Computer Graphics

Transformation

Teacher: Dr. Zhuo SU (苏卓)

E-mail: suzhuo3@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
  - Point
  - Scalar
  - Vector

Coordinate-independent Geometry

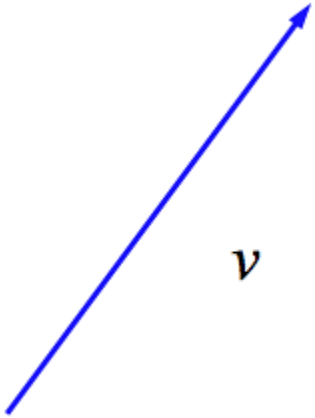
- In elementary geometry (初等几何), we use Cartesian
  - Position of a point in space p = (x, y, z)
  - Use algebraic operations on the coordinates to get the result
- This method is not based on physics
  - In physics, the existence of points are irrelevant to their position
  - Most geometric results are not based on positions
  - Euclidean geometry: congruent triangle (全等三角形)

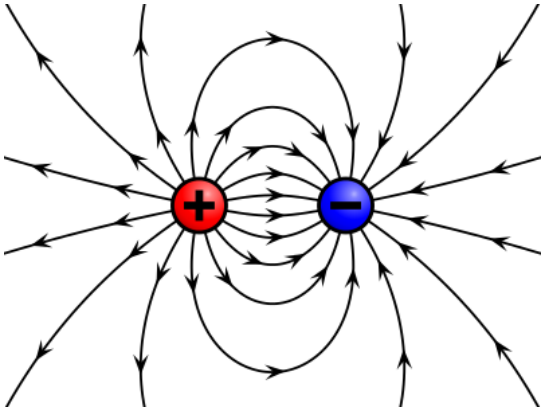
Scalar (标量)

- **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 (矢量)

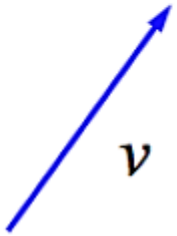
- Definition: vector is a line having the two properties
  - Direction
  - Length: $|v|$
- Examples:
  - Power
  - Speed
  - Directed line segment

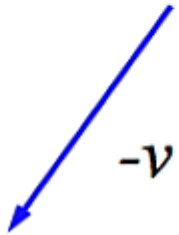


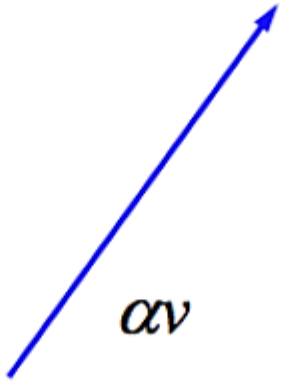


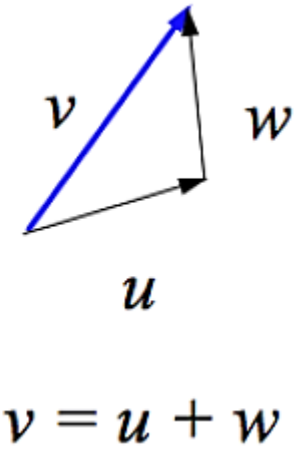
Vector operations

- **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







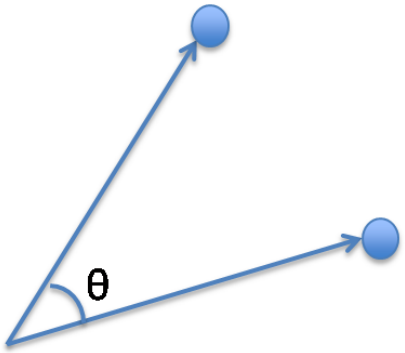


Inner and outer product

- **Inner product:** $\mathbf{u} \bullet \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \cos\theta$
  - $u \cdot v = 0 \Leftrightarrow u \perp v$

- **Outer product: u x v is a vector, its length is $|u|\,|v| \sin\theta$, its direction is vertical to the plane that contains u and v**
  - u x v = 0 means u is parallel to v

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$



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

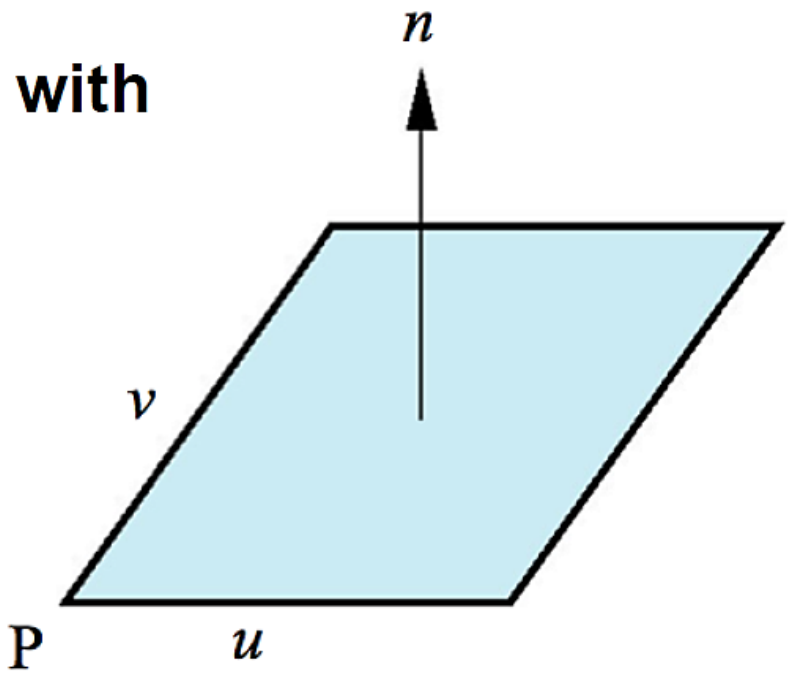
Normals

- 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

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



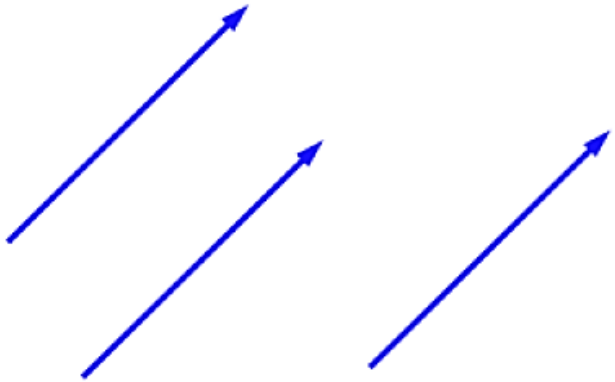
Linear space

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

简单的说，线性空间是这样一种集合，其中任意两元素相加可构成此集合内的另一元素，任意元素与任意数（可以是实数也可以是复数，也可以是任意给定域中的元素）相乘后得到此集合内的另一元素。

Vectors have no positions

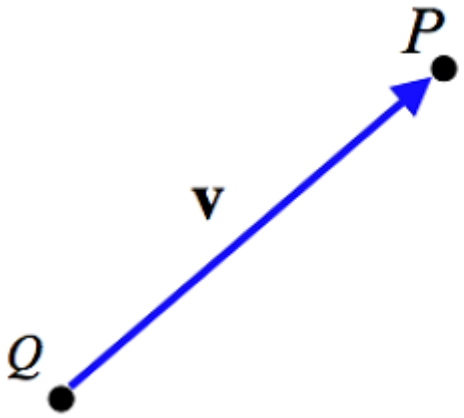
- **The following vectors are equal**
  - As they have same length and direction



- **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

- Affine Space (仿射空间) constructed by points and vectors
- Operational:
  - Vector + Vector = Vector
  - Scalar x Vector = Vector
  - Point + Vector = Point
  - Scalar + Scalar = Scalar

从基本数学概念上来说，一个坐标系对应了一个仿射空间 (Affine Space)，当矢量从一个坐标系变换到另一个坐标系时要进行线性变换 (Linear Transformation)。

Linear combination

- Given n vectors v_1, v_2, …, v_n and n scalar a_1, a_2, …, a_n, 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

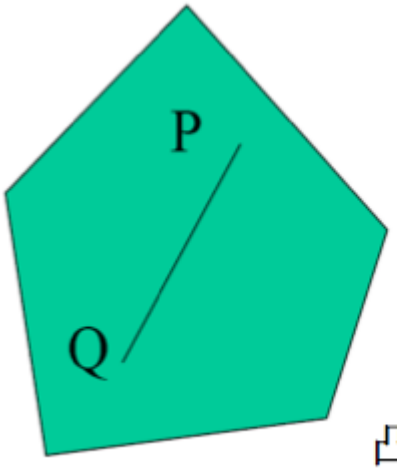
Linear combination of points

- **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

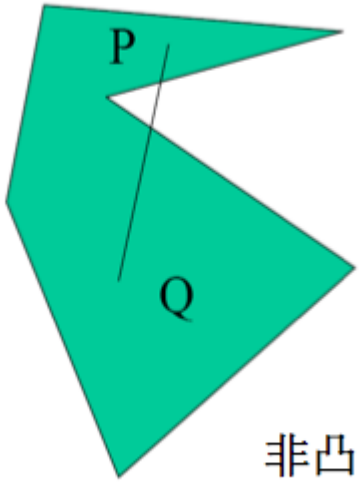
- **Combination coefficients have limitations**
  - When $\alpha_1 + \alpha_2 + \ldots + \alpha_n = 1$, linear combination of points is a point
  - $\frac{1}{2}\mathbf{P}_1 + \frac{1}{2}\mathbf{P}_2 = \mathbf{P}_1 + \frac{1}{2}(\mathbf{P}_2 - \mathbf{P}_1)$ = point + vector = point

Convex

- An object is convex if and only if the line segment connecting any two points on the object is also within the object



凸



非凸

Affine convex combination

- **Consider:**

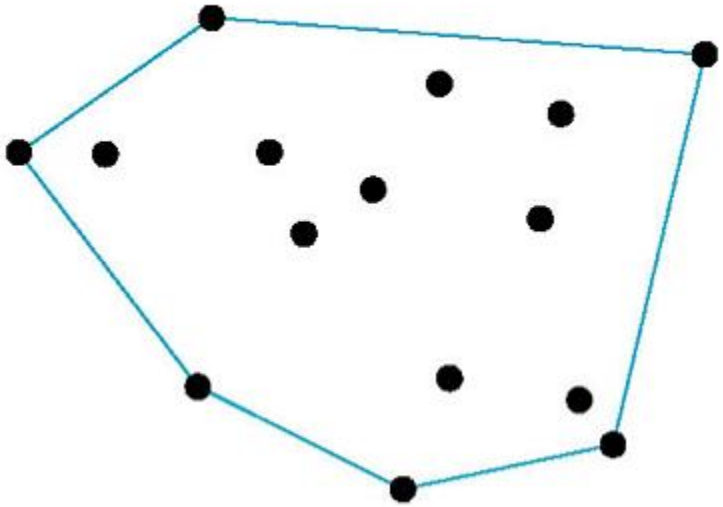
$$\mathbf{P} = \alpha_1\mathbf{P}_1 + \alpha_2\mathbf{P}_2 + \ldots + \alpha_n\mathbf{P}_n$$

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

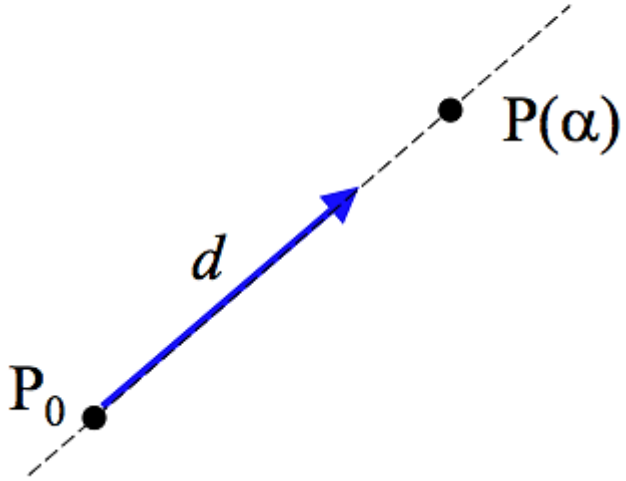
- The minimum convex contains P_1, P_2, …, P_n
- Can use the “Shrink” method to get it



Line

- All points comply with the following form

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



Parametric form

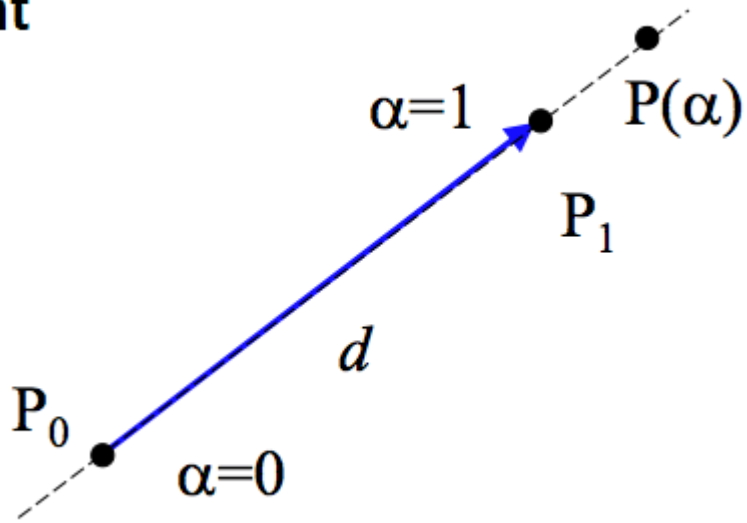
- **It is the parametric form definition for line**
  - More general and stable
  - Can be used in curves and surfaces
- **Two-dimensional form**
  - Explicit: y = mx + h
  - Implicit: ax + by + c = 0
  - Parametric: $x(\alpha) = x_0 + (1 - \alpha)\, x_1$
    $y(\alpha) = y_0 + (1 - \alpha)\, y_1$

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**

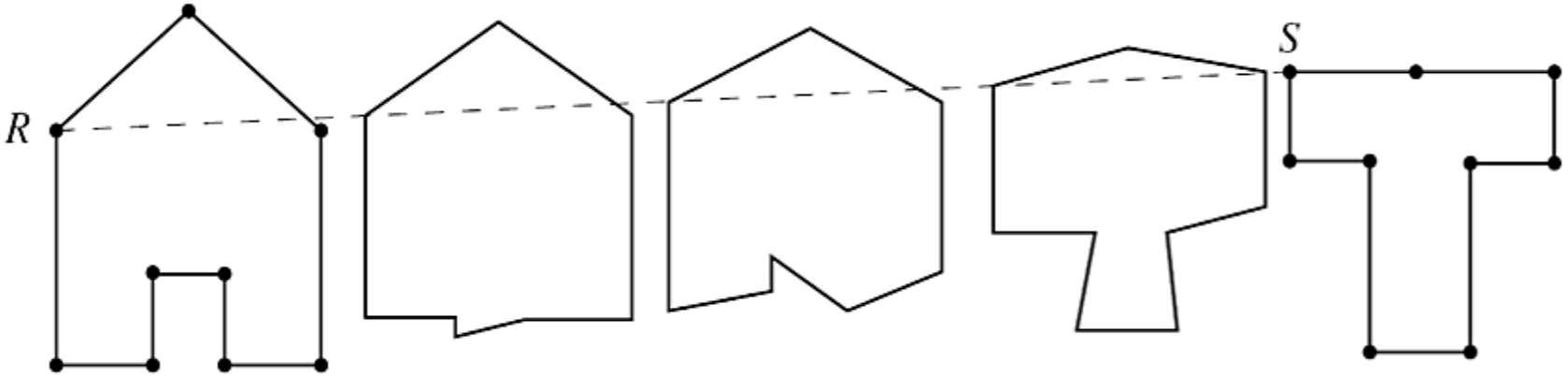
$$\mathbf{P}(t) = (1 - t)\,\mathbf{A} + t\,\mathbf{B}$$

defines a line pass these two points.

