# 컴퓨터그래픽스

김준호

Visual Computing Lab.

국민대학교 소프트웨어학부

# Transformations

- Affine Space & Homogeneous Coordinates
- Linear transformations
- Model transformations
- View transformations

# Affine Space & Homogeneous Coordinates

# OpenGL ES uses Homogeneous coordinate System

- Graphics cards support homogeneous coordinates
  - 4x1 vectors for 3D points & 3D vectors

$$\boldsymbol{p} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} \qquad \boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix}$$

  - In general, the following holds (if a ≠ 0, w ≠ 0)
    - It is related to projective geometry

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az \\ aw \end{bmatrix} = \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix}$$
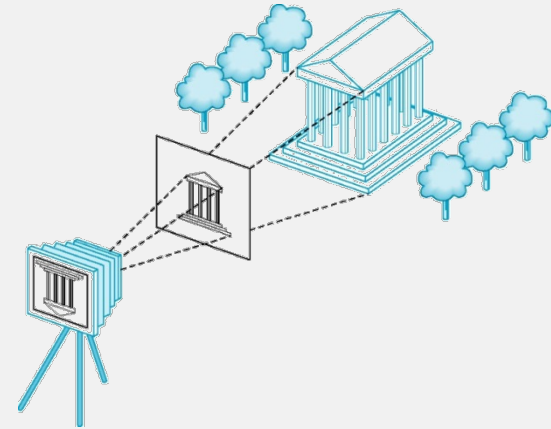
# Affine Space

- The affine space contains three types of object
  - Scalars
  - Vectors
  - Points
- Why affine space?
  - We need to clearly distinguish the concepts of vectors and points
    - There was no strong differences in vectors and points, especially in high school

**p** = (3,2)

**v** = (3,2)
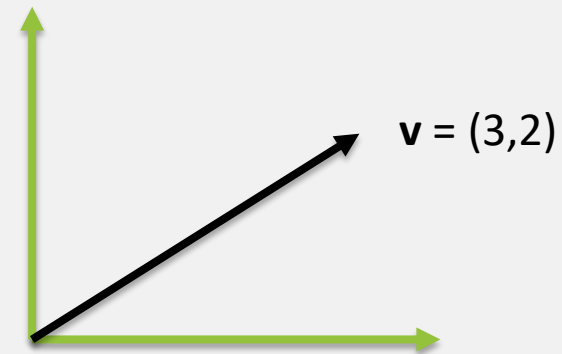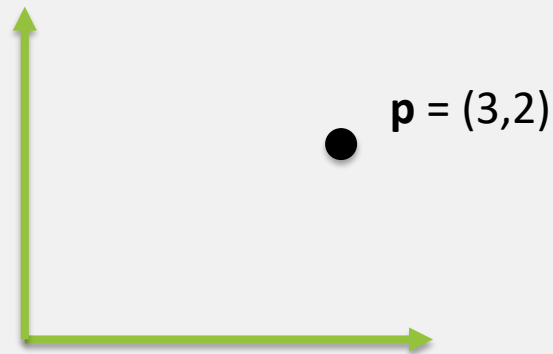
# Affine Space

- Scalar $\oplus$ Vector $\oplus$ Point

- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar opertions

- For any point define
  - $1 \bullet \mathbf{P} = \mathbf{P}$
  - $0 \bullet \mathbf{P} = \mathbf{0}$ (zero vector)

# Scalars

- Scalars represent the concept of quantity
  - Ex) 7, 3.14, -1, …
  - Combined with two basic operations
    - Addition
    - Multiplication

- Scalars alone have no geometric propoerties

# Vectors

- A vector is a quantity of two attributes
  - Direction
  - Magnitute

- A vector usually represented with a directed line segment

$v$

- Examples include
  - Force
  - Velocity
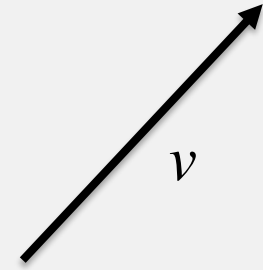
# Vector Operations

- scalar * vector
  - Ex) 2**v**, -**v**, …
  - Result: a vector
- vector + vector
  - Ex) **u** + **w**
  - Result: a vector
- Products
  - Dot product
    - Ex) **u** • **w**
    - Result: a scalar
      - Physical meaning: amount of projection
  - Cross product
    - Ex) **u** x **w**
    - Result: a vector
      - Physical meaning: amount of rotation

$v$   $-v$   $\alpha v$

v   $w$   $u$

A   $\theta$   B
$|A|\cos\theta$

**a**×**b**
**b**   $\theta$   $|\mathbf{a}\times\mathbf{b}|$
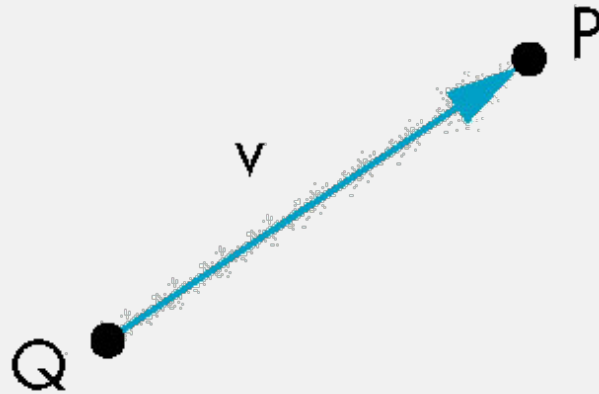**a**
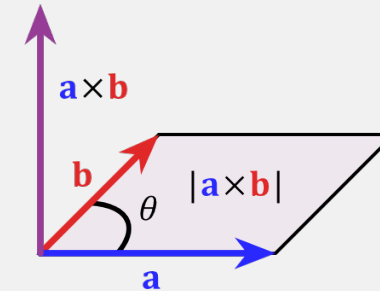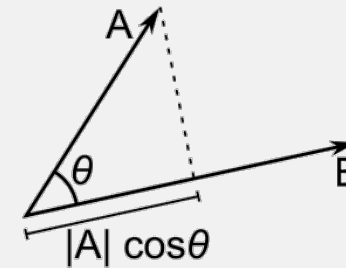
# Vectors Lack Point

- The following vectors are identical
  - Same length and magnitute
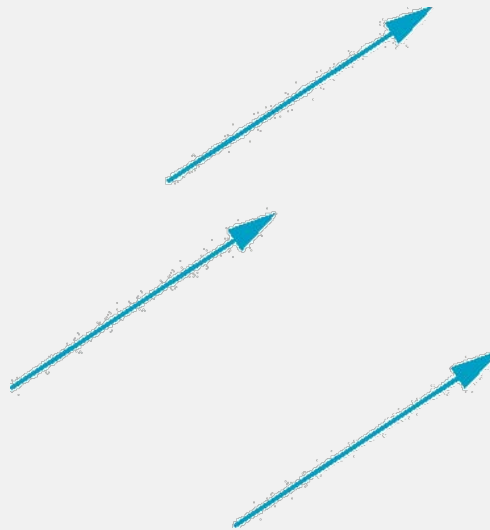
- (Scalars + Vectors) are insufficient for representing geometry
  - Need points

# Points

- Location in space
- Operations allowed between points and vectors
  - Point-point subtraction yields a vector
    - **P** - **Q** = **v**
    - ~~**P + Q**~~ → no physical meaning
  - Equivalent to point-vector addtion
    - **Q** + **v** = **P**
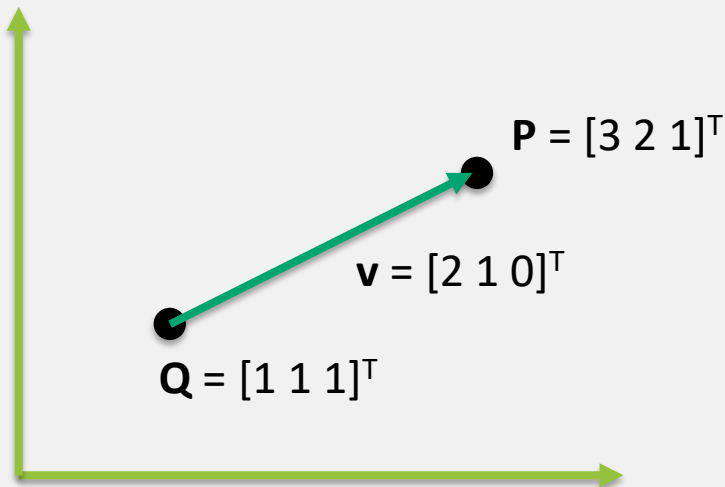


$$v=P-Q$$

$$P=v+Q$$

# Homogeneous Coordinates

- Homogeneous coordinate systems clearly distnguish the concepts of vectors and points
  - $n$D point is represented with a ($n+1$)D vector, whose last component is 1
  - $n$D vector is represented with a ($n+1$)D vector, whose last component is 0
- Example) 2D points, 2D vectors → 3D vectors
  - $\mathbf{p} = [3\ 2\ 1]^T$
  - $\mathbf{v} = [3\ 2\ 0]^T$

**p** = (3,2)

**v** = (3,2)
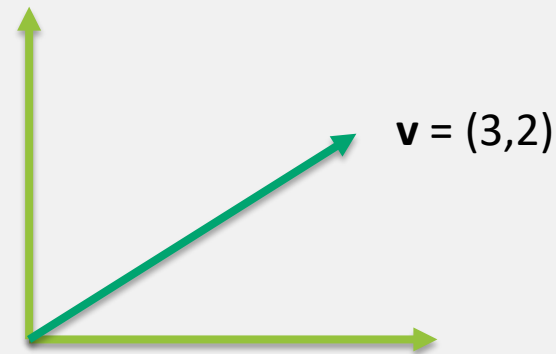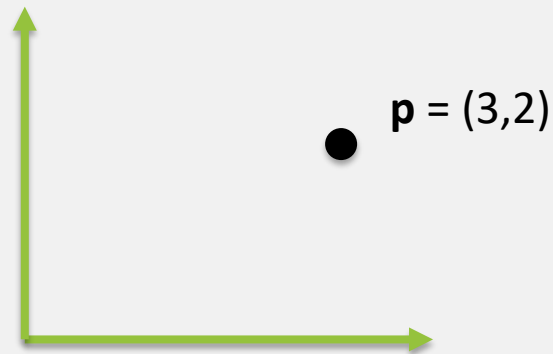
# Homogeneous Coordinates

- Homogeneous coordinates hold the operations of affine space
    - E.g.) **P** = [3 2 **1**]$^\text{T}$, **Q** = [1 1 **1**]$^\text{T}$, **v** = [2 1 **0**]$^\text{T}$
        - **v** = **P** - **Q** = [3 2 **1**]$^\text{T}$ - [1 1 **1**]$^\text{T}$ = [2 1 **0**]$^\text{T}$
        - **P** = **Q** + **v** = [1 1 **1**]$^\text{T}$ + [2 1 **0**]$^\text{T}$ = [3 2 **1**]$^\text{T}$
        - 2**v**
        - ~~3**Q**~~

**P** = [3 2 1]$^\text{T}$

**v** = [2 1 0]$^\text{T}$
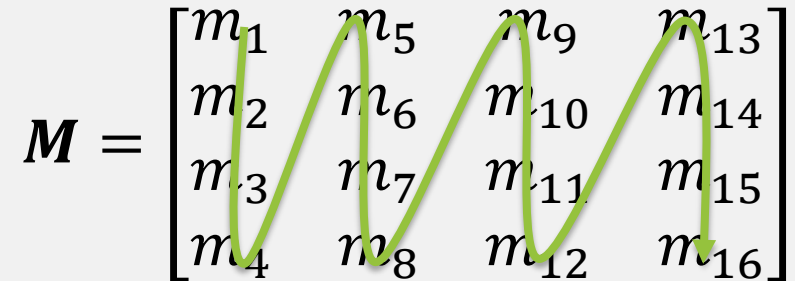
**Q** = [1 1 1]$^\text{T}$

# Why Homogeneous Coordinate?

- Graphics cards support homogeneous coordinates
  - 4x1 vectors for 3D points & 3D vectors

$$\boldsymbol{p} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} \qquad \boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix}$$

