
Computer Graphics

3 - Transformation 1

Yoonsang Lee
Spring 2022

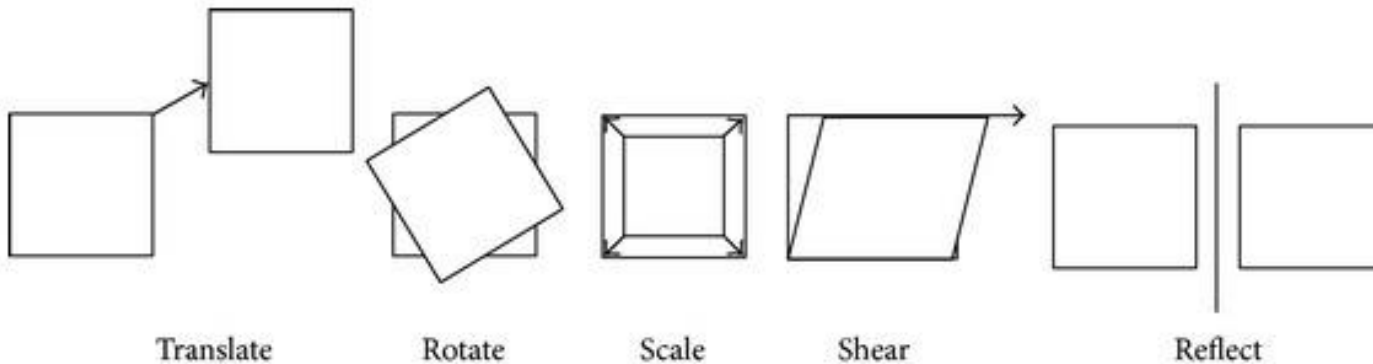
Topics Covered

- 2D Transformation
 - Scale, rotation, translation...
- Composing Transformations & Homogeneous Coordinates
- 3D Cartesian Coordinate System

2D Transformations

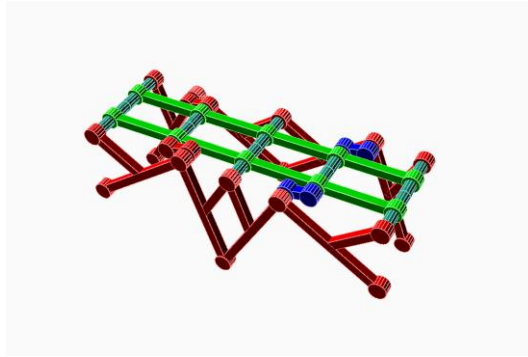
What is Transformation?

- Geometric **Transformation** - 기하 변환
 - One-to-one mapping (function) of a set having some geometric structure to itself or another such set.
 - More easily, “*moving a set of points*”
- Examples:

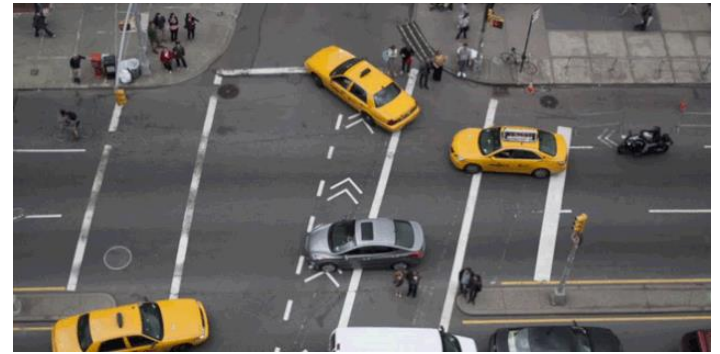


Where are Transformations used?

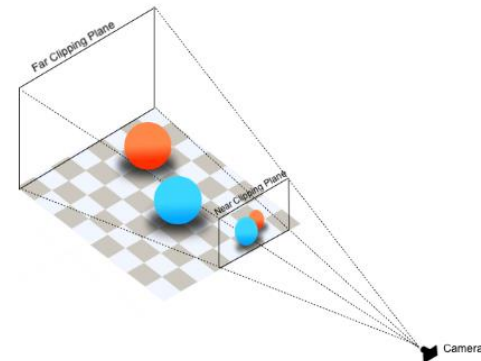
- Movement



- Image/object manipulation



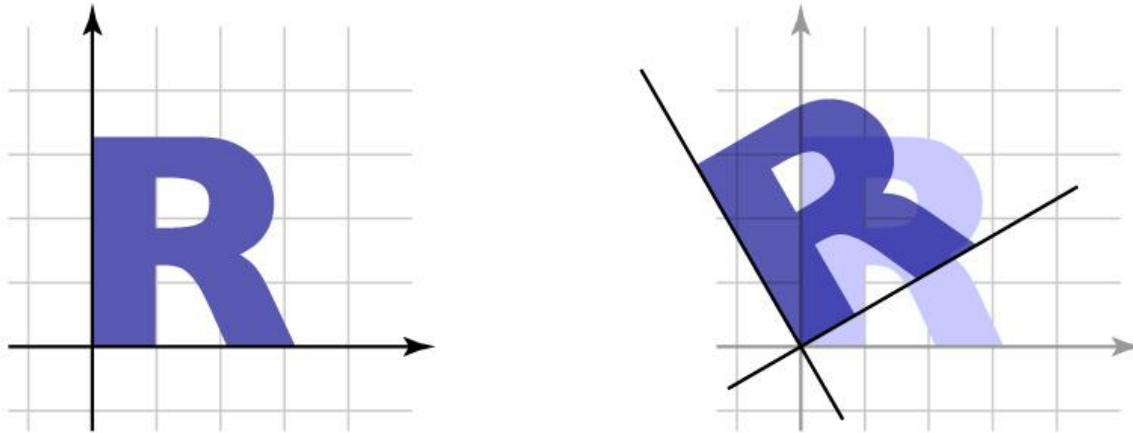
- Viewing, projection transform



Transformation

- “Moving a set of points”
 - Transformation T maps any input vector \mathbf{v} in the vector space S to $T(\mathbf{v})$.

$$S \rightarrow \{T(\mathbf{v}) \mid \mathbf{v} \in S\}$$



Linear Transformation

- One way to define a transformation is by matrix multiplication:

$$T(\mathbf{v}) = M\mathbf{v}$$

- This is called a **linear transformation** because a matrix multiplication represents a linear mapping.

$$T(a\mathbf{u} + \mathbf{v}) = aT(\mathbf{u}) + T(\mathbf{v})$$

$$M \cdot (a\mathbf{u} + \mathbf{v}) = aM\mathbf{u} + M\mathbf{v}$$

2D Linear Transformation

- 2x2 matrices represent 2D linear transformations such as:
 - uniform scaling
 - non-uniform scaling
 - rotation
 - shearing
 - reflection