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

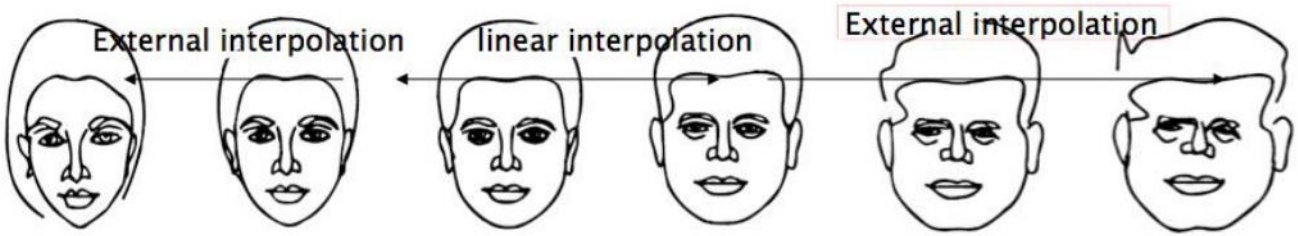
Polygon deformation

- 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

Face Transform



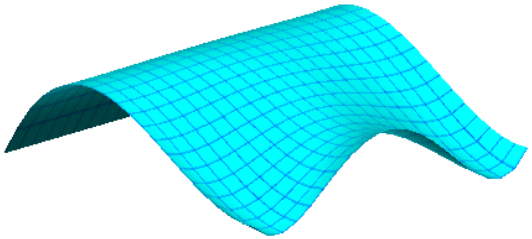
Elizabath Taylor

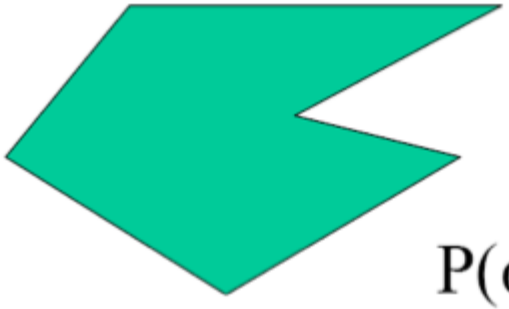
John F. Kennedy

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

$P(\alpha)$

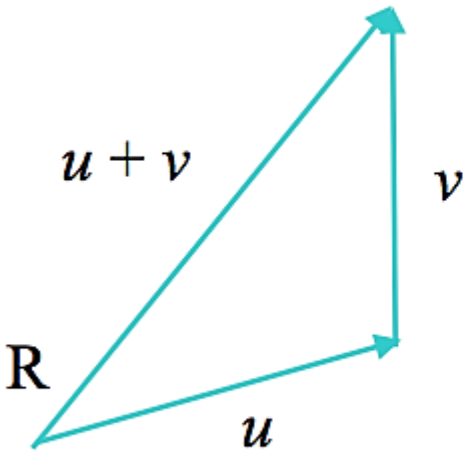




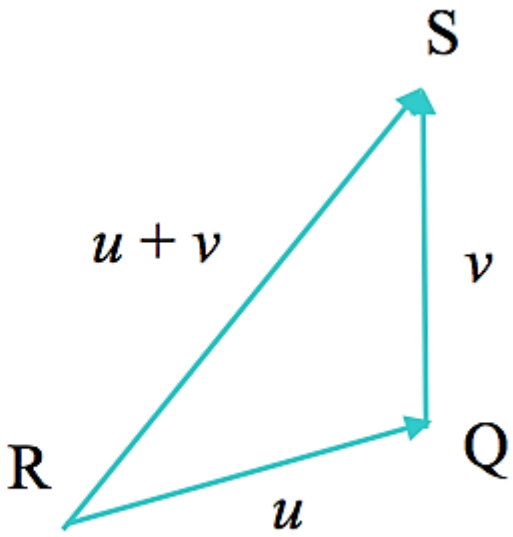
$P(\alpha, \beta)$

Plane

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

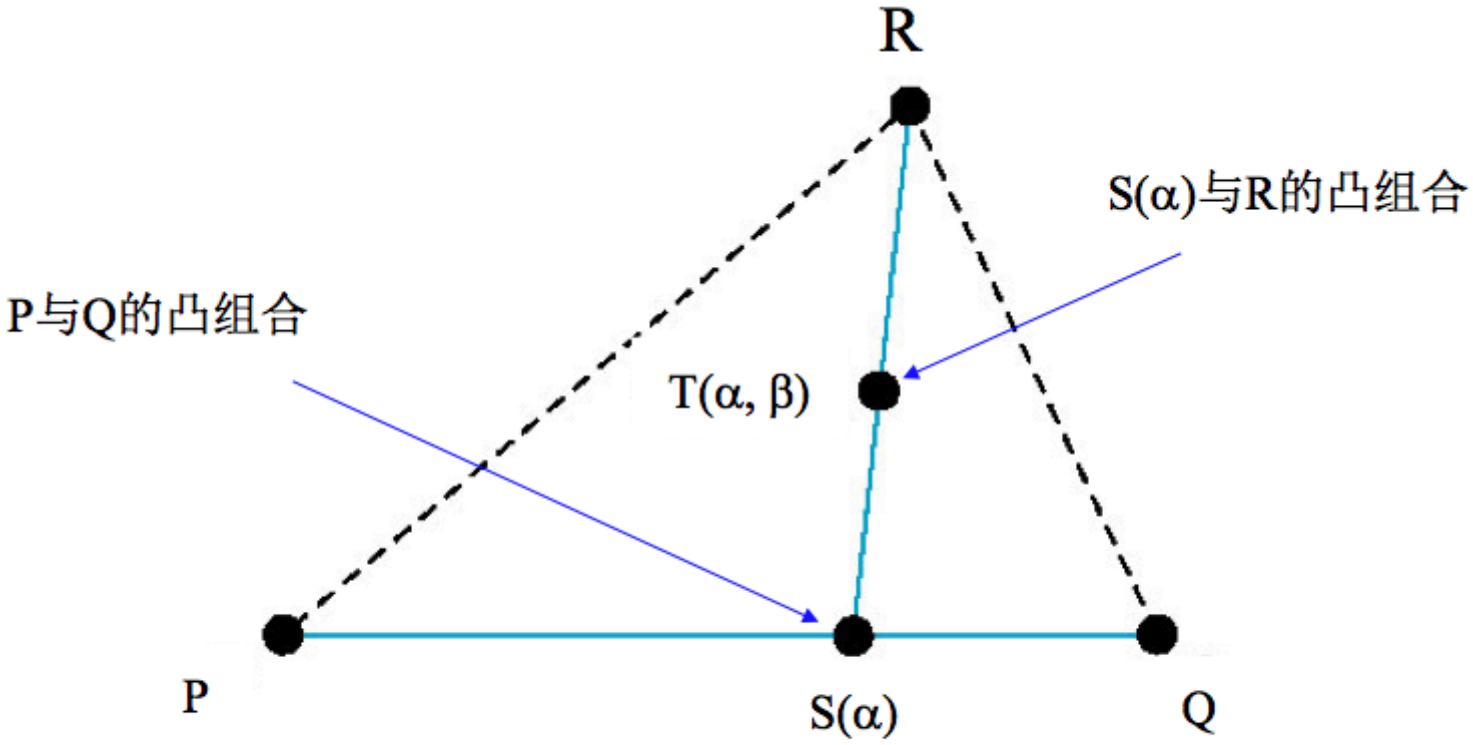


$$P(\alpha, \beta) = R + \alpha u + \beta v$$



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

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**

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

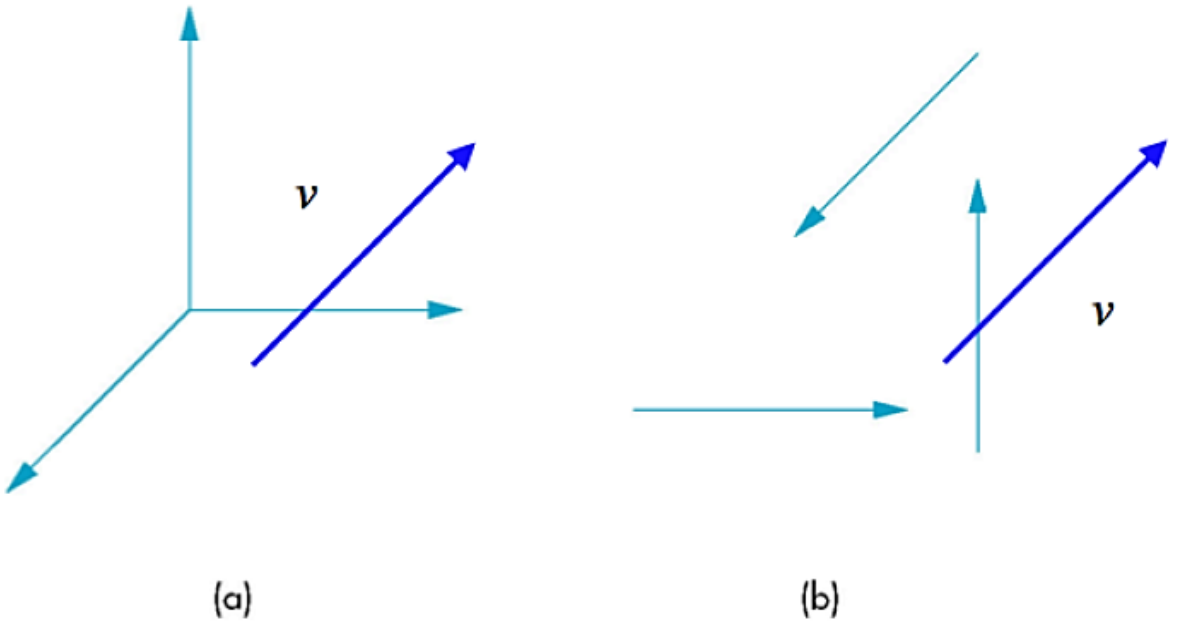
Example

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

- Note that the above statement is relative to a particular basis
- E.g. OpenGL represents a vector with respect to the world coordinate system, it is necessary to transform to the camera coordinate (viewpoint coordinate system) .

Coordinate

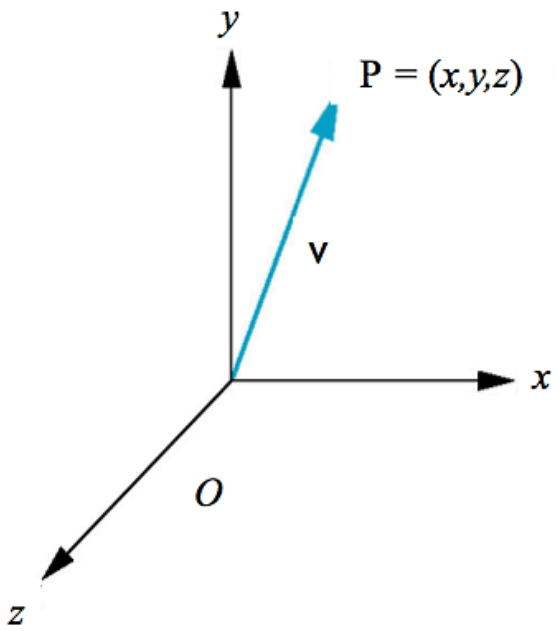
- **Which is right?**



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

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$
$$= [v_1, v_2, \ldots, v_n, O][\alpha_1, \alpha_2, \ldots, \alpha_n, 0]^T$$
$$P = O + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$$
$$= [v_1, v_2, \ldots, v_n, O][\beta_1, \beta_2, \ldots, \beta_n, 1]^T$$

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

$$v = [\alpha_1, \alpha_2, \ldots, \alpha_n, 0]^T$$
$$P = [\beta_1, \beta_2, \ldots, \beta_n, 1]^T$$

Homogeneous coordinate (齐次坐标)

- **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, translation, 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

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

Coordinate transformation

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

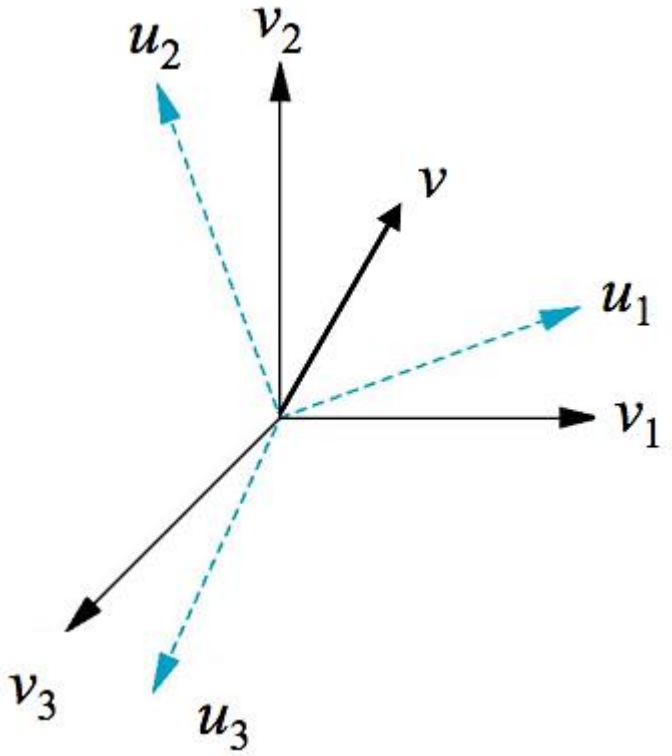
$$a = [\alpha_1, \alpha_2, \alpha_3]^T$$
$$b = [\beta_1, \beta_2, \beta_3]^T$$

- **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**

$$u_1 = \gamma_{11} v_1 + \gamma_{12} v_2 + \gamma_{13} v_3$$
$$u_2 = \gamma_{21} v_1 + \gamma_{22} v_2 + \gamma_{23} v_3$$
$$u_3 = \gamma_{31} v_1 + \gamma_{32} v_2 + \gamma_{33} v_3$$
$$\mathbf{Q}_0 = \gamma_{41} v_1 + \gamma_{42} v_2 + \gamma_{43} v_3 + \mathbf{P}_0$$

- **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**
  - **1st frame:** $a = [\alpha_1, \alpha_2, \alpha_3, \alpha_4]^T$
  - **2nd frame:** $b = [\beta_1, \beta_2, \beta_3, \beta_4]^T$

