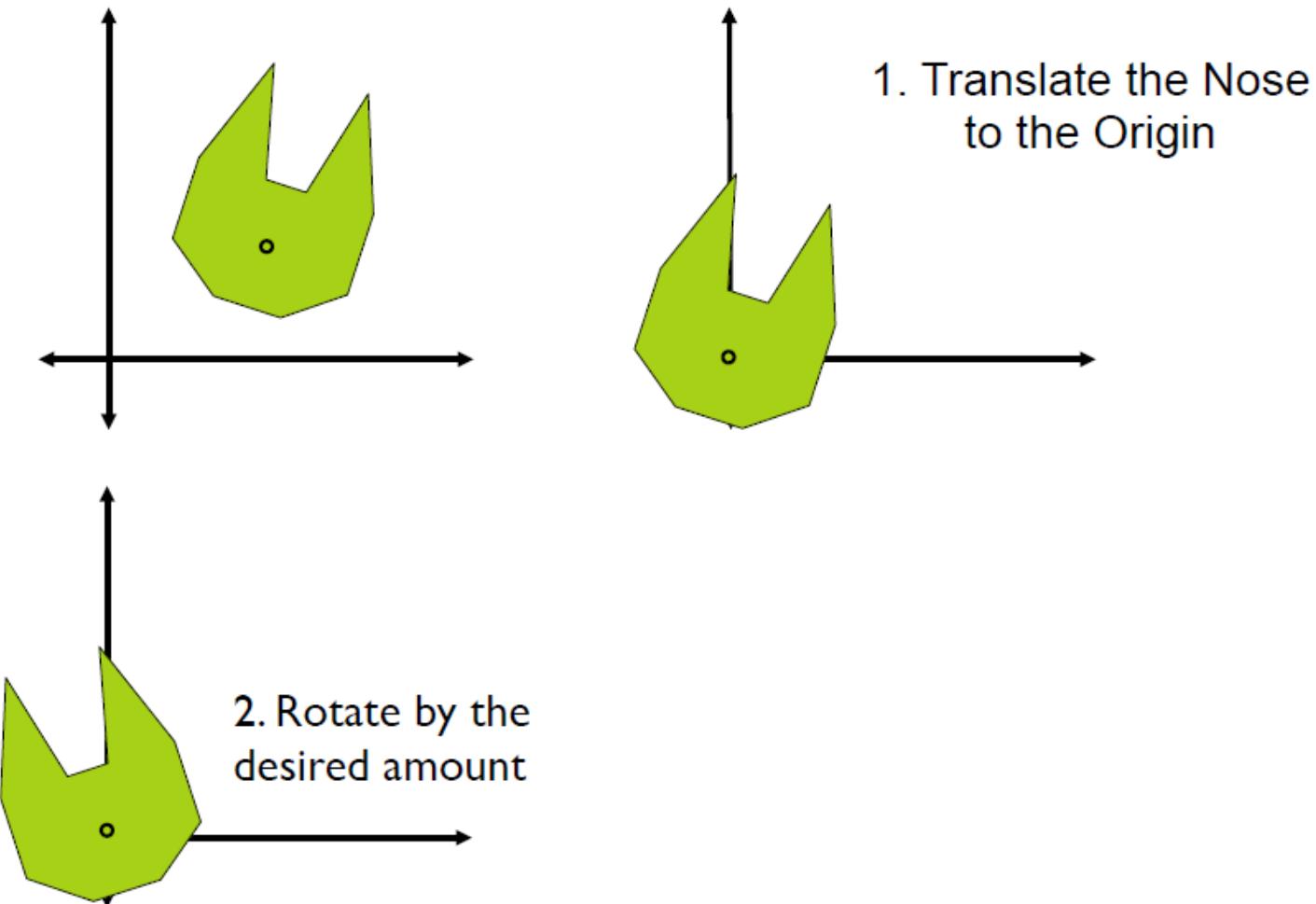
To rotate the cat's head about its nose



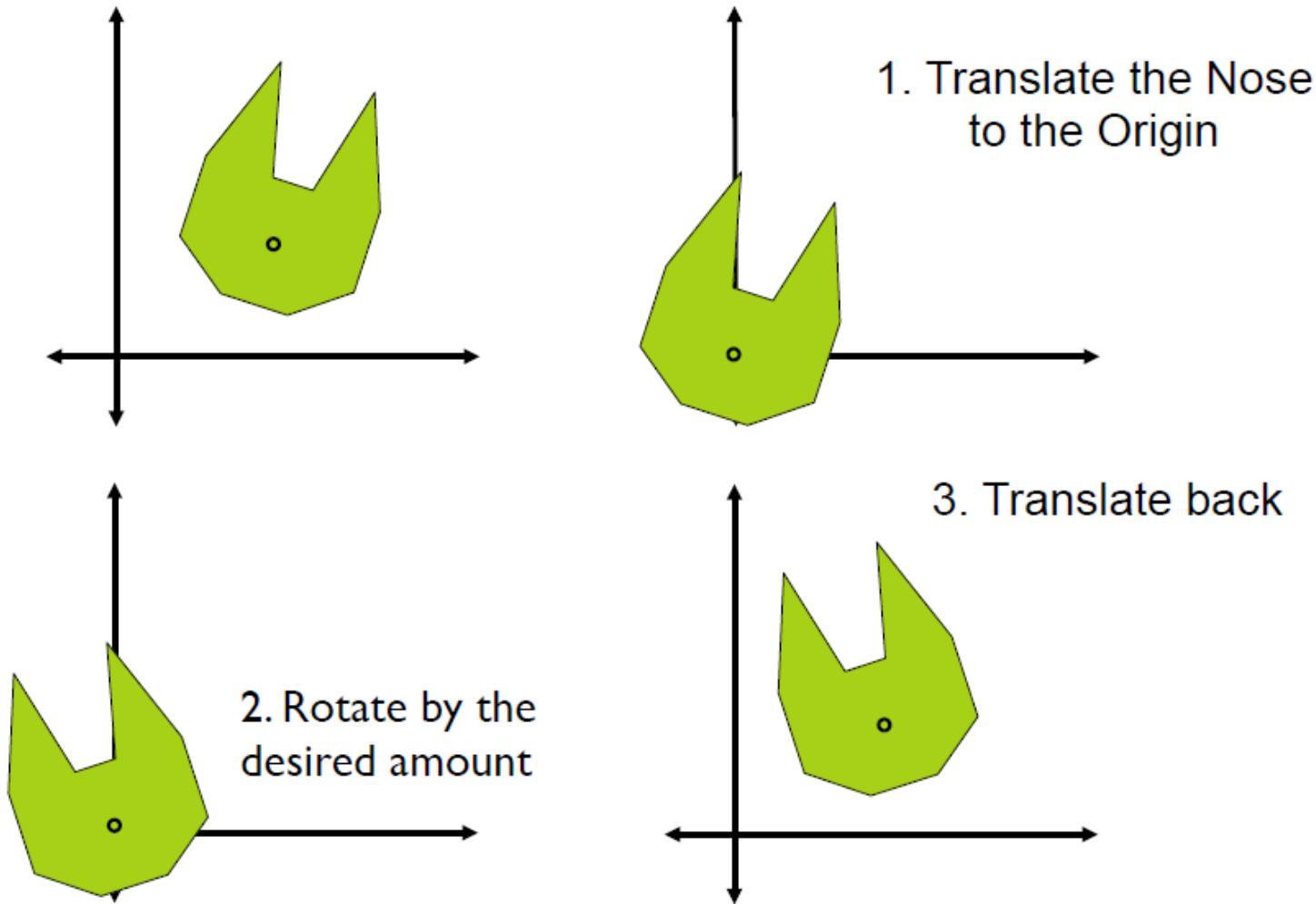
1. Translate the Nose to the Origin



Simple Rotate



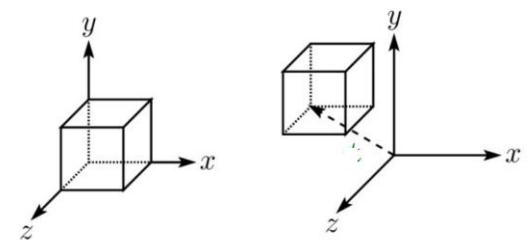
Simple Rotate



Translation in 3D

- Using a 4×4 homogeneous coordinates matrix T to represents the 3D translation transformation
- $p' = Tp$

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

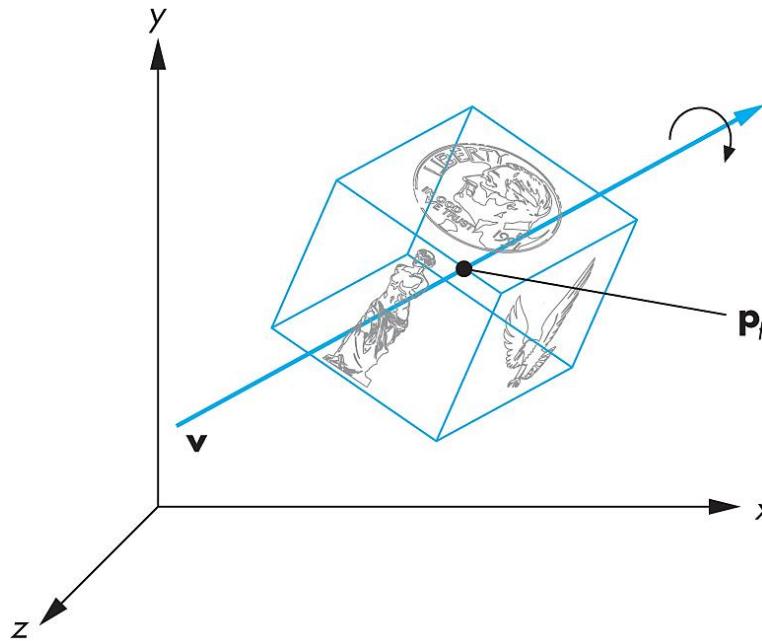


- This form is more easily achieved, because all the affine transformation can be used in this kind of form