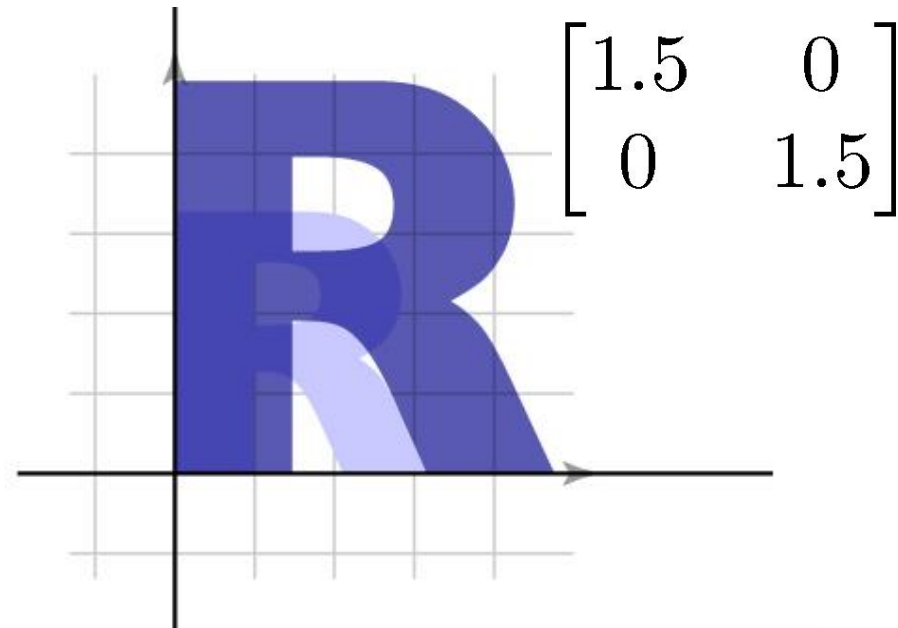
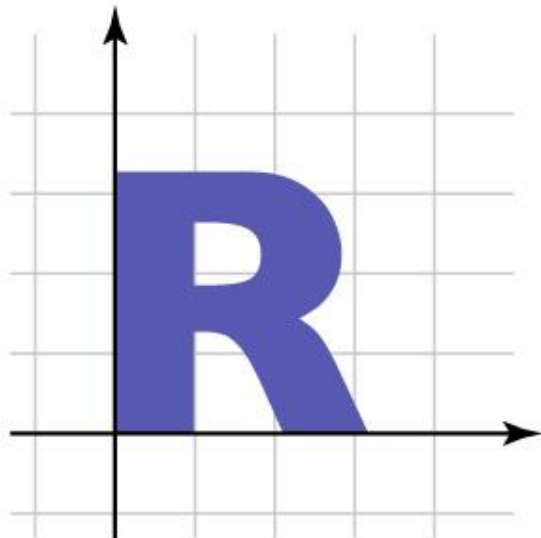
2D Linear Trans. – Uniform Scale

- Uniformly shrinks or enlarges both in x and y directions.

2x2 scale matrix S

$$\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sx \\ sy \end{bmatrix}$$

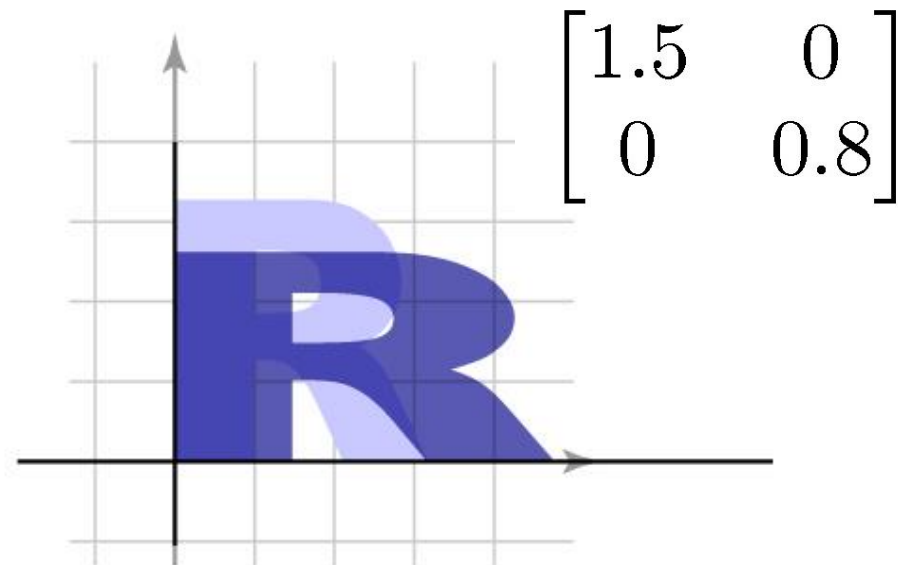
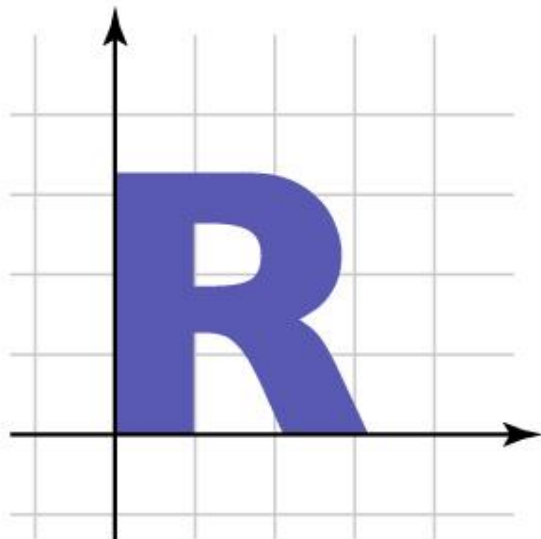
$p = p'$