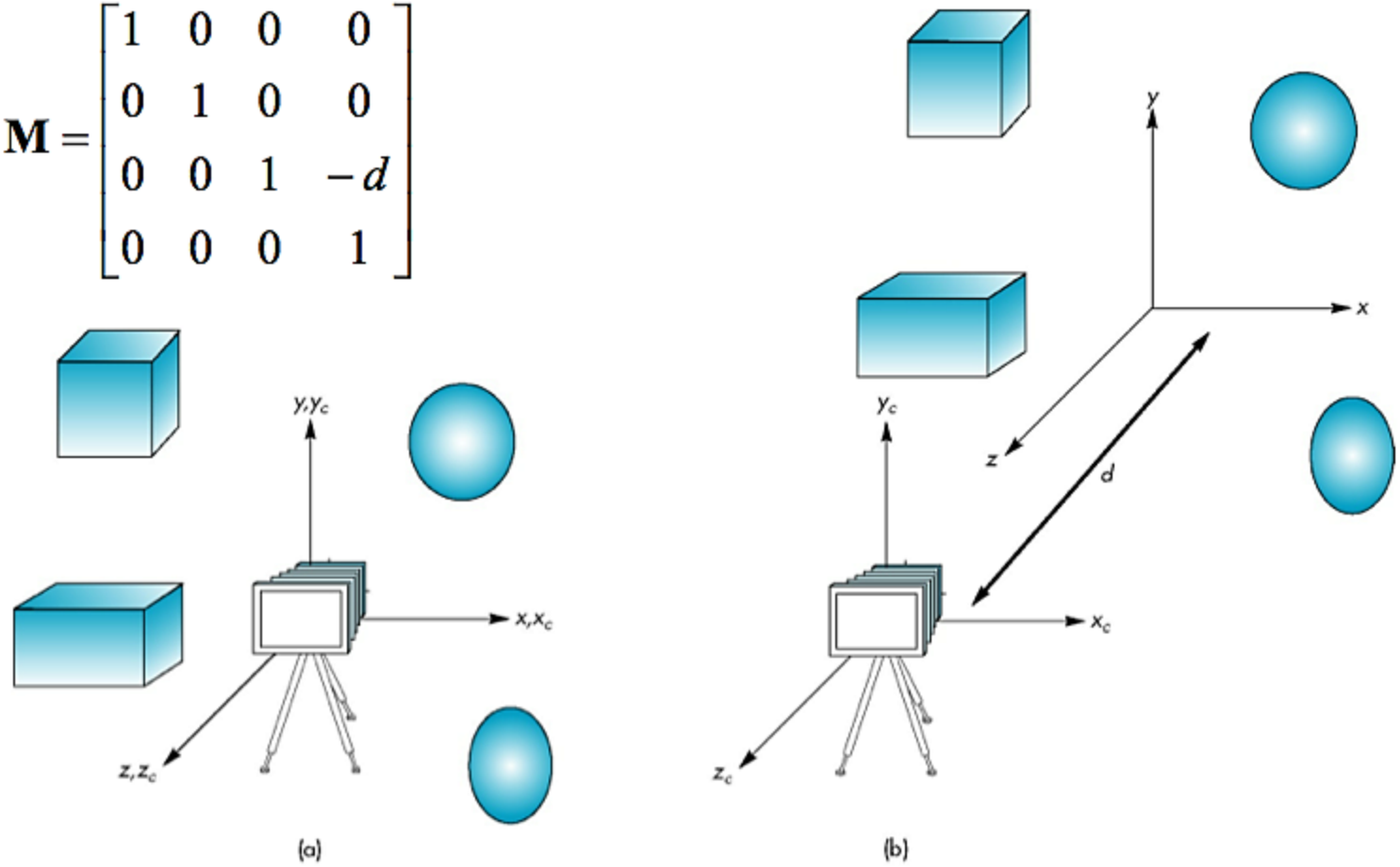
When represents a point $\alpha_4 = \beta_4 = 1$ **,When represents a vector** $\alpha_4 = \beta_4 = 0$ **, and** $a = M^T b$ **,The size of matrix M is 4x4, which defines a affine transformation with homogeneous coordinate.**

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.

Movement of the camera

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

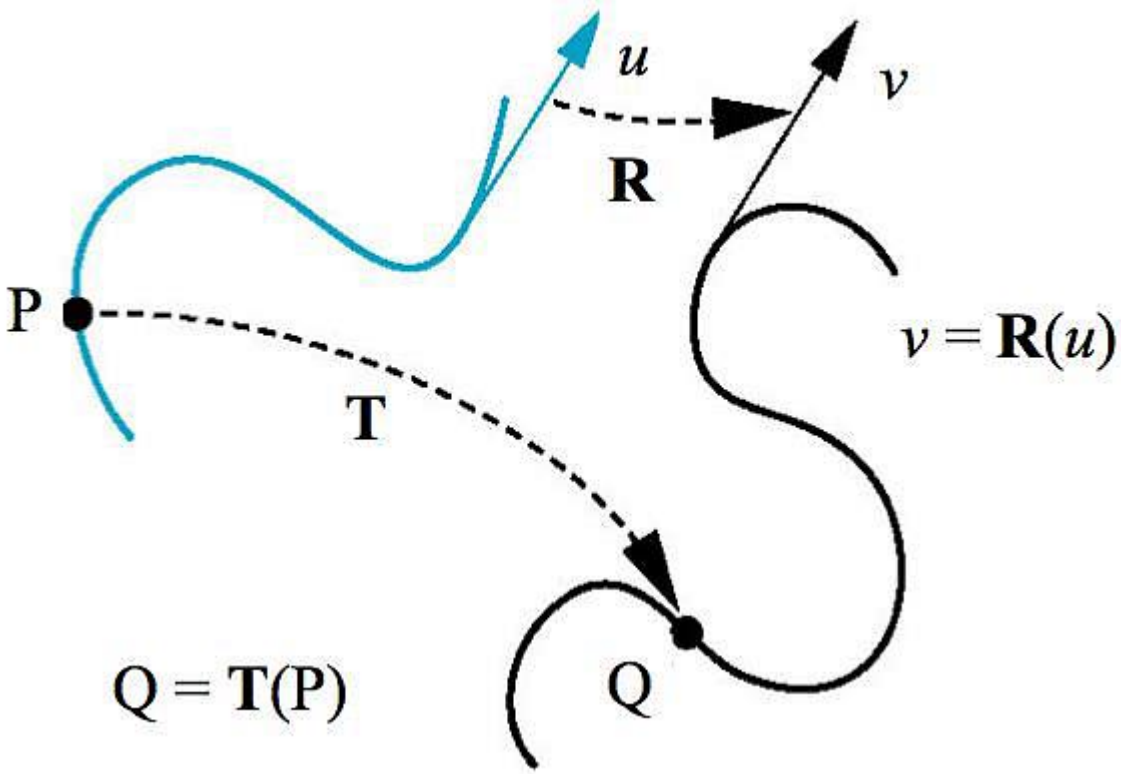


Outline

- 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



Transformation

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

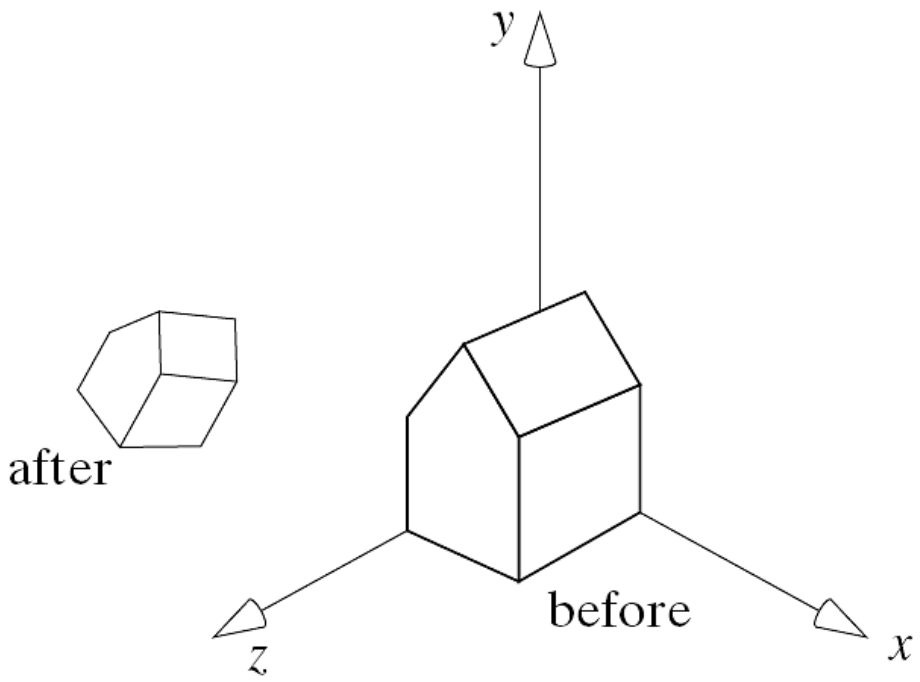
As all objects are eventually represented using points, it is enough to know how to transform points.

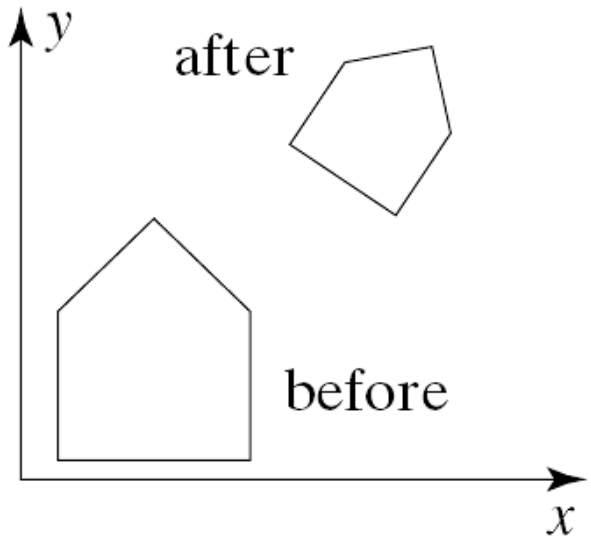
Affine transformation

- Many important physical feature of transformation
  - Rigid transformation: rotation, translation
  - Scaling, shear
- 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 (Maintaining collinearity 共线性)

Why we need transformation ?

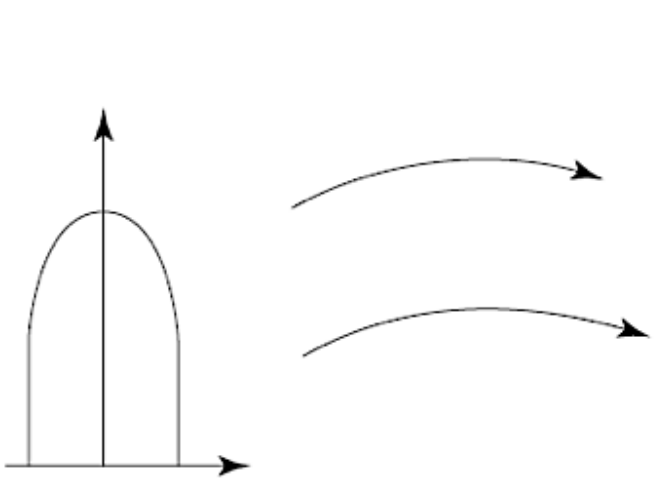
- Procedures to compute new positions of objects
- Used to modify objects or to transform (map) from one coordinate system to another coordinate system

