Computer Graphics

4 - Transformation 2

Yoonsang Lee
Spring 2019

Topics Covered

- 3D Affine Transformation
- Reference Frame & Composite Transformations
  - Coordinate System & Reference Frame
  - Global & Local Coordinate System
  - Interpretation of Composite Transformations
- OpenGL Transformation Functions
  - OpenGL “Current” Transformation Matrix
  - OpenGL Transformation Functions
- **Fundamental Idea of Transformation**
- Composing Transformations using OpenGL Functions

3D Affine Transformation

Point Representation in Cartesian & Homogeneous Coordinate System

	Cartesian coordinate system	Homogeneous coordinate system
A **2D point** is represented as…	$\begin{bmatrix} p_x \\ p_y \end{bmatrix}$	$\begin{bmatrix} p_x \\ p_y \\ \boxed{1} \end{bmatrix}$
A **3D point** is represented as…	$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$	$\begin{bmatrix} p_x \\ p_y \\ p_z \\ \boxed{1} \end{bmatrix}$

Review of Linear Transform in 2D

- Linear transformation in **2D** can be represented as matrix multiplication of …

2x2 matrix (in Cartesian coordinates)

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix}$$

or

3x3 matrix (in homogeneous coordinates)

$$\begin{bmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$$

Linear Transformation in 3D

- Linear transformation in **3D** can be represented as matrix multiplication of …

3x3 matrix
(in Cartesian coordinates)

or

4x4 matrix
(in homogeneous coordinates)

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Linear Transformation in 3D

Scale:

3D

$$\mathbf{S_s} = \begin{bmatrix} \mathbf{S}_x & 0 & 0 \\ 0 & \mathbf{S}_y & 0 \\ 0 & 0 & \mathbf{S}_z \end{bmatrix}$$

3D-H

$$\mathbf{S_s} = \begin{bmatrix} \mathbf{S}_x & 0 & 0 & 0 \\ 0 & \mathbf{S}_y & 0 & 0 \\ 0 & 0 & \mathbf{S}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

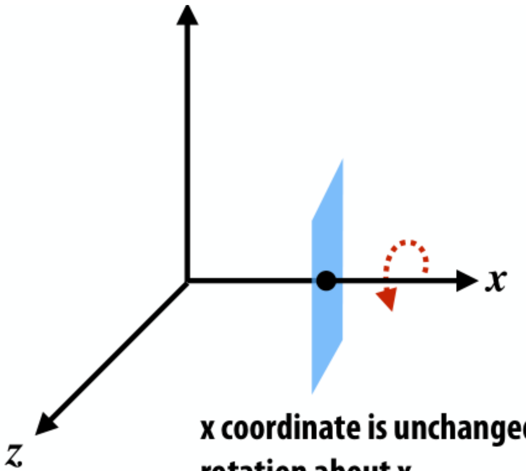
Shear (in x, based on y,z position):

$$\mathbf{H}_{x,\mathbf{d}} = \begin{bmatrix} 1 & \mathbf{d}_y & \mathbf{d}_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{H}_{x,\mathbf{d}} = \begin{bmatrix} 1 & \mathbf{d}_y & \mathbf{d}_z & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Linear Transformation in 3D

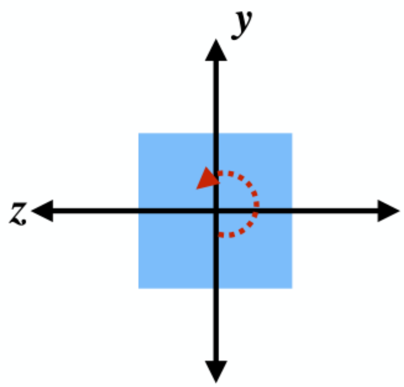
Rotation about x axis:

$$\mathbf{R}_{x,\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$



x coordinate is unchanged by rotation about x

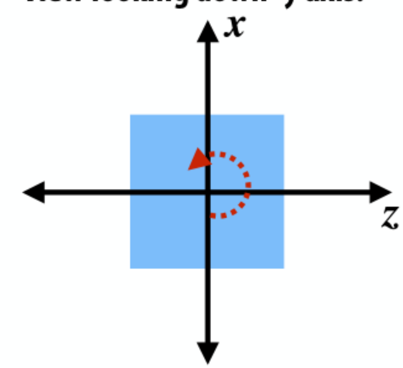
View looking down -x axis:



Rotation about y axis:

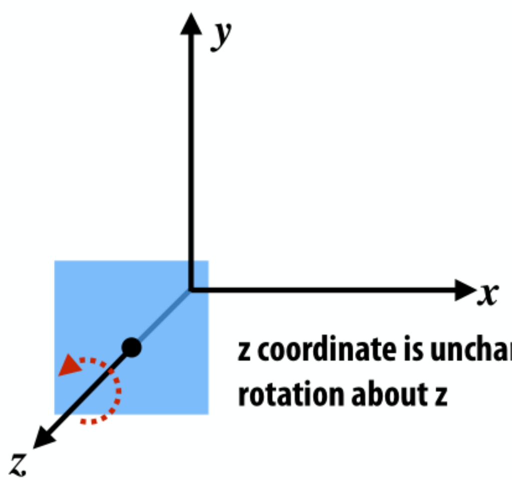
$$\mathbf{R}_{y,\theta} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

View looking down -y axis:



Rotation about z axis:

$$\mathbf{R}_{z,\theta} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



z coordinate is unchanged by rotation about z

Review of Translation in 2D

- Translation in **2D** can be represented as …

Vector addition
(in Cartesian coordinates)

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} + \begin{bmatrix} u_x \\ u_y \end{bmatrix}$$

Matrix multiplication of
3x3 matrix
(in homogeneous coordinates)

$$\begin{bmatrix} 1 & 0 & u_x \\ 0 & 1 & u_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$$

Translation in 3D

- Translation in **3D** can be represented as …

Vector addition
(in Cartesian coordinates)

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} + \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$$

Matrix multiplication of
4x4 matrix
(in homogeneous coordinates)

$$\begin{bmatrix} 1 & 0 & 0 & u_x \\ 0 & 1 & 0 & u_y \\ 0 & 0 & 1 & u_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Review of Affine Transformation in 2D

- In homogeneous coordinates, **2D** affine transformation can be represented as multiplication of **3x3 matrix**:

linear part

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

translational part

Affine Transformation in 3D

- In homogeneous coordinates, **3D** affine transformation can be represented as multiplication of **4x4 matrix**:

linear part

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & u_x \\ m_{21} & m_{22} & m_{23} & u_y \\ m_{31} & m_{32} & m_{33} & u_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translational part

[Practice] 3D Transformations

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

def render(M):
    # enable depth test (we'll see details later)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glEnable(GL_DEPTH_TEST)

    glLoadIdentity()

    # use orthogonal projection (we'll see details later)
    glOrtho(-1,1, -1,1, -1,1)

    # rotate "camera" position to see this 3D space better (we'll see details later)
    t = glfw.get_time()
    gluLookAt(.1*np.sin(t),.1, .1*np.cos(t), 0,0,0, 0,1,0)
```

```
    # draw coordinate: x in red, y in green, z in blue
    glBegin(GL_LINES)
    glColor3ub(255, 0, 0)
    glVertex3fv(np.array([0.,0.,0.]))
    glVertex3fv(np.array([1.,0.,0.]))
    glColor3ub(0, 255, 0)
    glVertex3fv(np.array([0.,0.,0.]))
    glVertex3fv(np.array([0.,1.,0.]))
    glColor3ub(0, 0, 255)
    glVertex3fv(np.array([0.,0.,0]))
    glVertex3fv(np.array([0.,0.,1.]))
    glEnd()

    # draw triangle
    glBegin(GL_TRIANGLES)
    glColor3ub(255, 255, 255)
    glVertex3fv((M @ np.array([.0,.5,0.,1.]))[:-1])
    glVertex3fv((M @ np.array([.0,.0,0.,1.]))[:-1])
    glVertex3fv((M @ np.array([.5,.0,0.,1.]))[:-1])
    glEnd()
```

```python
def main():
    if not glfw.init():
        return
    window = glfw.create_window(640,640,
"3D Trans", None,None)
    if not window:
        glfw.terminate()
        return
    glfw.make_context_current(window)
    glfw.swap_interval(1)
    count = 0
    while not
glfw.window_should_close(window):
        glfw.poll_events()

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

        # translate by (.4, 0., .2)
        T = np.array([[1.,0.,0.,.4],
                       [0.,1.,0.,0.],
                       [0.,0.,1.,.2],
                       [0.,0.,0.,1.]])
```

```python
        render(R, camAng)
        # render(T)
        # render(T @ R)
        # render(R @ T)

        glfw.swap_buffers(window)

    glfw.terminate()

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

[Practice] Use Slicing

- You can use **slicing** for cleaner code (the behavior is the same as the previous page)

```
# ...

# rotate 60 deg about x axis
th = np.radians(-60)
R = np.identity(4)
R[:3,:3] = [[1.,0.,0.],
            [0., np.cos(th), -np.sin(th)],
            [0., np.sin(th), np.cos(th)]]

# translate by (.4, 0., .2)
T = np.identity(4)
T[:3,3] = [.4, 0., .2]

# ...
```

Quiz #1

- Go to https://www.slido.com/
- Join #cg-hyu
- 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”.

Reference Frame & Composite Transformations

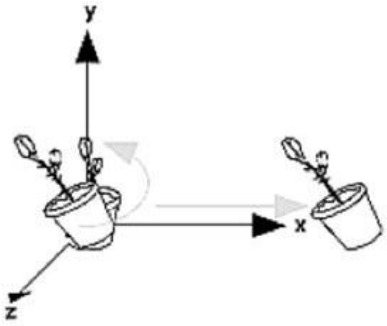
Revisit: Order Matters!