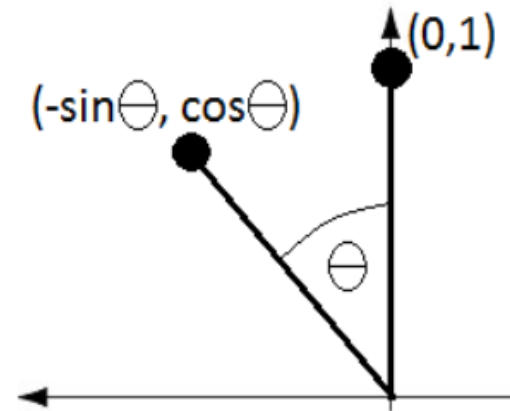
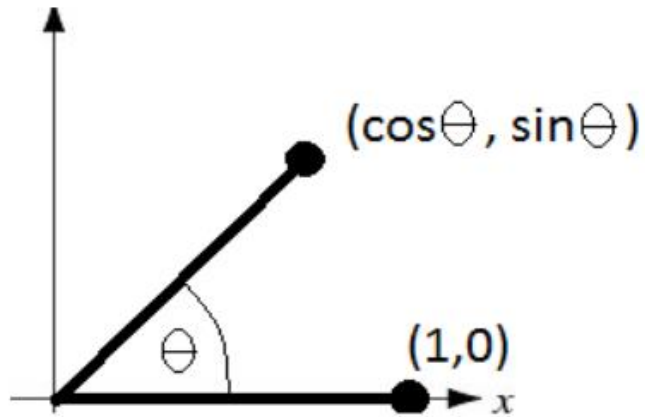
2D Linear Trans. – Nonuniform Scale

- Non-uniformly shrinks or enlarges in x and y directions.

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}$$



Rotation

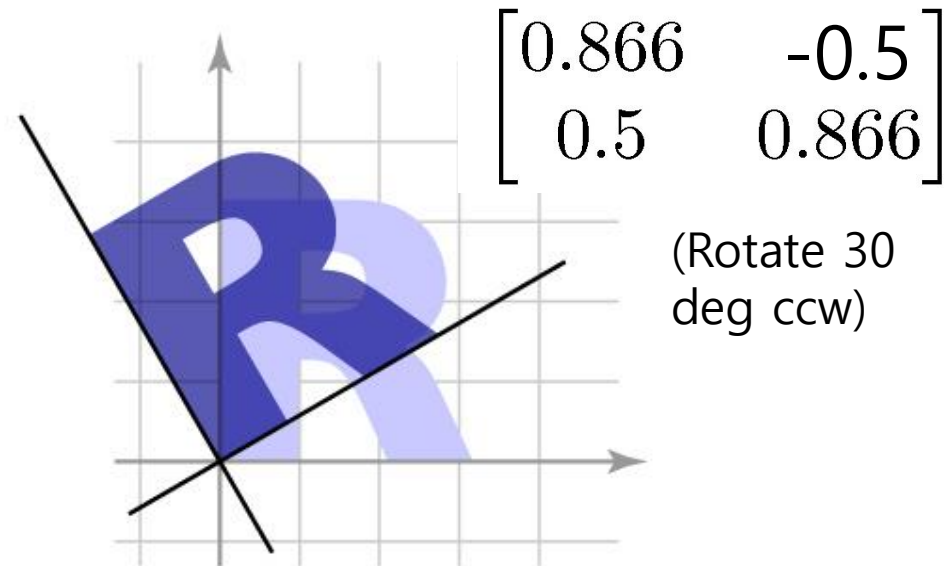
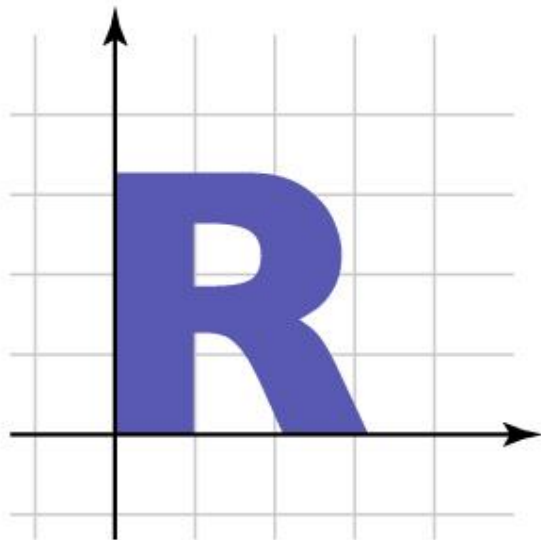


$$\Rightarrow R_{\theta} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \quad : \text{Rotation matrix}$$

2D Linear Trans. – Rotation

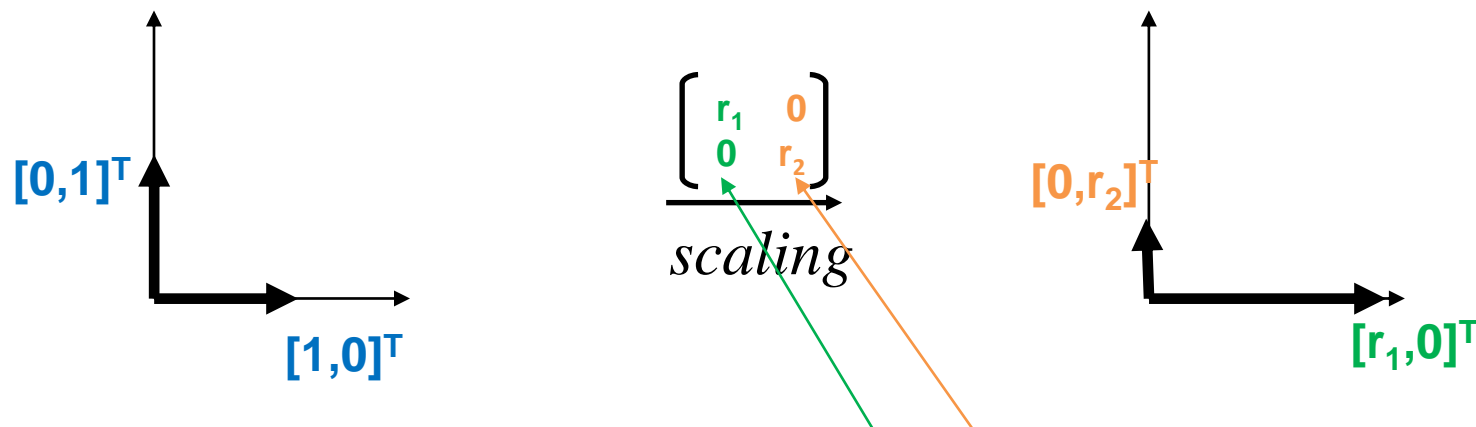
- Rotation can be written in matrix multiplication, so it's also a linear transformation.
 - Note that positive angle means CCW rotation.