  - 4x4 matrices for
    - GL_MODELVIEW_MATRIX, GL_PROJECTION_MATRIX, GL_TEXTURE_MATRIX
    - Note: OpenGL ES represents a matrix in the column-major way

$$\boldsymbol{M} = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

# Linear Transformations
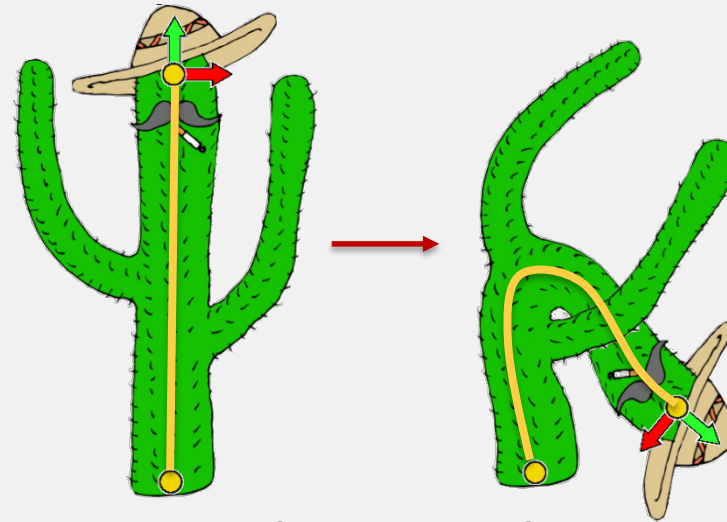
# Transformation (변환)

- In computer graphics, transformation refers to
  - Change of shape
    - Linear transformation: line to line
    - Non-linear transformation: line to curve

Linear transformation

Non-linear transformation

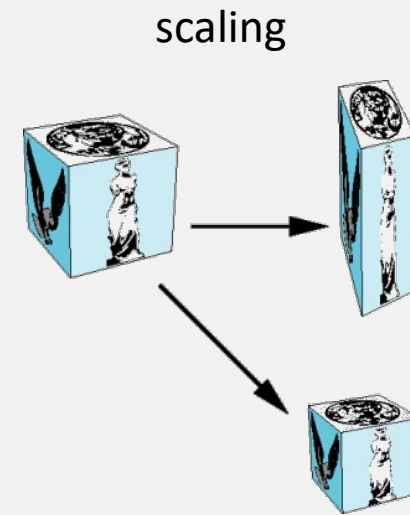$$\mathbf{T}(x,y) = \begin{bmatrix} -1 & -1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$
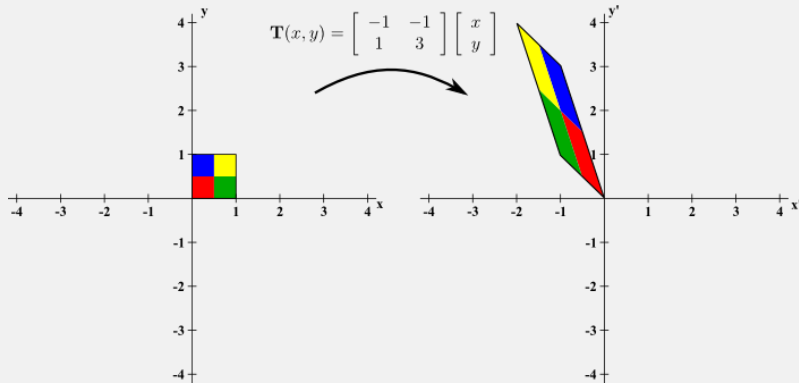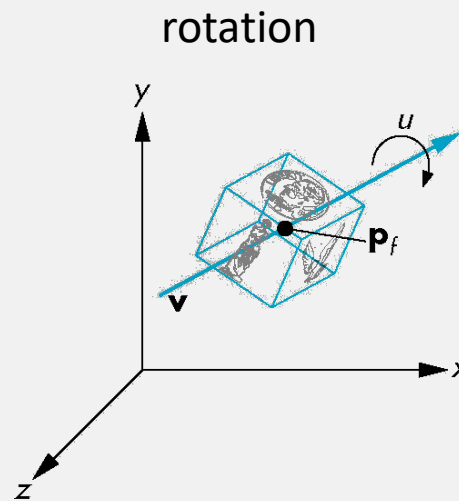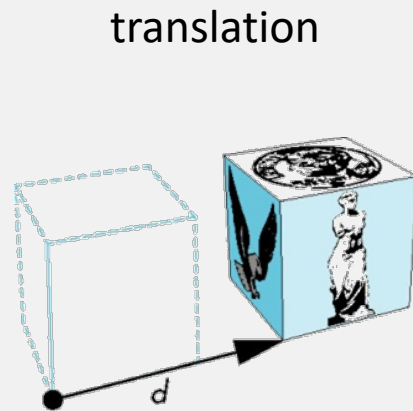
http://mathinsight.org/determinant_linear_transformation
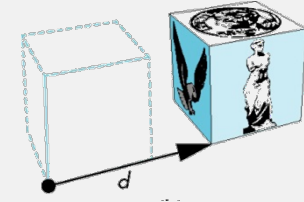
[Jacobson et al. 2012]

# Linear Transformation (선형 변환)

- It can be represented as linear a matrix
- Standard transformation
  - Translation / Rotation / Scaling
- Composition of linear transformation is linear

translation          rotation          scaling

# Translation (이동)

- Move (translate, displace) a point to a new location
  - Translation of an object: every point displaced by same vector
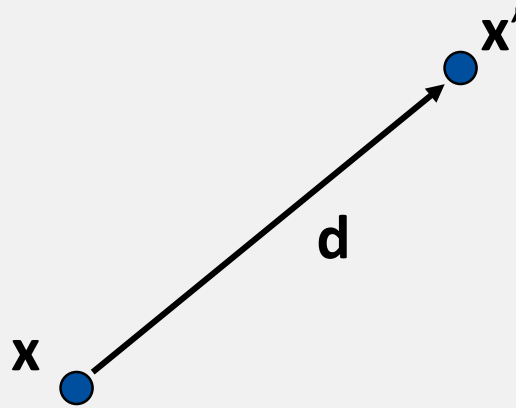


- Displacement determined by a vector **d**
  - Three degrees of freedom for **d**, in 3D case: **d** = $(d_x, d_y, d_z)$
  - **x′** = **x** + **d**

# Translation (이동)

- Translation matrix **T**
  - Translation can also be expressed by using a 4x4 matrix **T** in homogeneous coordinates
  - **x'=Tx**
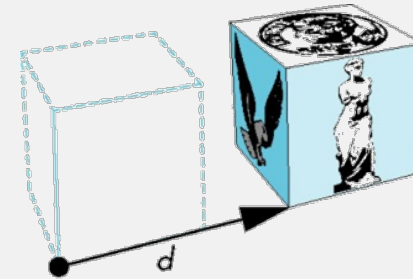    - **T** = **T**($d_x$, $d_y$, $d_z$) =

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

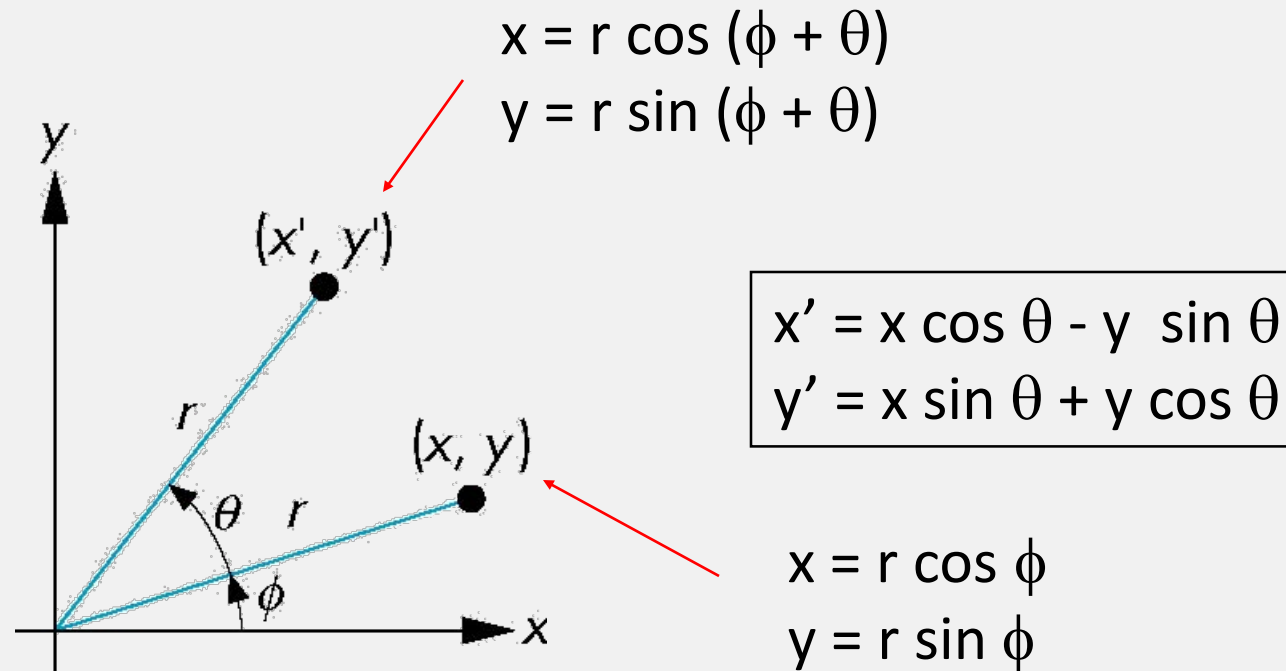    - glTranslate($d_x$, $d_y$, $d_z$)
    - This form is better for H/W implementation

# Rotation (회전)

- Consider rotation about the origin by $\theta$ degrees
    - Radius stays the same, angle increases by $\theta$

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

$(x', y')$

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

$(x, y)$

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

# Rotation (회전)
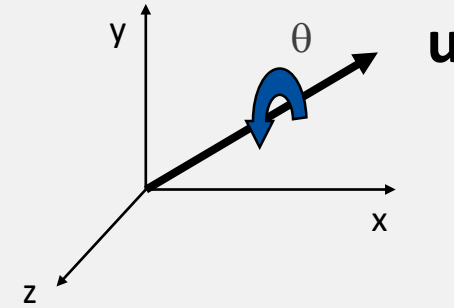
- General rotation about the <span style="color:red">origin</span>
  - A rotation by $\theta$ rotation an arbitrary axis **u**
    - [Quaternion](#) can express general rotation
  - **x'=R$_u$($\theta$)x**
    - **R$_u$($\theta$)=** $\begin{bmatrix} \cos\theta + u_x^2(1-cos\theta) & u_xu_y(1-cos\theta) - u_z sin\theta & u_xu_z(1-cos\theta) + u_y sin\theta & 0 \\ u_yu_x(1-cos\theta) + u_z sin\theta & \cos\theta + u_y^2(1-cos\theta) & u_yu_z(1-cos\theta) - u_x sin\theta & 0 \\ u_zu_x(1-cos\theta) - u_y sin\theta & u_zu_y(1-cos\theta) + u_x sin\theta & \cos\theta + u_z^2(1-cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

    - [glRotate](#)($\theta$, $u_x$, $u_y$, $u_z$)
      - **u**=($u_x$, $u_y$, $u_z$)

# Scaling (확대축소)

- Scaling matrix **S**
  - Expand or contract along each axis (fixed point of origin)
  - **x'=Sx**
    - **S** = **S**($s_x$, $s_y$, $s_z$) =

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
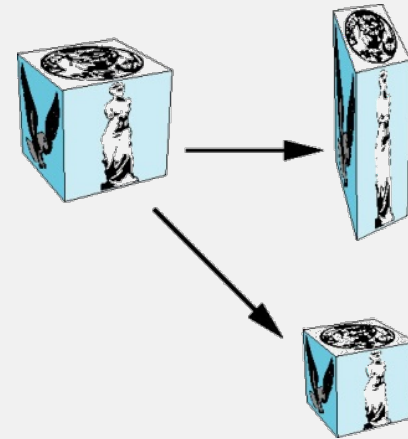