- If T and R are matrices representing affine transformations,

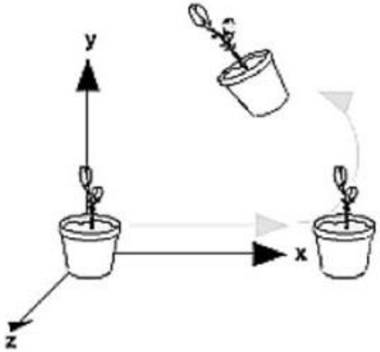
- **p**’ = TR**p**
  - First apply transformation R to point **p**, then apply transformation T to transformed point R**p**

- **p**’ = RT**p**
  - First apply transformation T to point **p**, then apply transformation R to transformed point T**p**

- Note that these are done **w.r.t. global coordinate system**



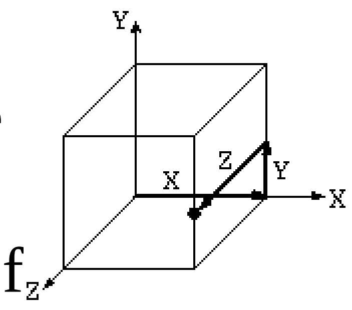
Rotate then Translate



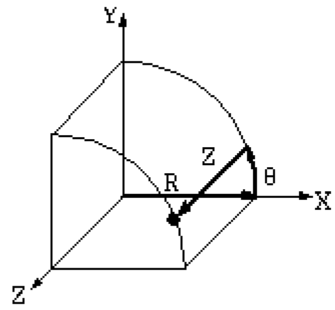
Translate then Rotate

Coordinate System & Reference Frame

- Coordinate system
  - A system which uses one or more numbers, or coordinates, to uniquely determine the position of the points.

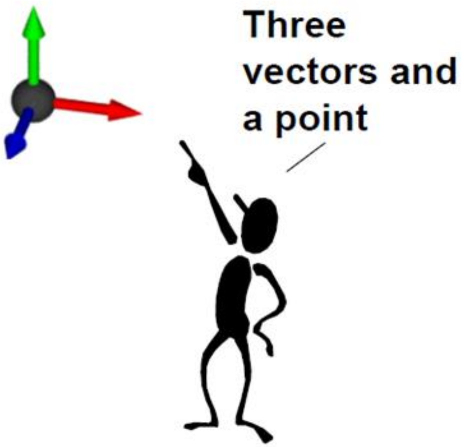


Cartesian (X,Y,Z components) coordinate system 0 (C.S. 0)



Cylindrical (R,q,Z components) coordinate system 1 (C.S. 1)

- Reference frame
  - Abstract coordinate system + physical reference points (to uniquely fix the coordinate system).



Coordinate System & Reference Frame

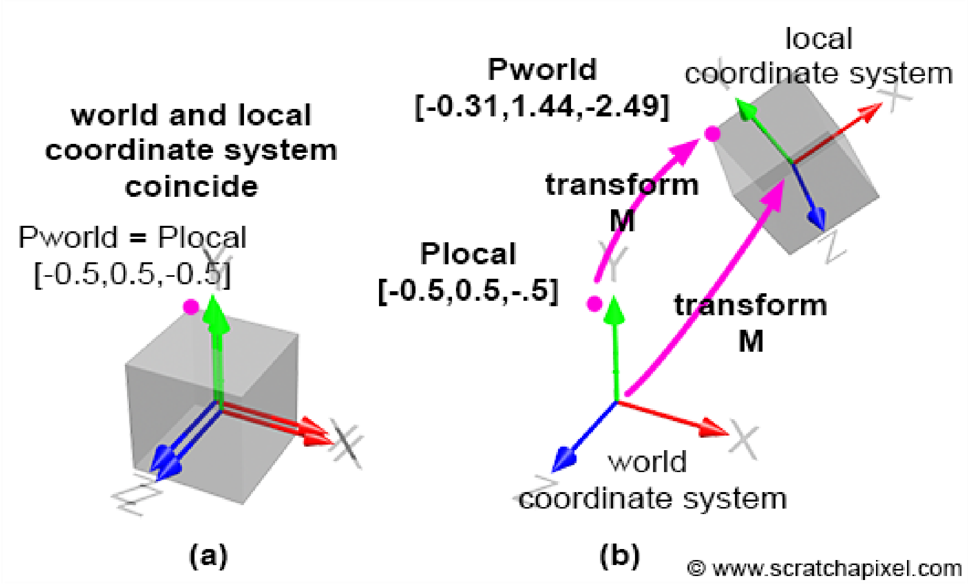
- Two terms are slightly different:
  - **Coordinate system** is a mathematical concept, about a choice of “language” used to describe observations.
  - **Reference frame** is a physical concept related to state of motion.
  - You can think the coordinate system determines the way one describes/observes the motion in each reference frame.

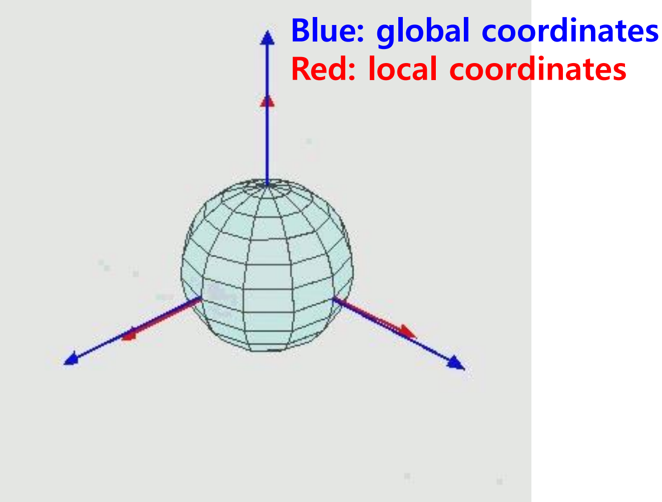
- But these two terms are often mixed.