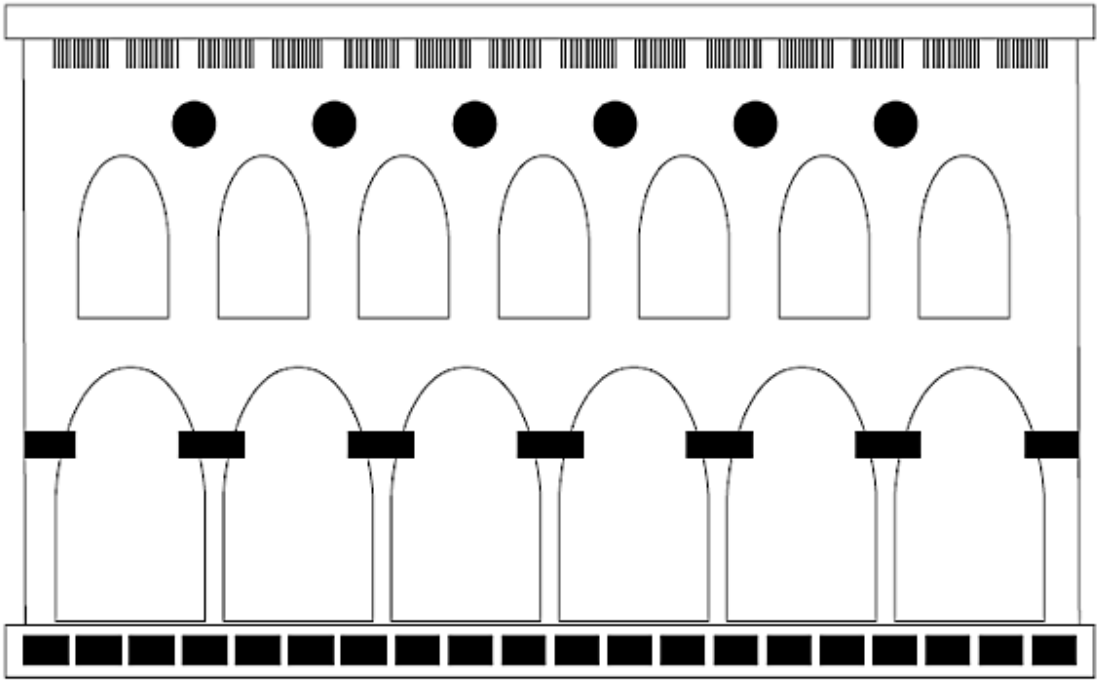




Action 1

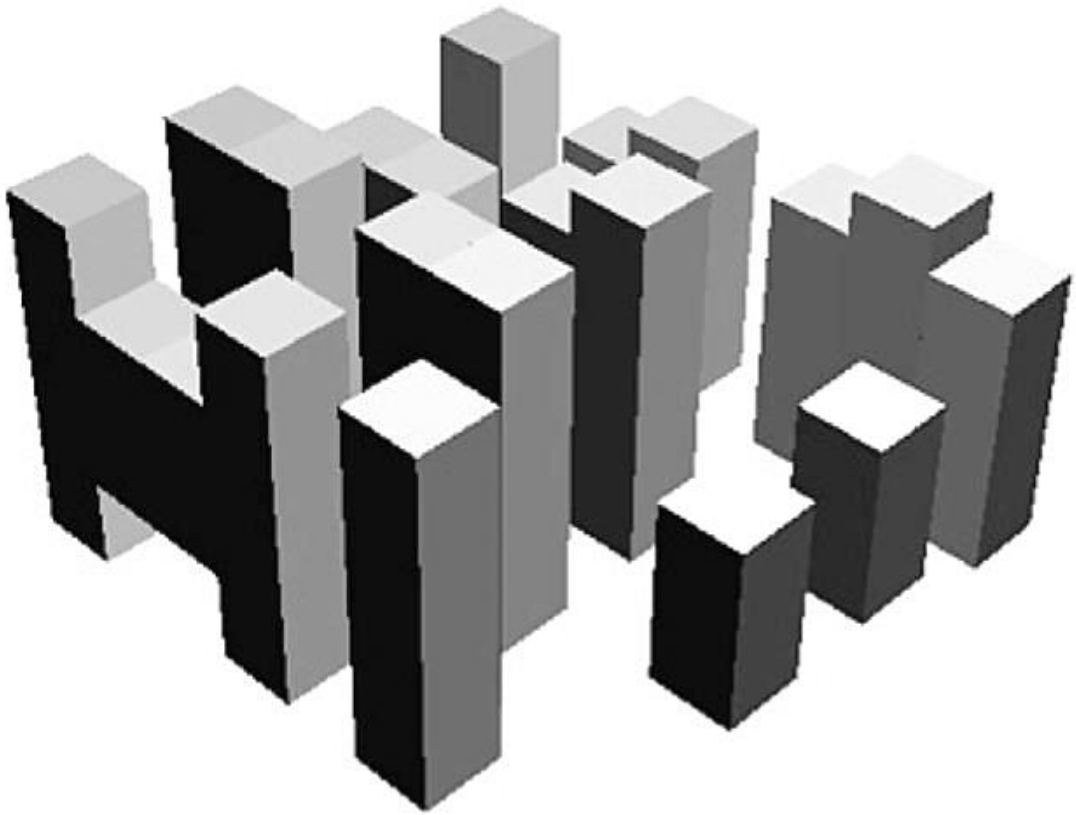
- Construct scenes





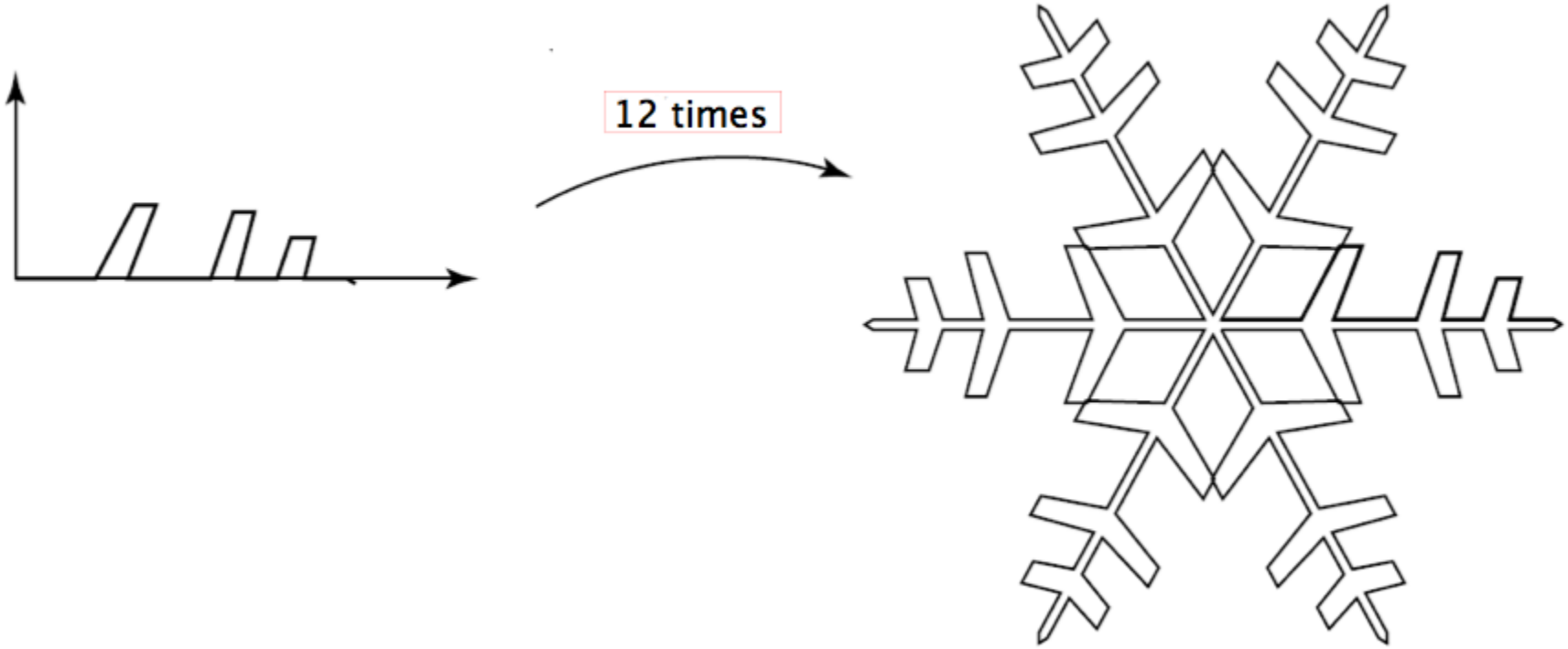
Action 1

- Construct 3D scene



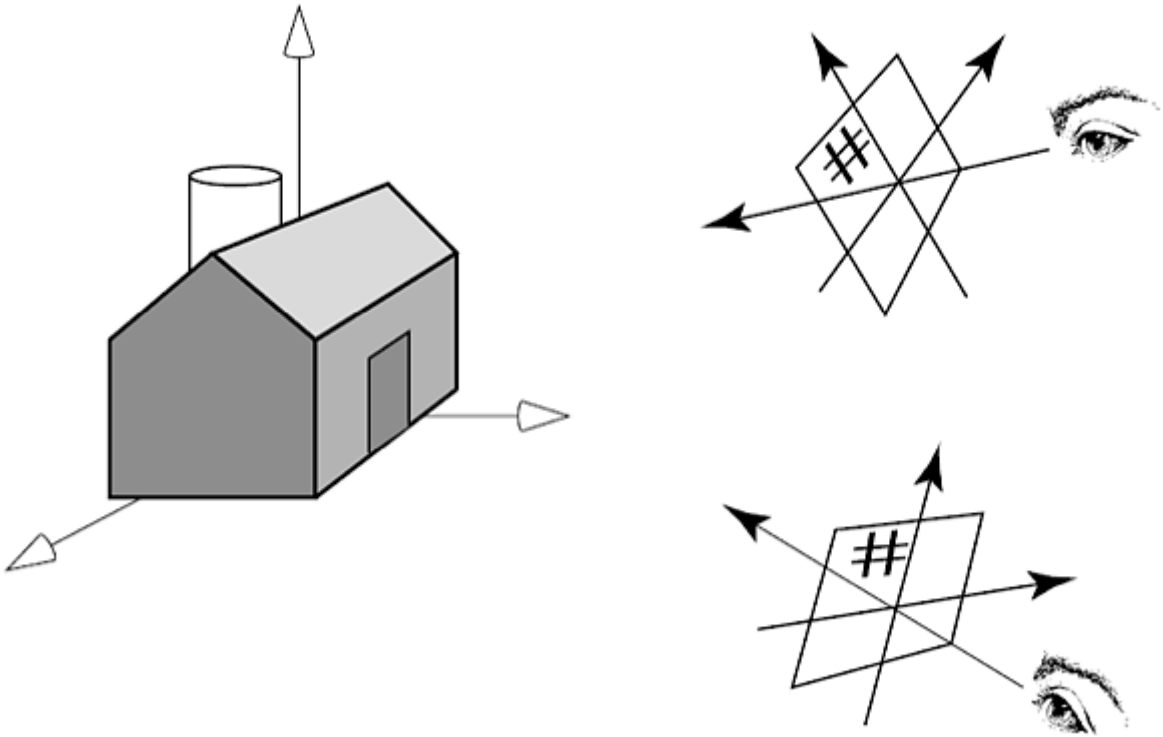
Action 1

- Snowflake structure



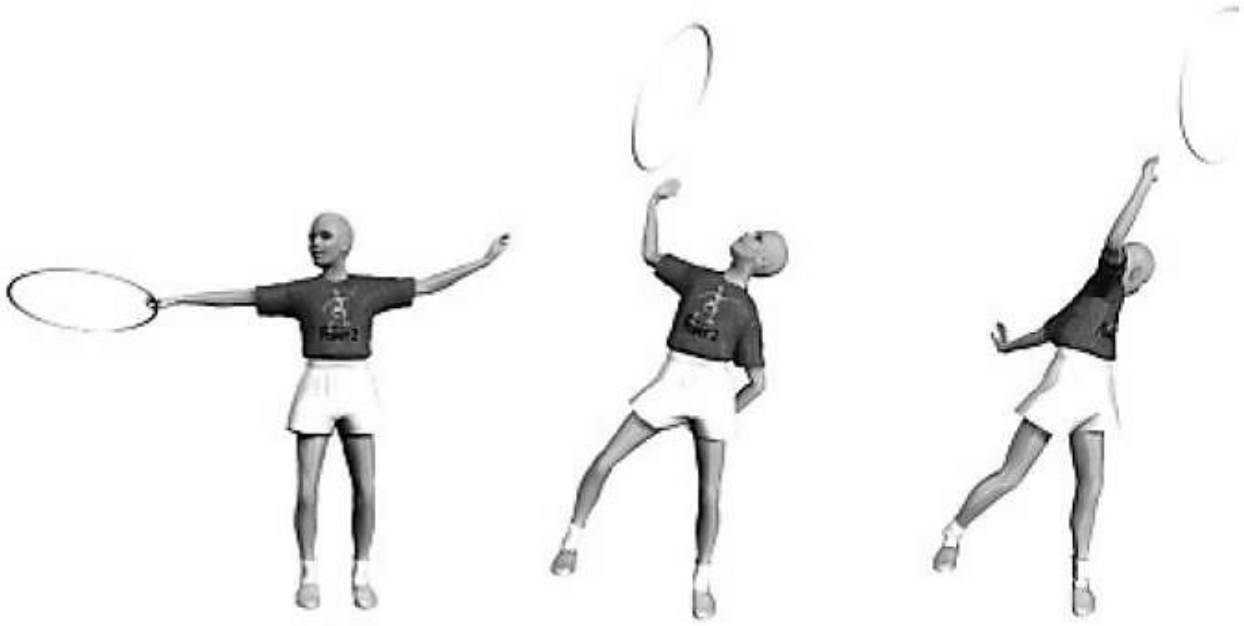
Action 2

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

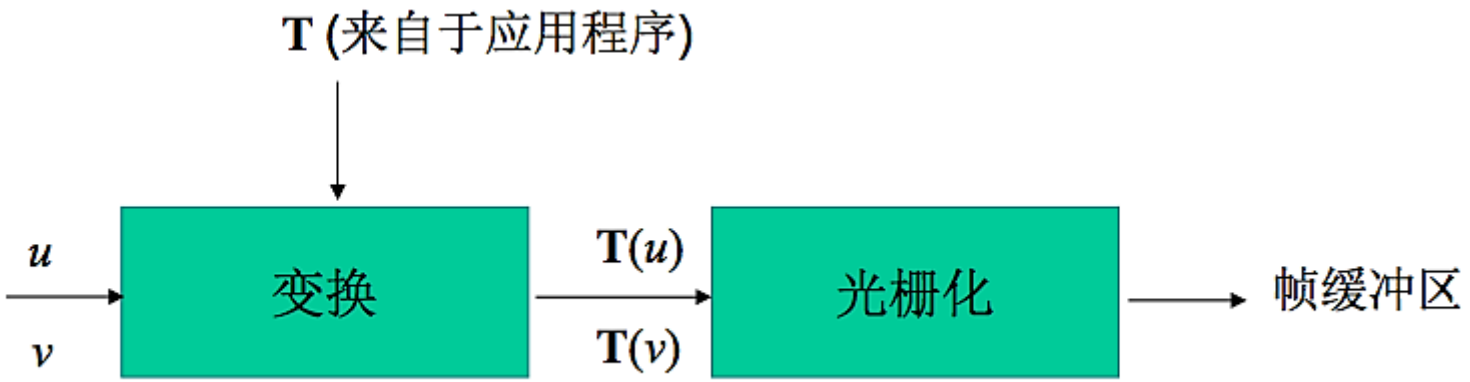


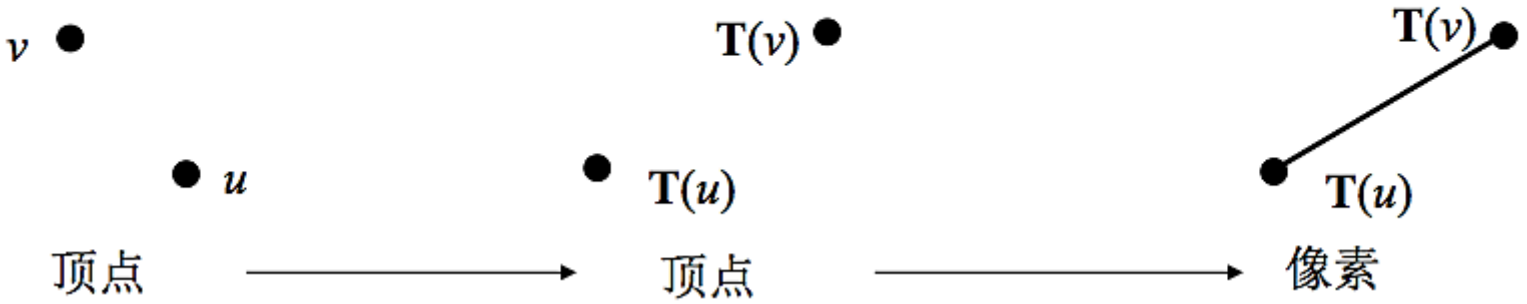
Action 3

- 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.



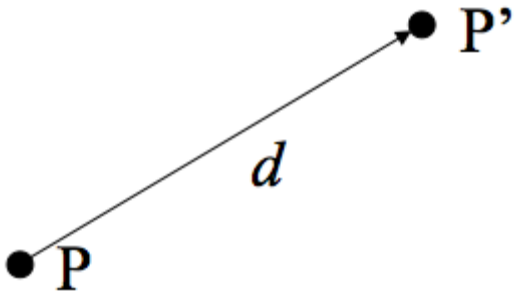
Pipeline





Translation

- Put a point to a new position

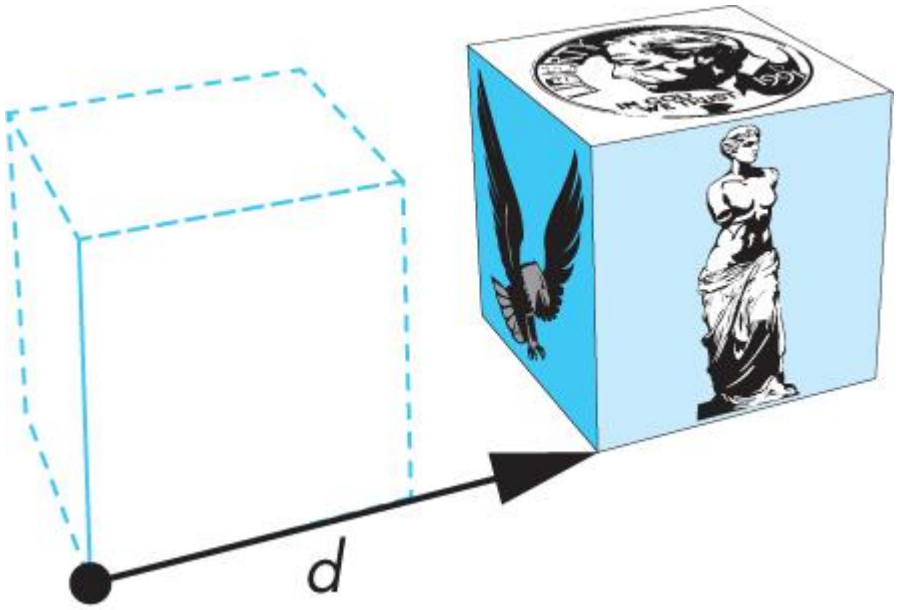


- Determined by a vector d
  - Three free degrees
  - P' = P + d

Translation of objects

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

Before



After

Representation of Translation

- Homogeneous coordinates in a frame

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

- 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

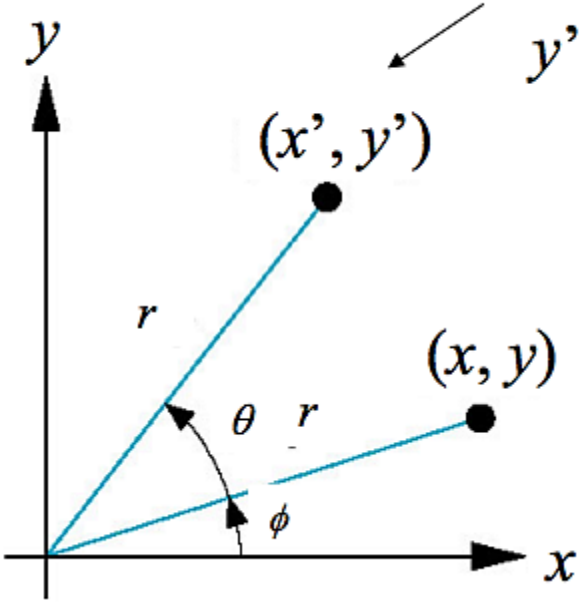
- Using a 4 × 4 homogeneous coordinates matrix **T** to represents the translation
- p’ = **T**p