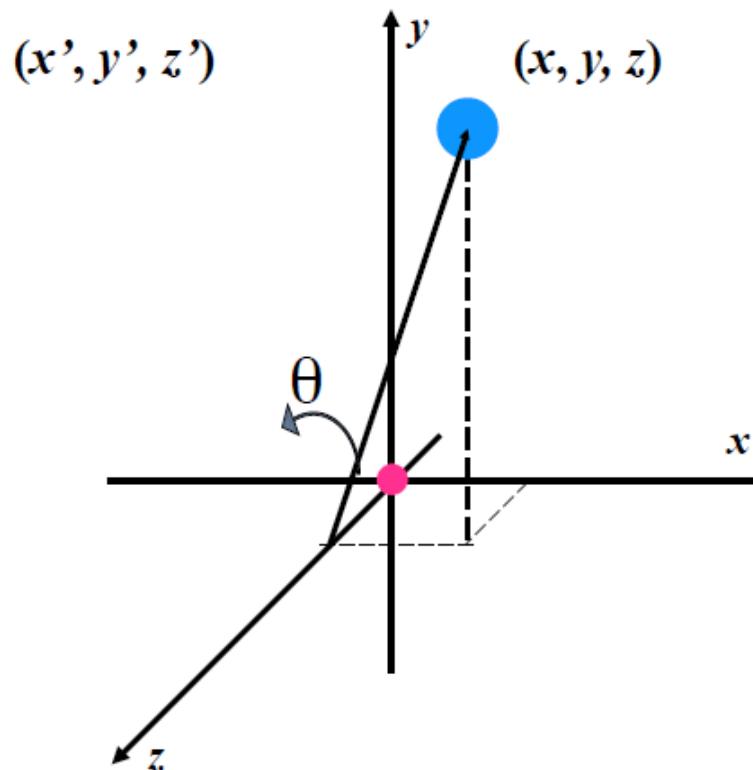
Rotation in 3D

- 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)



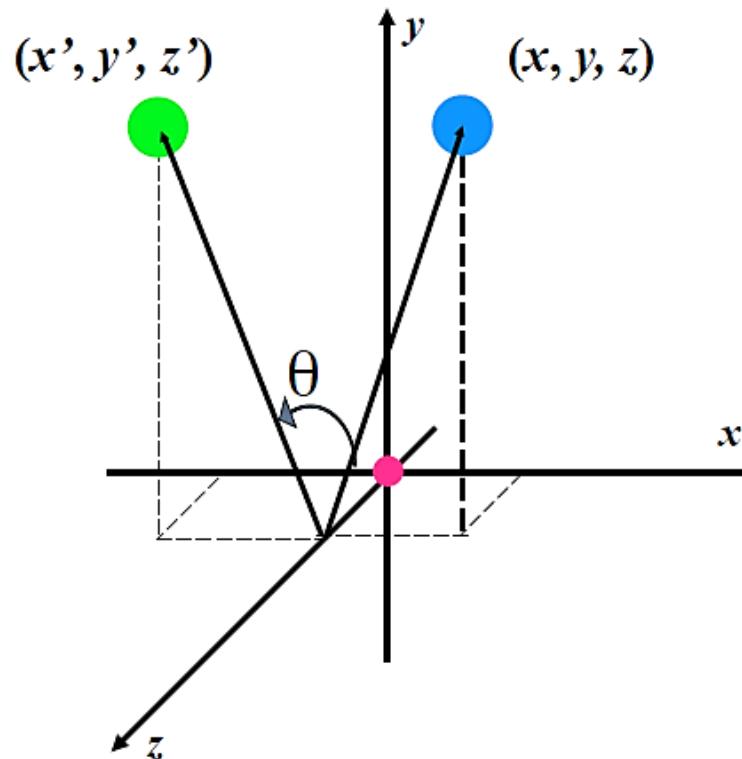
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

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



3D rotation around Z-axis

Rotation (around Z axis)



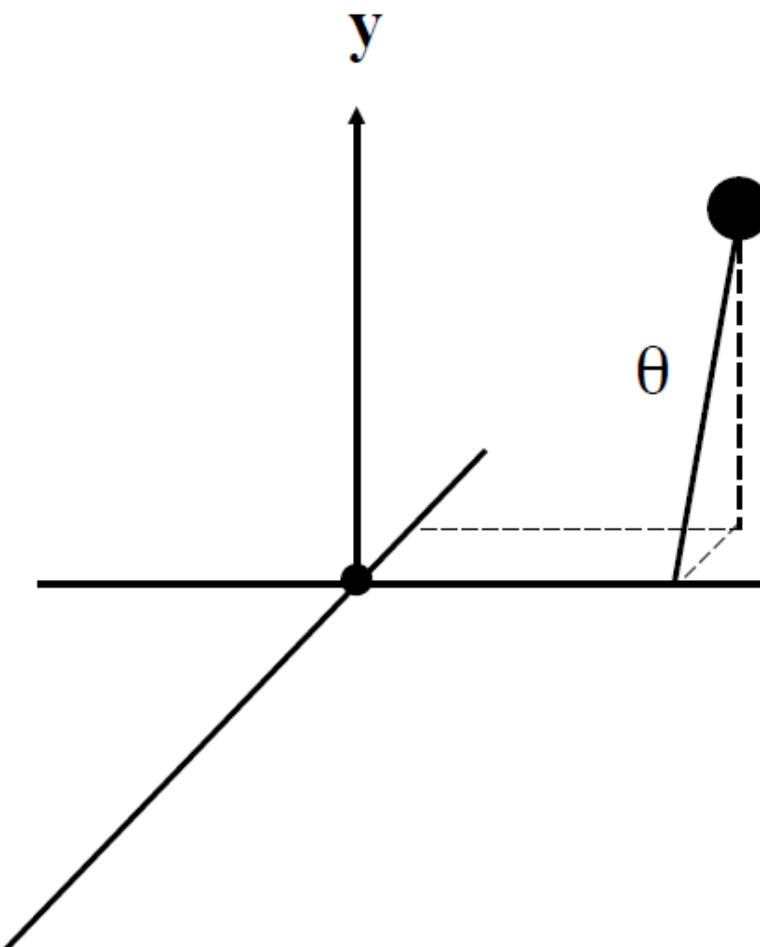
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

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



3D rotation around X-axis

Rotation (around X axis)

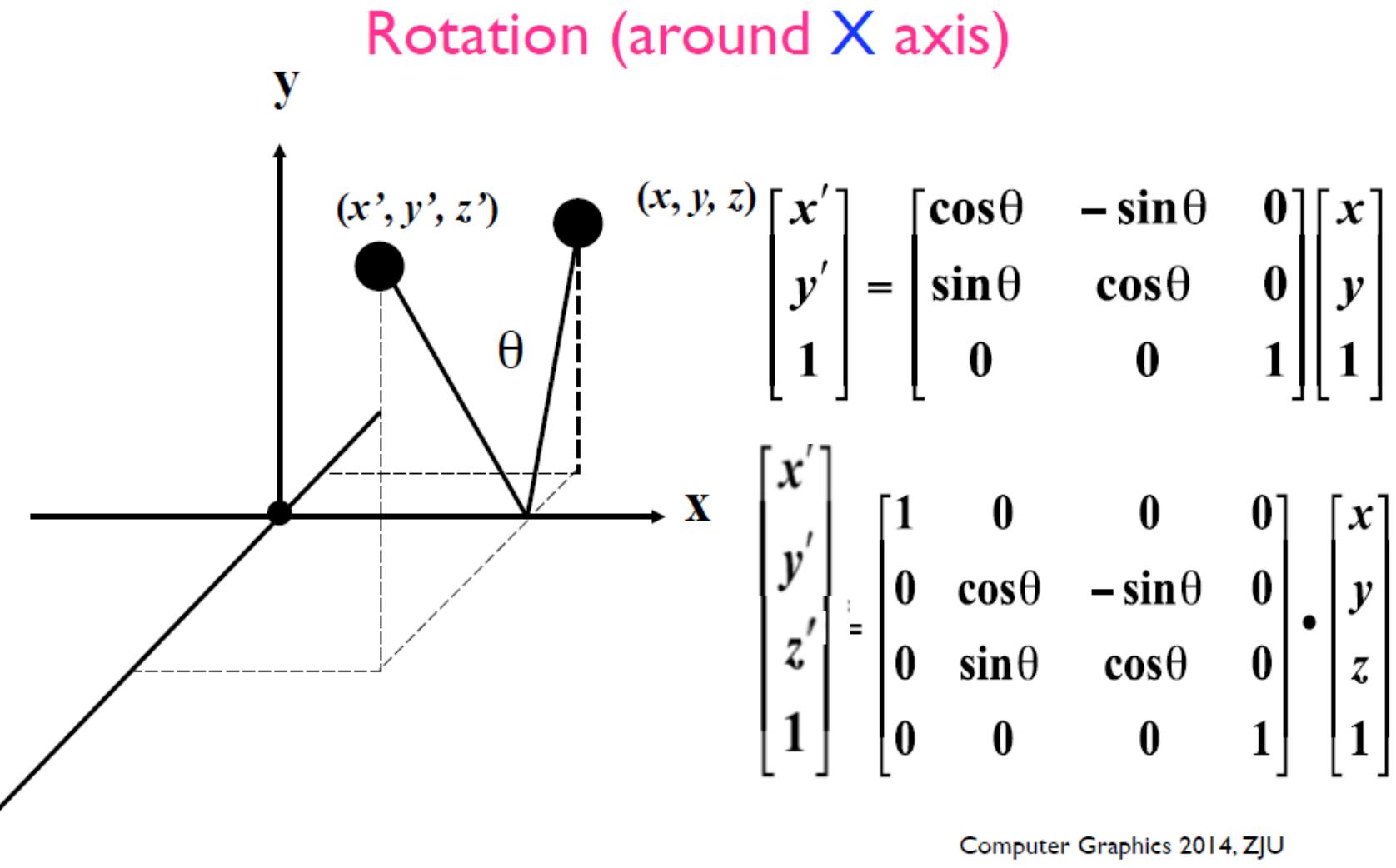


$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

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



3D rotation around X-axis



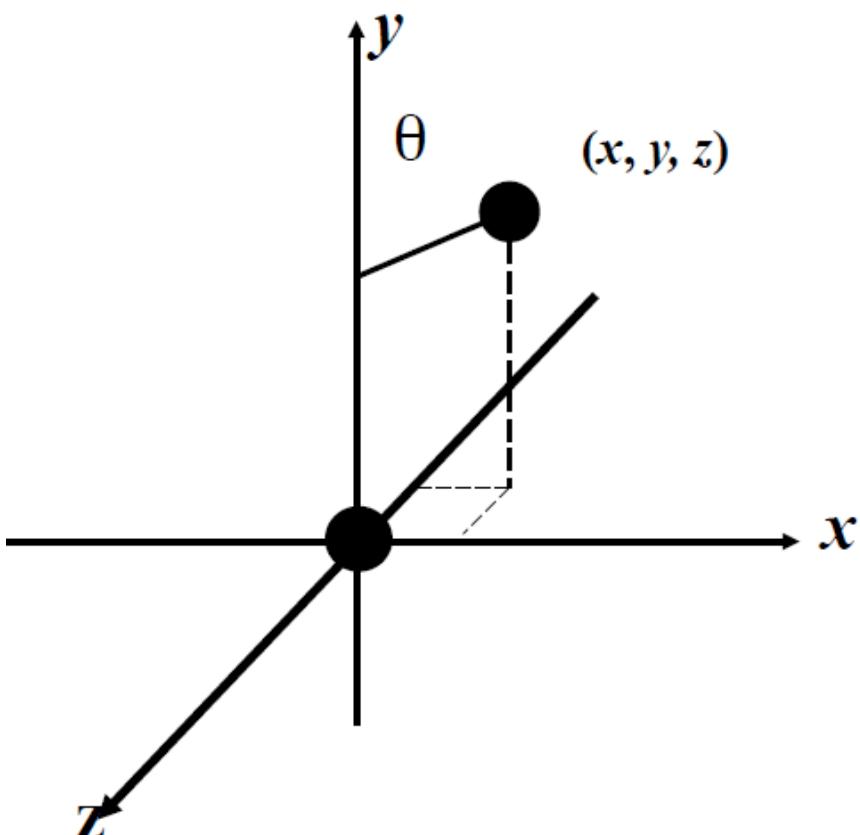
Computer Graphics 2014, ZJU



3D rotation around Y-axis

-

Rotation (around Y axis)