Global & Local Coordinate System(or Frame)

- **global coordinate system** (or **global frame**)
  - A coordinate system(or frame) attached to the **world.**
  - A.k.a. **world** coordinate system, **fixed** coordinate system

- **local coordinate system** (or **local frame**)
  - A coordinate system(or frame) attached to a **moving object.**





https://commons.wikimedia.org/wiki/File:Euler2a.gif

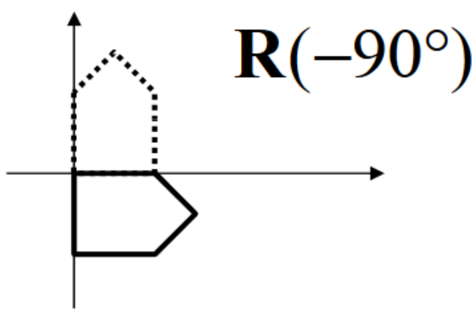
Interpretation of Composite Transformations #1

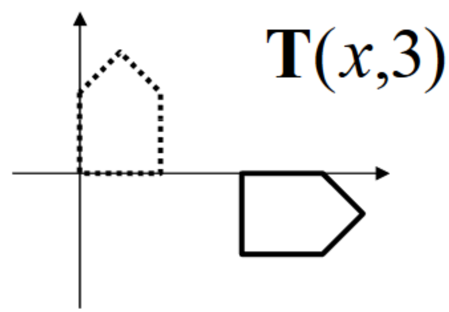
- An example transformation:

$$T = \mathbf{T}(x,3) \cdot \mathbf{R}(-90°)$$

- This is how we've interpreted so far:
  - R-to-L : interpret operations w.r.t. fixed coordinates

=global coordinates
=world coordinates



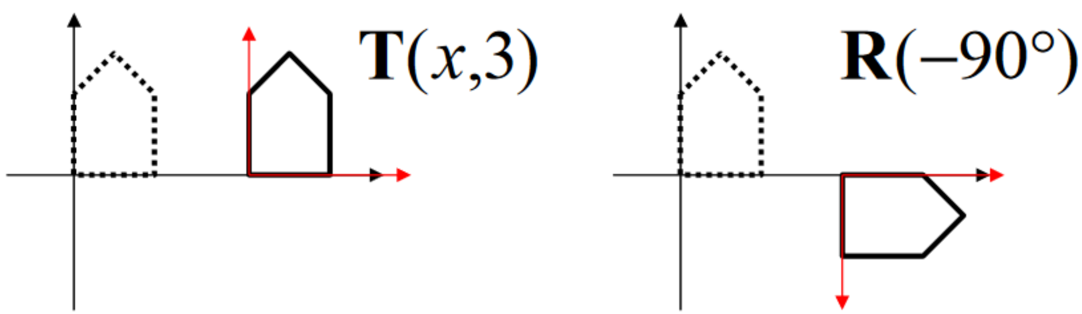


Interpretation of Composite Transformations #2

- An example transformation:

$$T = \mathbf{T}(x,3) \cdot \mathbf{R}(-90°)$$

- **Another way of interpretation:**
  - L-to-R : interpret operations w.r.t local coordinates



Left & Right Multiplication

- Thinking it deeper, we can see:

- p’ = **R**Tp (left-multiplication by **R**)
  - Apply transformation **R** to point Tp w.r.t. global coordinates

- p’ = T**R**p (right-multiplication by **R**)
  - Apply transformation **R** to point Tp w.r.t. local coordinates

[Practice] Interpretation of Composite Transformations

- Just use the same code as the previous practice [Practice] 3D Transformations.

- Test render(T @ R) and render(R @ T), and interpret them in two other ways.

Quiz #2

- Go to https://www.slido.com/
- Join #cg-hyu
- 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”.

OpenGL Transformation Functions

OpenGL "Current" Transformation Matrix

- OpenGL is a "state machine".
  - If you set a value for a state, it remains in effect until you change it.
  - ex1) current color
  - ex2) **current transformation matrix**

- An OpenGL context keeps the "current" transformation matrix somewhere in the memory.

OpenGL "Current" Transformation Matrix

- OpenGL always draws an object with the **current transformation matrix**.

- Let's say **p** is a vertex position of an object w.r.t. its local coordinates,
- and **C** is the current transformation matrix,

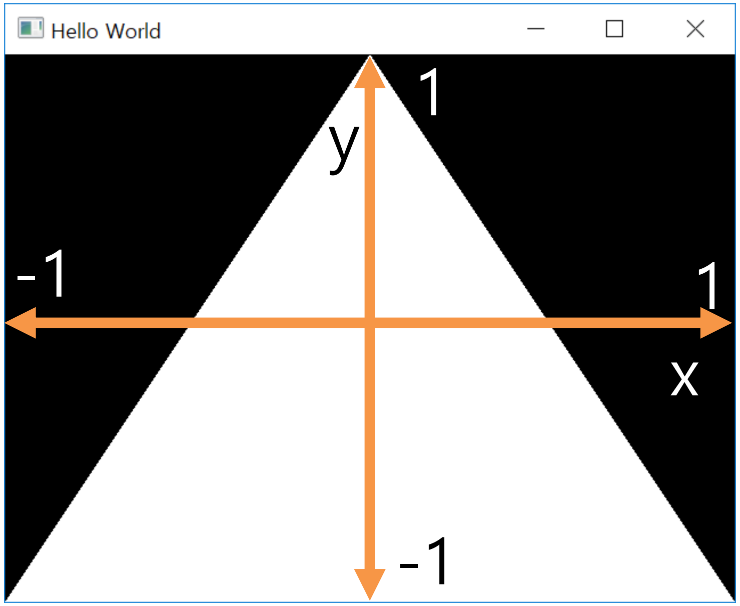
- If you set the vertex position using glVertex3fv(**p**),
- OpenGL will draw the vertex at the position of **Cp**

OpenGL "Current" Transformation Matrix

- Except today's practice code (which use glOrtho() and gluLookAt()), the current transformation matrix we've used so far is the **identity matrix.**

- This is done by **glLoadIdentity()** - replace the current matrix with the identity matrix.