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



Numbers in Matrices: Scale, Rotation

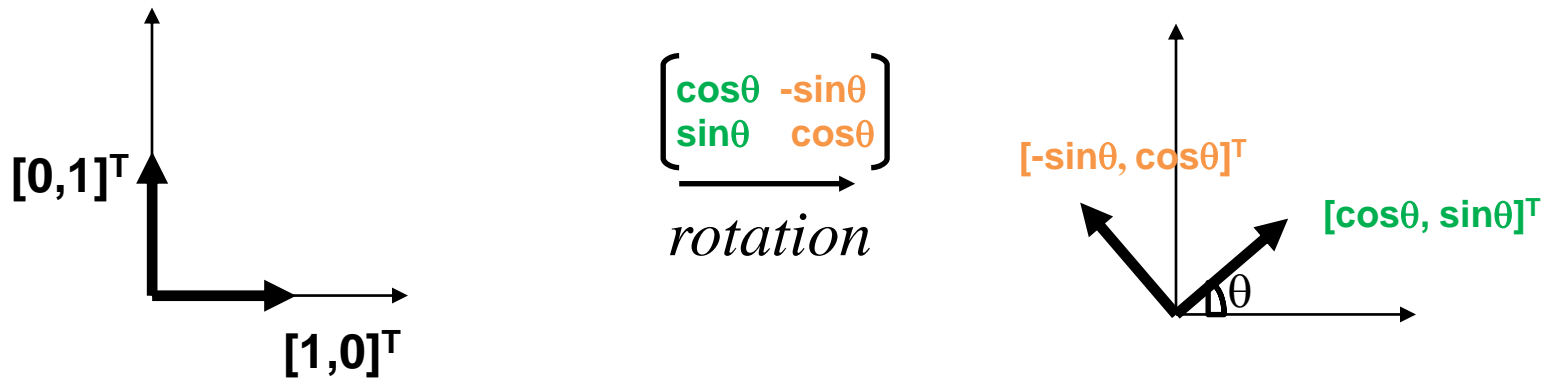
- Let's think about what the numbers in the matrix means.



Canonical basis vectors: unit vectors pointing in the direction of the axes of a Cartesian coordinate system.

1st & 2nd basis vector of the transformed coordinates

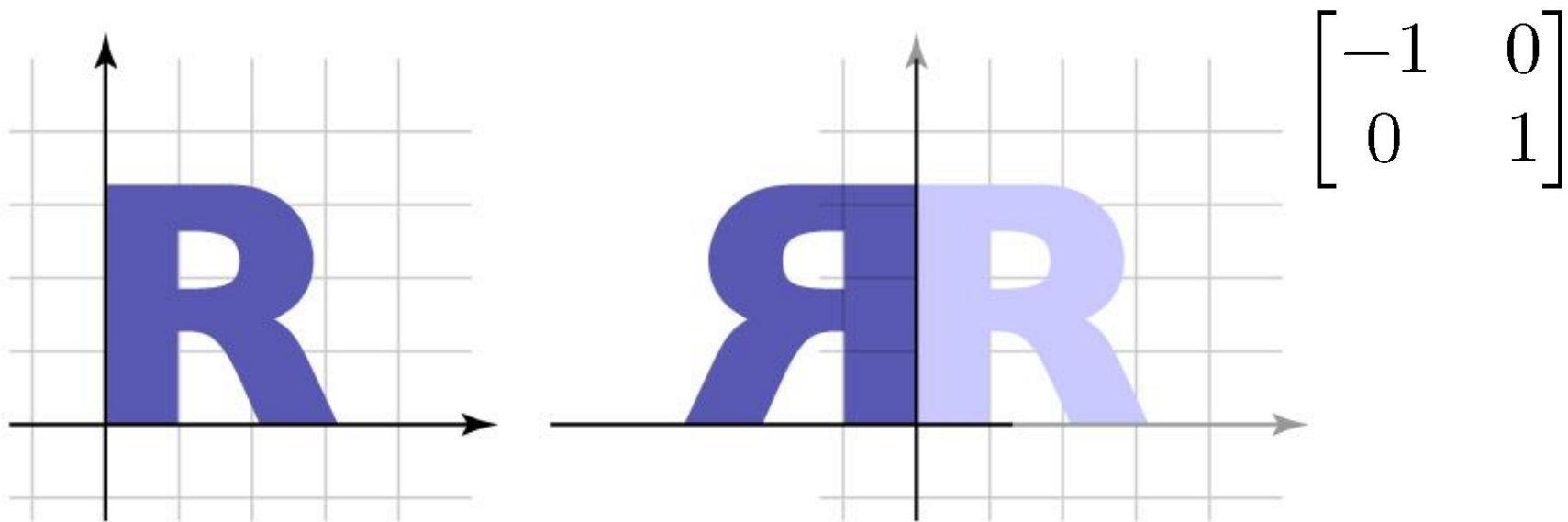
Numbers in Matrices: Scale, Rotation



- *Column vectors* of a matrix is the *basis vectors* of the *column space (range)* of the matrix.
 - *Column space* of a matrix: The span (a set of all possible linear combinations) of its column vectors.

2D Linear Trans. – Reflection

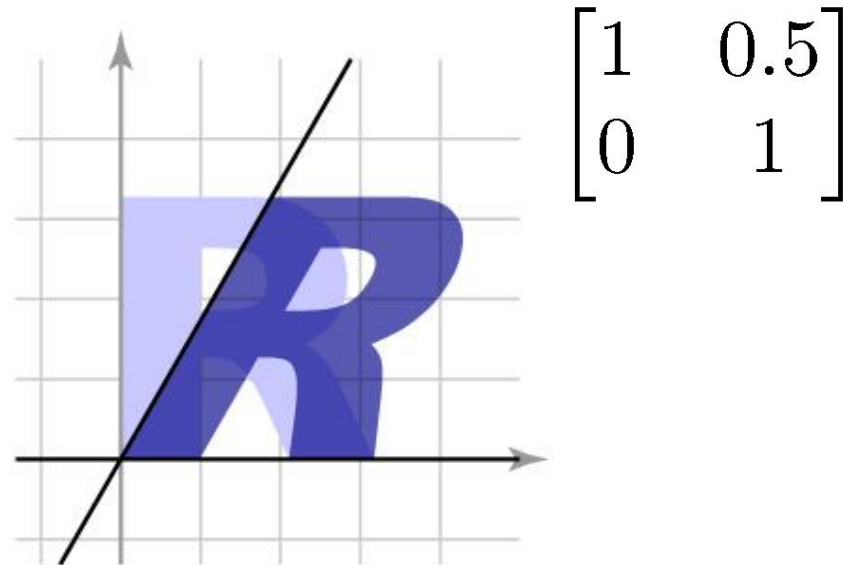
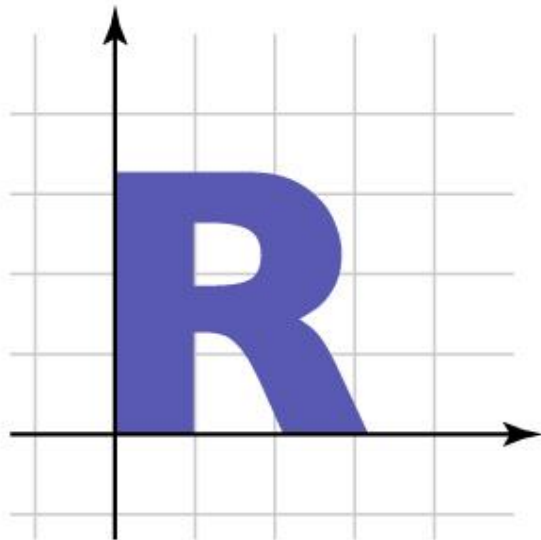
- Reflection can be considered as a special case of non-uniform scale.