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

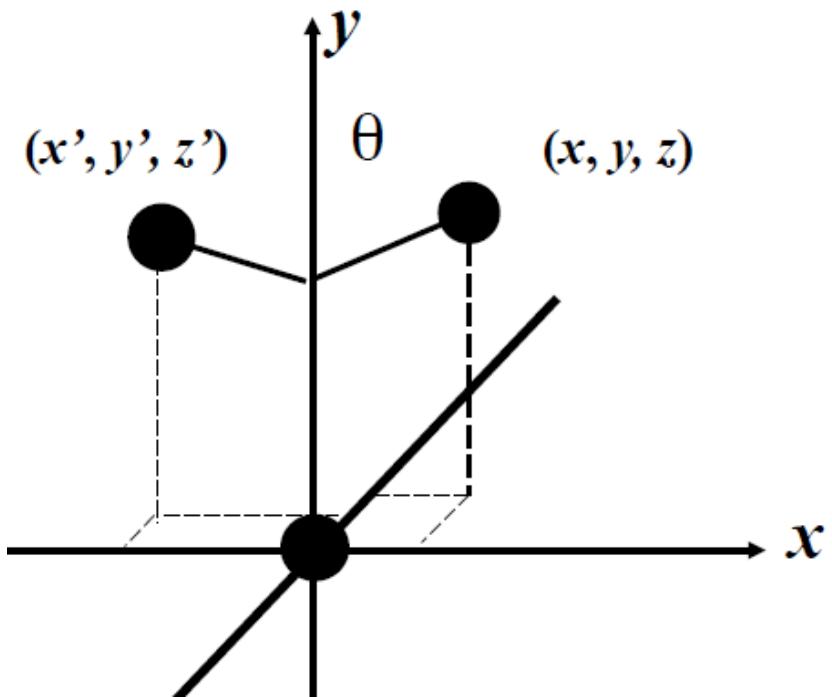
question : why?

Computer Graphics 2014, ZJU



3D rotation around Y-axis

Rotation (around Y axis)



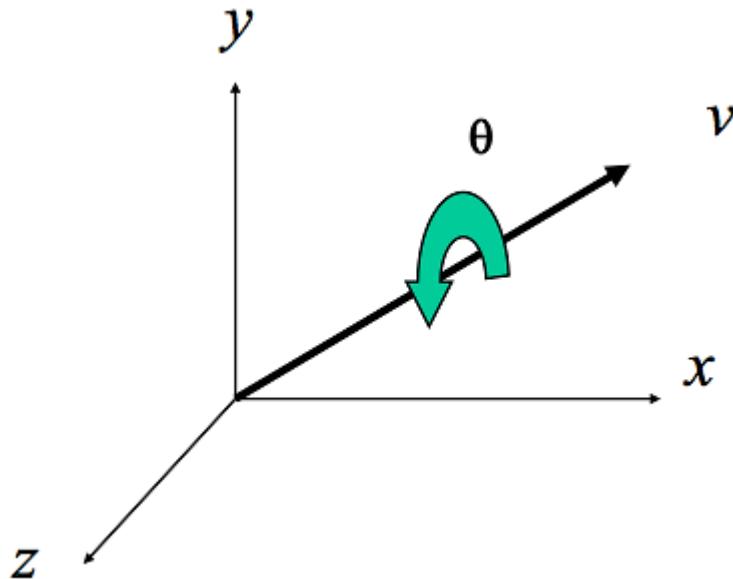
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotate along the general axis through the origin

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

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

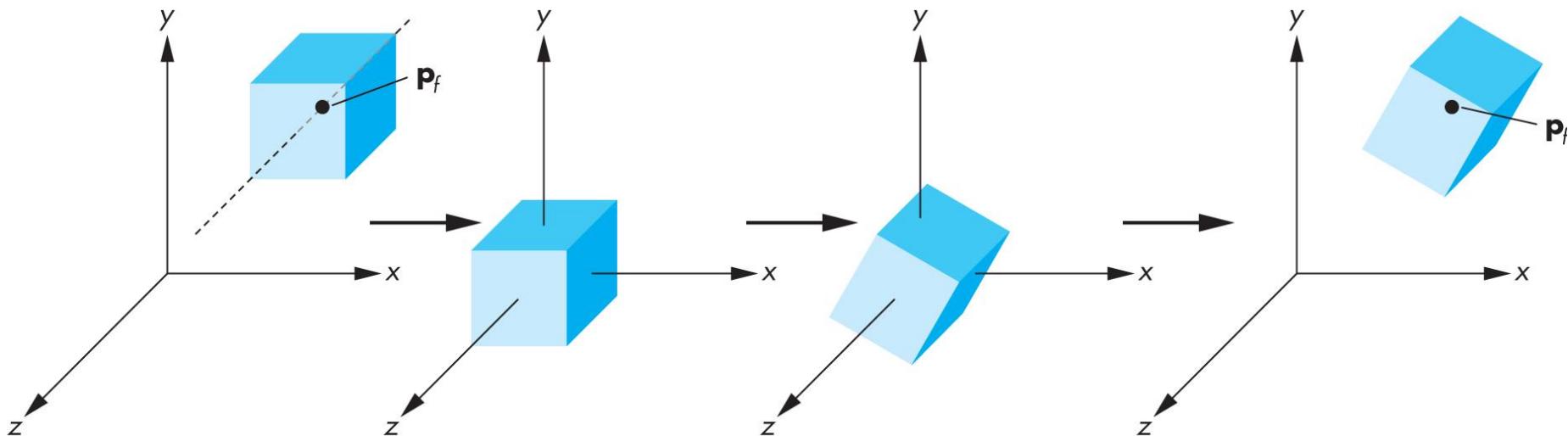
- Note that the rotation order **can not be exchanged**



Rotate along an arbitrary axis not through the origin

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

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

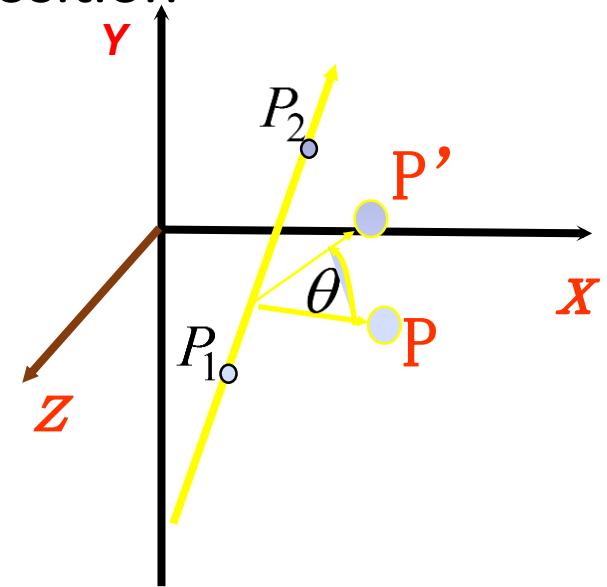


Rotate along an arbitrary axis not through the origin

- Given an axis L defined by two points:

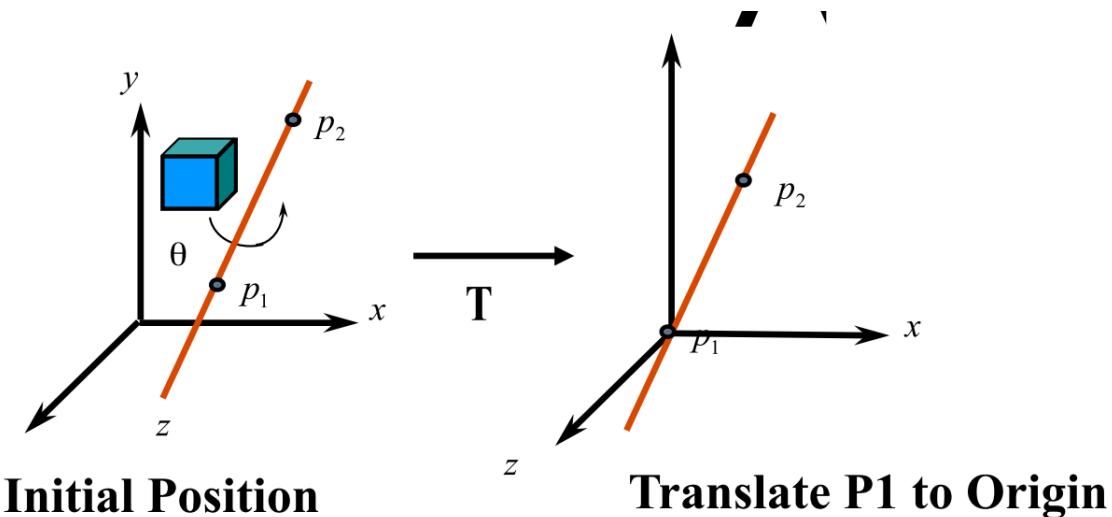
$$P_1 = (x_1, y_1, z_1) \quad P_2 = (x_2, y_2, z_2)$$

- P rotates to P' with respect to the axis L 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