- If the current transformation matrix is the **identity**, all objects are drawn in the Normalized Device Coordinate (**NDC**) space.



OpenGL Transformation Functions

- OpenGL provides a number of functions *to manipulate the current transformation matrix.*

- At the beginning of each rendering iteration, you have to set the current matrix to the identity matrix with **glLoadIdentity().**

- Then you can manipulate the current matrix with following functions:

- Direct manipulation of the current matrix
  - **glMultMatrix*()**

- Scale, rotate, translate with parameters
  - **glScale*()**
  - **glRotate*()**
  - **glTranslate*()**
  - OpenGL doesn’t provide functions like glShear*() and glReflect*()

glMultMatrix*()

- glMultiMatrix*(*m*) - multiply the current transformation matrix with the matrix *m*
  - *m* : 4x4 **column-major** matrix
  - Note that you have to pass the **transpose of np.ndarray** because np.ndarray is **row-major**

If this is the memory layout of a stored matrix:

m[0]	m[1]	m[2]	m[3]	m[4]	m[5]	m[6]	m[7]	m[8]	m[9]	m[10]	m[11]	m[12]	m[13]	m[14]	m[15]

$$\begin{bmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{bmatrix}$$

Column-major

$$\begin{bmatrix} m[0] & m[1] & m[2] & m[3] \\ m[4] & m[5] & m[6] & m[7] \\ m[8] & m[9] & m[10] & m[11] \\ m[12] & m[13] & m[14] & m[15] \end{bmatrix}$$

Row-major

glMultMatrix*()

- Let's call the current matrix C
- Calling glMultMatrix*(M) will update the current matrix as follows:
- C ← C**M** (**right-multiplication by M**)

[Practice] OpenGL Trans. Functions

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

gCamAng = 0.

def render(camAng):
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glEnable(GL_DEPTH_TEST)

    # set the current matrix to the identity matrix
    glLoadIdentity()

    # use orthogonal projection (multiply the current
matrix by "projection" matrix - we'll see details
later)
    glOrtho(-1,1, -1,1, -1,1)

    # rotate "camera" position (multiply the current
matrix by "camera" matrix - we'll see details later)
    gluLookAt(.1*np.sin(camAng),.1,.1*np.cos(camAng),
0,0,0, 0,1,0)

    # draw coordinates
    glBegin(GL_LINES)
    glColor3ub(255, 0, 0)
    glVertex3fv(np.array([0.,0.,0.]))
    glVertex3fv(np.array([1.,0.,0.]))
    glColor3ub(0, 255, 0)
    glVertex3fv(np.array([0.,0.,0.]))
    glVertex3fv(np.array([0.,1.,0.]))
    glColor3ub(0, 0, 255)
    glVertex3fv(np.array([0.,0.,0]))
    glVertex3fv(np.array([0.,0.,1.]))
    glEnd()

    ##############################
    # edit here
```

```
def key_callback(window, key, scancode, action,
mods):
    global gCamAng
    # rotate the camera when 1 or 3 key is pressed
or repeated
    if action==glfw.PRESS or action==glfw.REPEAT:
        if key==glfw.KEY_1:
            gCamAng += np.radians(-10)
        elif key==glfw.KEY_3:
            gCamAng += np.radians(10)

def main():
    if not glfw.init():
        return
    window = glfw.create_window(640,640, 'OpenGL
Trans. Functions', None,None)
    if not window:
        glfw.terminate()
        return
    glfw.make_context_current(window)
    glfw.set_key_callback(window, key_callback)

    while not glfw.window_should_close(window):
        glfw.poll_events()
        render(gCamAng)
        glfw.swap_buffers(window)

    glfw.terminate()

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

[Practice] OpenGL Trans. Functions

```
def drawTriangleTransformedBy(M):
    glBegin(GL_TRIANGLES)
    glVertex3fv((M @ np.array([.0,.5,0.,1.]))[:-1])
    glVertex3fv((M @ np.array([.0,.0,0.,1.]))[:-1])
    glVertex3fv((M @ np.array([.5,.0,0.,1.]))[:-1])
    glEnd()

def drawTriangle():
    glBegin(GL_TRIANGLES)
    glVertex3fv(np.array([.0,.5,0.]))
    glVertex3fv(np.array([.0,.0,0.]))
    glVertex3fv(np.array([.5,.0,0.]))
    glEnd()
```