  - glScale($s_x$, $s_y$, $s_z$)

# Model Transformations

# Composite Transformation (합성 변환)

- We can composite transformation by multiplying matrices of rotation, translation, and scaling.
  - Example:
    1) Uniformly scale 2x:           **S = S(2,2,2)**
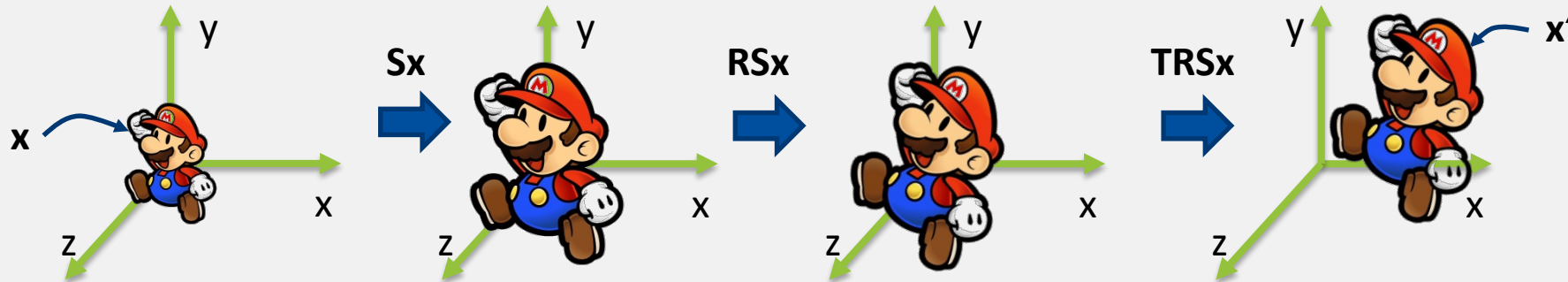    2) Rotate -30 degrees by +z axis:     **R = $R_z$(-30)**
    3) Translate by (2, 1, 0):           **T = T(2,1,0)**
    - **x' = T(R(Sp)) = TRSx**
      - **x**: original object
      - **x'**: transformed object

# Composite Transformation (합성 변환)

- We can composite transformation by multiplying matrices of rotation, translation, and scaling.
  - Example:
    1) Uniformly scale 2x:
    2) Rotate -30 degrees by +z axis:
    3) Translate by (2, 1, 0):
       - $x' = T(R(Sp)) = TRSx$
         - **x**: original object
         - **x'**: transformed object

```
glTranslatef(2, 1, 0);          // T
glRotatef(-30, 0, 0, 1);        // R
glScalef(2, 2, 2);              // S
glDrawElements(…);              // x
```
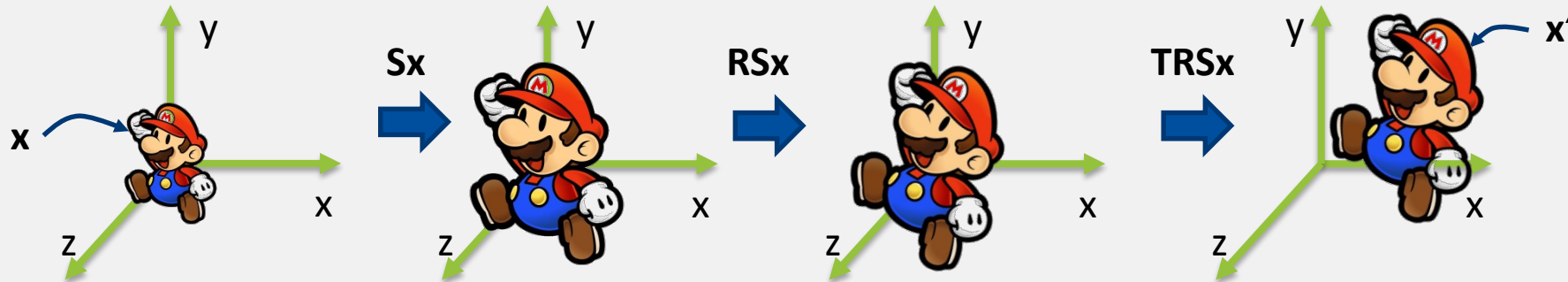
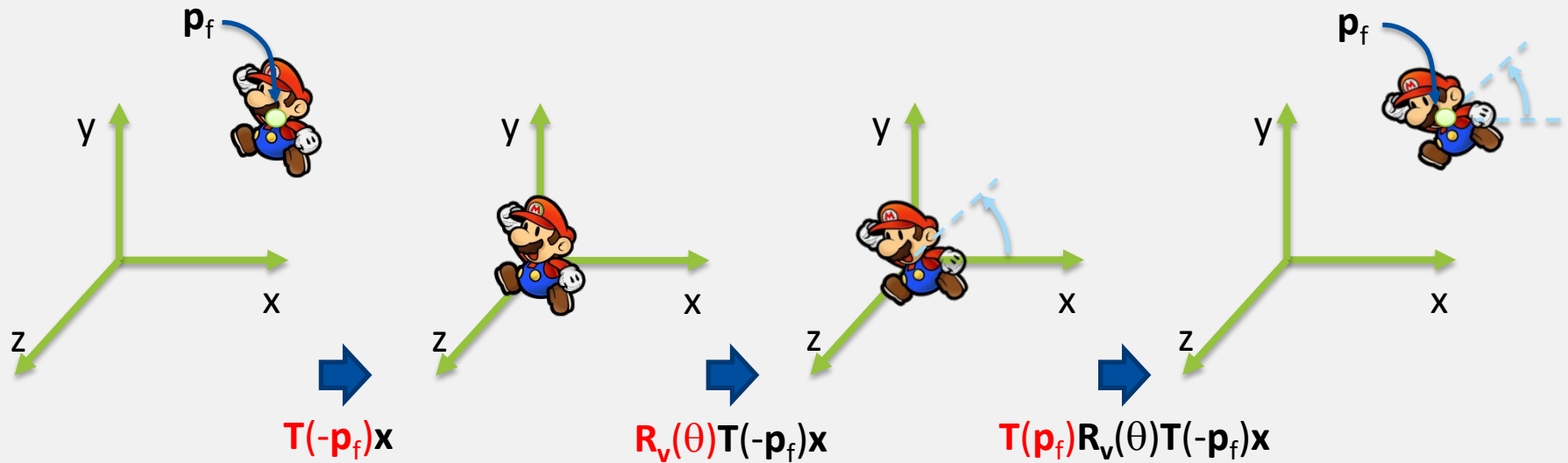# Rotation about a Fixed Point other than the Origin

- Rotation: originally, a fixed point is the origin
  - Example
    - Rotate 30 degrees by +z axis:      **R** = **R$_z$(30)**

- But, ration about a general fixed point **p**$_f$ is necessary, in general
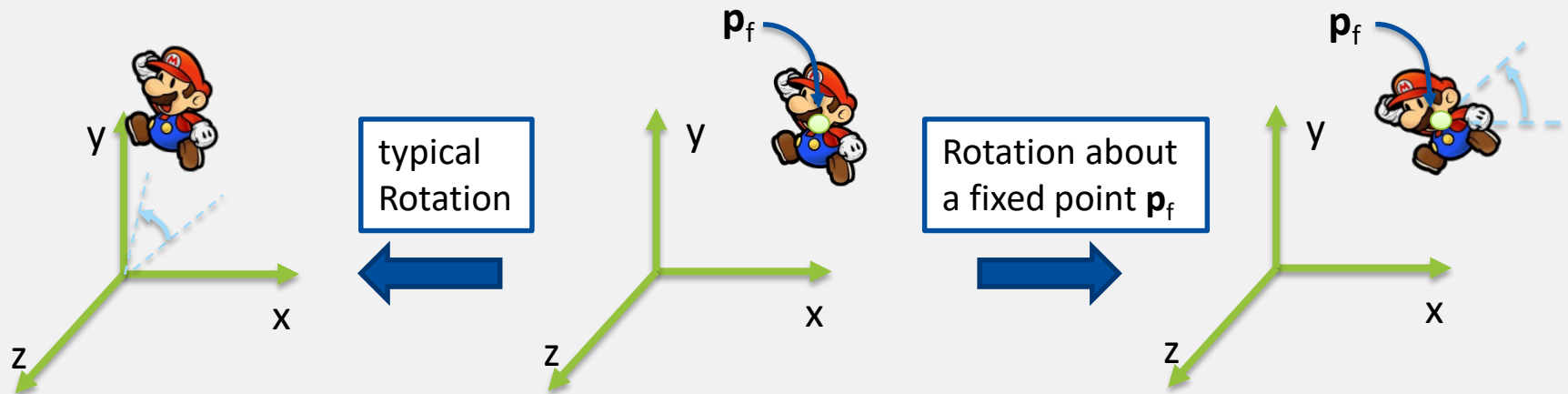
# Rotation about a Fixed Point other than the Origin

- Composite transformation
  - Move fixed point to origin
  - Rotate
  - Move fixed point back
- $\mathbf{x}' = \mathbf{T}(\mathbf{p}_f)\mathbf{R}_v(\theta)\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$

```
glTranslate(pfx, pfy, pfz);        // T(pf)
glRotate(theta, vx, vy, vz);       // Rv(θ)
glTranslate(-pfx, -pfy, -pfz);     // T(-pf)
glDrawElements(…);                 // x
```



$\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$ → $\mathbf{R}_v(\theta)\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$ → $\mathbf{T}(\mathbf{p}_f)\mathbf{R}_v(\theta)\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$

# Scale about a Fixed Point other than the Origin

- Scaling: originally, a fixed point is the origin
  - Example
    - Uniformly scale 2x:         $S(2,2,2)$