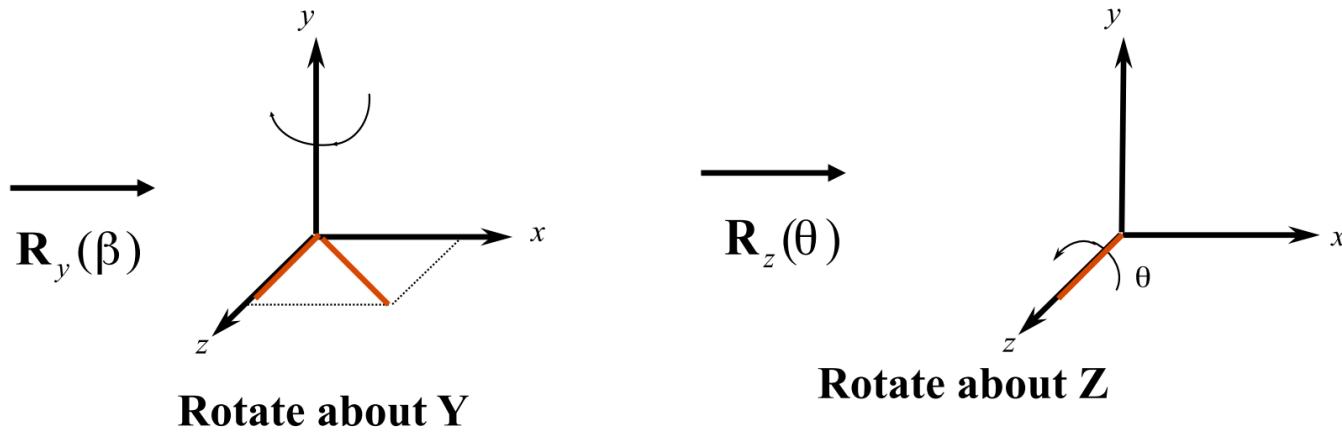
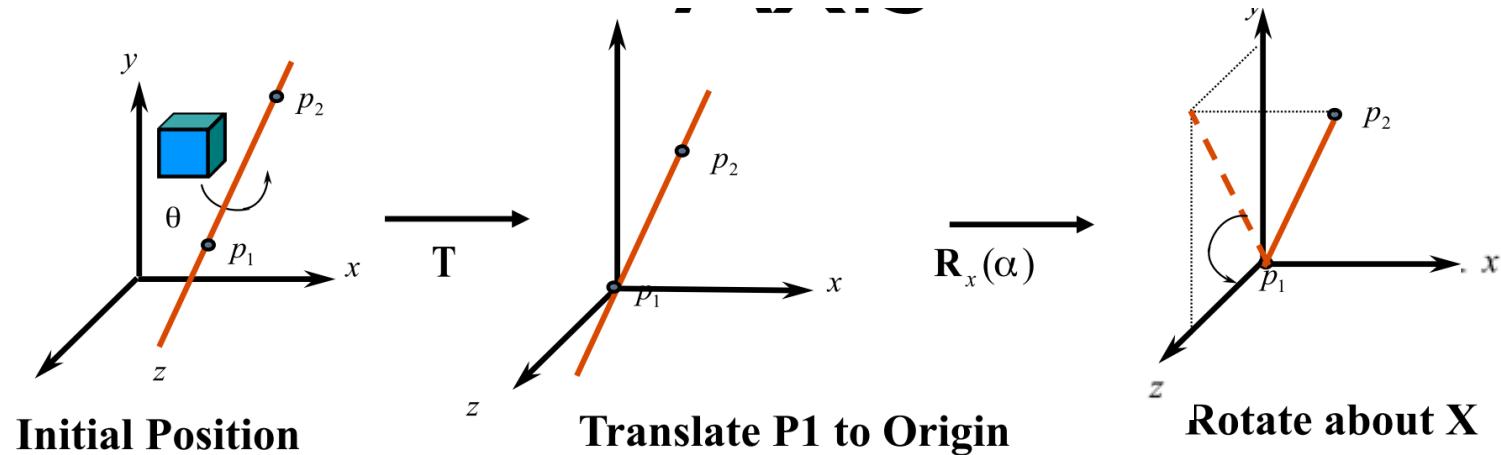
(1a) Translate such that l passes through the origin

(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



Step1



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

$$\begin{array}{c} R(\theta) = T(x_1, y_1, z_1) \bullet R_x^{-1}(\alpha) \bullet R_y^{-1}(\beta) \bullet R_z(\theta) \\ \hline \bullet R_y(\beta) \bullet R_x(\alpha) \bullet T(-x_1, -y_1, -z_1) \\ \hline \end{array}$$

—————



Scaling in 3D

- Scale along each coordinate (origin is fixed point)

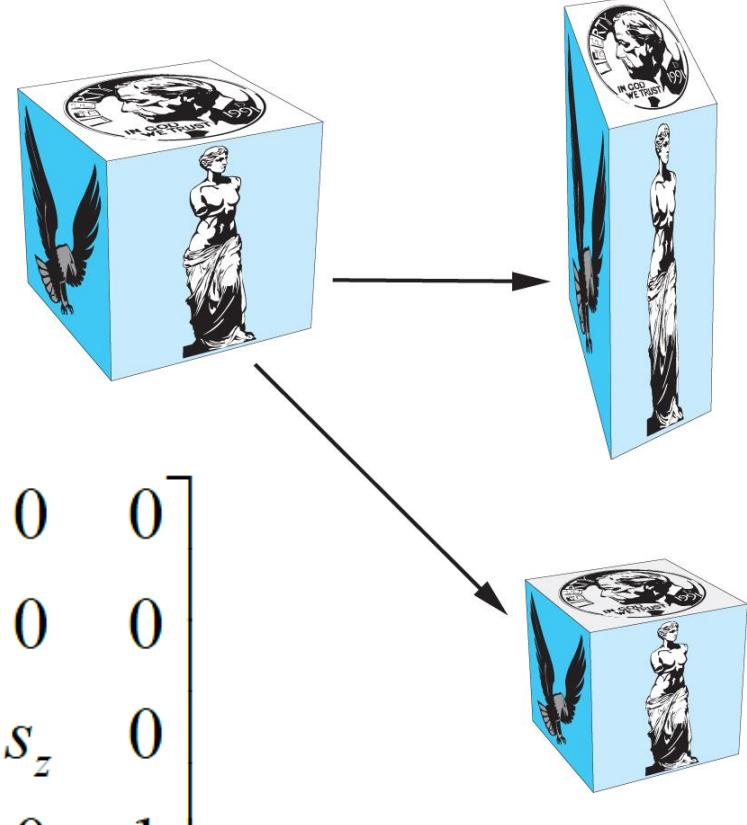
$$x' = s_x x$$

$$y' = s_y y$$

$$z' = s_z z$$

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

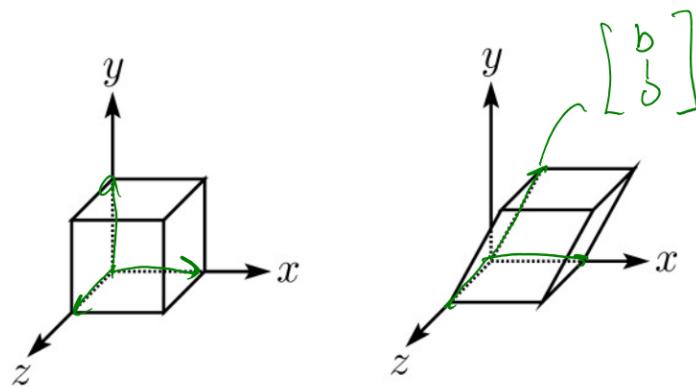
$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Shearing in 3D

Shearing is also more complicated. Here is one example:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & b & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



We call this a shear with respect to the x-z plane.



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 4×4 matrix
- Composition of transformations is represented by product of matrices
- So composition of transformations is also represented by 4×4 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



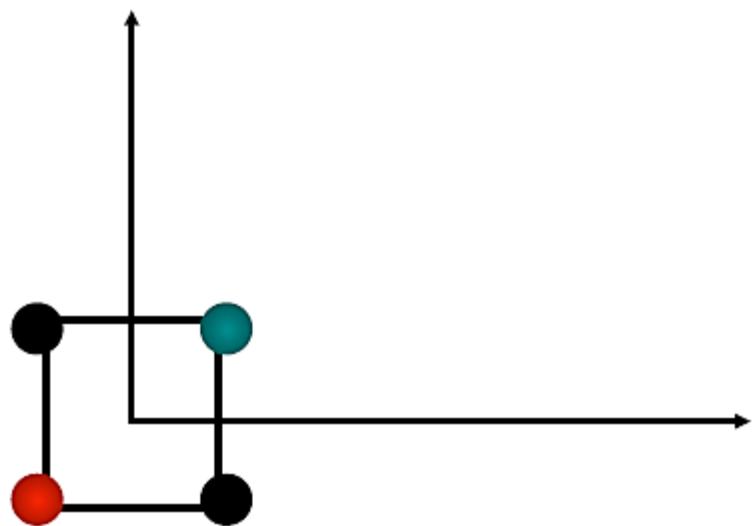
Matrix Composition

- 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
 - $\mathbf{p}' = (T^*(R^*(S^*\mathbf{p})))$
 - $\mathbf{p}' = (T^*R^*S)^*\mathbf{p}$
- 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



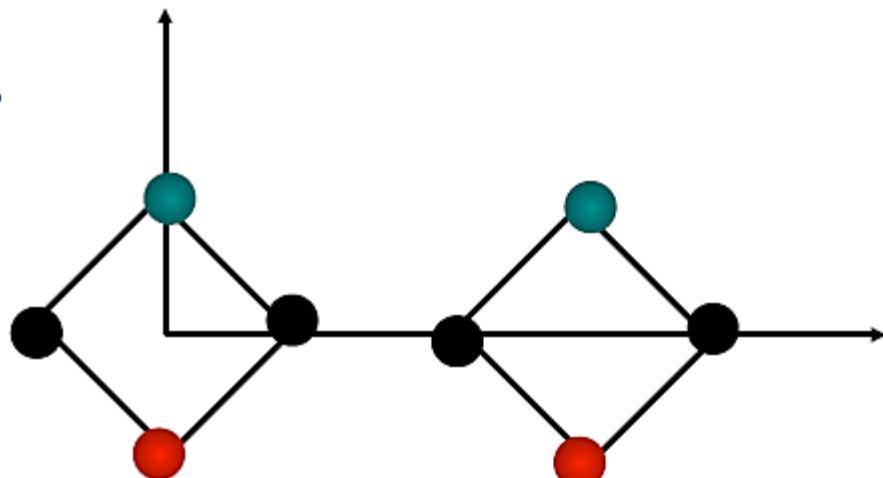
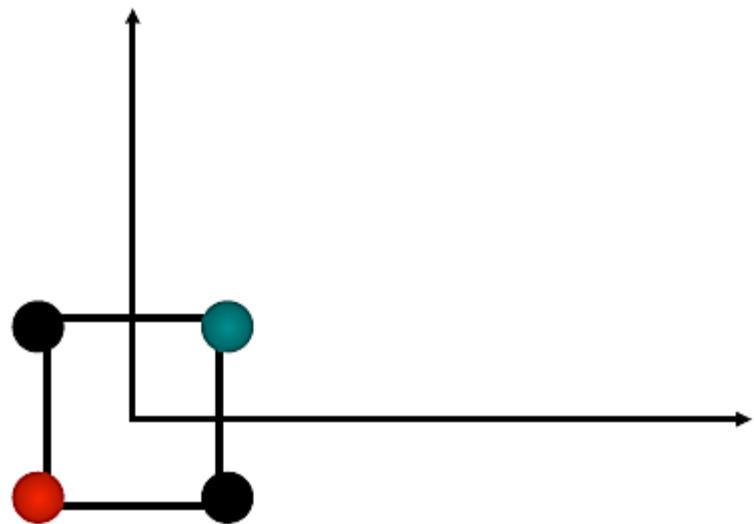
Matrix Multiplication is Not Commutative (不可交換)