[Practice] glMultMatrix*()

```
def render():
    # ...
    # edit here

    # rotate 30 deg about x axis
    th = np.radians(30)
    R = np.identity(4)
    R[:3,:3] = [[1.,0.,0.],
                [0., np.cos(th), -np.sin(th)],
                [0., np.sin(th), np.cos(th)]]

    # translate by (.4, 0., .2)
    T = np.identity(4)
    T[:3,3] = [.4, 0., .2]

    glColor3ub(255, 255, 255)

    # 1)& 2)& 3) all draw a triangle with the
same transformation

    # 1)
    glMultMatrixf(R.T)
    glMultMatrixf(T.T)
    drawTriangle()

    # 2)
    # glMultMatrixf((R@T).T)
    # drawTriangle()

    # 3)
    # drawTriangleTransformedBy(R@T)
```

glScale*()

- glScale*(x, y, z) - multiply the current matrix by a general scaling matrix
  - x, y, z : scale factors along the x, y, and z axes

- Calling glScale*(x, y, z) will update the current matrix as follows:
- C ← C**S** (**right-multiplication by S**)

$$S = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[Practice] glScale*()

```
def render():
    # ...
    # edit here
    glColor3ub(255, 255, 255)

    # 1)& 2) all draw a triangle with the same transformation
    # (scale by [2., .5, 0.])

    # 1)
    glScalef(2., .5, 0.)
    drawTriangle()

    # 2)
    # S = np.identity(4)
    # S[0,0] = 2.
    # S[1,1] = .5
    # S[2,2] = 0.
    # drawTriangleTransformedBy(S)
```

glRotate*()

- glRotate*(*angle*, *x*, *y*, *z*) - multiply the current matrix by a rotation matrix
  - *angle* : angle of rotation, **in degrees**
  - *x*, *y*, *z* : x, y, z coord. value of rotation axis vector

- Calling glRotate*(*angle*, *x*, *y*, *z*) will update the current matrix as follows:
- C ← C**R** (**right-multiplication by R**)

R is a rotation matrix

[Practice] glRotate*()

```
def render():
    # ...
    # edit here
    glColor3ub(255, 255, 255)

    # 1)& 2) all draw a triangle with the same transformation
    # (rotate 60 deg about x axis)

    # 1)
    glRotatef(60, 1, 0, 0)
    drawTriangle()

    # 2)
    # th = np.radians(60)
    # R = np.identity(4)
    # R[:3,:3] = [[1.,0.,0.],
                # [0., np.cos(th), -np.sin(th)],
                # [0., np.sin(th), np.cos(th)]]
    # drawTriangleTransformedBy(R)
```

glTranslate*()

- glTranslate*(*x*, *y*, *z*) - multiply the current matrix by a translation matrix
  - *x*, *y*, *z* : x, y, z coord. value of a translation vector

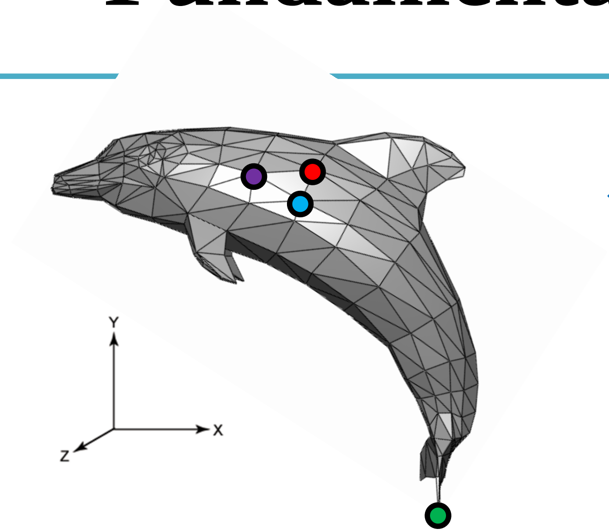
- Calling glTranslate*(*x*, *y*, *z*) will update the current matrix as follows:
- C ← C**T** (**right-multiplication by T**)

$$T = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[Practice] glTranslate*()

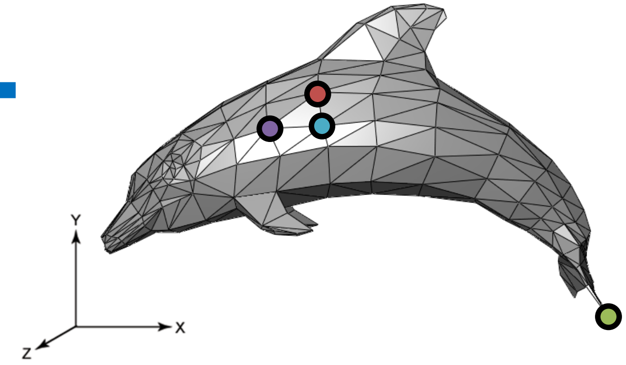
```
def render():
    # ...
    # edit here
    glColor3ub(255, 255, 255)

    # 1)& 2) all draw a triangle with the same transformation
    # (translate by [.4, 0, .2])

    # 1)
    glTranslatef(.4, 0, .2)
    drawTriangle()

    # 2)
    # T = np.identity(4)
    # T[:3,3] = [.4, 0., .2]
    # drawTriangleTransformedBy(T)
```