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

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

2D rotation

- Consider θ degrees rotation about the origin

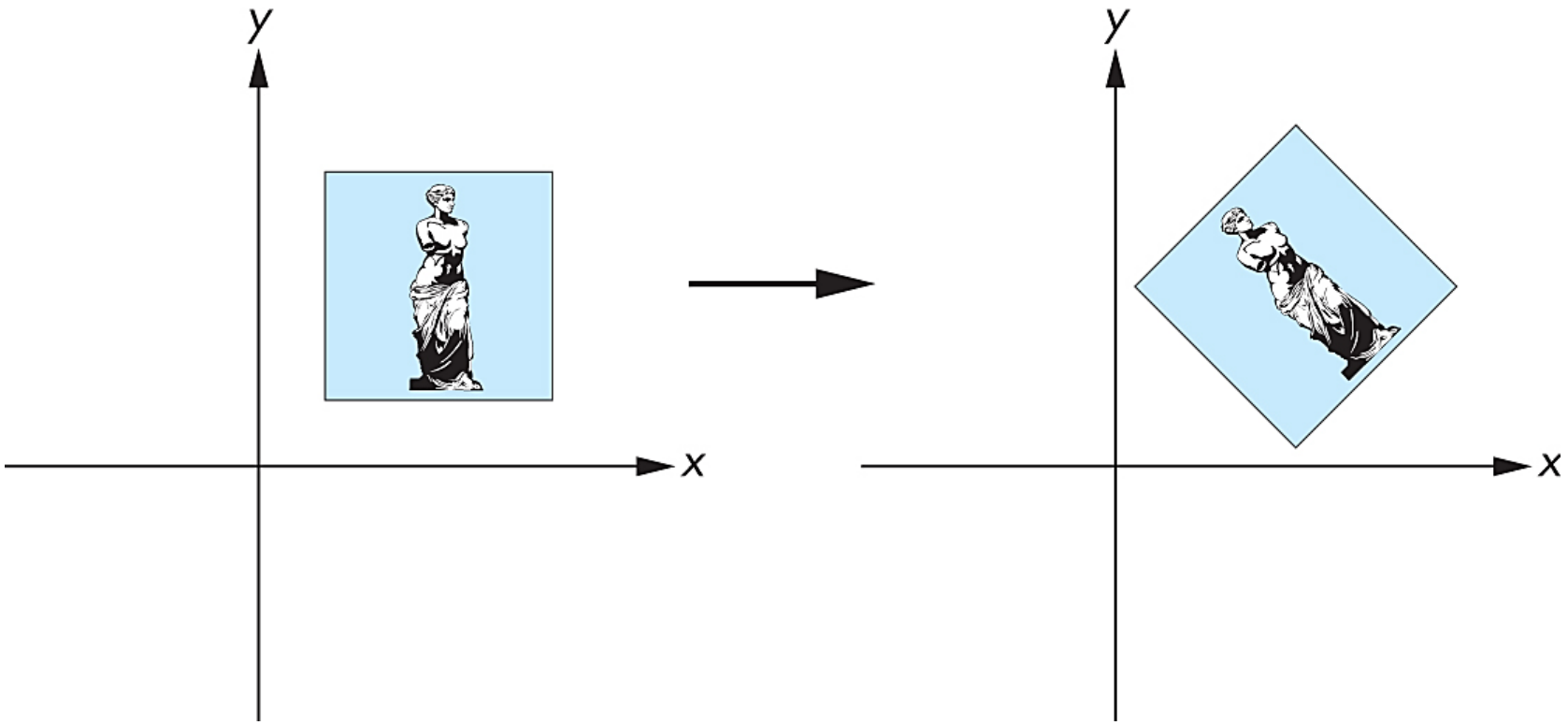


$$x' = r\cos(\phi + \theta)$$
$$y' = r\sin(\phi + \theta)$$

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

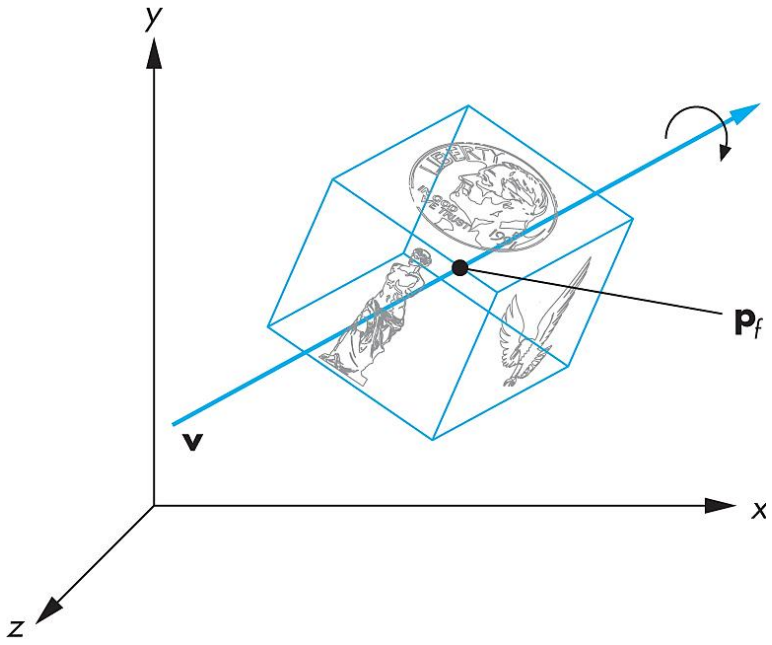
$$x = r\cos\phi$$
$$y = r\sin\phi$$

2D rotation



3D rotation

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



Rotate along z axis

- Z coordinates unchanged

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

- Homogenous coordinate: $\mathbf{p}' = \mathbf{R}_z(\theta)\mathbf{p}$

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

Rotation along x and y axis

- Similar to the rotation along z axis
  - Along x axis, x coordinate unchanged
  - Along y axis, y coordinate unchanged

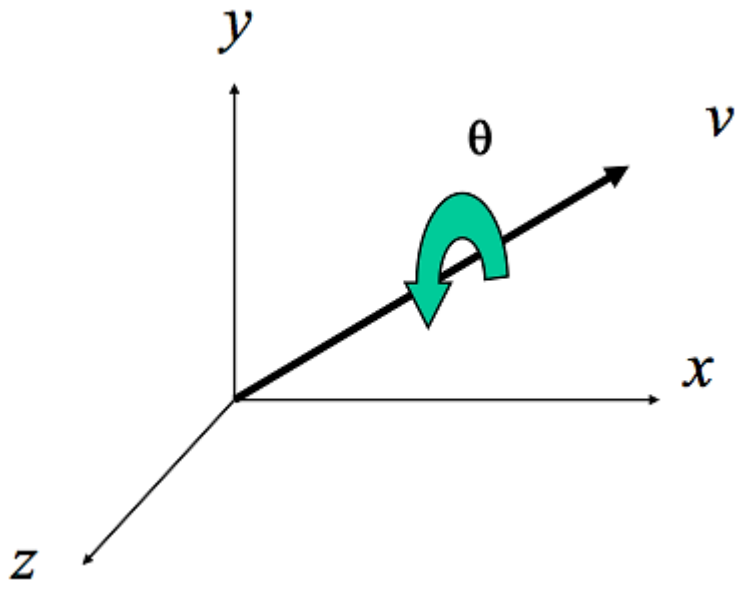
$$\mathbf{R}_x = \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_y = \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 &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

Rotate along the general axis through the origin

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

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

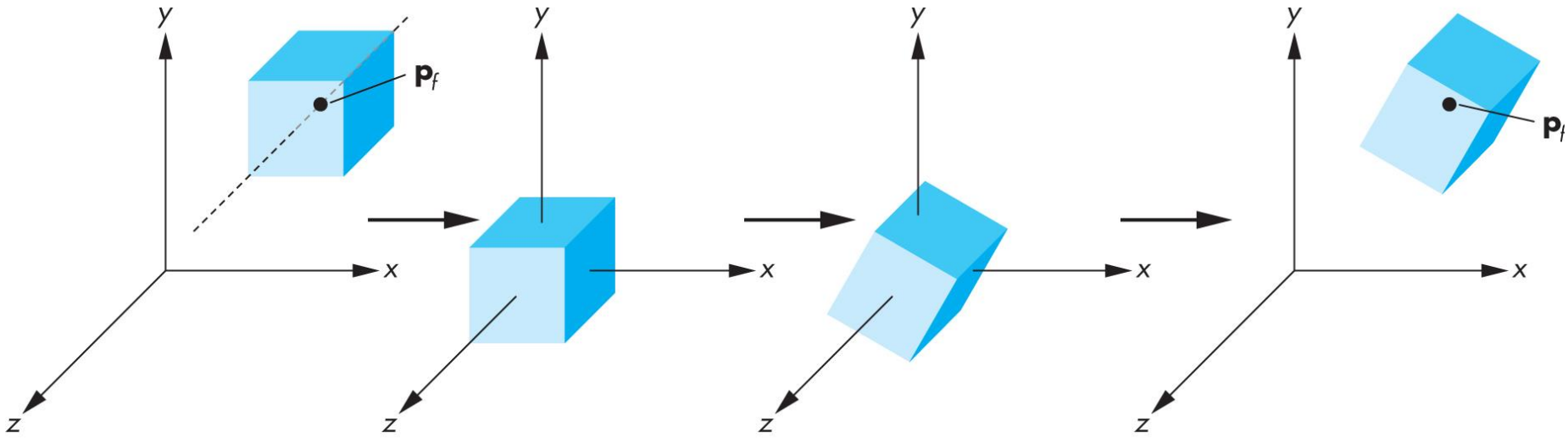
- Note that the rotation order can not be exchanged



Rotate around a fixed point except origin

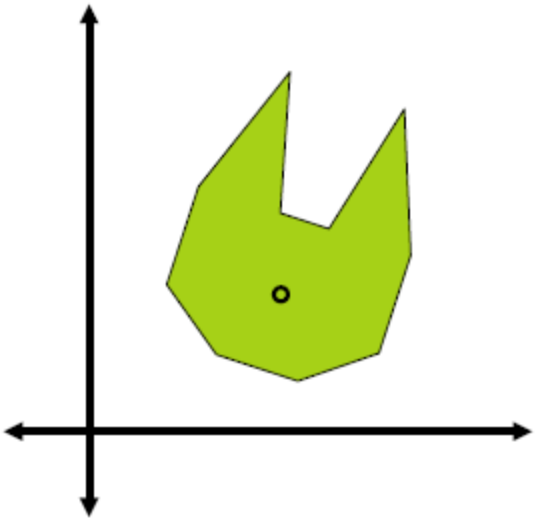
- Move the fixed point to origin
- Rotate
- Move the fixed point back to its initial place