- But, scaling about a general fixed point $p_f$ is necessary, in general

# Scale about a Fixed Point other than the Origin
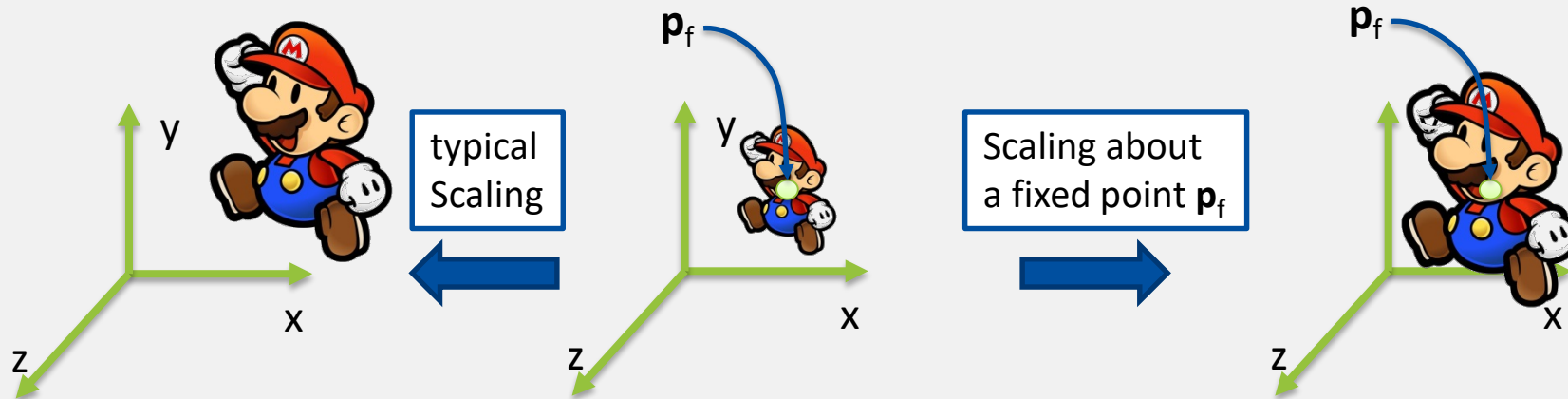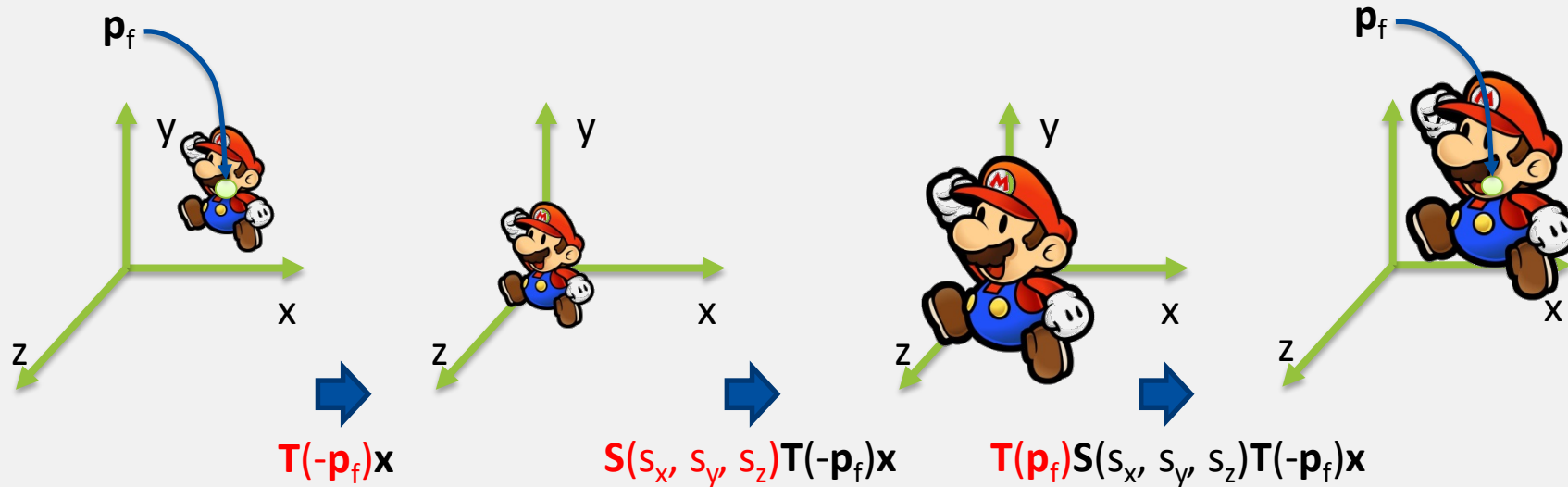
- Composite transformation
  - Move fixed point to origin
  - Scale
  - Move fixed point back
- $\mathbf{x}' = \mathbf{T}(\mathbf{p}_f)\mathbf{S}(s_x, s_y, s_z)\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$

```
glTranslate(pfx, pfy, pfz);        // T(pf)
glScale(sx, sy, sz);               // S(sx, sy, sz)
glTranslate(-pfx, -pfy, -pfz);     // T(-pf)
glDrawElements();                  // x
```



$\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$    $\mathbf{S}(s_x, s_y, s_z)\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$    $\mathbf{T}(\mathbf{p}_f)\mathbf{S}(s_x, s_y, s_z)\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$

# Instancing

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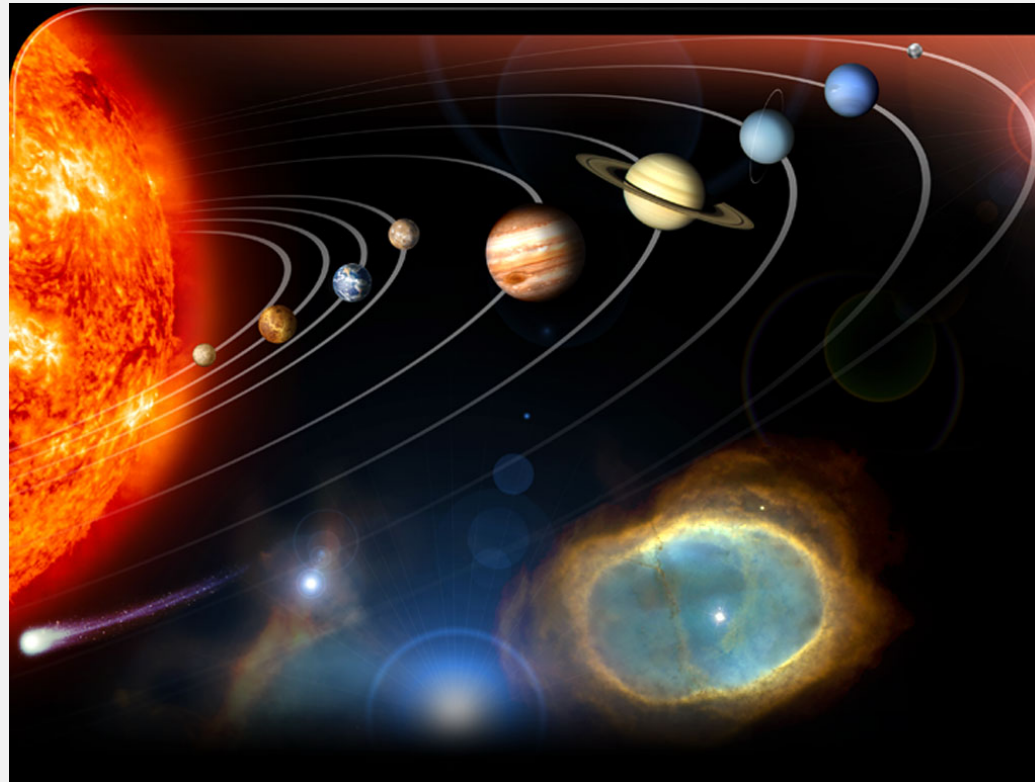  - Scale
  - Orient
  - Locate
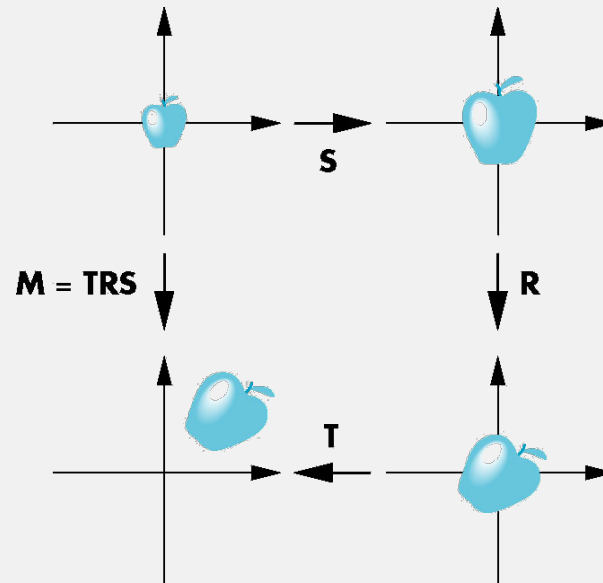
*Advanced Transformation*

# Transformations
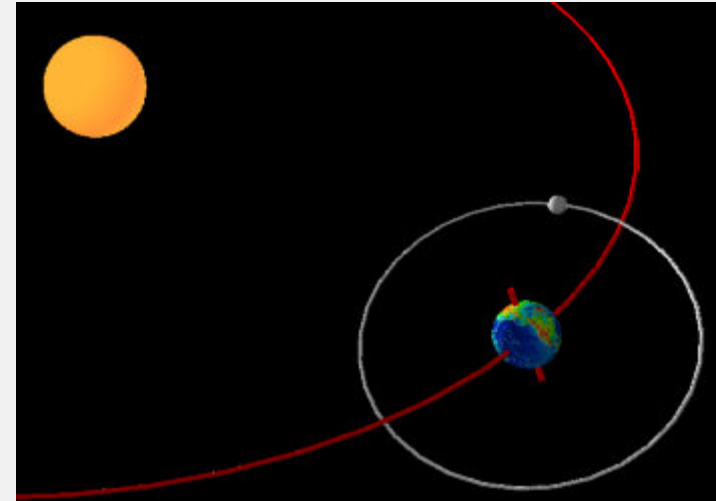# for Hierarchical Objects

# Solar System

- An example of hierarchical objects
  - How can we design transformations for hierarhical objects


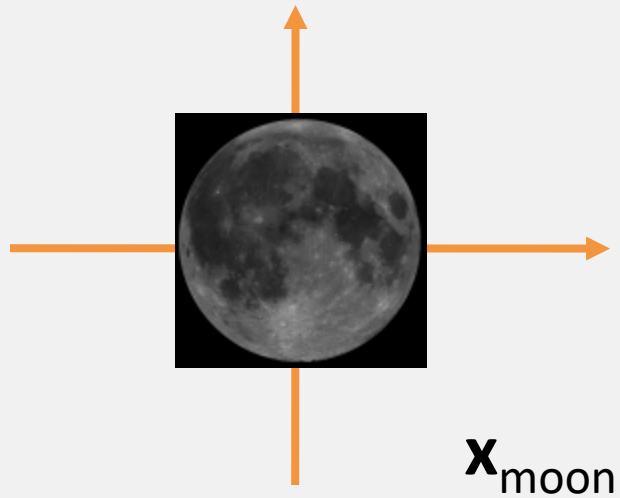
([video](#), [youtube](#))

# Sun – Earth – Moon

- Sun
- Earth
  - rotating itself
  - orbiting around the sun
- Moon
  - rotating itself
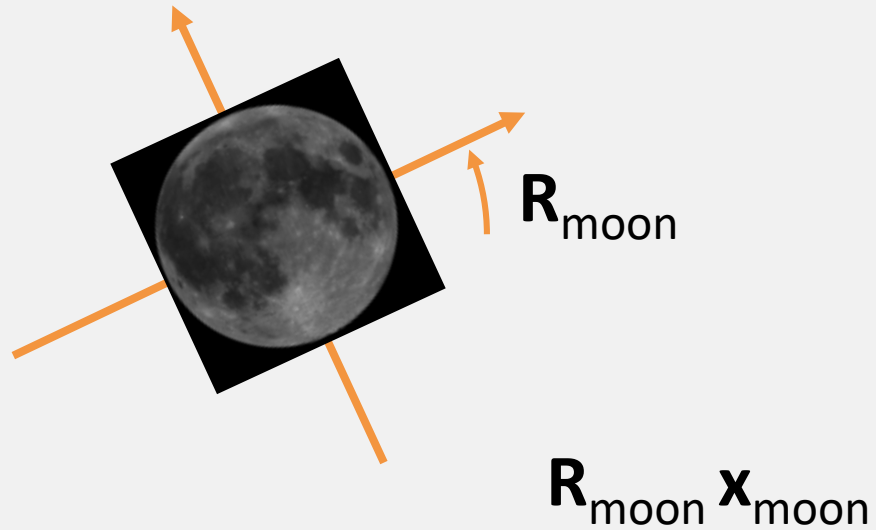  - orbiting around the earth

# Hierarchical Transformation

- Moon
  - Modeling the moon



$\mathbf{x}_{moon}$

# Hierarchical Transformation

- Moon
  - Modeling the moon
  - Rotating itself



$\mathbf{R}_{moon}$

$\mathbf{R}_{moon}\,\mathbf{x}_{moon}$

# Hierarchical Transformation

- Earth – Moon
  - The moon is orbiting around the earth

$$\mathbf{x}_{earth} = \mathbf{T}_{moon}(d_{e-m}\cos\theta, d_{e-m}\sin\theta)\mathbf{R}_{moon}\,\mathbf{x}_{moon}$$

$d_{e-m}$

$\mathbf{R}_{moon}$

$\theta$

$\mathbf{T}_{moon}(d_{e-m}\cos\theta, d_{e-m}\sin\theta)$

# Hierarchical Transformation

- Sun – Earth
  - The earth is orbiting around the sun

$$\mathbf{x}_{sun} = \mathbf{T}_{earth}(d_{s-e}\cos\phi,\ d_{s-e}\sin\phi)\mathbf{R}_{earth}\ \mathbf{x}_{earth}$$