2D Linear Trans. – Shear

- "Push things sideways"

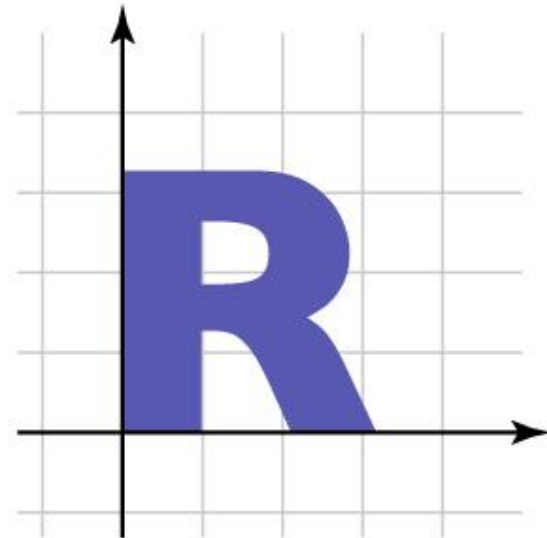
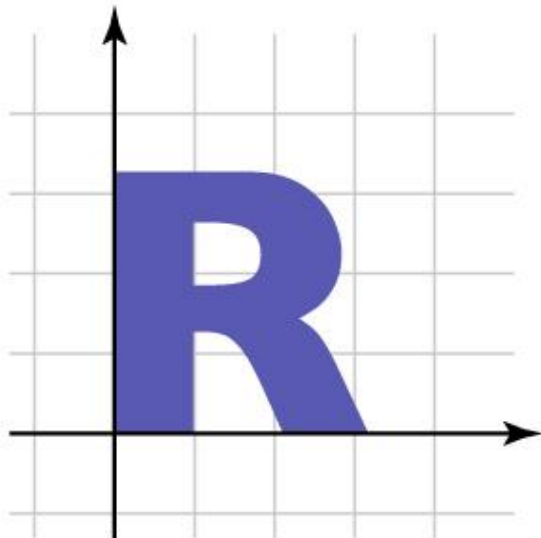
$$\begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + ay \\ y \end{bmatrix}$$



Identity Matrix

- "Doing nothing"

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



[Practice]

Uniform Scale

```
import glfw
from OpenGL.GL import *
import numpy as np

def render(M):
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()

    # draw coordinate
    glBegin(GL_LINES)
    glColor3ub(255, 0, 0)
    glVertex2fv(np.array([0.,0.]))
    glVertex2fv(np.array([1.,0.]))
    glColor3ub(0, 255, 0)
    glVertex2fv(np.array([0.,0.]))
    glVertex2fv(np.array([0.,1.]))
    glEnd()

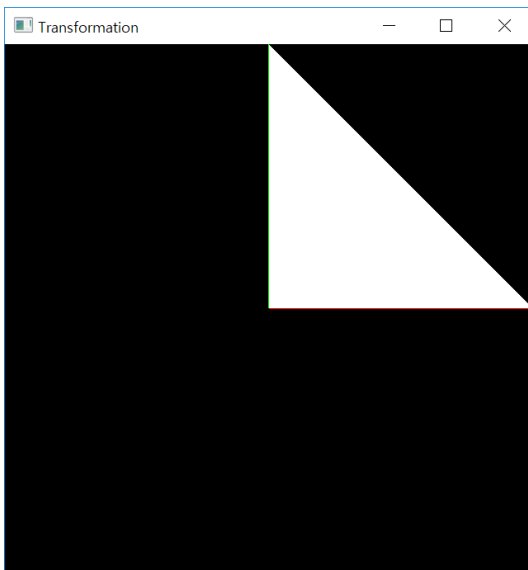
    # draw triangle - p'=Mp
    glBegin(GL_TRIANGLES)
    glColor3ub(255, 255, 255)
    glVertex2fv(M @ np.array([0.0,0.5]))
    glVertex2fv(M @ np.array([0.0,0.0]))
    glVertex2fv(M @ np.array([0.5,0.0]))
    glEnd()
```

[Practice] Uniform Scale

```
def main():  
    if not glfw.init():  
        return  
    window = glfw.create_window(640, 640, "2D  
Trans", None, None)  
    if not window:  
        glfw.terminate()  
        return  
    glfw.make_context_current(window)
```

```
while not glfw.window_should_close(window):  
    glfw.poll_events()  
  
    M = np.array([[2., 0.],  
                  [0., 2.]])  
    render(M)  
    glfw.swap_buffers(window)  
  
    glfw.terminate()
```

```
if __name__ == "__main__":  
    main()
```



[Practice] Animate It!

```
def main():
    if not glfw.init():
        return
    window = glfw.create_window(640, 640, "2D Trans", None, None)
    if not window:
        glfw.terminate()
        return
    glfw.make_context_current(window)

    # set the number of screen refresh to wait before calling glfw.swap_buffer().
    # if your monitor refresh rate is 60Hz, the while loop is repeated every 1/60 sec
    glfw.swap_interval(1)

    while not glfw.window_should_close(window):
        glfw.poll_events()

        # get the current time, in seconds
        t = glfw.get_time()

        s = np.sin(t)
        M = np.array([[s, 0.],
                     [0., s]])

        render(M)

        glfw.swap_buffers(window)
    glfw.terminate()
```

[Practice] Nonuniform Scale, Rotation, Reflection, Shear

```
while not glfw.window_should_close(window):
    glfw.poll_events()
    t = glfw.get_time()

    # nonuniform scale
    s = np.sin(t)
    M = np.array([[s, 0.],
                  [0., s*.5]])

    # rotation
    th = t
    M = np.array([[np.cos(th), -np.sin(th)],
                  [np.sin(th), np.cos(th)]])

    # reflection
    M = np.array([[-1., 0.],
                  [0., 1.]])

    # shear
    a = np.sin(t)
    M = np.array([[1., a],
                  [0., 1.]])

    # identity matrix
    M = np.identity(2)

    render(M)
    glfw.swap_buffers(window)
```

Quiz #1

- Go to <https://www.slido.com/>
- Join #cg-ys
- Click “Polls”

- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. **2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked for “attendance”.