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



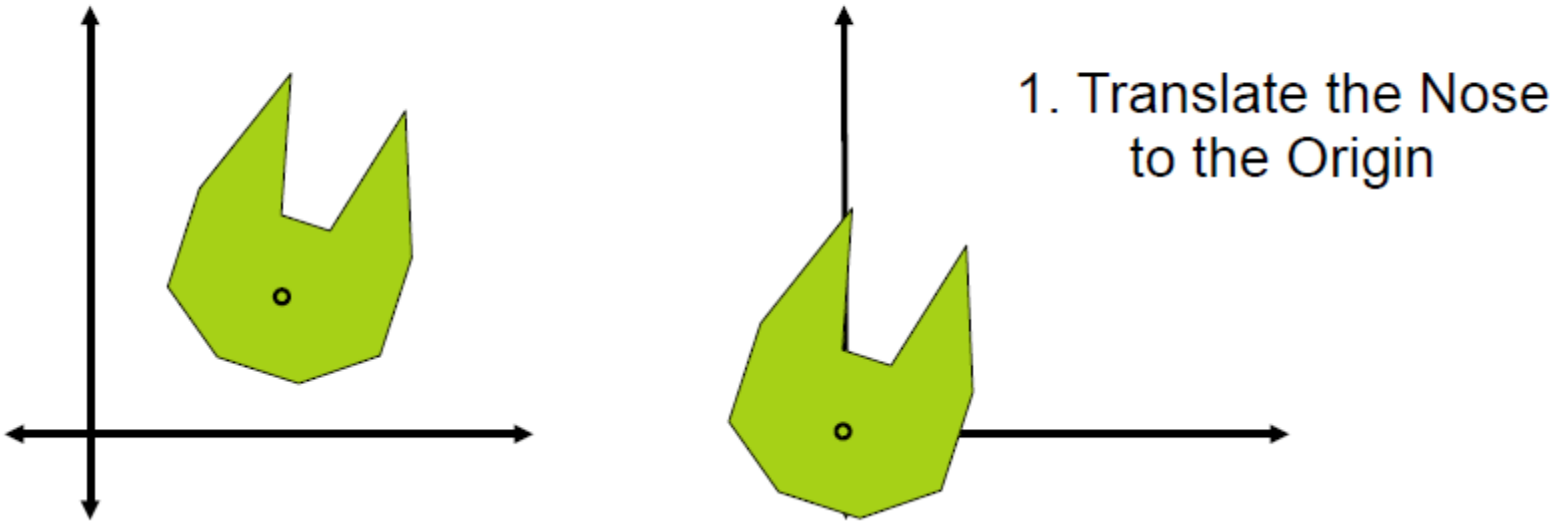
Simple Rotate

To rotate the cat's head about its nose

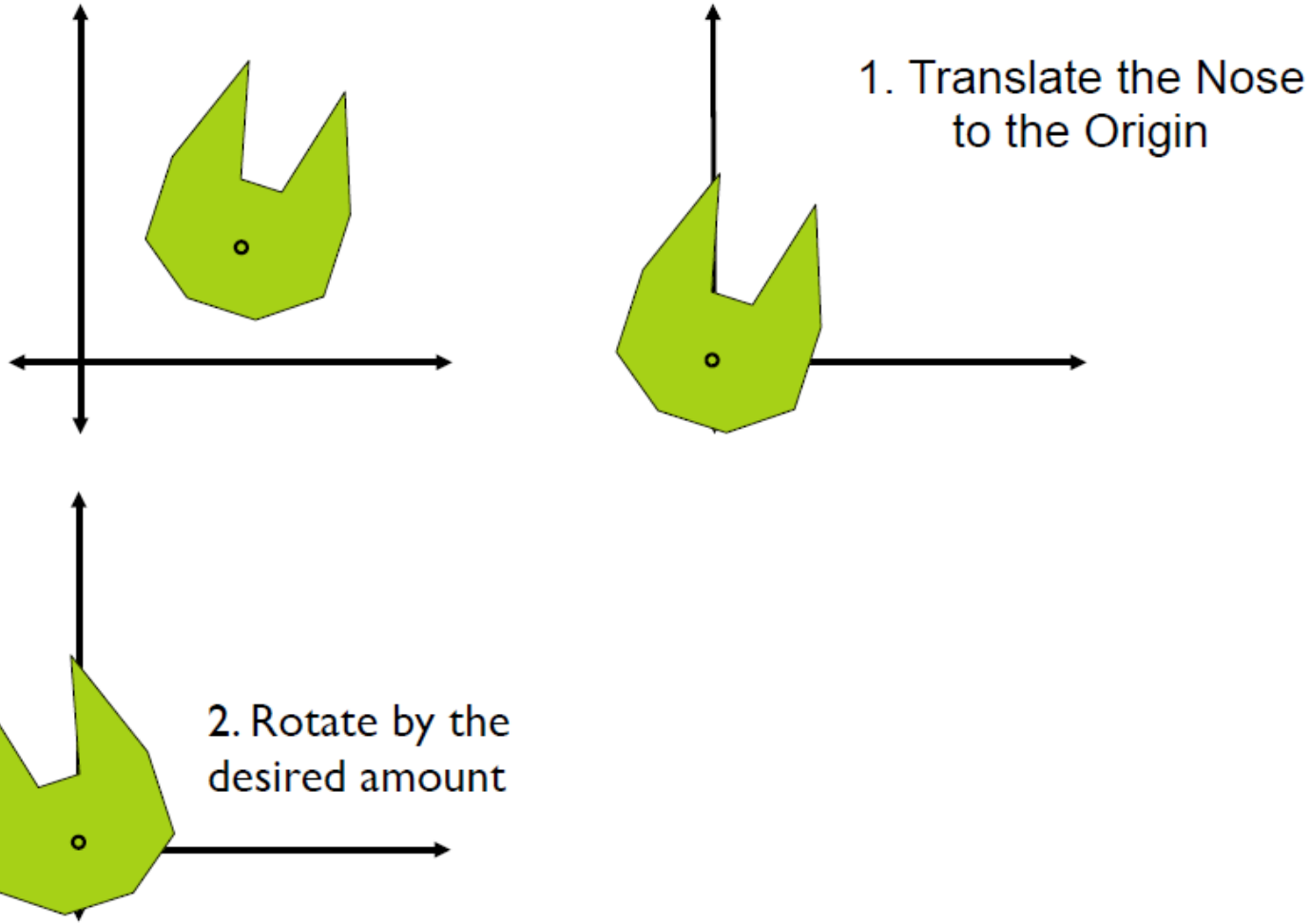


Simple Rotate

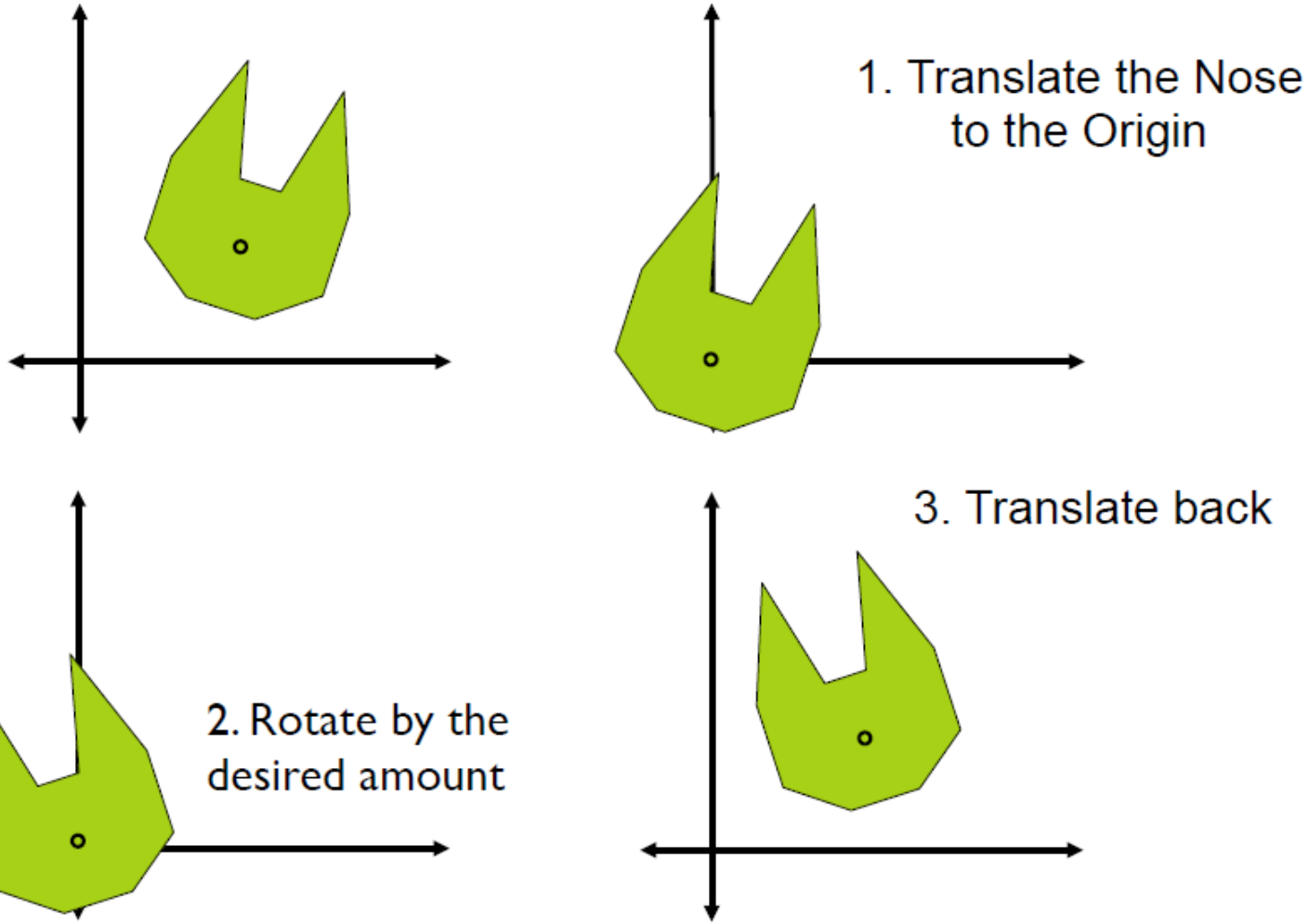
To rotate the cat's head about its nose



Simple Rotate

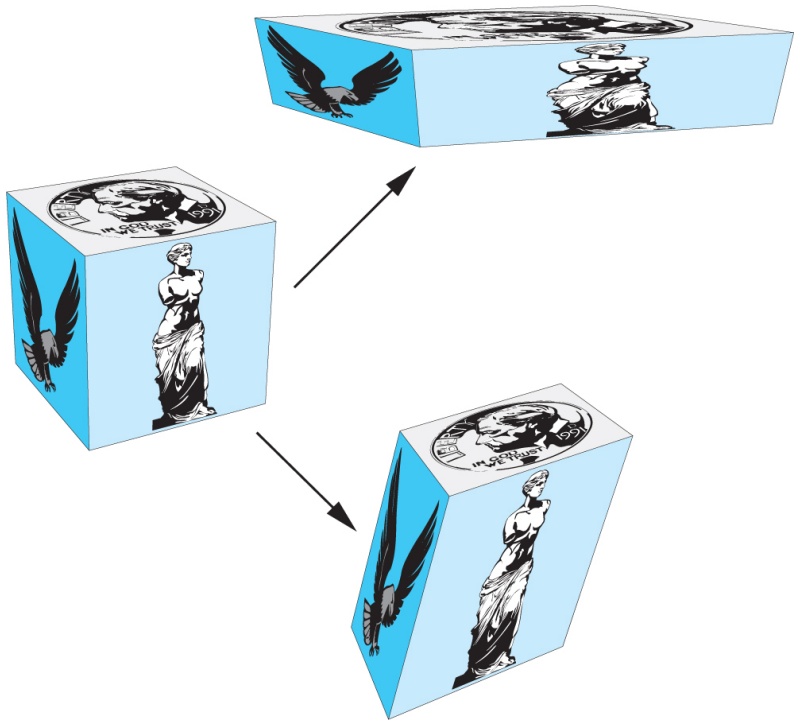


Simple Rotate



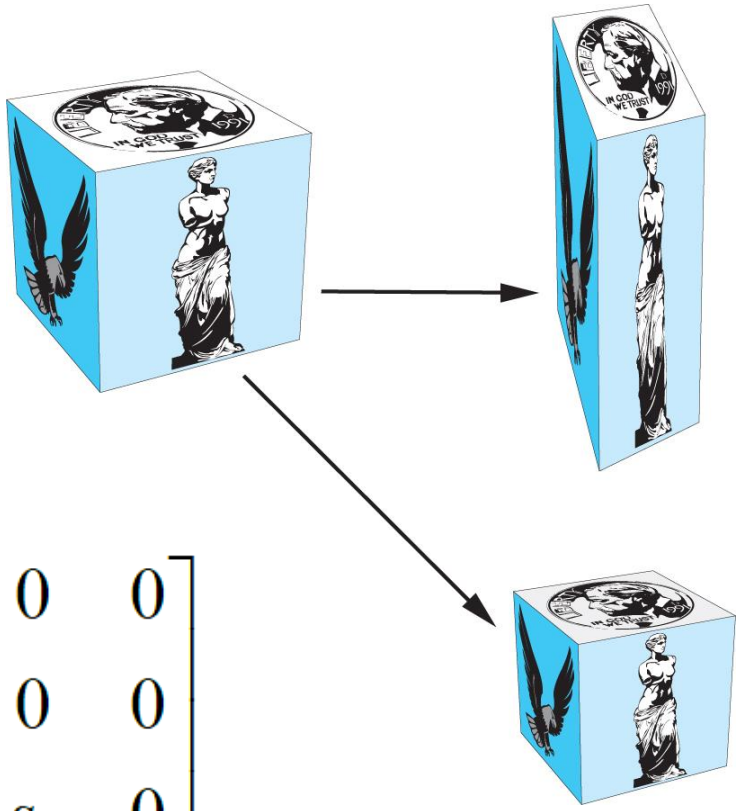
Rigid Body Transformation

- Translation and Rotate are both rigid body transformation.
  - Only alter position and Orientation
- Other affine transformations will alter object's shape.



Scaling

$$x' = s_x x$$
$$y' = s_y y$$
$$z' = s_z z$$
$$\mathrm{p}' = \mathrm{Sp}$$



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

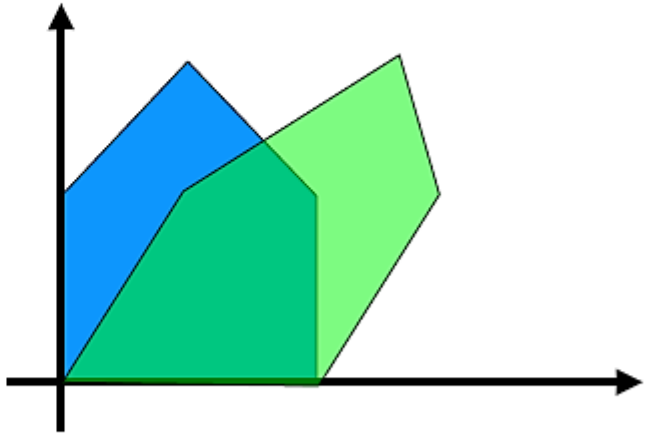
Shearing

- Produces shape distortions
- Shearing in x-direction

$x \Rightarrow x + a * y$

$y \Rightarrow y$

$z \Rightarrow z$



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

• Translation

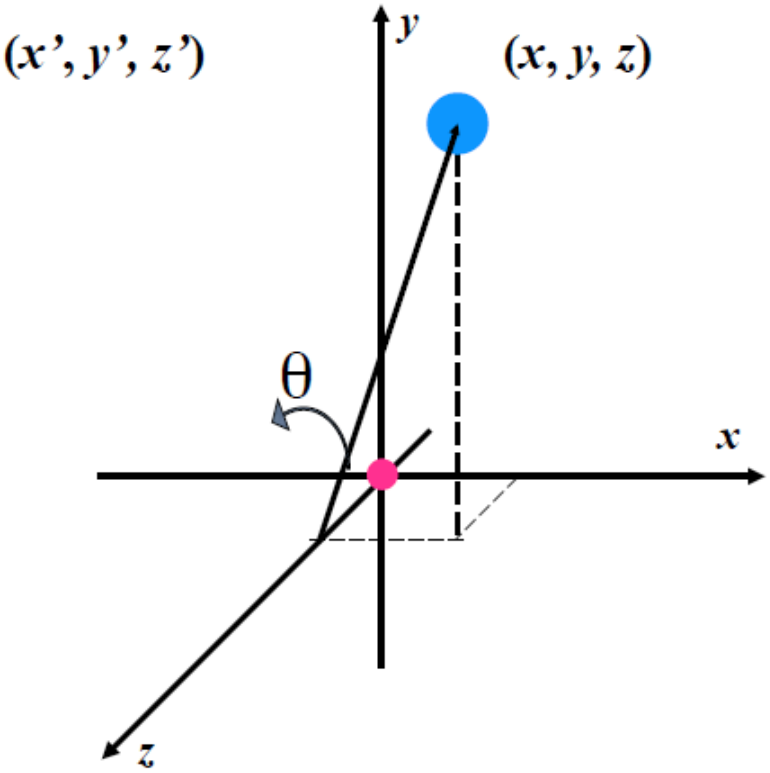
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

• Scaling

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

Matrix Representations of Various Transformations

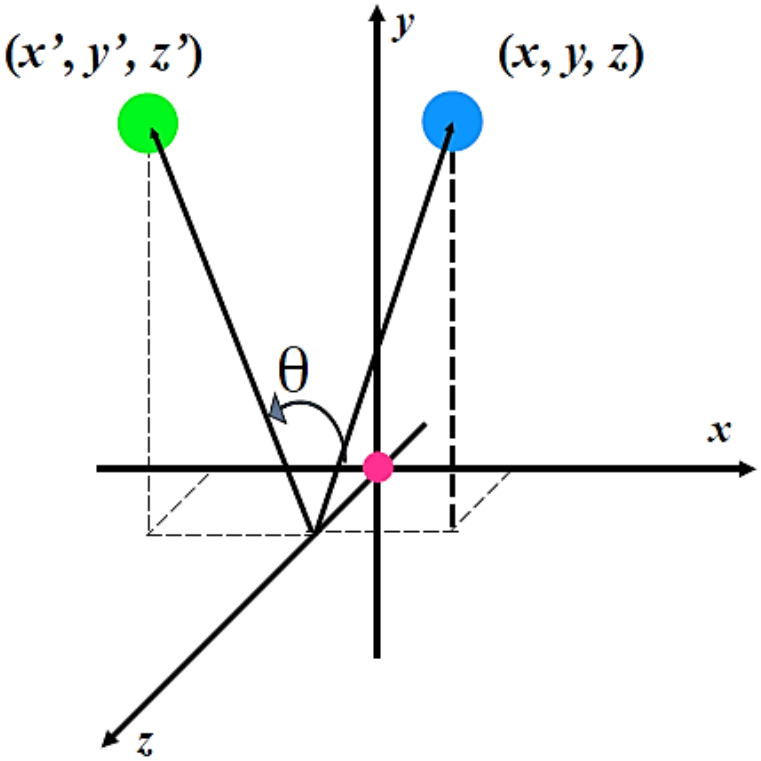
Rotation (around Z axis)