$\mathbf{R}_{earth}$

$d_{s-e}$

$\phi$

$\mathbf{T}_{earth}(d_{s-e}\cos\phi,\ d_{s-e}\sin\phi\ )$

# Hierarchical Transformation

- Sun – Earth – Moon
  - $\mathbf{x}_{sun}=\mathbf{T}_{earth}(d_{s-e}\cos\phi, d_{s-e}\sin\phi)\mathbf{R}_{earth}\,\mathbf{x}_{earth}$
  - $\mathbf{x}_{earth}=\mathbf{T}_{moon}(d_{e-m}\cos\theta, d_{e-m}\sin\theta)\mathbf{R}_{moon}\,\mathbf{x}_{moon}$

# Hierarchical Transformation

- Sun – Earth – Moon
    - $\mathbf{x}_{sun} = \mathbf{T}_{earth}(d_{s-e}\cos\phi, d_{s-e}\sin\phi)\mathbf{R}_{earth}\,\mathbf{x}_{earth}$
    - $\mathbf{x}_{earth} = \mathbf{T}_{moon}(d_{e-m}\cos\theta, d_{e-m}\sin\theta)\mathbf{R}_{moon}\,\mathbf{x}_{moon}$

$\mathbf{R}_{moon}$

**Transformation of the Earth w.r.t. the frame of the Sun**
$$\mathbf{x}_{sun} = \mathbf{T}_{earth}(d_{s-e}\cos\phi, d_{s-e}\sin\phi)\mathbf{R}_{earth}\,\mathbf{x}_{earth}$$

**Transformation of the Moon w.r.t. the frame of the Sun**
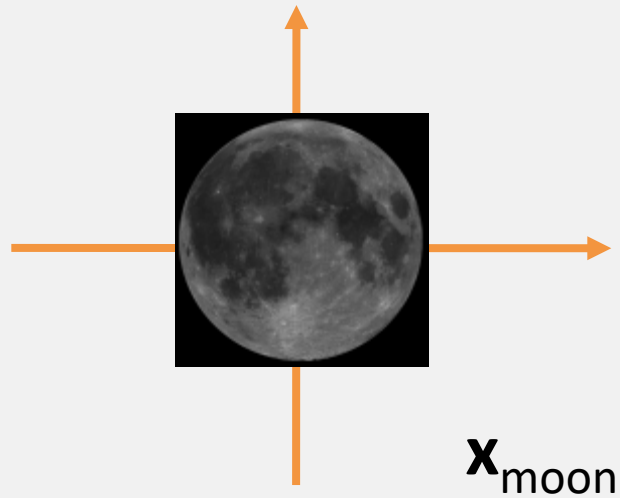$$\mathbf{x}_{sun} = \mathbf{T}_{earth}(d_{s-e}\cos\phi, d_{s-e}\sin\phi)\mathbf{R}_{earth}\mathbf{T}_{moon}(d_{e-m}\cos\theta, d_{e-m}\sin\theta)\mathbf{R}_{moon}\,\mathbf{x}_{moon}$$
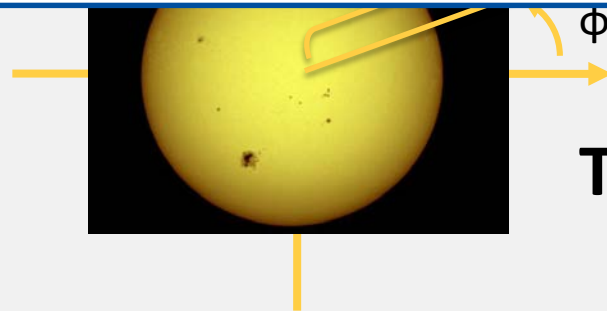
$\phi$

$\mathbf{T}_{earth}(d_{s-e}\cos\phi, d_{s-e}\sin\phi)$

# OpenGL ES 1.x Codes – Hierarchical Transformation

- Moon
  - Modeling the moon



$\mathbf{x}_{moon}$

```
void draw_moon()
{
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_COLOR_ARRAY);
  glVertexPointer(…);
  glColorPointer(…);
  glDrawElements(…);
  glDiableClientState(GL_VERTEX_ARRAY);
  glDiableClientState(GL_COLOR_ARRAY);
}
```

# OpenGL ES 1.x Codes – Hierarchical Transformation

- Moon
  - Modeling the moon
  - Rotating itself
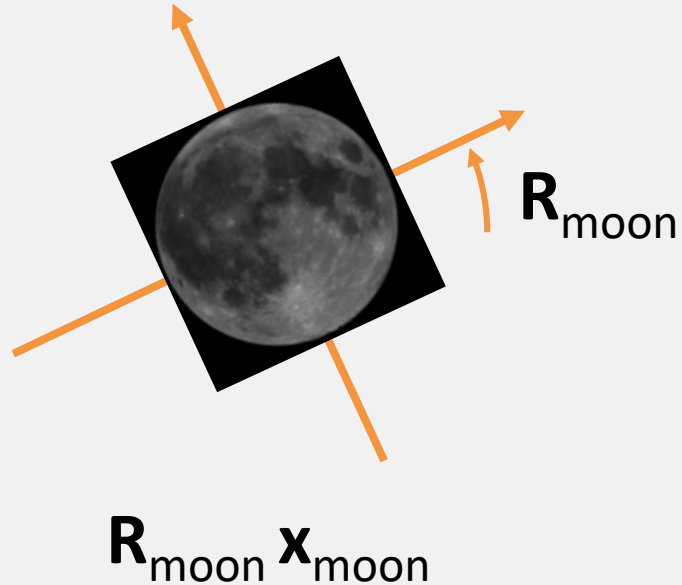
```
void draw_earth_system()
{
  // ...
  glRotatef(…);      // rotating the Moon
  draw_moon();
}

void draw_moon()    { // … glDrawElements(); … }
```



$R_{moon}$

$R_{moon} x_{moon}$

# OpenGL ES 1.x Codes – Hierarchical Transformation

- Earth – Moon
  - The moon is orbiting around the earth

$$\mathbf{x}_{earth} = \mathbf{T}_{moon}(d_{e-m}\cos\theta, d_{e-m}\sin\theta)\mathbf{R}_{moon}\,\mathbf{x}_{moon}$$

```
void draw_earth_system()
{
  draw_earth();

  glTranslatef(…); // orbiting around the Earth
  glRotatef(…);    // rotating the Moon
  draw_moon();
}

void draw_moon()  { // … glDrawElements(); … }
void draw_earth() { // … glDrawElements(); … }
```



$d_{e-m}$

$\mathbf{R}_{moon}$

$\theta$

$\mathbf{T}_{moon}(d_{e-m}\cos\theta, d_{e-m}\sin\theta)$
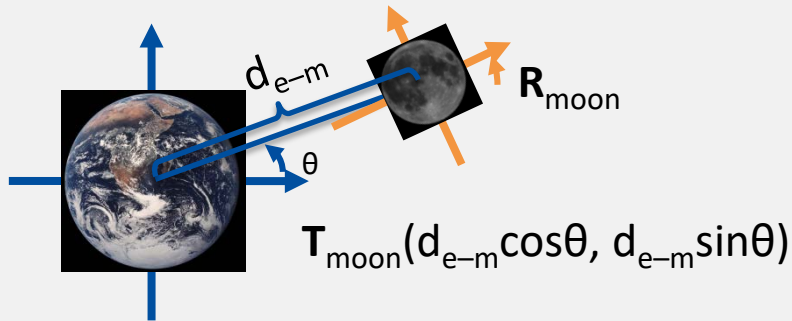
# OpenGL ES 1.x Codes – Hierarchical Transformation

- ## Sun – Earth
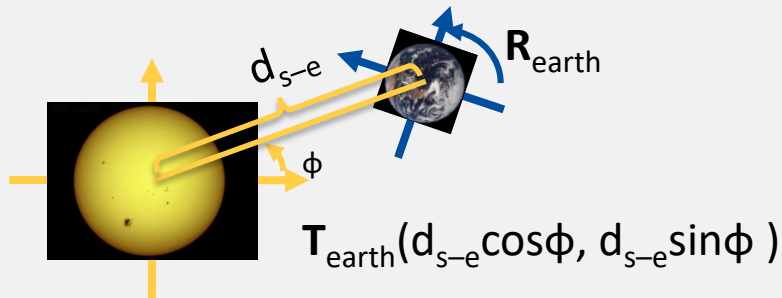  - The earth is orbiting around the sun

$$\mathbf{x}_{sun} = \mathbf{T}_{earth}(d_{s-e}\cos\phi, d_{s-e}\sin\phi)\mathbf{R}_{earth}\,\mathbf{x}_{earth}$$

```
void draw_sun_system()
{
  draw_sun();

  glTranslatef(…); // orbiting around the Sun
  glRotatef(…);    // rotating the Earth system
  draw_earth_system();
}

void draw_moon()   { // … glDrawElements(); … }
void draw_earth()  { // … glDrawElements(); … }
void draw_sun()    { // … glDrawElements(); … }
```



$d_{s-e}$　$\mathbf{R}_{earth}$　$\phi$

$$\mathbf{T}_{earth}(d_{s-e}\cos\phi, d_{s-e}\sin\phi)$$

# OpenGL ES 1.x Codes – Hierarchical Transformation

- Sun – Earth – Moon

$$\mathbf{x}_{earth}=\mathbf{T}_{moon}(d_{e-m}\cos\theta, d_{e-m}\sin\theta)\mathbf{R}_{moon}\,\mathbf{x}_{moon}$$

$$\mathbf{x}_{sun}=\mathbf{T}_{earth}(d_{s-e}\cos\phi, d_{s-e}\sin\phi)\mathbf{R}_{earth}\,\mathbf{x}_{earth}$$

$\mathbf{R}_{moon}$

$d_{e-m}$ $\theta$

$d_{s-e}$ $\mathbf{R}_{earth}$

$\phi$

$\mathbf{T}_{earth}(d_{s-e}\cos\phi, d_{s-e}\sin\phi )$

```c
void draw_sun_system()
{
  draw_sun();

  glTranslatef(…); // orbiting around the Sun
  glRotatef(…);    // rotating the Earth system
  draw_earth_system();
}

void draw_earth_system()
{
  draw_earth();

  glTranslatef(…); // orbiting around the Earth
  glRotatef(…);    // rotating the Moon
  draw_moon();
}

void draw_moon()   { // … glDrawElements(); … }
void draw_earth()  { // … glDrawElements(); … }
void draw_sun()    { // … glDrawElements(); … }
```
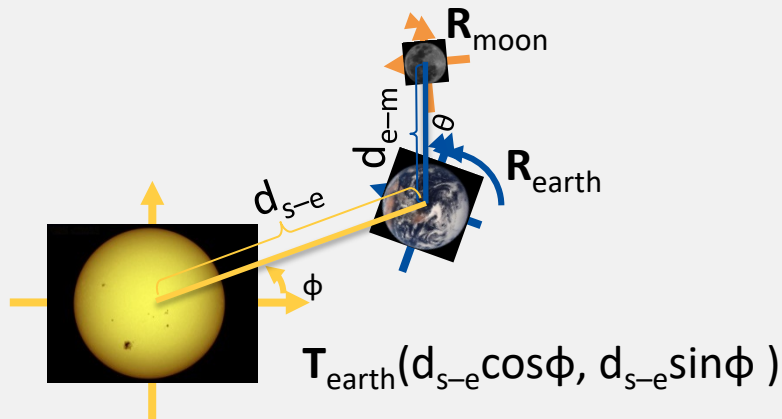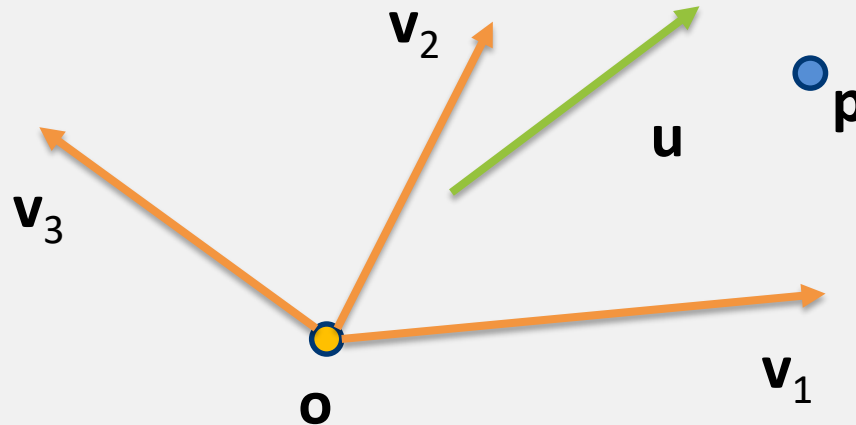
# View Transformations

# Coordinate System (Frame)

- A Coordinate system (or frame) consists of a set of basis vectors and an origin
  - A set of basis vectors:    $v_1, v_2, ..., v_n$
  - An origin:    $o$

- How to representing a vector **u** and a point **p** w.r.t. a given coordinate system?