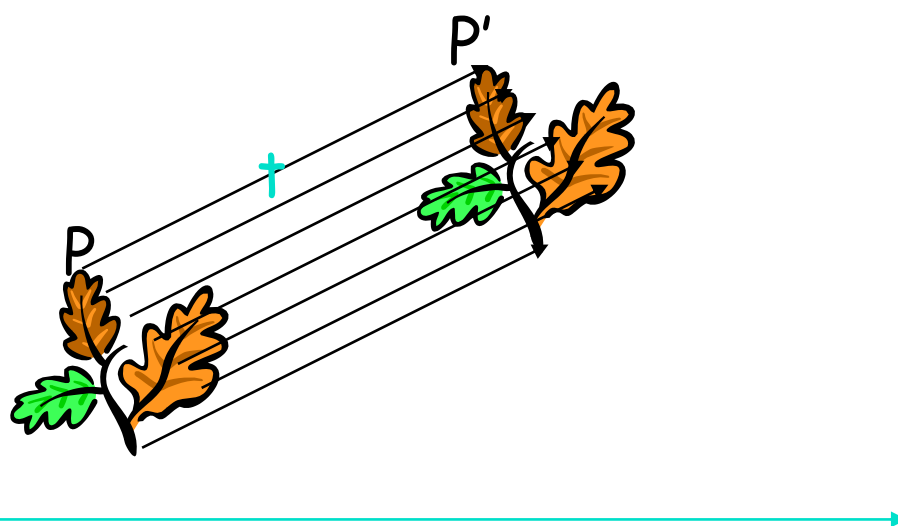
2D Translation

- Translation is the simplest transformation:

$$T(\mathbf{v}) = \mathbf{v} + \mathbf{u}$$

- Inverse:

$$T^{-1}(\mathbf{v}) = \mathbf{v} - \mathbf{u}$$



[Practice] Translation

```
def render(u):
    # ...
    glBegin(GL_TRIANGLES)
    glColor3ub(255, 255, 255)
    glVertex2fv(np.array([0.0, 0.5]) + u)
    glVertex2fv(np.array([0.0, 0.0]) + u)
    glVertex2fv(np.array([0.5, 0.0]) + u)
    glEnd()

def main():
    # ...
    while not glfw.window_should_close(window):
        glfw.poll_events()
        t = glfw.get_time()

        u = np.array([np.sin(t), 0.])
        render(u)
    # ...
```


Is translation linear transformation?

- No, because it cannot be represented using a simple matrix multiplication.

- We can express it using vector addition:

$$T(\mathbf{v}) = \mathbf{v} + \mathbf{u}$$

- Combining with linear transformation:

$$T(\mathbf{v}) = M\mathbf{v} + \mathbf{u}$$

 **Affine transformation**

Let's check again

- Linear transformation
 - Scaling, rotation, reflection, shearing
 - Represented as matrix multiplications

$$T(\mathbf{v}) = M\mathbf{v}$$

- Translation
 - Not a linear transformation
 - Can be expressed using vector addition

$$T(\mathbf{v}) = \mathbf{v} + \mathbf{u}$$

Affine Transformation

- Linear transformation + Translation

$$T(\mathbf{v}) = M\mathbf{v} + \mathbf{u}$$

- Preserves lines
- Preserves parallel lines
- Preserves ratios of distance along a line
- → These properties are inherited from linear transformations.

Rigid Transformation

- Rotation + Translation

$$T(\mathbf{v}) = R\mathbf{v} + \mathbf{u} \quad , \text{ where } R \text{ is a rotation matrix.}$$

- Preserves distances between all points
- Preserves cross product for all vectors

[Practice] Affine Transformation

```
def render(M, u):
    # ...
    glBegin(GL_TRIANGLES)
    glColor3ub(255, 255, 255)
    glVertex2fv(M @ np.array([0.0, 0.5]) + u)
    glVertex2fv(M @ np.array([0.0, 0.0]) + u)
    glVertex2fv(M @ np.array([0.5, 0.0]) + u)
    glEnd()

def main():
    # ...
    while not glfw.window_should_close(window):
        glfw.poll_events()
        t = glfw.get_time()

        th = t
        R = np.array([[np.cos(th), -np.sin(th)],
                     [np.sin(th), np.cos(th)]])
        u = np.array([np.sin(t), 0.])
        render(R, u)
    # ...
```

Quiz #2

- Go to <https://www.slido.com/>
- Join #cg-ys
- Click “Polls”

- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. **2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked for “attendance”.

Composing Transformations & Homogeneous Coordinates

Composing Transformations

- Move an object, then move it some more

$$\mathbf{p} \rightarrow T(\mathbf{p}) \rightarrow S(T(\mathbf{p})) = (S \circ T)(\mathbf{p})$$

- **Composing 2D linear transformations just works by 2x2 matrix multiplication**

$$T(\mathbf{p}) = M_T \mathbf{p}; S(\mathbf{p}) = M_S \mathbf{p}$$

$$(S \circ T)(\mathbf{p}) = M_S M_T \mathbf{p} = (M_S M_T) \mathbf{p} = M_S (M_T \mathbf{p})$$