Transformation sequence is not commutative



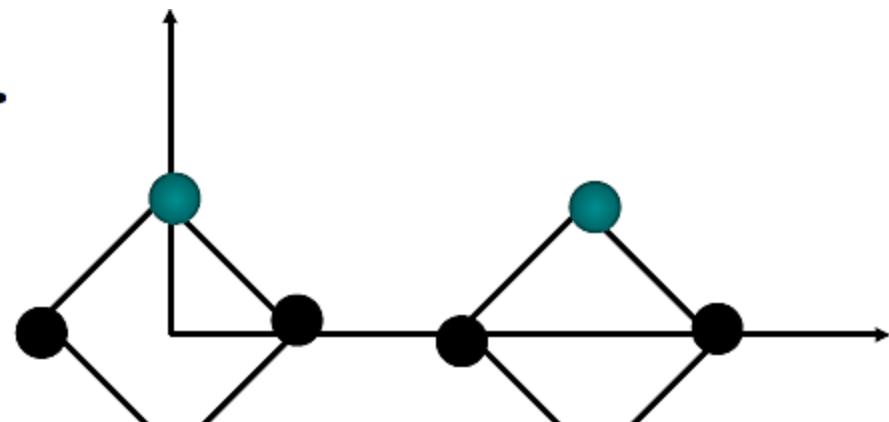
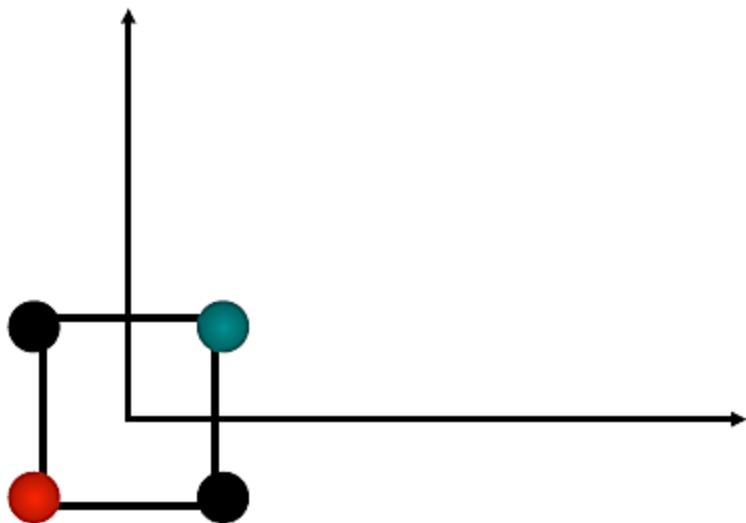
Matrix Multiplication is Not Commutative

First rotate, then translate =>

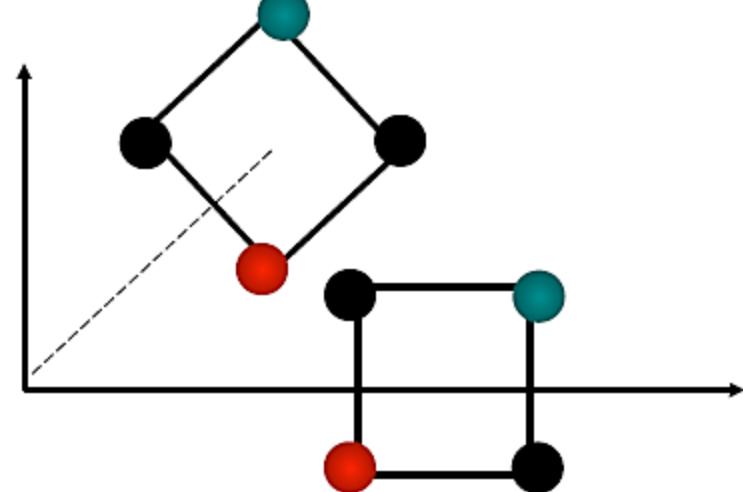


Matrix Multiplication is Not Commutative

First rotate, then translate =>



First translate, then rotate =>



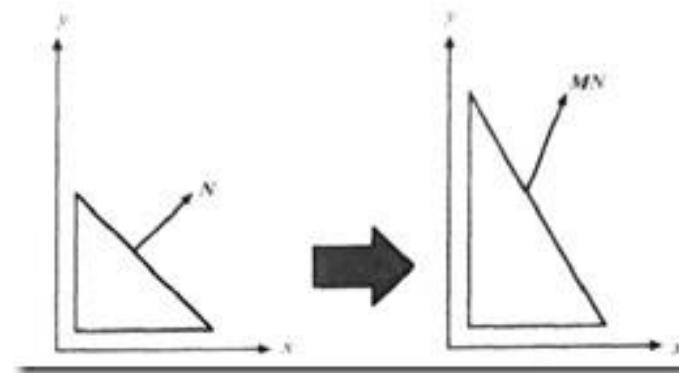
Properties of Transformations

Type	Rigid Body:	Linear	Affine	Projective
Preserves	Rotation & translation	General 3x3 matrix	Linear + translation	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

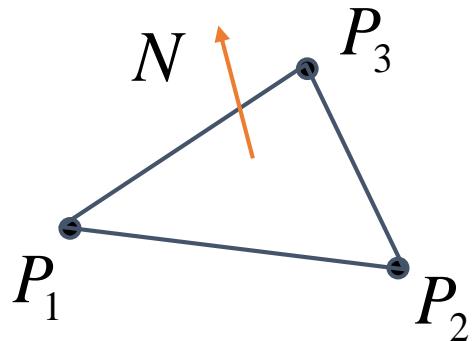
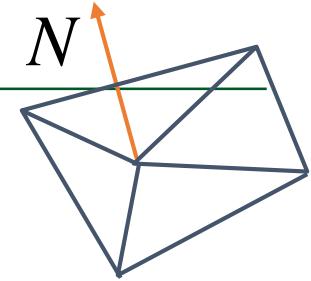
- lines, polygons made up of vertices
 - transform the vertices
 - interpolate between
- does this work for everything? no!
 - normals are trickier



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

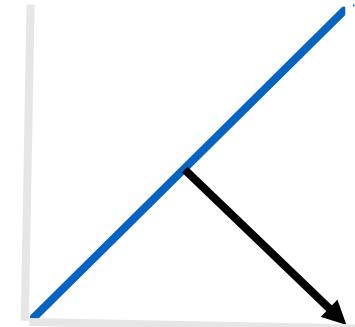
$$\begin{bmatrix} x' \\ y' \\ z' \\ 0 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & T_x \\ m_{21} & m_{22} & m_{23} & T_y \\ m_{31} & m_{32} & m_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

- 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



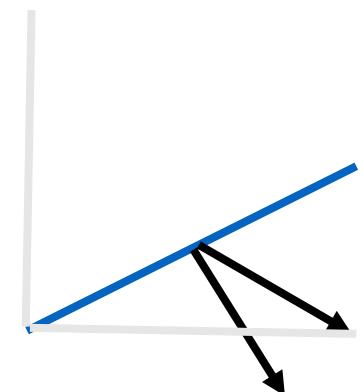
Transforming Normals

- **nonuniform scaling does not work**
- $x - y = 0$ plane
 - line $x = y$
 - normal: $[1, -1, 0]$
- apply nonuniform scale: stretch along x by 2: new plane $x = 2y$
 - normal: $[1, -2, 0]$
 - transformed normal: $[2, -1, 0]$



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

- not perpendicular to plane!



Planes and Normals

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

$$\begin{array}{ccc} P & \xrightarrow{\hspace{1cm}} & P' = MP \\ N & \longrightarrow & N' = QN \end{array} \quad \begin{array}{l} \text{given } M, \\ \text{what should } Q \text{ be?} \end{array}$$

$$N'^T P' = 0 \quad \begin{array}{l} \text{stay perpendicular} \end{array}$$

$$(QN)^T (MP) = 0 \quad \begin{array}{l} \text{substitute from above} \end{array}$$

$$N^T Q^T M P = 0 \quad (\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

$$\boxed{Q^T M = I}$$

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

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

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



Outline

- Geometry
- Representation
- Transformation
- Transformation in OpenGL



固定管线 VS 可编程管线

- 早期的OpenGL使用立即渲染模式（ Immediate mode ，也就是固定渲染管线） ，这个模式下绘制图形很方便。 OpenGL的大多数功能都被库隐藏起来，开发者很少能控制OpenGL进行自由地计算。
- 随着时间推移，规范越来越灵活，开发者对绘图细节有了更多的掌控。立即渲染模式确实容易使用和理解，但是效率太低。因此从OpenGL3.2开始，规范文档开始废弃立即渲染模式，鼓励开发者在OpenGL的核心模式(Core-profile)下进行开发，并要求开发者使用现代的函数。
- 立即渲染模式从OpenGL实际运作中抽象掉了很多细节，因此它在易于学习的同时，也很难让人去把握OpenGL具体是如何运作的。现代函数要求使用者真正理解OpenGL和图形编程，它有一些难度，然而提供了更多的灵活性，更高的效率，更重要的是可以更深入的理解图形编程。



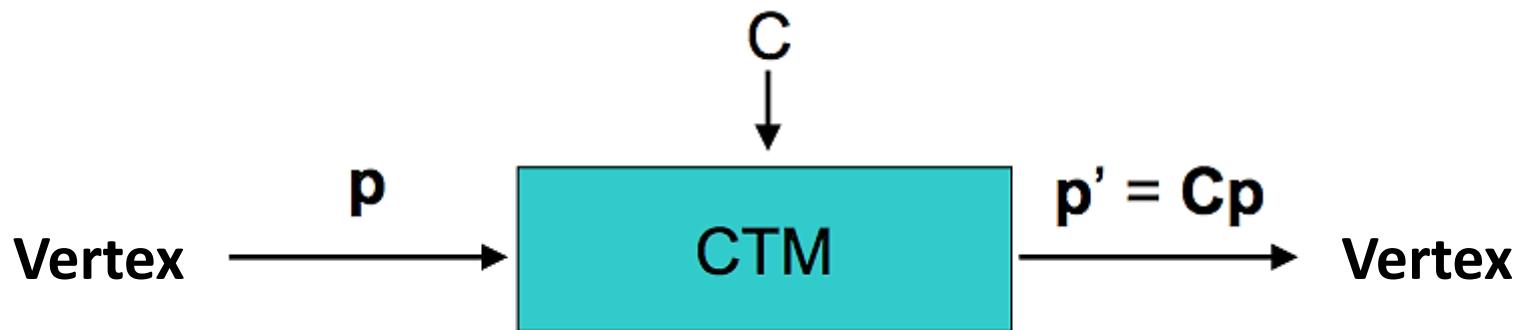
Programming Transformations(固定管线)

- 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.