# Coordinate System (Frame)

- Consider a set of basis vectors and an origin
  - $v_1$, $v_2$, …, $v_n$
  - $o$
- A vector **u** is written
  - $u = \alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n + 0 \cdot o$
  - The list of scalars, $\{\alpha_1, \alpha_2, \dots \alpha_n, 0\}$ is the representation of **u** w.r.t. the given coordinate system
- A point **p** is written
  - $p = \beta_1 v_1 + \beta_2 v_2 + \dots + \beta_n v_n + 1 \cdot o$
  - The list of scalars, $\{\beta_1, \beta_2, \dots \beta_n, 1\}$ is the representation of **p** w.r.t . the given coordinate system
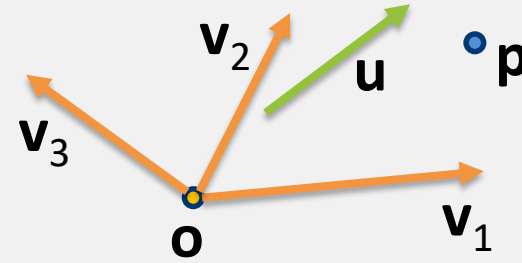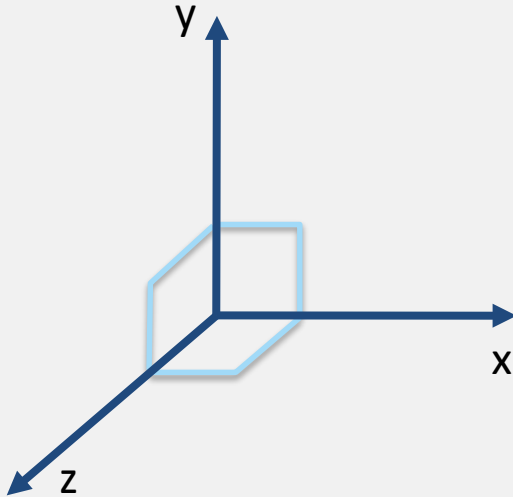
$$u = [\alpha_1 \quad \alpha_2 \quad \cdots \quad \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

$$p = [\beta_1 \quad \beta_2 \quad \cdots \quad \beta_n]^T = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$
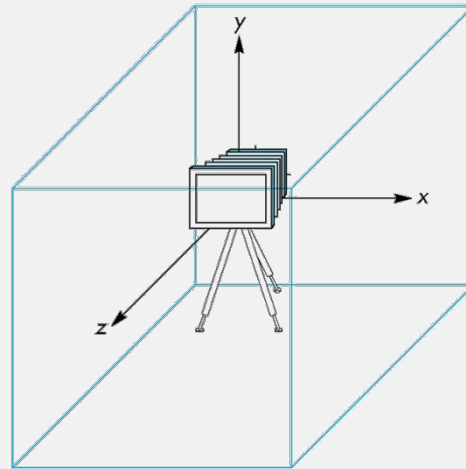
# Coordinate System (Frame)

- In OpenGL ES, we just care about **orthonormal** frames
  - **Ortho** means that the basis vectors are orthogonal to each other
    - x-axis $\perp$ y-axis $\perp$ z-axis
  - **normal** means that the length of each basis vector is 1
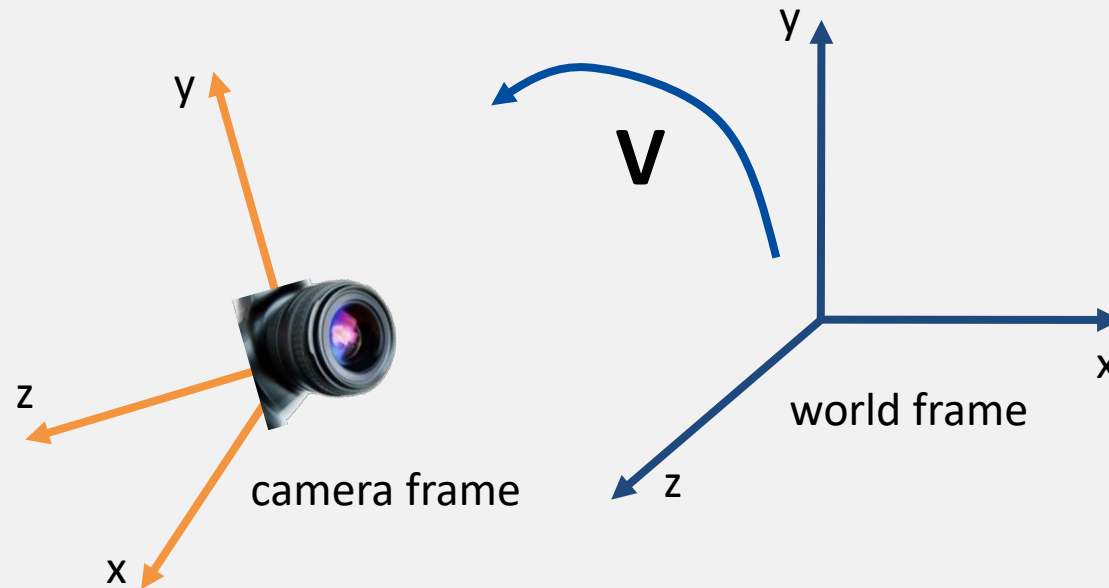    - The unit length of each axis is equal to 1

# Inside of gluLookAt()

- Initially, OpenGL ES camera coordinate is as follows
  - Center of projection (COP) is placed in the origin
  - Right direction is the positive direction of x-axis
  - Up direction is the positive direction of y-axis
  - Viewing direction is the negative direction of z-axis

# Inside of gluLookAt()

- When we apply gluLookAt()
  - Rotate the camera frame on the world frame: $\mathbf{R}_c$
  - Translate the camera frame on the world frame: $\mathbf{T}_c$
  - Therefore, $\mathbf{V} = \mathbf{T}_c\mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1}\mathbf{T}_c^{-1}$



**V**

camera frame

world frame

# Inside of gluLookAt()

- When we apply gluLookAt()
  - Rotate the camera frame on the world frame: $\mathbf{R}_c$
  - Translate the camera frame on the world frame: $\mathbf{T}_c$
  - Therefore, $\mathbf{V} = \mathbf{T}_c\mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1}\mathbf{T}_c^{-1}$



$$\mathbf{R}_c = \begin{bmatrix} & & & 0 \\ & & & 0 \\ & & & 0 \\ & & & 1 \end{bmatrix}$$
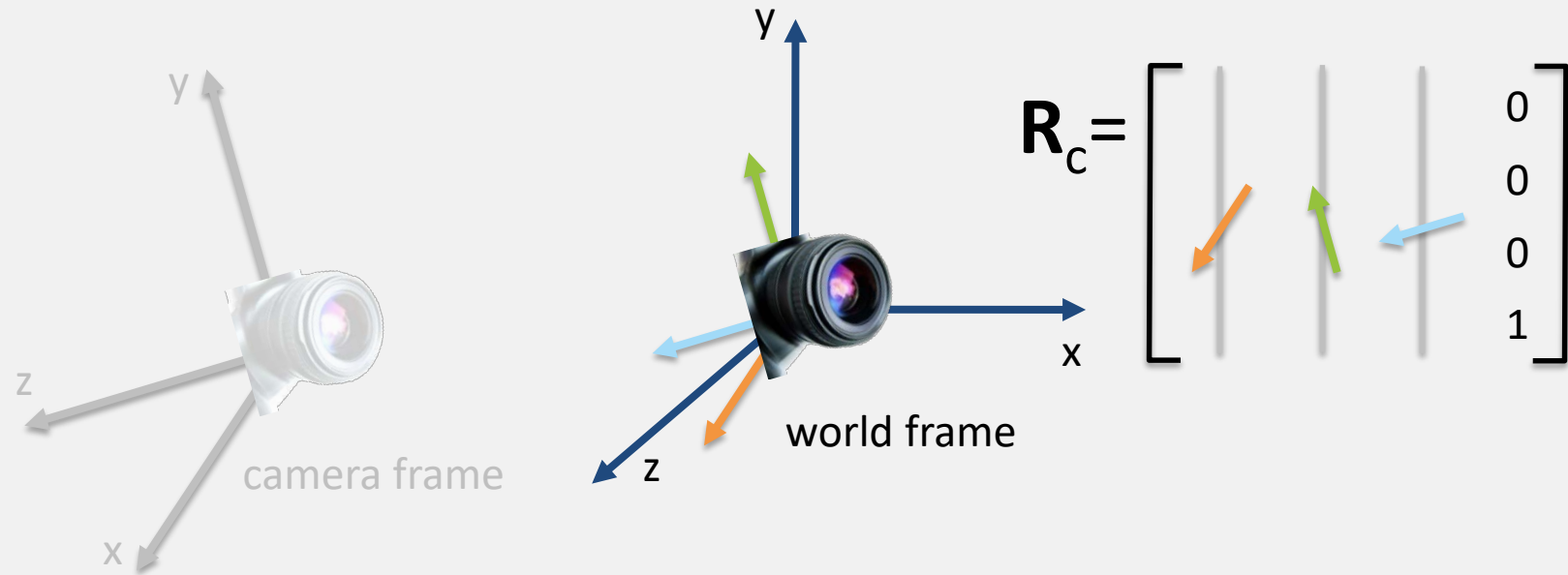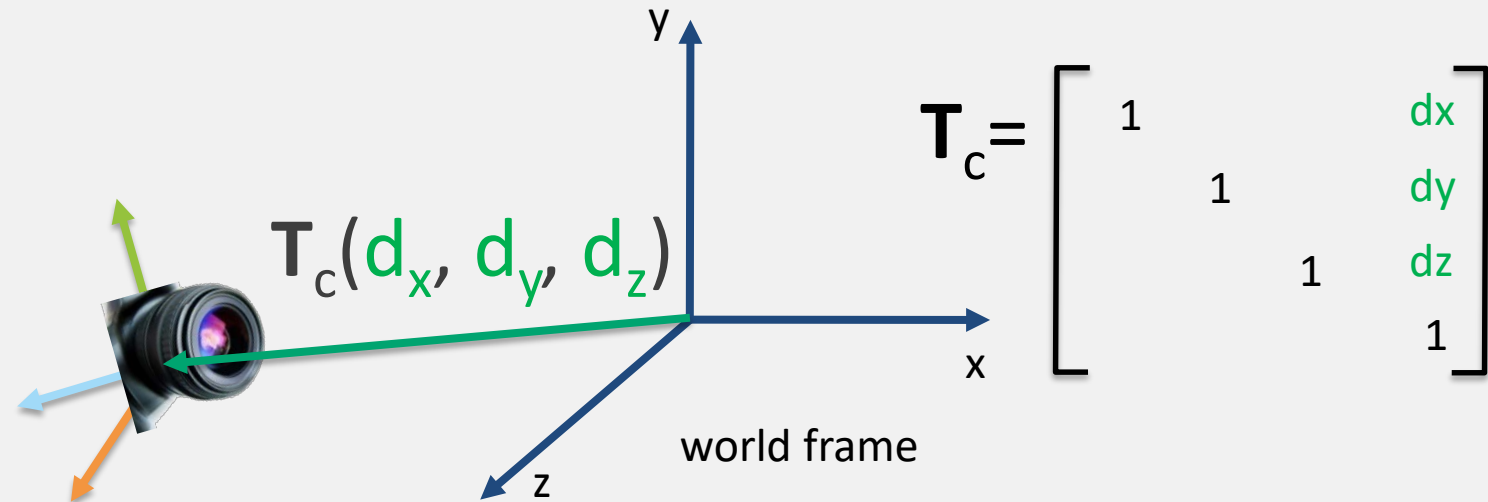
# Inside of gluLookAt()