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

Matrix Representations of Various Transformations

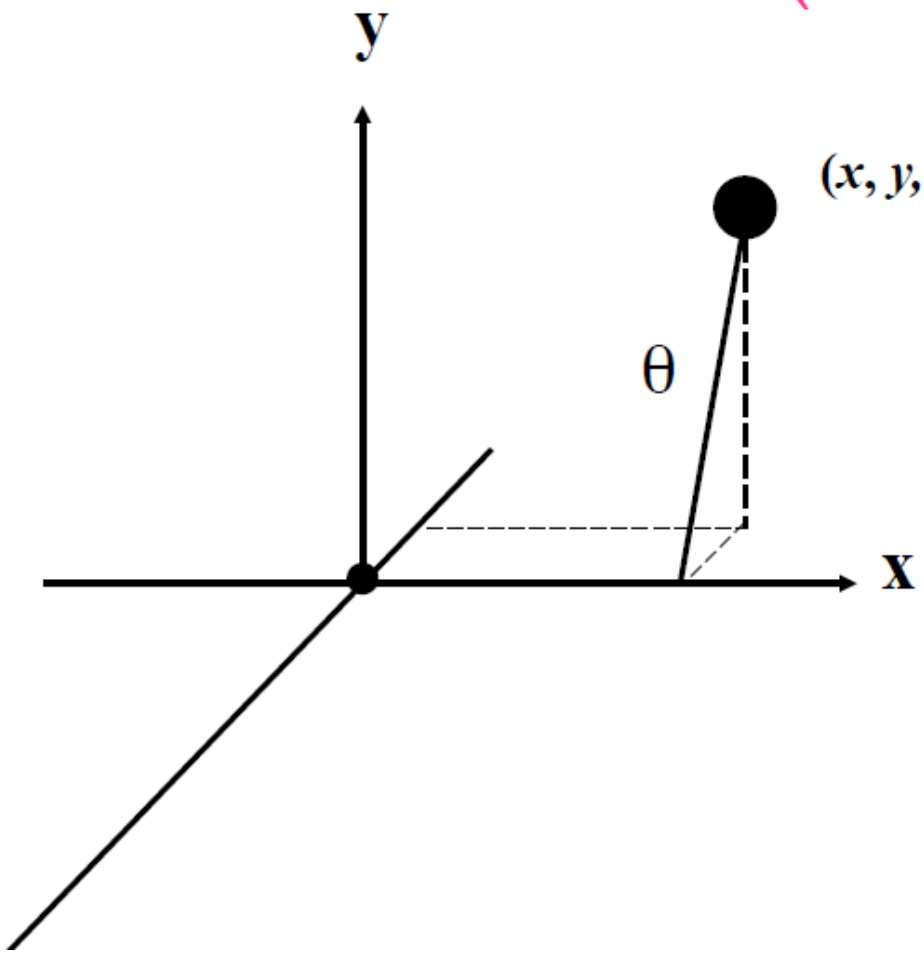
Rotation (around Z axis)



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

Matrix Representations of Various Transformations

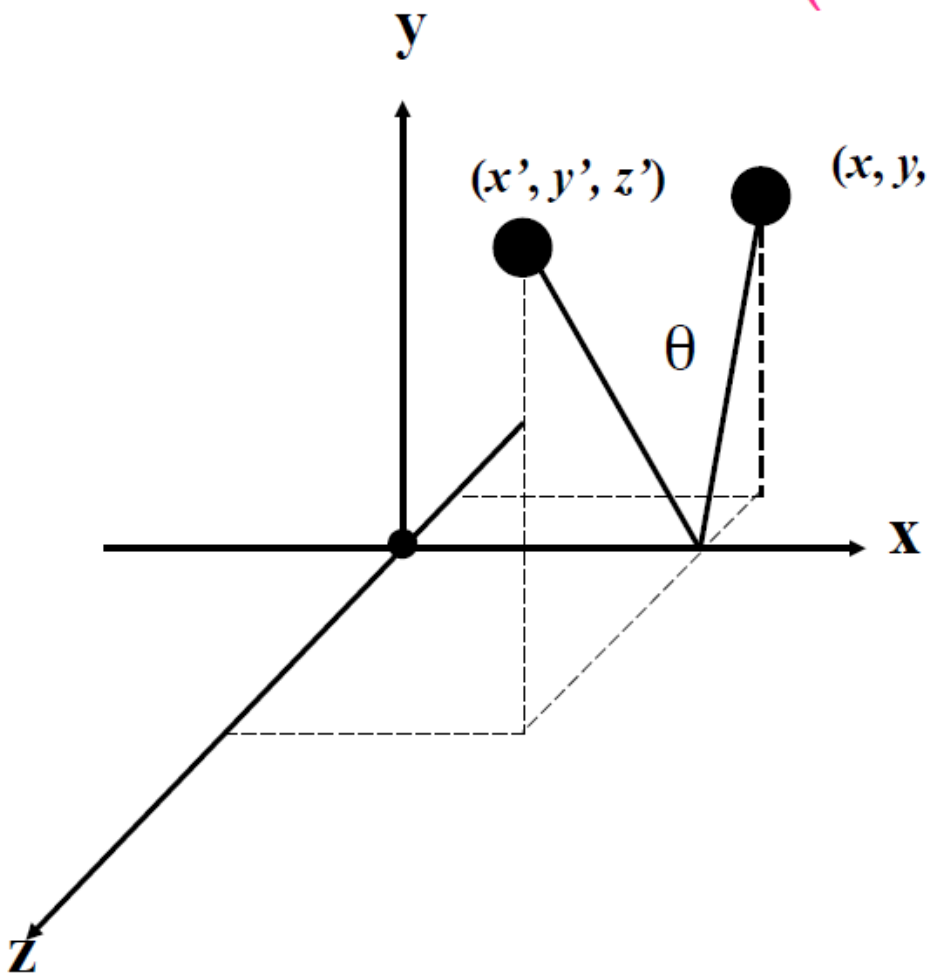
Rotation (around X axis)



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

Matrix Representations of Various Transformations

Rotation (around X axis)

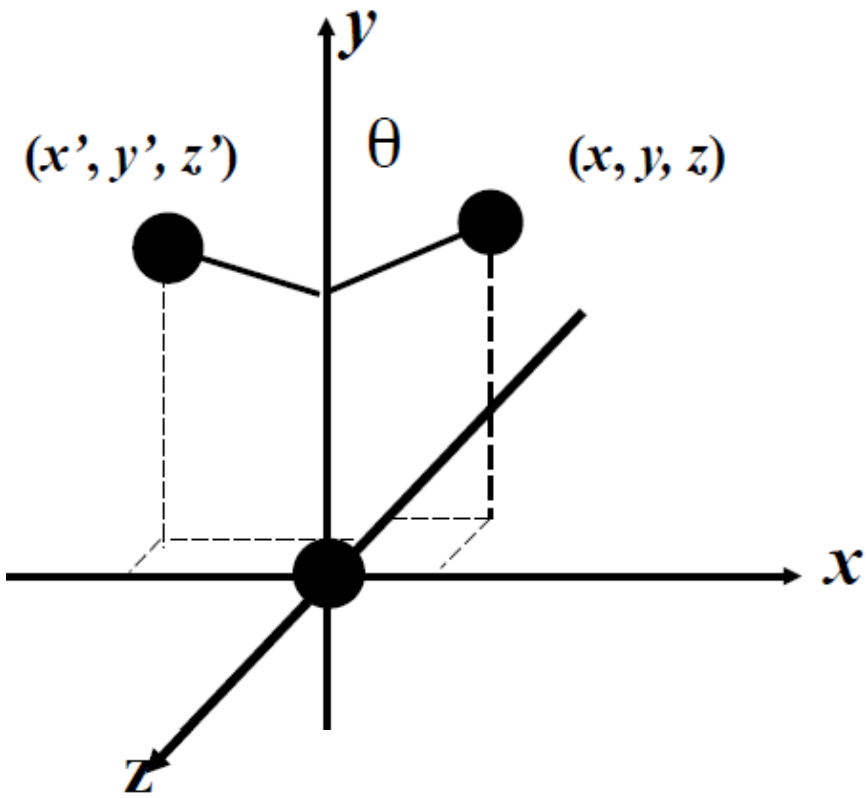


$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Computer Graphics 2014, ZJU

Matrix Representations of Various Transformations

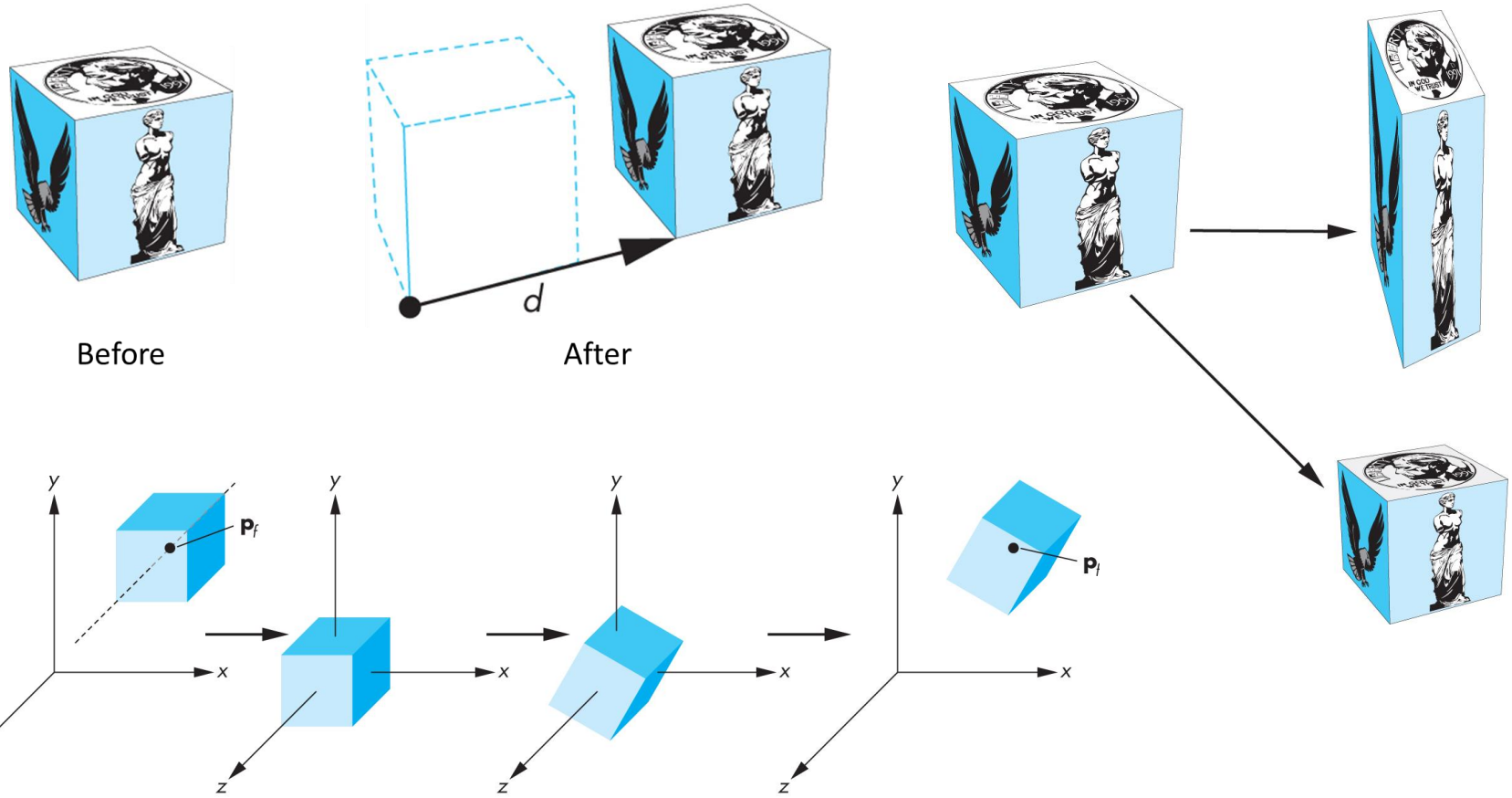
- Rotation (around Y axis)



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

Computer Graphics 2014, ZJU

Transformation



Properties of Transformations

Type Preserves	Rigid Body:	Linear	Affine	Projective
	Rotation & translation	General 3x3 matrix	Linear + translation	4x4 matrix with last row ≠(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

Inverse transformation

- **According to geometric meaning we can give the inverse for various transform**
  - Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathrm{T}(-d_x, -d_y, -d_z)$
  - Rotation: $\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$
    - For any rotation matrix
    - Note: $\cos(-\theta) = \cos\theta$, $\sin(-\theta) = -\sin\theta$, so $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{T}(\theta)$
  - Scale: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Composite transformation

- Multiplication among the rotation, translation and scaling matrices can form arbitrary affine transformation
- 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

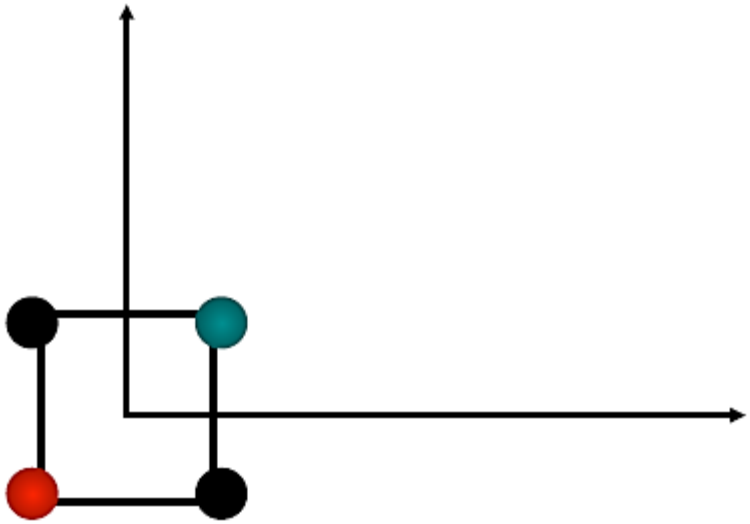
Order of transformations

- Note that the matrix rightmost is the matrix first applied
- From the mathematical point of view, the following representation is equivalent