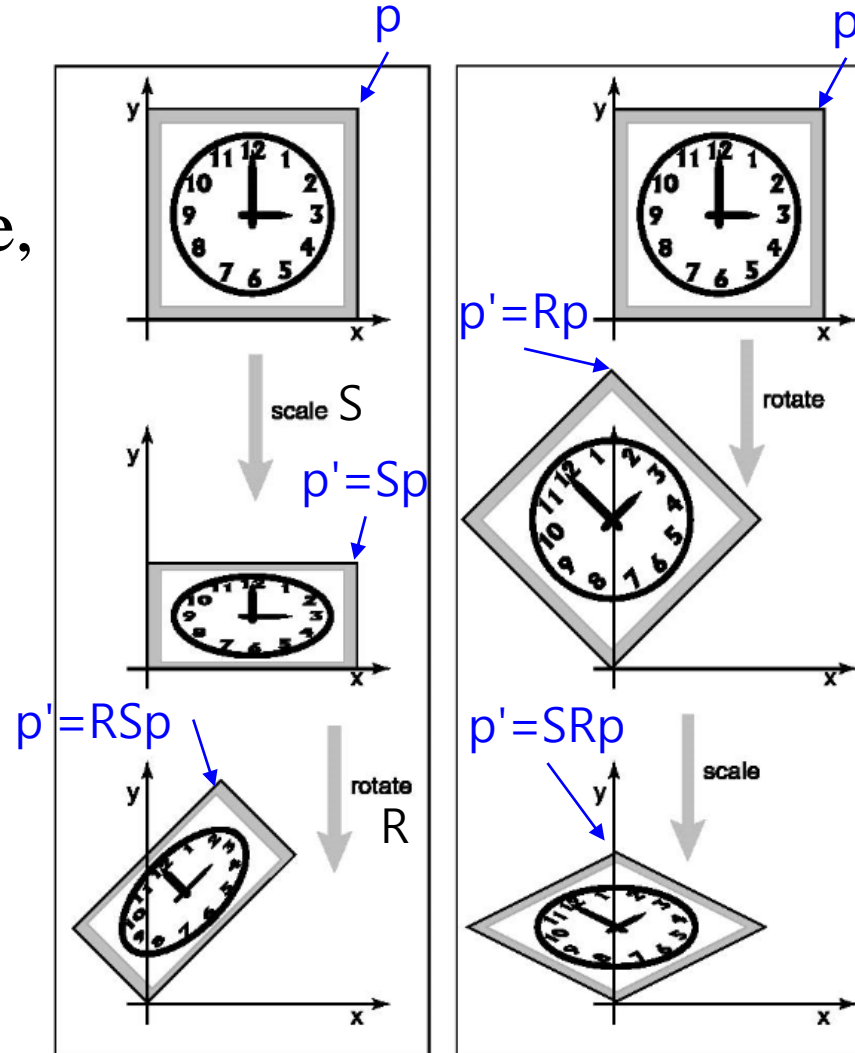
Order Matters!

- Note that matrix multiplication is associative, but **not commutative**.

$$(AB)C = A(BC)$$

$$AB \neq BA$$

- As a result, the **order of transforms is very important**.



[Practice] Composition

```
def main():
    # ...
    while not glfw.window_should_close(window):
        glfw.poll_events()

    S = np.array([[1., 0.],
                  [0., 2.]])

    th = np.radians(60)
    R = np.array([[np.cos(th), -np.sin(th)],
                  [np.sin(th), np.cos(th)]])

    u = np.zeros(2)

    # compare results of these two lines
    render(R @ S, u)      # p'=RSp
    # render(S @ R, u)    # p'=SRp

    # ...
```

Problems when handling Translation as Vector Addition

- Cannot treat linear transformation (rotation, scale,...) and translation in a consistent manner.
- Composing affine transformations is complicated

$$\begin{aligned} T(\mathbf{p}) &= M_T \mathbf{p} + \mathbf{u}_T & (S \circ T)(\mathbf{p}) &= M_S(M_T \mathbf{p} + \mathbf{u}_T) + \mathbf{u}_S \\ S(\mathbf{p}) &= M_S \mathbf{p} + \mathbf{u}_S & &= (M_S M_T) \mathbf{p} + (M_S \mathbf{u}_T + \mathbf{u}_S) \end{aligned}$$

- We need a cleaner way!

 **Homogeneous coordinates**

Homogeneous Coordinates

- Key idea: Represent 2D points in 3D coordinate space
- Extra component w for vectors, extra row/column for matrices
 - For points, can always keep $w = 1$
 - 2D point $[x, y]^T \rightarrow 3D$ point $[x, y, 1]^T$.
- 2D linear transformations are represented as:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \\ 1 \end{bmatrix}$$

Homogeneous Coordinates

- 2D translations are represented as:

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

- 2D affine transformations are represented as:

linear part

$$\begin{bmatrix} m_{11} & m_{12} & u_x \\ m_{21} & m_{22} & u_y \\ 0 & 0 & 1 \end{bmatrix}$$

translational part

Homogeneous Coordinates

- **Composing affine transformations just works by 3x3 matrix multiplication**

$$T(\mathbf{p}) = M_T \mathbf{p} + \mathbf{u}_T$$

$$S(\mathbf{p}) = M_S \mathbf{p} + \mathbf{u}_S$$



$$T(\mathbf{p}) = \begin{bmatrix} M_T^{2 \times 2} & \mathbf{u}_T^{2 \times 1} \\ 0 & 1 \end{bmatrix} \quad S(\mathbf{p}) = \begin{bmatrix} M_S^{2 \times 2} & \mathbf{u}_S^{2 \times 1} \\ 0 & 1 \end{bmatrix}$$

Homogeneous Coordinates

- **Composing affine transformations just works by 3x3 matrix multiplication**

$$(S \circ T)(\mathbf{p}) = \begin{bmatrix} M_S^{2 \times 2} & \mathbf{u}_S^{2 \times 1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} M_T^{2 \times 2} & \mathbf{u}_T^{2 \times 1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p}^{2 \times 1} \\ 1 \end{bmatrix} \\ = \begin{bmatrix} (M_S M_T) \mathbf{p} + (M_S \mathbf{u}_T + \mathbf{u}_S) \\ 1 \end{bmatrix}$$

- Much cleaner

[Practice] Homogeneous Coordinates

```
def render(M):  
    # ...  
    glBegin(GL_TRIANGLES)  
    glColor3ub(255, 255, 255)  
    glVertex2fv( (M @ np.array([.0, .5, 1.]))[:-1] )  
    glVertex2fv( (M @ np.array([.0, .0, 1.]))[:-1] )  
    glVertex2fv( (M @ np.array([.5, .0, 1.]))[:-1] )  
    glEnd()
```


[Practice] Homogeneous Coordinates

```
def main():
    # ...
    while not glfw.window_should_close(window):
        glfw.poll_events()

        # rotate 60 deg about z axis
        th = np.radians(60)
        R = np.array([[np.cos(th), -np.sin(th), 0.],
                     [np.sin(th), np.cos(th), 0.],
                     [0.,          0.,          1.]])

        # translate by (.4, .1)
        T = np.array([[1., 0., .4],
                     [0., 1., .1],
                     [0., 0., 1.]])

        render(R)      # p'=Rp
        # render(T)    # p'=Tp
        # render(T @ R) # p'=TRp
        # render(R @ T) # p'=RTP
        # ...
```