- When we apply gluLookAt()
  - Rotate the camera frame on the world frame: $\mathbf{R}_c$
  - Translate the camera frame on the world frame: $\mathbf{T}_c$
  - Therefore, $\mathbf{V} = \mathbf{T}_c\mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1}\mathbf{T}_c^{-1}$

$$\mathbf{T}_c = \begin{bmatrix} 1 & & & dx \\ & 1 & & dy \\ & & 1 & dz \\ & & & 1 \end{bmatrix}$$

$\mathbf{T}_c(d_x, d_y, d_z)$

world frame

# Inside of [gluLookAt]()

- When we apply [gluLookAt]()
  - Rotate the camera frame on the world frame: $\mathbf{R}_c$
  - Translate the camera frame on the world frame: $\mathbf{T}_c$
  - Therefore, $\mathbf{V} = \mathbf{T}_c\mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1}\mathbf{T}_c^{-1}$
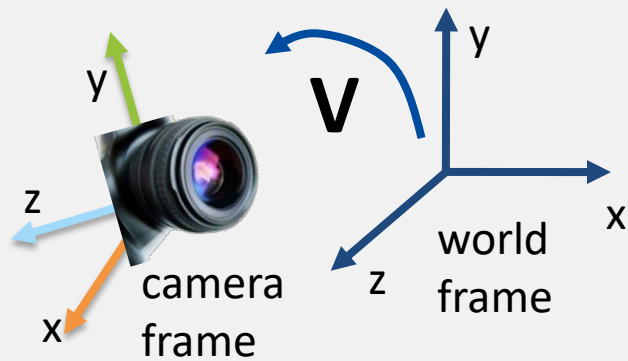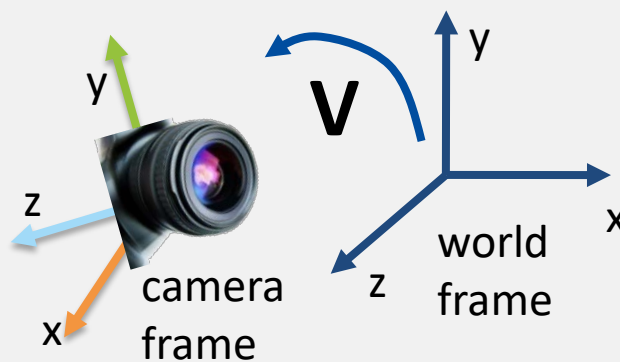


$$\mathbf{V=T_cR_c}$$

$$= \begin{bmatrix} 1 & & & dx \\ & 1 & & dy \\ & & 1 & dz \\ & & & 1 \end{bmatrix} \begin{bmatrix} & & & 0 \\ & & & 0 \\ & & & 0 \\ & & & 1 \end{bmatrix}$$

# Inside of gluLookAt()

- When we apply gluLookAt()
  - Rotate the camera frame on the world frame: $R_c$
  - Translate the camera frame on the world frame: $T_c$
  - Therefore, $V = T_c R_c$ and gluLookAt() generates $V^{-1} = R_c^{-1} T_c^{-1}$



$$V^{-1} = R_c^{-1} T_c^{-1}$$

# Revisiting
# Linear Transformations

# Translation (이동)

- **Translation matrix T**
  - Translation can also be expressed by using a 4x4 matrix **T** in homogeneous coordinates
  - **x'=Tx**
    - **T** = **T**($d_x$, $d_y$, $d_z$) =

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- glTranslate($d_x$, $d_y$, $d_z$)
- This form is better for H/W implementation
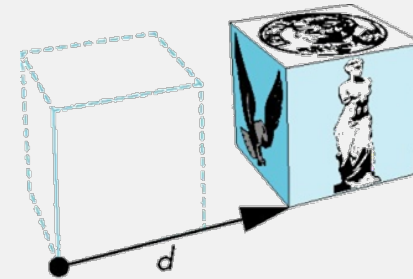
# Translation (이동) 행렬의 해석: Homogeneous Coordinate 활용

$$\mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{3 \times 3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix} = \mathbf{T}(\mathbf{d}) = \mathbf{T_d}$$

- 단, $\mathbf{d} = (d_x, d_y, d_z)$.

# Translation about Points or Vectors

**Translation about Points**

- 위치 $\mathbf{p} = (p_x, p_y, p_z)$를 $\mathbf{d} = (d_x, d_y, d_z)$ 만큼 이동
  - Homogeneous coordinate 이용
  $$\mathbf{T_d} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{p} + \mathbf{d} \\ 1 \end{bmatrix}$$

  - Homogeneous coordinate 이용치 않은 경우
    - $\mathbf{p} + \mathbf{d}$

**Translation about Vectors**

- 벡터 $\mathbf{v} = (v_x, v_y, v_z)$를 $\mathbf{d} = (d_x, d_y, d_z)$ 만큼 이동
  - Homogeneous coordinate 이용한 경우
  $$\mathbf{T_d} \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix}$$
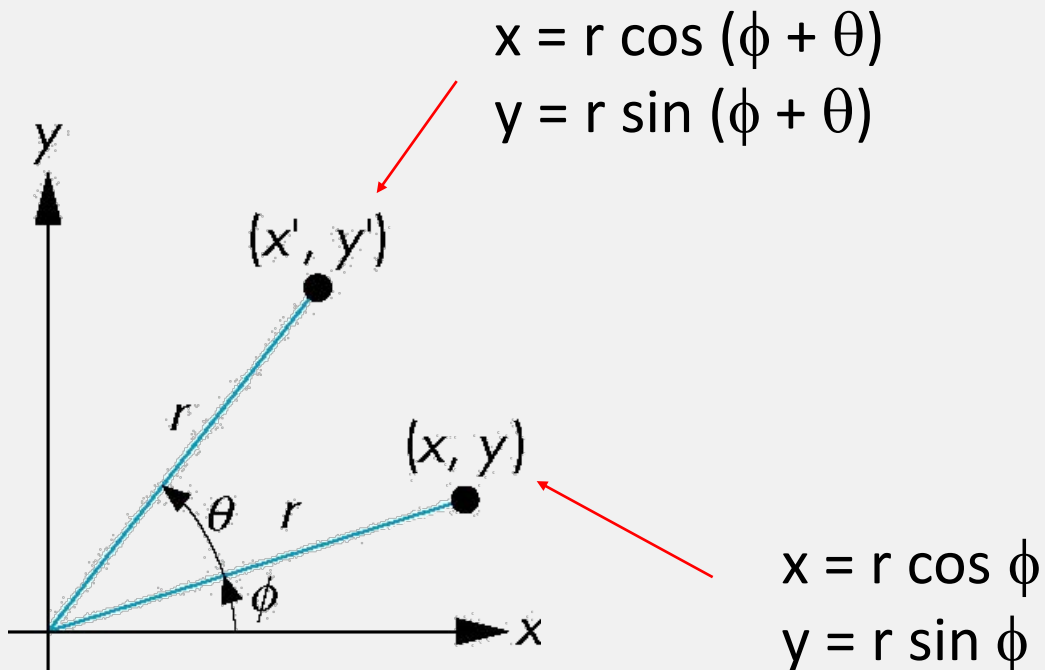
  - Homogeneous coordinate 이용치 않은 경우
    - $\mathbf{v}$
      - 즉, 벡터 $\mathbf{v}$는 이동 $\mathbf{d}$에 영향이 없음

# Rotation (회전)
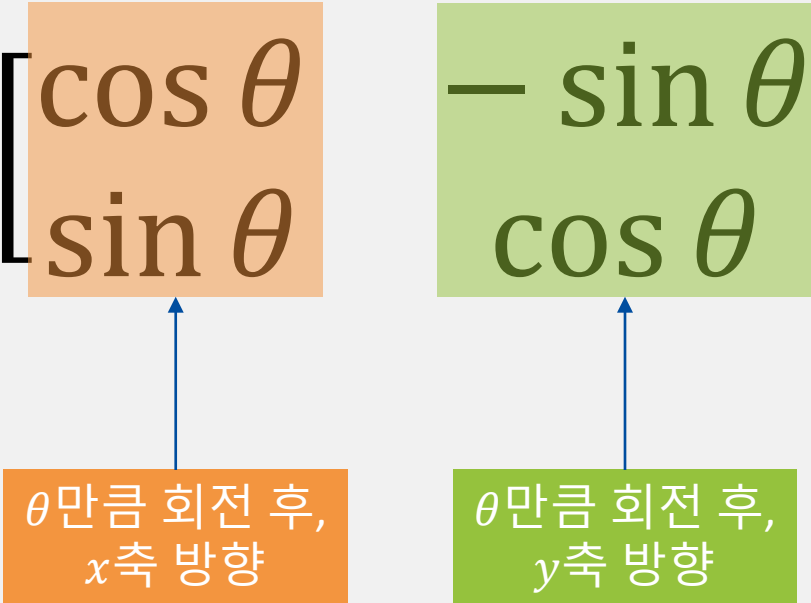
- Consider rotation about the origin by $\theta$ degrees
  - Radius stays the same, angle increases by $\theta$

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

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

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

# Rotation (회전) 행렬의 해석

- 2차원 회전 행렬 해석
  - 고교과정에서 배운 내용에 대한 재해석

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

$\theta$만큼 회전 후, $x$축 방향

$\theta$만큼 회전 후, $y$축 방향

# Rotation (회전) 행렬의 해석: Homogeneous Coordinate 활용

- 2차원 회전 행렬 해석
  - Homogeneous coordinate를 이용한 해석

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}(\theta)_{2\times2} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

$\theta$ 만큼 회전 후, $x$ 축 방향

$\theta$ 만큼 회전 후, $y$ 축 방향

원점

# Rotation (회전) 행렬의 해석: Homogeneous Coordinate 활용

**2D Rotation of Points**

- 위치 $\mathbf{p} = (p_x, p_y)$를 $\theta$만큼 회전
  - Homogeneous coordinate를 이용한 해석

$$\mathbf{R}(\theta)\begin{bmatrix}\mathbf{p}\\1\end{bmatrix} = \begin{bmatrix}\mathbf{R}(\theta)_{2\times2} & \mathbf{0}\\\mathbf{0}^T & 1\end{bmatrix}\begin{bmatrix}\mathbf{p}\\1\end{bmatrix} = \begin{bmatrix}\mathbf{R}(\theta)_{2\times2}\mathbf{p}\\1\end{bmatrix}$$

  - Homogeneous coordinate 이용치 않은 경우
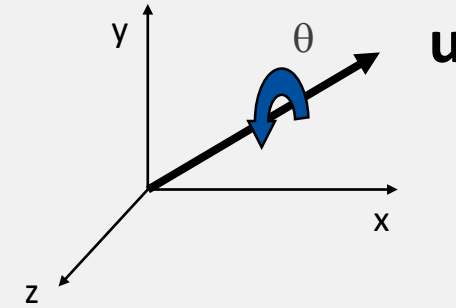    - $\mathbf{R}(\theta)_{2\times2}\mathbf{p}$

**2D Rotation of Vectors**

- 벡터 $\mathbf{v} = (v_x, v_y)$를 $\theta$만큼 회전
  - Homogeneous coordinate를 이용한 해석

$$\mathbf{R}(\theta)\begin{bmatrix}\mathbf{v}\\0\end{bmatrix} = \begin{bmatrix}\mathbf{R}(\theta)_{2\times2} & \mathbf{0}\\\mathbf{0}^T & 1\end{bmatrix}\begin{bmatrix}\mathbf{v}\\0\end{bmatrix} = \begin{bmatrix}\mathbf{R}(\theta)_{2\times2}\mathbf{v}\\0\end{bmatrix}$$