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

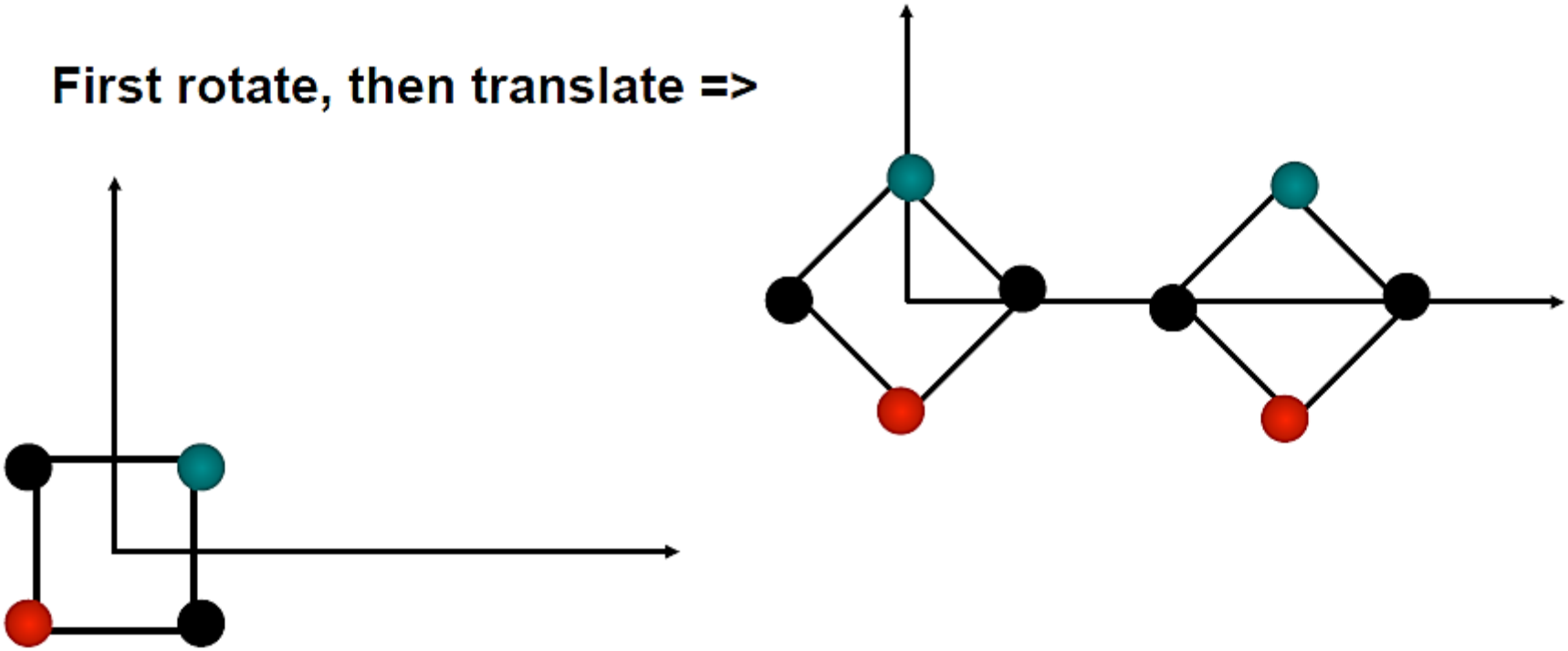
- Transformation sequence is not commutative

Matrix Multiplication is Not Commutative (不可交换)



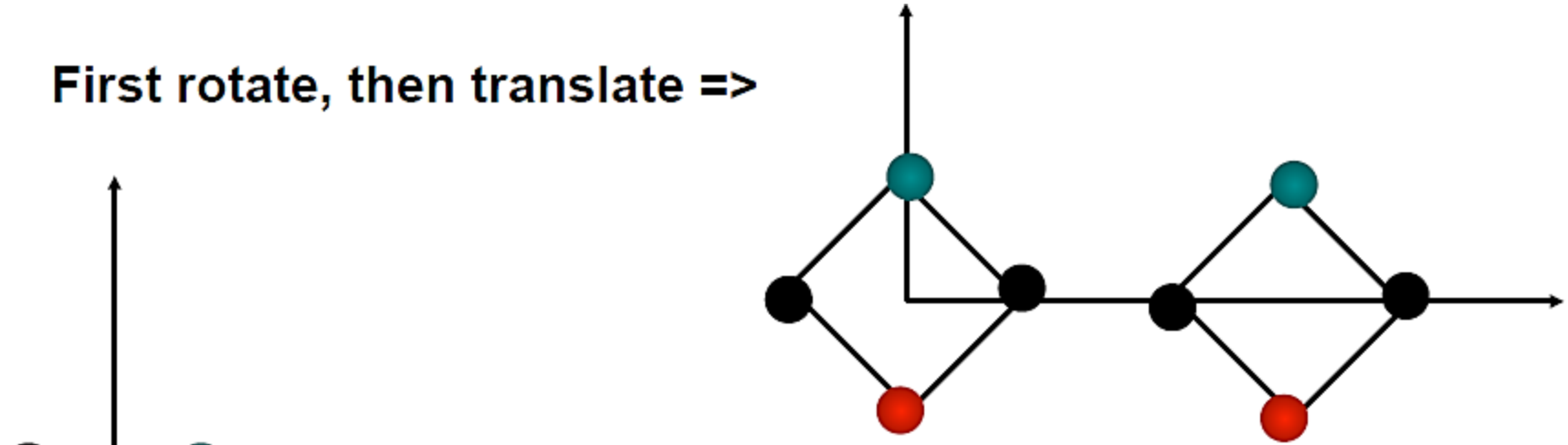
Matrix Multiplication is Not Commutative

First rotate, then translate =>

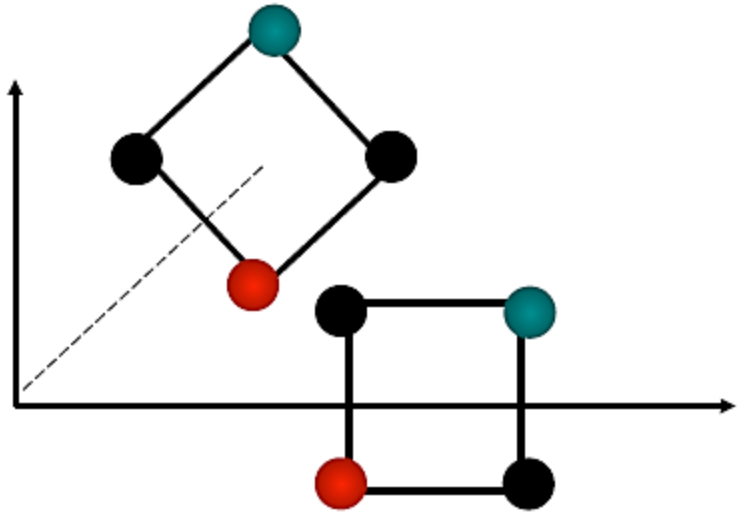


Matrix Multiplication is Not Commutative

First rotate, then translate =>



First translate, then rotate =>



Outline

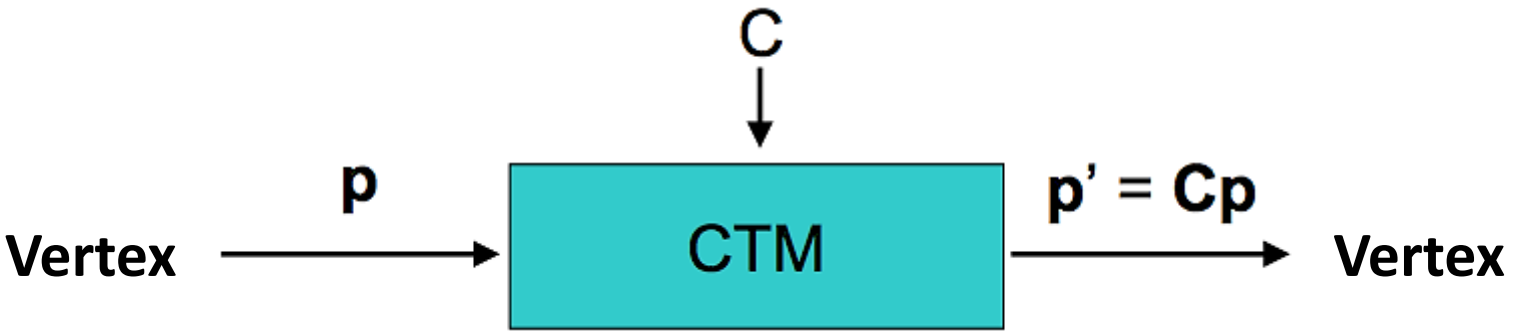
- Geometry
- Representation
- Transformation
- **Transformation in OpenGL**

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);

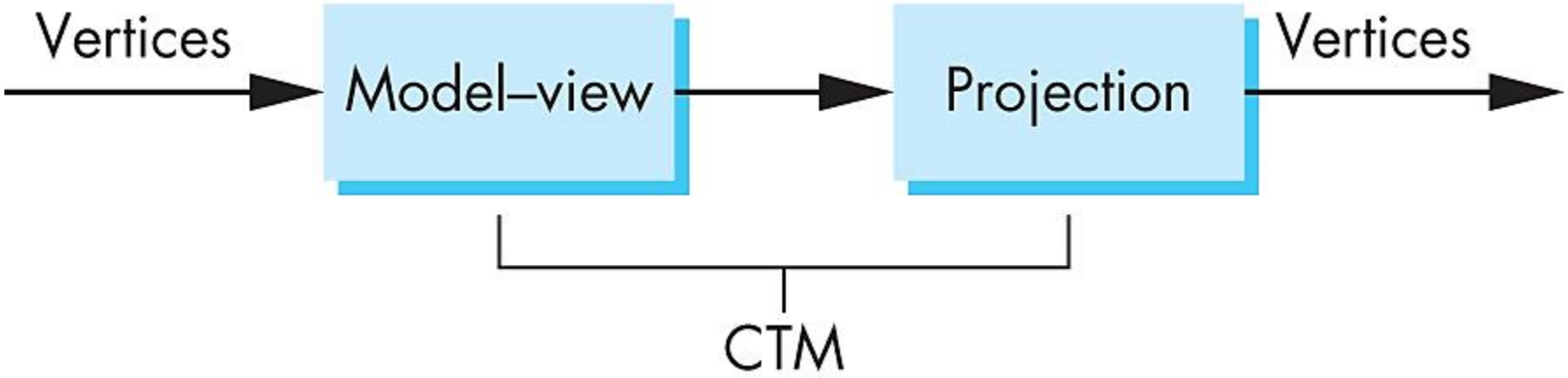
  rotate counterclockwise around ray (0,0,0) to (x, y, z)

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

Rotation, Translation, Scale

- **Upload Identity:** `glLoadIdentity();`
- **Multiply from the right side:**

```
glRotatef(theta, vx, vy, vz);
glTranslatef(dx, dy, dz);
glScalef(sx, sy, sz);
```

Example

- Fixed point (1.0, 2.0, 3.0),rotate 30 degrees along the z axis

```
glMatrixMode(GL_MODEL_VIEW);
glLoadIdentity();

glTranslated(1.0, 2.0, 3.0);
glRotated(30.0,0.0,0.0,1.0);
glTranslated(-1.0,-2.0,-3.0);
```

- The last operation will be executed first

OpenGL Matrices

- In C/C++, we are used to row-major matrices
- In OpenGL, matrices are specified in column-major order

$$\begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix}$$

Row-Major Order

$$\begin{bmatrix} A_0 & A_4 & A_8 & A_{12} \\ A_1 & A_5 & A_9 & A_{13} \\ A_2 & A_6 & A_{10} & A_{14} \\ A_3 & A_7 & A_{11} & A_{15} \end{bmatrix}$$

Column-Major Order

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]**

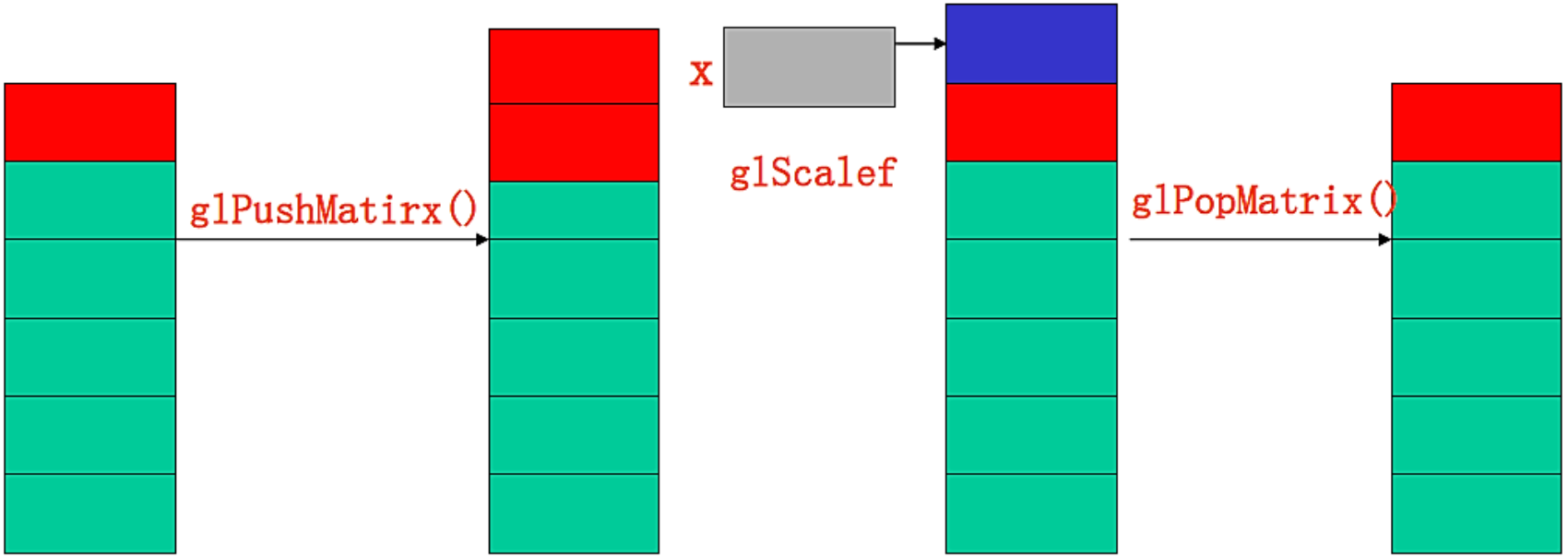
Matrix stack

- 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
- OpenGL maintain a stack for each kind of matrices
  - Use the following statements to operate the stacks

```
glPushMatrix()
glPopMatrix()
```

Matrix in OpenGL

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



Matrix in OpenGL

```
glPushMatrix(); //a此处的push是为了表明堆栈当前状态，测试的每次画图之前都需要用push表明当前状态
  glTranslatef (-1.0, 0.0, 0.0);
  glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
  glTranslatef (1.0, 0.0, 0.0);
  glPushMatrix(); // b此处的push保存了以上三个变换的矩阵
    glScalef (2.0, 0.4, 1.0);
    glutWireCube (1.0); //这里画出正方形之后对其作用一个scale加上面的三个变换
  glPopMatrix(); //这里的pop使得堆栈状态回复到b的状态，也就是有三个变换矩阵的状态

  glPushMatrix(); //作用同a，表明当前堆栈状态，此时状态与b同，保存了三个变换的矩阵
    glTranslatef (1.0, 0.0, 0.0);
    glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
    glTranslatef (1.0, 0.0, 0.0);
    glPushMatrix(); //c 这里的pushMatrix保存了六个变换矩阵的状态，自己的三个变换加上初始时就已经存在的三个变换矩阵
      glScalef (2.0, 0.4, 1.0);
      glutWireCube (1.0); //这里的正方形要作用一个scale加上面的六个变换
    glPopMatrix(); //此处回复到第二次画图时初始的三个变换矩阵状态
  glPopMatrix(); //此处pop回复到最初始的堆栈状态
```