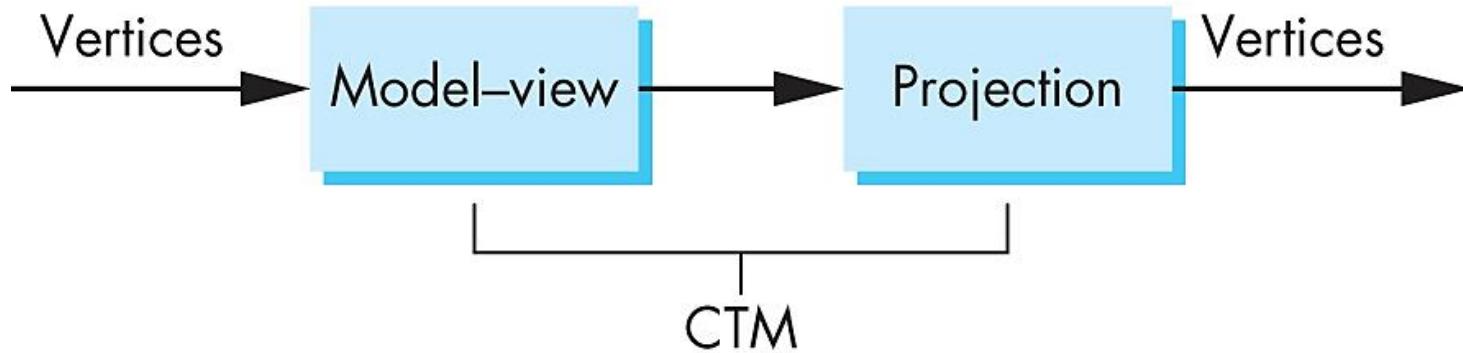
Current Transformation Matrix (CTM)

- CTM is a 4×4 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.



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.



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)



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
<code>glLoadMatrixf(A)</code>	<code>stack = [A]</code>
<code>glPushMatrix()</code>	<code>stack = [A, A]</code>
<code>glMultMatrixf(B)</code>	<code>stack = [AB, A]</code>
<code>glPopMatrix()</code>	<code>stack = [A]</code>



Rotation around a fixed point

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



Rotation about an arbitrary axis

Task:

Rotate an object by 45.0 degrees about the line from (4.0, 5.0, 6.0) to (5.0, 7.0, 9.0). (T_{-p1} , R_{45} , T_{+p1})

```
glMatrixMode (GL_MODEVIEW);  
glLoadIdentity ();  
glTranslatef (4.0, 5.0, 6.0);  
glRotatef (45.0, 1.0, 2.0, 3.0);  
glTranslatef (-4.0, -5.0, -6.0);
```



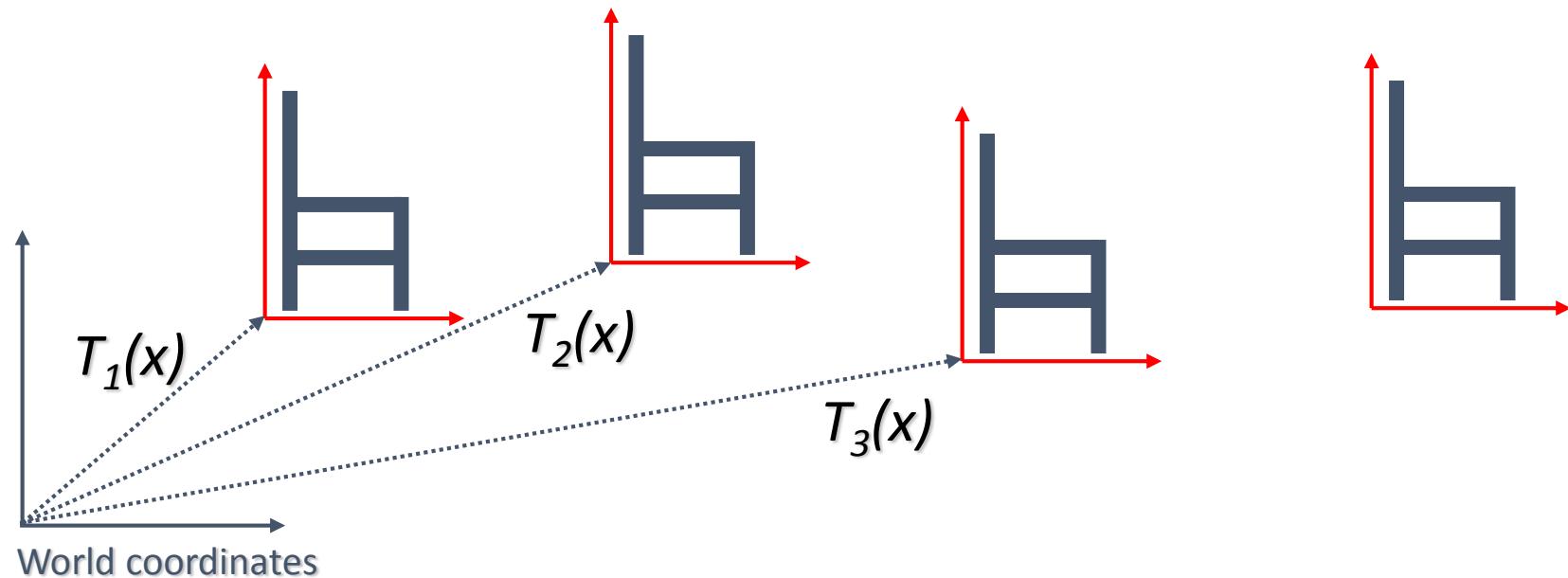
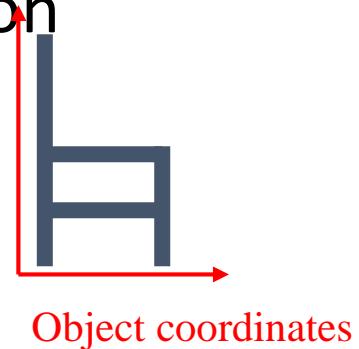
Matrix Stacks

- 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 $T_1 \rightarrow T_2$

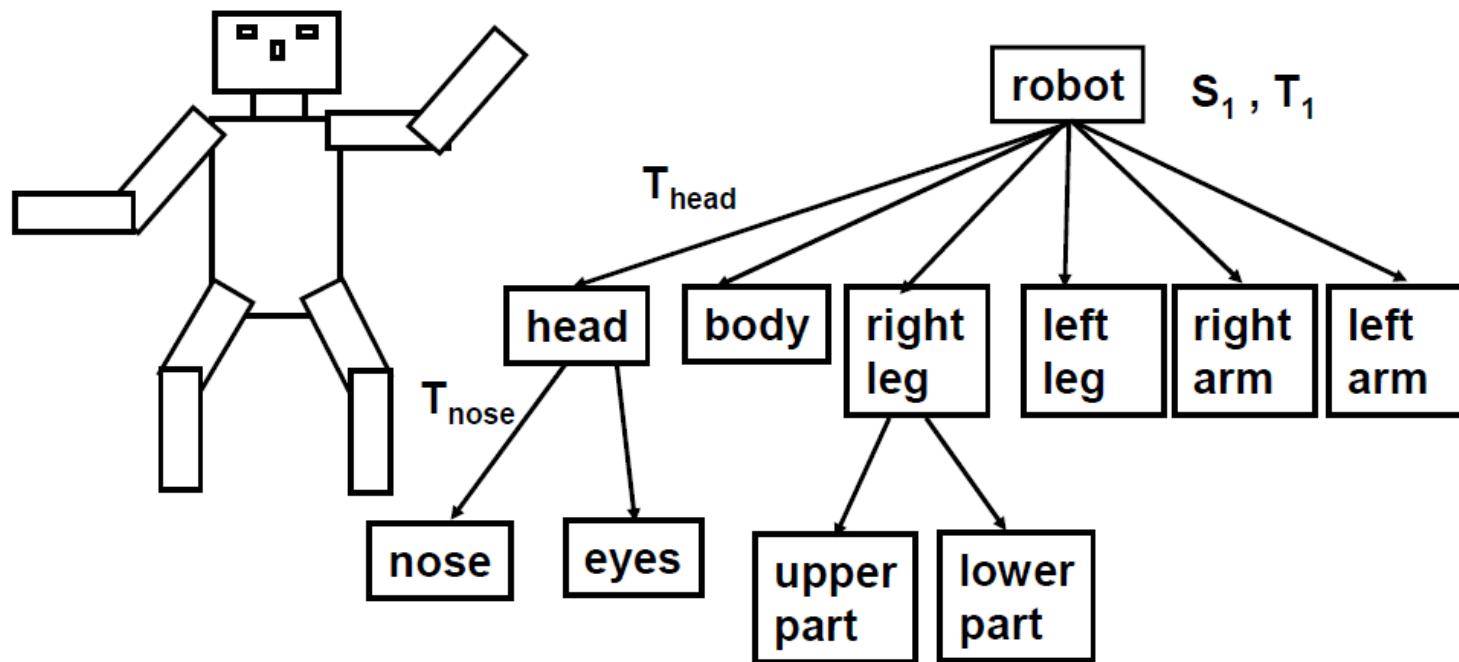


Matrix Stacks

- 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:

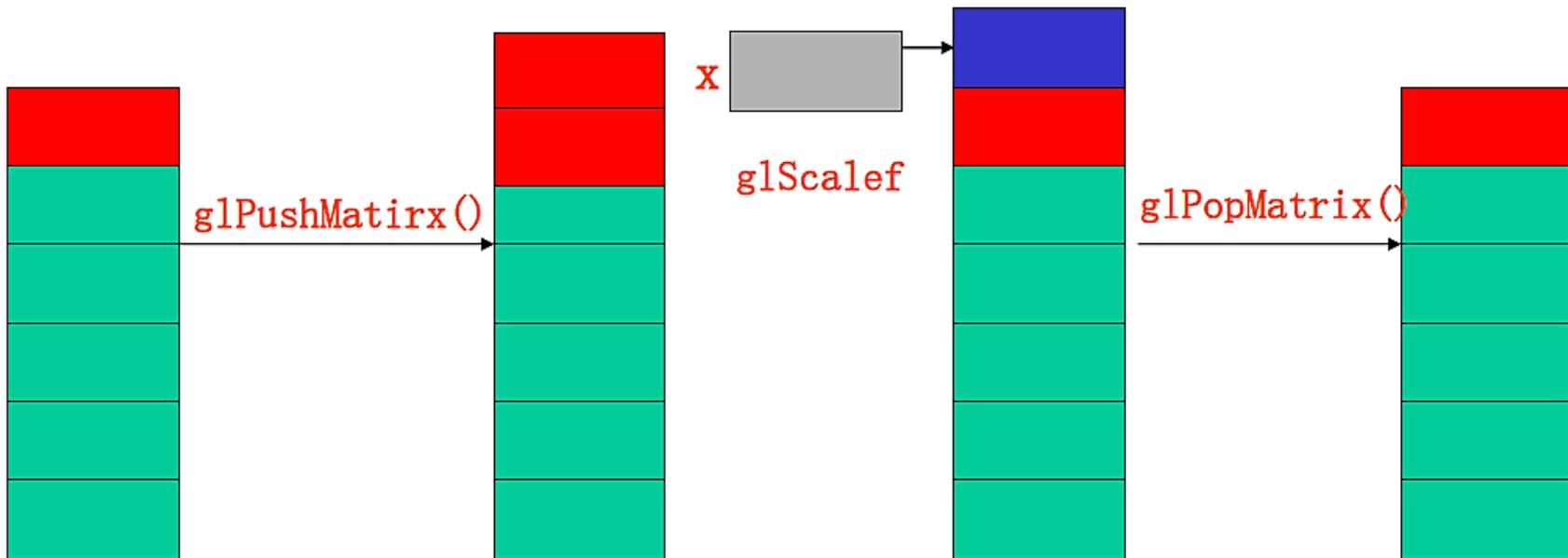
- push: create a copy of the matrix on the top
and put it on the top
- pop: remove the matrix on the top
- multiply: multiply the top by the given matrix
- load: replace the top matrix with a given
matrix



Matrix in OpenGL

- **Maintain matrix stack**

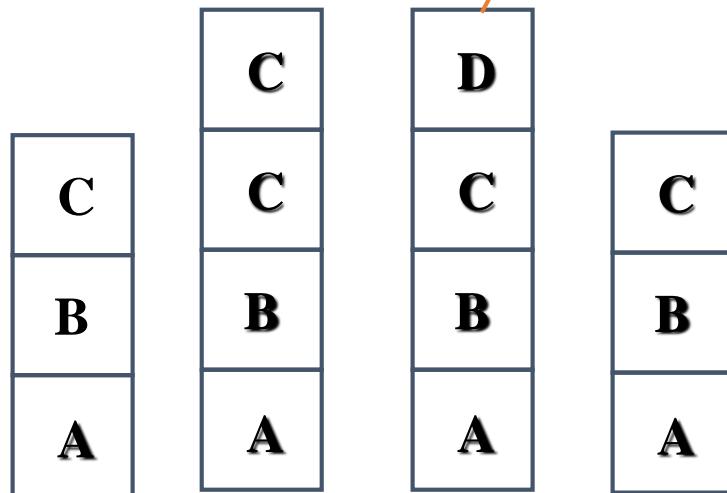
- `glPushMatrix()` : used to save current stack
- `glPopMatrix()` : used to restore previous stack



Matrix Stacks

glPushMatrix()

glPopMatrix()



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

DrawSquare()

glPushMatrix()

glScale3f(2,2,2)

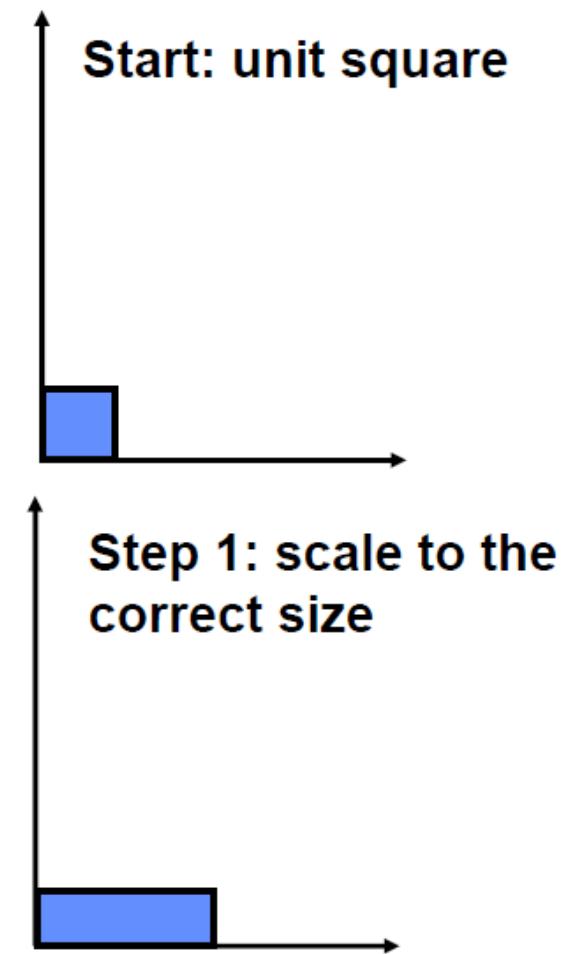
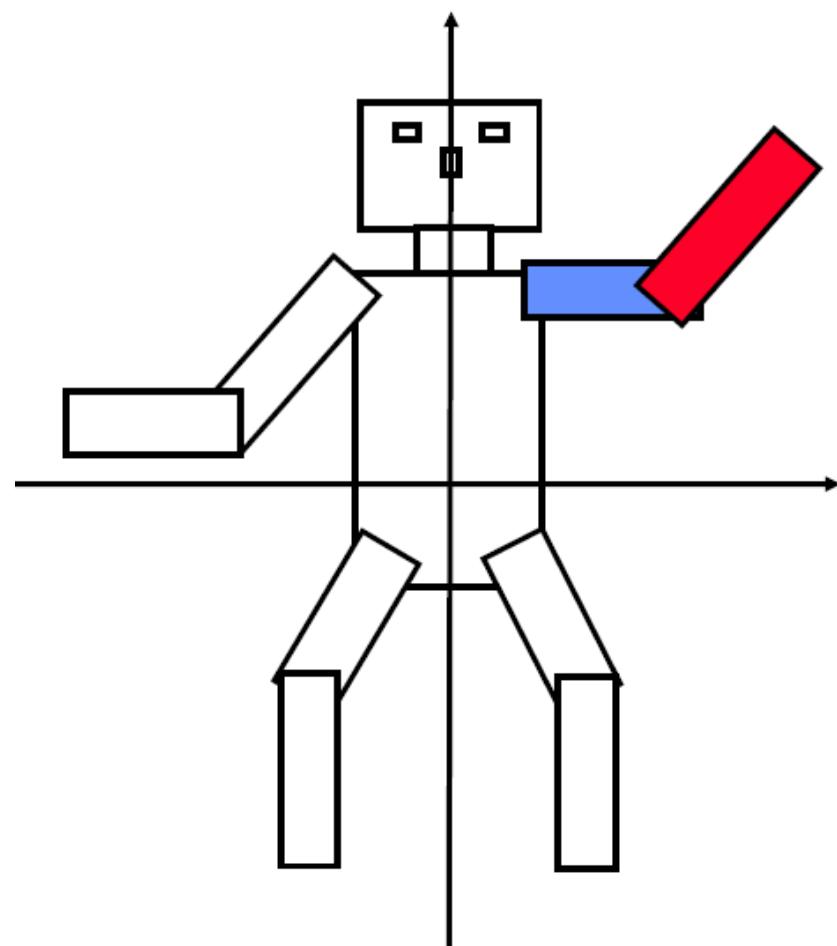
glTranslate3f(1,0,0)

DrawSquare()

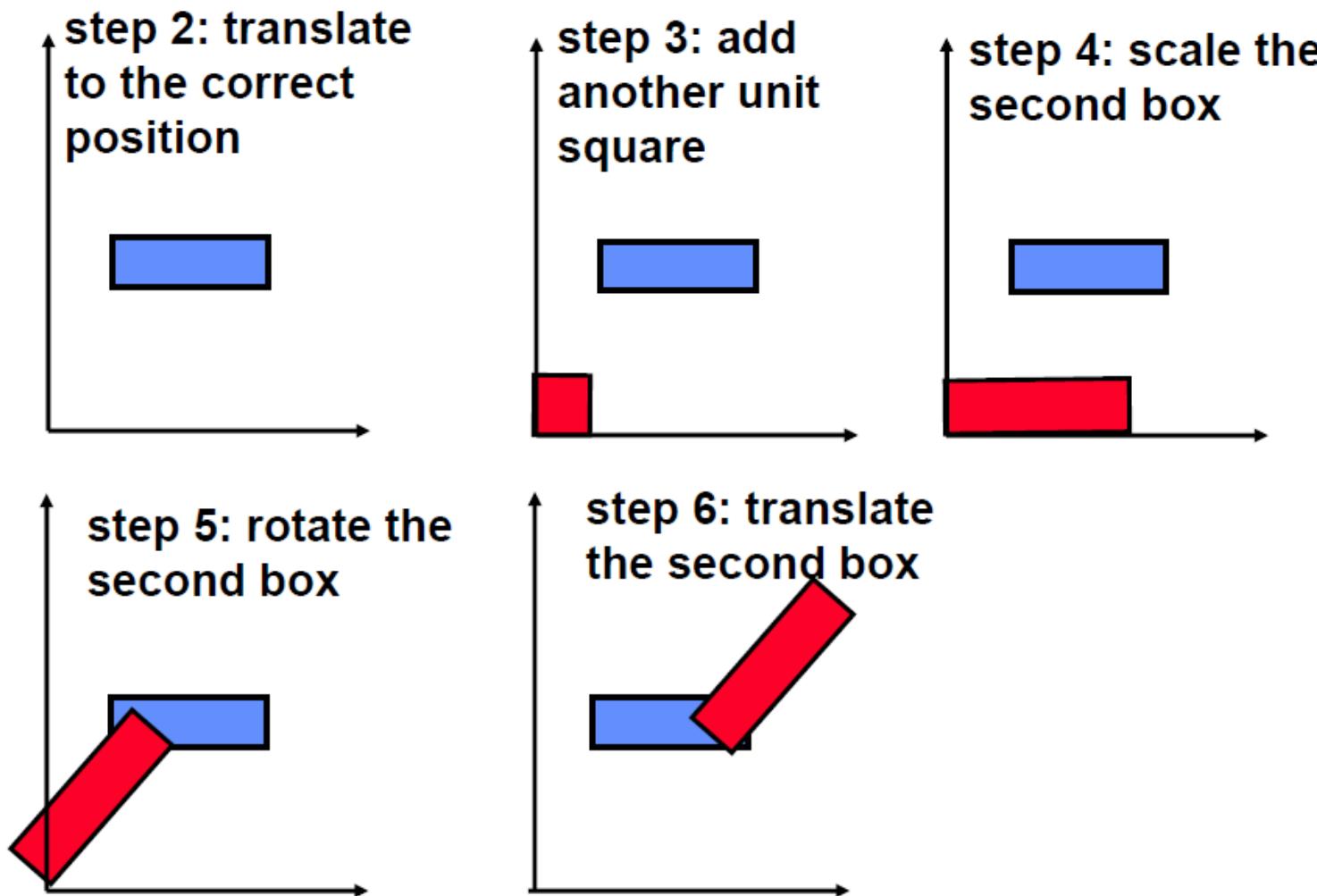
glPopMatrix()