  - Homogeneous coordinate 이용치 않은 경우
    - $\mathbf{R}(\theta)_{2\times2}\mathbf{v}$

# Rotation (회전)

- General rotation about the origin
  - A rotation by $\theta$ rotation an arbitrary axis **u**
    - Quaternion can express general rotation
  - **x'**=**R$_u$($\theta$)x**
    - **where,**

$$\mathbf{R_u}(\theta) = \begin{bmatrix} \cos\theta + u_x^2(1-cos\theta) & u_xu_y(1-cos\theta)-u_zsin\theta & u_xu_z(1-cos\theta)+u_ysin\theta & 0 \\ u_yu_x(1-cos\theta)+u_zsin\theta & \cos\theta+u_y^2(1-cos\theta) & u_yu_z(1-cos\theta)-u_xsin\theta & 0 \\ u_zu_x(1-cos\theta)-u_ysin\theta & u_zu_y(1-cos\theta)+u_xsin\theta & \cos\theta+u_z^2(1-cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R_u}(\theta)_{3\times3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

**u**축을 중심으로 $\theta$만큼 회전 후, $x$축 방향

**u**축을 중심으로 $\theta$만큼 회전 후, $y$축 방향

**u**축을 중심으로 $\theta$만큼 회전 후, $z$축 방향

원점

- glRotate($\theta$, u$_x$, u$_y$, u$_z$), where **u**=(u$_x$, u$_y$, u$_z$)

# Rotation (회전) 행렬의 해석: Homogeneous Coordinate 활용

**3D Rotation of Points**

- 위치 $\mathbf{p} = (p_x, p_y, p_z)$를
  $\mathbf{u}$축을 중심으로 $\theta$만큼 회전

  – Homogeneous coordinate를 이용한 해석

$$\mathbf{R_u}(\theta) \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R_u}(\theta)_{3\times3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R_u}(\theta)_{3\times3}\mathbf{p} \\ 1 \end{bmatrix}$$

  – Homogeneous coordinate 이용치 않은 경우
    - $\mathbf{R_u}(\theta)_{3\times3}\mathbf{p}$

**3D Rotation of Vectors**

- 벡터 $\mathbf{v} = (v_x, v_y, v_z)$를
  $\mathbf{u}$축을 중심으로 $\theta$만큼 회전

  – Homogeneous coordinate를 이용한 해석

$$\mathbf{R_u}(\theta) \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R_u}(\theta)_{3\times3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R_u}(\theta)_{3\times3}\mathbf{v} \\ 0 \end{bmatrix}$$

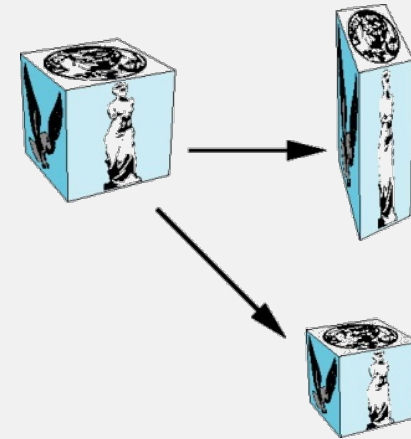  – Homogeneous coordinate 이용치 않은 경우
    - $\mathbf{R_u}(\theta)_{3\times3}\mathbf{v}$

# Scaling (확대축소)

- ## Scaling matrix **S**
  - Expand or contract along each axis (fixed point of <span style="color:red">origin</span>)
  - **x'=Sx**
    - **S** = **S**($s_x$, $s_y$, $s_z$) =

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



  - glScale($s_x$, $s_y$, $s_z$)

# Scale (확대축소) 행렬의 해석: Homogeneous Coordinate 활용

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{S}_{3\times3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \mathbf{S}(\mathbf{s}) = \mathbf{S_s}$$

- 단, $\mathbf{s} = (s_x, s_y, s_z)$.

# Revisiting
# Composite Transformation

# Composite Transformation (합성 변환)

- We can composite transformation by multiplying matrices of rotation, translation, and scaling.
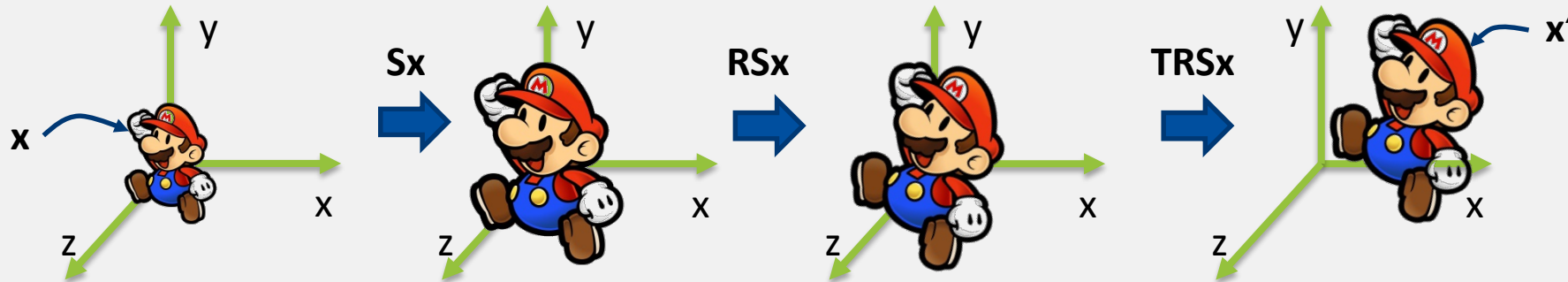  - Example:
    1) Uniformly scale 2x:
    2) Rotate -30 degrees by +z axis:
    3) Translate by (2, 1, 0):
       - $x' = T(R(Sp)) = TRSx$
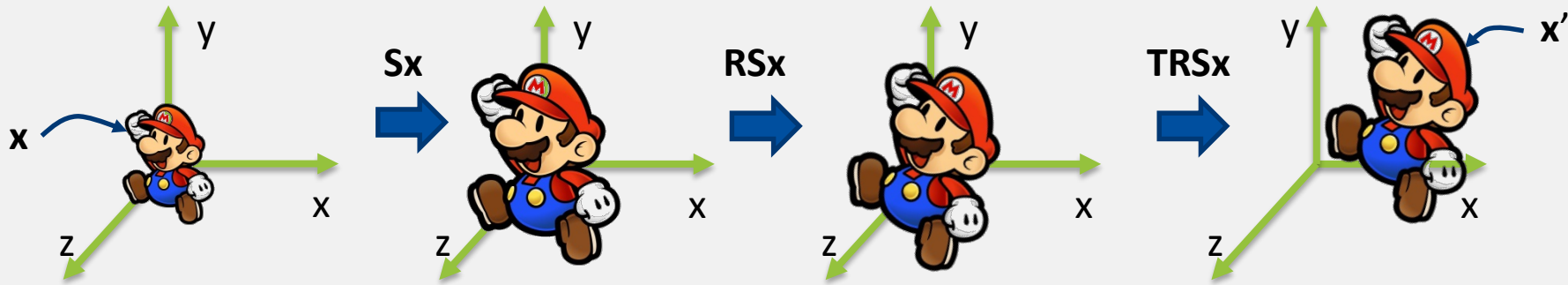         - x: original object
         - x': transformed object

```
glTranslatef(2, 1, 0);          // T
glRotatef(-30, 0, 0, 1);        // R
glScalef(2, 2, 2);              // S
glDrawElements(…);              // x
```

# Composite Transformation (합성 변환)

- Composite transformation w/ "scale, rotation, and then translation"



$$\mathbf{T_d R_u}(\theta)\mathbf{S_s} = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R_u}(\theta)_{3\times3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{S}_{3\times3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{SR_u}(\theta)_{3\times3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix}$$
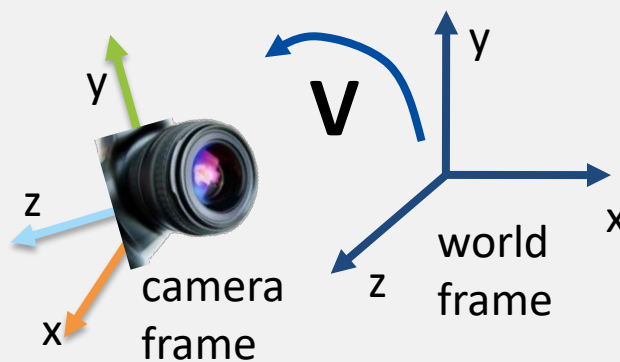
# Inside of gluLookAt()

- When we apply gluLookAt()
  - Rotate the camera frame on the world frame: $\mathbf{R}_c$
  - Translate the camera frame on the world frame: $\mathbf{T}_c$
  - Therefore, $\mathbf{V} = \mathbf{T}_c\mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1}\mathbf{T}_c^{-1}$
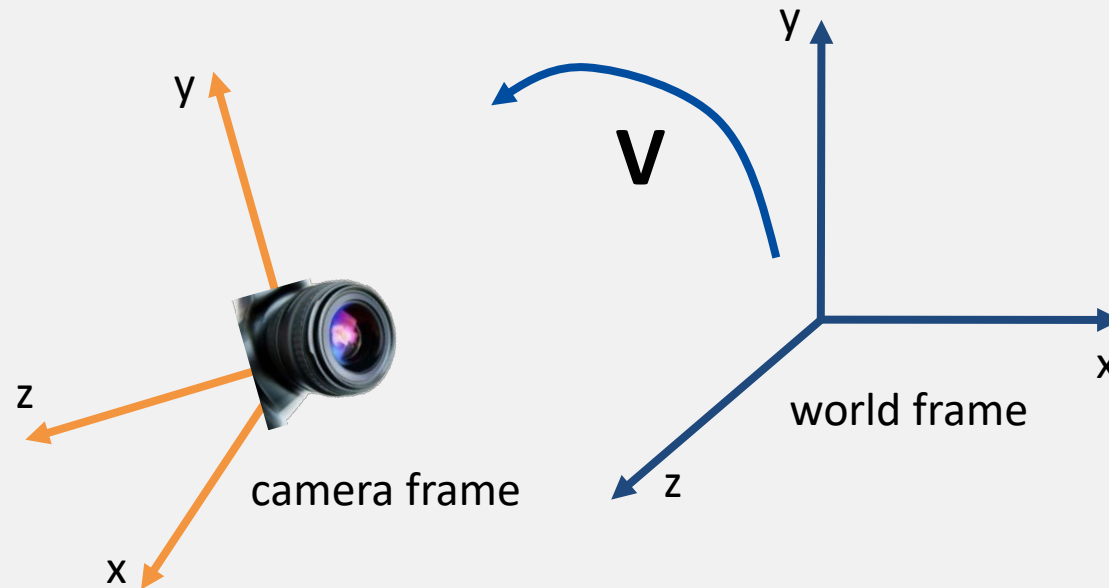
# Inside of gluLookAt()

- When we apply gluLookAt()
  - Rotate the camera frame on the world frame: $R_c$
  - Translate the camera frame on the world frame: $T_c$
  - Therefore, $V = T_c R_c$ and gluLookAt() generates $V^{-1} = R_c^{-1} T_c^{-1}$



$$V^{-1} = R_c^{-1} T_c^{-1}$$

# Inside of [gluLookAt]()

- ViewMatrix $V^{-1}$

$$\mathbf{V} = \mathbf{T_d}\mathbf{R_u}(\theta) = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R_u}(\theta)_{3\times3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R_u}(\theta)_{3\times3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

$$\mathbf{V}^{-1} = \mathbf{R_u}^{-1}(\theta)\mathbf{T_d}^{-1} = \mathbf{R_u}^T(\theta)\mathbf{T_d}^{-1} = \begin{bmatrix} \mathbf{R_u}^T(\theta)_{3\times3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I}_{3\times3} & -\mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{R_u}^T(\theta)_{3\times3} & -\mathbf{R_u}^T(\theta)_{3\times3}\mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix}$$