Fundamental Idea of Transformation



Affine transformation ←

$$\mathbf{M}=\begin{bmatrix} m_{11} & m_{12} & m_{13} & u_1 \\ m_{21} & m_{22} & m_{23} & u_2 \\ m_{31} & m_{32} & m_{33} & u_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$$\mathbf{p}_1' \leftarrow \mathbf{M}\ \mathbf{p}_1$$

$$\mathbf{p}_2' \leftarrow \mathbf{M}\ \mathbf{p}_2$$

$$\mathbf{p}_3' \leftarrow \mathbf{M}\ \mathbf{p}_3$$

$$\vdots \qquad \vdots \quad \vdots$$

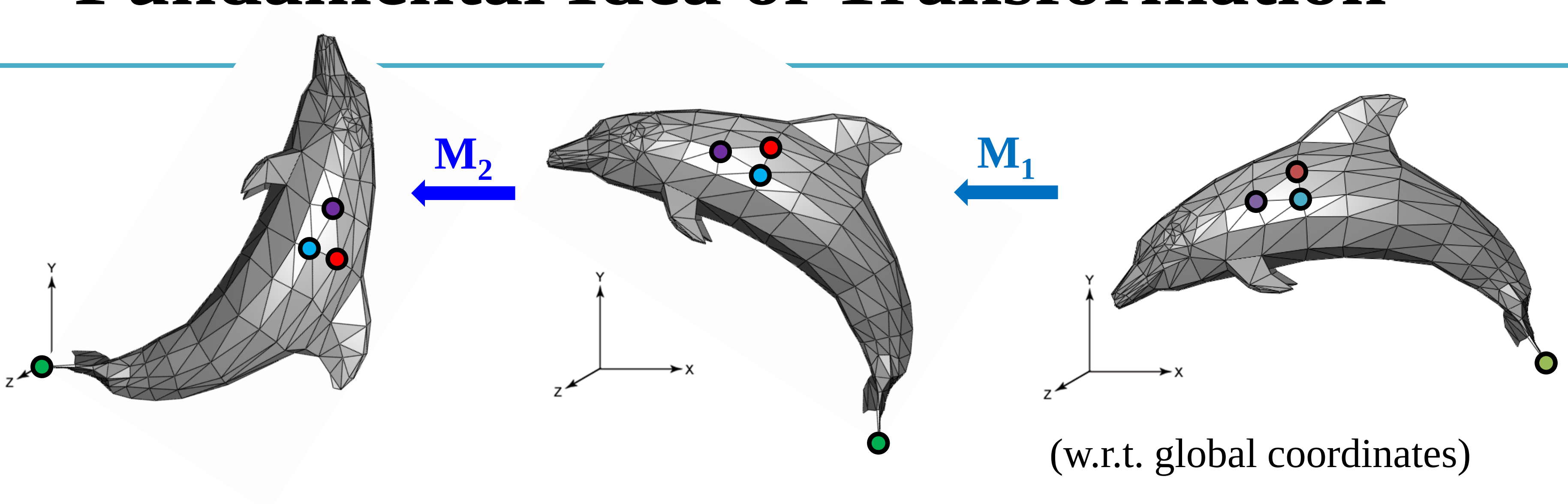
$$\mathbf{p}_N' \leftarrow \mathbf{M}\ \mathbf{p}_N$$

Fundamental idea	Implementation 1: Using numpy matrix multiplication	Implementation 2: Using OpenGL transformation functions
$\mathbf{p}_1' \leftarrow \mathbf{M}\,\mathbf{p}_1$ $\mathbf{p}_2' \leftarrow \mathbf{M}\,\mathbf{p}_2$ $\mathbf{p}_3' \leftarrow \mathbf{M}\,\mathbf{p}_3$ ⋮ $\mathbf{p}_N' \leftarrow \mathbf{M}\,\mathbf{p}_N$	glVertex3fv($\mathbf{M}\mathbf{p}_1$) glVertex3fv($\mathbf{M}\mathbf{p}_2$) glVertex3fv($\mathbf{M}\mathbf{p}_3$) . . glVertex3fv($\mathbf{M}\mathbf{p}_N$) (slicing is omitted)	**glMultMatrixf($\mathbf{M}^T$)** glVertex3fv($\mathbf{p}_1$) glVertex3fv($\mathbf{p}_2$) glVertex3fv($\mathbf{p}_3$) . . glVertex3fv($\mathbf{p}_N$) (or you can use **glScalef(x,y,z), glRotatef(ang,x,y,z), glTranslatef(x,y,z)**)
An array that stores all vertex data. This enables very fast drawing.		

Fundamental idea	Implementation 1: Using numpy matrix multiplication	Implementation 2: Using OpenGL transformation functions
$\mathbf{p}_1' \leftarrow \mathbf{M}\ \mathbf{p}_1$ $\mathbf{p}_2' \leftarrow \mathbf{M}\ \mathbf{p}_2$ $\mathbf{p}_3' \leftarrow \mathbf{M}\ \mathbf{p}_3$ . . . . $\mathbf{p}_N' \leftarrow \mathbf{M}\ \mathbf{p}_N$	glVertex3fv($\mathbf{M}\mathbf{p}_1$) glVertex3fv($\mathbf{M}\mathbf{p}_2$) glVertex3fv($\mathbf{M}\mathbf{p}_3$) . . glVertex3fv($\mathbf{M}\mathbf{p}_N$) (slicing is omitted)	**glMultMatrixf($\mathbf{M}^T$)** glVertex3fv($\mathbf{p}_1$) glVertex3fv($\mathbf{p}_2$) glVertex3fv($\mathbf{p}_3$) . . glVertex3fv($\mathbf{p}_N$) (or you can use **glScalef(x,y,z), glRotatef(ang,x,y,z), glTranslatef(x,y,z)**)
An array that stores all vertex data. This enables very fast drawing.	• Performance drawback: CPU performs all matrix multiplications	• Faster than the left method because GPU performs matrix multiplications

• (Actually, calling a large number of glVertex3f() is not applicable to serious OpenGL programs. Instead they use *vertex array*.)