Summary: Homogeneous Coordinates in 2D

- Use $(\mathbf{x}, \mathbf{y}, 1)^T$ instead of $(x, y)^T$ for **2D points**
- Use **3x3 matrices** instead of 2x2 matrices for **2D linear transformations**
- Use **3x3 matrices** instead of vector additions for **2D translations**

- → We can treat linear transformations and translations **in a consistent manner!**

Quiz #3

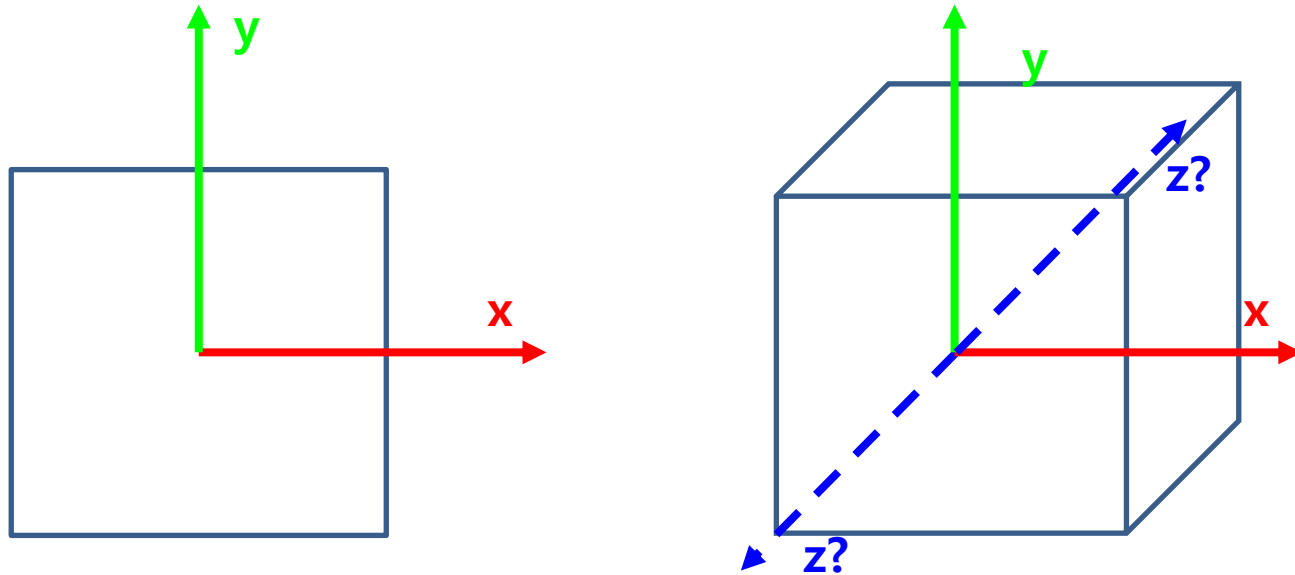
- Go to <https://www.slido.com/>
- Join #cg-ys
- Click “Polls”

- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. **2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked for “attendance”.

3D Cartesian Coordinate System

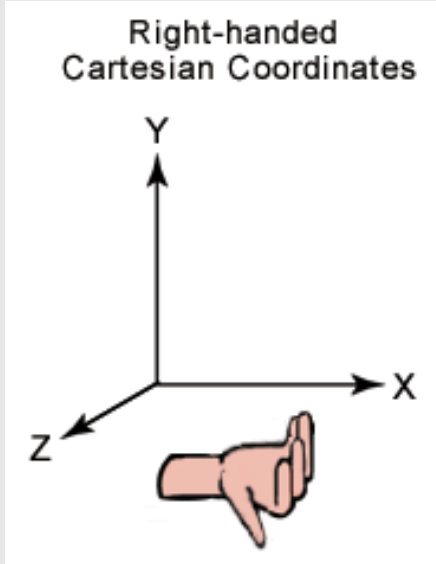
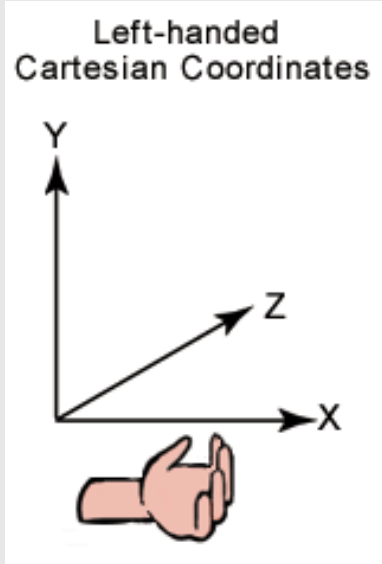


Now, Let's go to the 3D world!



- Coordinate system (좌표계)
 - Cartesian coordinate system (직교좌표계)

Two Types of 3D Cartesian Coordinate Systems

What we're using

	 <p>Right-handed Cartesian Coordinates</p> <p>The diagram shows a 3D coordinate system with X, Y, and Z axes. The Y-axis is vertical, the X-axis is horizontal to the right, and the Z-axis points diagonally down and to the left. A right hand is shown below the axes, with the thumb pointing up (Y), the index finger pointing right (X), and the middle finger pointing down (Z).</p>	 <p>Left-handed Cartesian Coordinates</p> <p>The diagram shows a 3D coordinate system with X, Y, and Z axes. The Y-axis is vertical, the X-axis is horizontal to the right, and the Z-axis points diagonally up and to the right. A left hand is shown below the axes, with the thumb pointing up (Y), the index finger pointing right (X), and the middle finger pointing up (Z).</p>
Positive rotation direction	counterclockwise about the axis of rotation  <p>A right hand is shown with the thumb pointing along a green arrow labeled 'X'. A red curved arrow indicates a counterclockwise rotation around the X-axis.</p>	clockwise about the axis of rotation  <p>A left hand is shown with the thumb pointing along a green arrow labeled 'X'. A red curved arrow indicates a clockwise rotation around the X-axis.</p>
Used in...	OpenGL , Maya, Houdini, AutoCAD, ... Standard for Physics & Math	DirectX, Unity, Unreal, ...

Next Time

- Lab for this lecture (next Monday):
 - Lab assignment 3

- Next lecture (next Wednesday):
 - 4 - Transformation 2

- Acknowledgement: Some materials come from the lecture slides of
 - Prof. Taesoo Kwon, Hanyang Univ., <http://calab.hanyang.ac.kr/cgi-bin/cg.cgi>
 - Prof. Steve Marschner, Cornell Univ., <http://www.cs.cornell.edu/courses/cs4620/2014fa/index.shtml>