Building the arm



Building the arm



Hierarchical Transformations

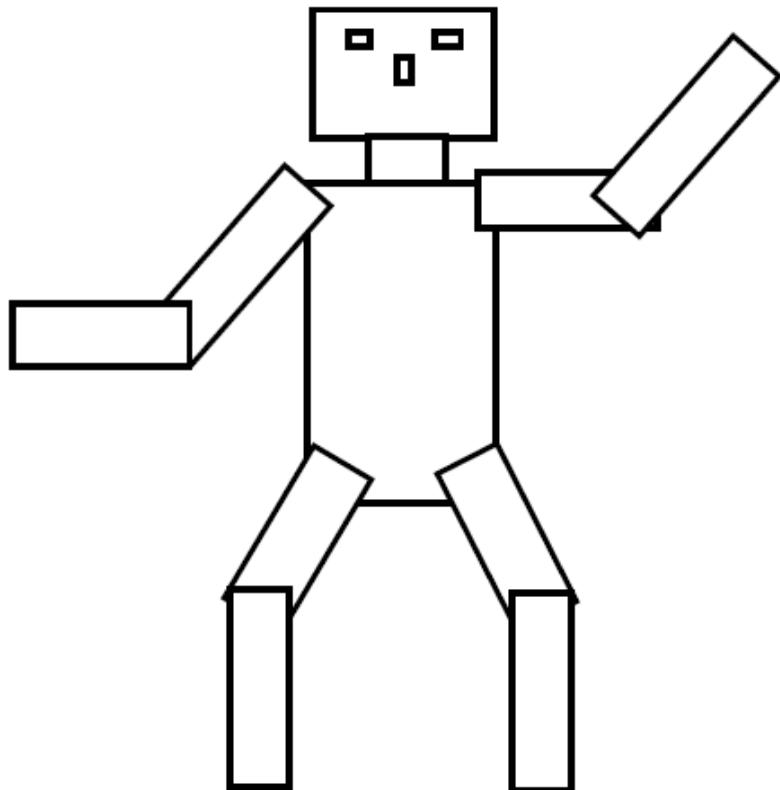
- Positioning each part of a complex object separately is difficult
- If we want to move whole complex objects consisting of many parts or complex parts of an object (for example, the arm of a robot) then we would have to modify transformations for each part
- solution: build objects hierarchically

Complex models

- can be built in a simple, modular fashion
- can be stored efficiently
- can be updated simply



Hierarchical Transformations

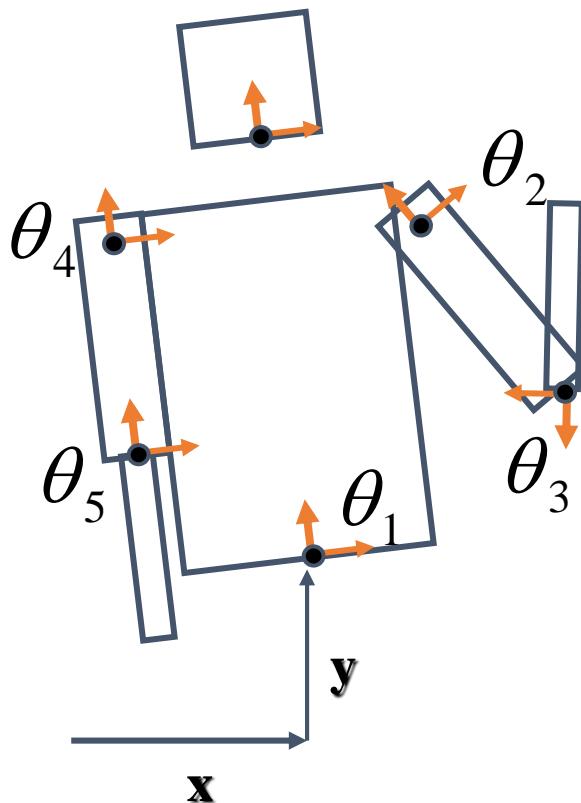
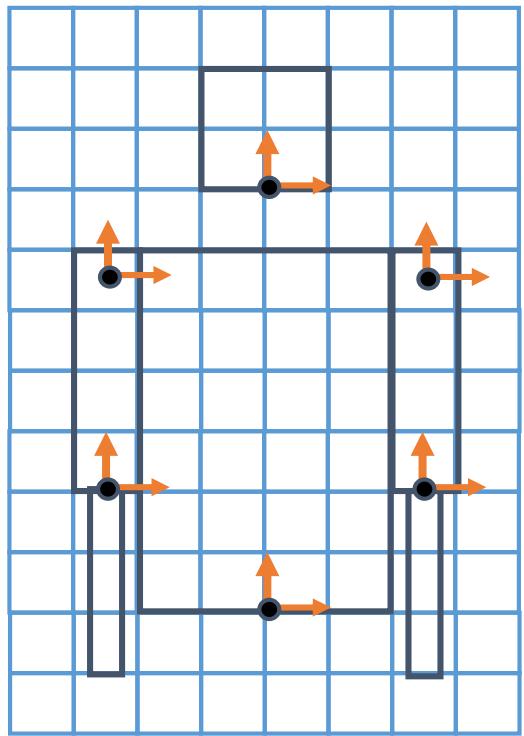


Idea: group parts hierarchically,
associate transforms with each
group.

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



Transformation Hierarchy Example



```
glTranslate3f(x,y,0);  
glRotatef( $\theta_1$ ,0,0,1);  
DrawBody();  
glPushMatrix();  
glTranslate3f(0,7,0);  
DrawHead();  
glPopMatrix();  
glPushMatrix();  
glTranslate(2.5,5.5,0);  
glRotatef( $\theta_2$ ,0,0,1);  
DrawUArm();  
glTranslate(0,-3.5,0);  
glRotatef( $\theta_3$ ,0,0,1);  
DrawLArm();  
glPopMatrix();  
... (draw other arm)
```

Programming Transformations(可编程管线)

- 对于可编程管线，变换的实现依赖于渲染管线中的顶点着色器。
- OpenGL需要对变换所需的Model、View以及Projection矩阵进行传入，从而在顶点着色器中实现从局部坐标系->世界坐标系->摄影坐标系->投影坐标系的坐标变换。
- OpenGL没有自带任何的矩阵和向量知识，所以必须定义自己的数学类和函数。使用已经实现的数学库是个不错的选择，它可以帮助我们抽象所有的数学细节。幸运的是，有个易于使用，专门为OpenGL量身定做的数学库，那就是GLM。



GLM(可编程管线)

- GLM是OpenGL Mathematics的缩写，它是一个只有头文件的库，只需包含对应的头文件即可，不用链接和编译。GLM可以在它们的网站上下载，把头文件的根目录复制到相应的includes文件夹就可以使用这个库了。
- Task：把一个向量 $(1, 0, 0)$ 位移 $(1, 1, 0)$ 个单位



```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);  
// 译注：下面就是矩阵初始化的一个例子，如果使用的是0.9.9及以上版本  
// 下面这行代码就需要改为：  
// glm::mat4 trans = glm::mat4(1.0f)  
// 之后将不再进行提示  
glm::mat4 trans; 单位矩阵  
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));  
vec = trans * vec;
```

Task：沿z轴逆时针旋转90度，然后缩放0.5倍

```
glm::mat4 trans;  
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));  
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```



如何把矩阵传递给着色器(可编程管线)

- 修改顶点着色器让其接收一个mat4的uniform变量，然后再用矩阵uniform乘以位置向量
- 顶点着色器：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 transform;

void main()
{
    gl_Position = transform * vec4(aPos, 1.0f);
    TexCoord = vec2(aTexCoord.x, 1.0 - aTexCoord.y);
}
```

- 传递给着色器：

```
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```



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 $\mathbf{q} = (q_1, q_2, q_3)$. The operations among quaternions are based on the use of three “complex” numbers i , j , and k with the properties $i^2 = j^2 = k^2 = ijk = -1$.

These numbers are analogous to the unit vectors in three dimensions, and we can write $\mathbf{q} = q_0\mathbf{i} + q_1\mathbf{j} + q_2\mathbf{k}$.



Quaternions

- 一个四元数（Quaternion）描述了一个旋转轴和一个旋转角度：
1. 四元数由4个数组成，旋转矩阵需要9个数,相比矩阵更加轻量
 2. 两个四元数之间更容易插值
 3. 四元数、矩阵在多次运算后会积攒误差，需要分别对其做规范化(normalize)和正交化(orthogonalize)，对四元数规范化更容易
 4. 与旋转矩阵类似，两个四元组相乘可表示两次旋转
 5. 解决万向节死锁（Gimbal Lock）问题
 6. 四元数无论是求逆、串联等操作，相比矩阵更加高效

所以综合考虑，现在主流游戏或动画引擎都会以缩放向量+旋转四元数+平移向量的形式进行存储角色的运动数据。



Quaternions

- “Quaternions For Computer Graphics”
- <http://www.cs.ucr.edu/~vzbz/resources/quatut.pdf>
- <https://zhuanlan.zhihu.com/p/27471300?from=singlmessage&isappinstalled=1>