Fundamental Idea of Transformation



(w.r.t. global coordinates)

$$p_1' \leftarrow M_2\, M_1\, p_1$$

$$p_2' \leftarrow M_2\, M_1\, p_2$$

$$p_3' \leftarrow M_2\, M_1\, p_3$$

$$\vdots \qquad \vdots \quad \vdots \quad \vdots$$

$$p_N' \leftarrow M_2\, M_1\, p_N$$

Fundamental idea	Implementation 1: Using numpy matrix multiplication	Implementation 2: Using OpenGL transformation functions
$\mathbf{p}_1' \leftarrow \mathbf{M}_2\,\mathbf{M}_1\,\mathbf{p}_1$ $\mathbf{p}_2' \leftarrow \mathbf{M}_2\,\mathbf{M}_1\,\mathbf{p}_2$ $\mathbf{p}_3' \leftarrow \mathbf{M}_2\,\mathbf{M}_1\,\mathbf{p}_3$ . . . . $\mathbf{p}_N' \leftarrow \mathbf{M}_2\,\mathbf{M}_1\,\mathbf{p}_N$	glVertex3fv($\mathbf{M}_2\mathbf{M}_1\mathbf{p}_1$) glVertex3fv($\mathbf{M}_2\mathbf{M}_1\mathbf{p}_2$) glVertex3fv($\mathbf{M}_2\mathbf{M}_1\mathbf{p}_3$) . . glVertex3fv($\mathbf{M}_2\mathbf{M}_1\mathbf{p}_N$) (slicing is omitted)	**glMultMatrixf($\mathbf{M}_2^T$)** **glMultMatrixf($\mathbf{M}_1^T$)** …or… **glMultMatrixf($(\mathbf{M}_2\mathbf{M}_1)^T$)** glVertex3fv($\mathbf{p}_1$) glVertex3fv($\mathbf{p}_2$) glVertex3fv($\mathbf{p}_3$) . . glVertex3fv($\mathbf{p}_N$) (or you can use combination of **glScalef(x,y,z), glRotatef(ang,x,y,z), glTranslatef(x,y,z)**)

Fundamental Idea is Most Important!

- If you see the term "transformation", what you have to think about is:

$$
\begin{aligned}
p_1' &\leftarrow M\, p_1 \\
p_2' &\leftarrow M\, p_2 \\
p_3' &\leftarrow M\, p_3 \\
&\vdots \\
p_N' &\leftarrow M\, p_N
\end{aligned}
$$

$$
\begin{aligned}
p_1' &\leftarrow M_2\, M_1\, p_1 \\
p_2' &\leftarrow M_2\, M_1\, p_2 \\
p_3' &\leftarrow M_2\, M_1\, p_3 \\
&\vdots \\
p_N' &\leftarrow M_2\, M_1\, p_N
\end{aligned}
$$

- Not this one:

```
glScalef(x, y, x)
glRotatef(angle, x, y, z)
glTranslatef(x, y, z)
```

Fundamental Idea is Most Important!

- `glScalef()`, `glRotatef()`, `glTranslatef()` are only in legacy OpenGL, not in DirectX, Unity, Unreal, modern OpenGL, …

- In modern OpenGL, one have to directly multiply a transformation matrix to a vertex position in *vertex shader*.
  - Very similar to our first method – using numpy matrix multiplication

- That’s why I started the transformation lectures with numpy matrix multiplication, not OpenGL transform functions.
  - The fundamental idea is the most important!

- But in this class, you have to know how to use these gl transformation functions anyway.

Composing Transformations using OpenGL Functions

- Let's say the current matrix is the identity **I**
-
```
glTranslatef(x, y, z) # T
glRotatef(angle, x, y, z) # R
drawTriangle() # p
```
will update the current matrix to **TR**
- A vertex **p** of the triangle will be drawn at **TRp**

- Two possible interpretations:
- 1) Rotate first by **R**, then translate by **T** **w.r.t. global coordinates** or,
- 2) Translate first by **T**, then rotate by **R** **w.r.t. local coordinates**

[Practice] Composing Transformations

```
def render():
    # ...
    # edit here
    glColor3ub(255, 255, 255)

    glTranslatef(.4, .0, 0)
    glRotatef(60, 0, 0, 1)

    # now swap the order
    glRotatef(60, 0, 0, 1)
    glTranslatef(.4, .0, 0)

    drawTriangle()
```

Quiz #3

- Go to https://www.slido.com/
- Join #cg-hyu
- 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”.

Next Time

- Lab in this week:
  - Lab assignment 4

- Next lecture:
  - 5 - Affine Geometry, Rendering Pipeline

- Acknowledgement: Some materials come from the lecture slides of
  - Prof. Kayvon Fatahalian and Prof. Keenan Crane, CMU, http://15462.courses.cs.cmu.edu/fall2015/
  - Prof. Jehee Lee, SNU, http://mrl.snu.ac.kr/courses/CourseGraphics/index_2017spring.html
  - Prof. Sung-eui Yoon, KAIST, https://sglab.kaist.ac.kr/~sungeui/CG/