# Advanced Computer Graphics

## 5 - Reference Frame & Composite Trans., OpenGL Transformation Functions

Yoonsang Lee
Fall 2018

# Today's Topics

- Reference Frame & Composite Transformations
  - Coordinate System & Reference Frame
  - Global & Local Coordinate System
  - Composite Transformations

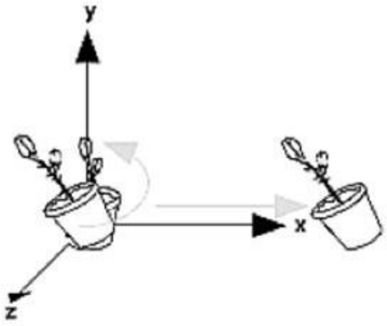
- OpenGL Transformation Functions
  - OpenGL “Current” Transformation Matrix
  - OpenGL Transformation Functions
  - Fundamental Concept of Transformation
  - Composing Transformations using OpenGL Functions

# Reference Frame & Composite Transformations

# Revisit: Order Matters!

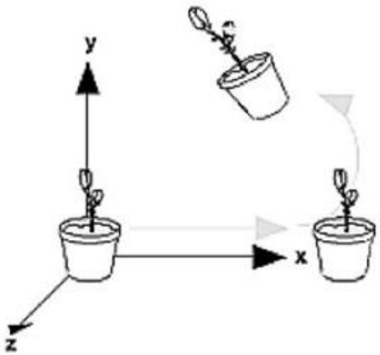
- If T and R are matrices representing affine transformations,

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




Rotate then Translate

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




Translate then Rotate

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

# [Review] 3D Transformations

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

def render(M, camAng):
    # 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)
    gluLookAt(.1*np.sin(camAng),.1, .1*np.cos(camAng), 0,0,0, 0,1,0)
```

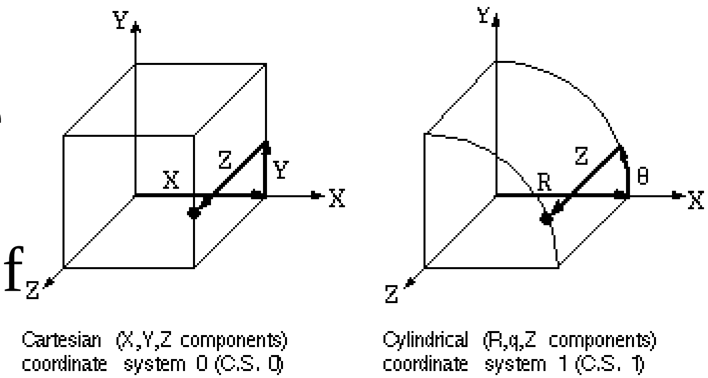
```
    # draw coordinate: x in red, y in green, z in blue
    glBegin(GL_LINES)
    glColor3ub(255, 0, 0)
    glVertex3fv(np.array([0.,0.,0.]))
    glVertex3fv(np.array([1.,0.,0.]))
    glColor3ub(0, 255, 0)
    glVertex3fv(np.array([0.,0.,0.]))
    glVertex3fv(np.array([0.,1.,0.]))
    glColor3ub(0,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

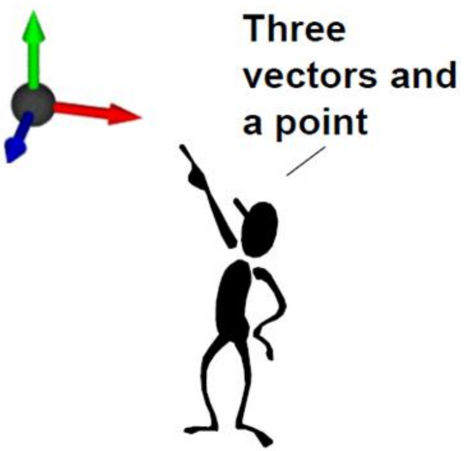
```python
        camAng = np.radians(count% 360)
        render(R, camAng)
        # render(T, camAng)
        # render(T @ R, camAng)
        # render(R @ T, camAng)
        count += 1

        glfw.swap_buffers(window)

    glfw.terminate()

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

# Coordinate System & Reference Frame

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




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



# Coordinate System & Reference Frame

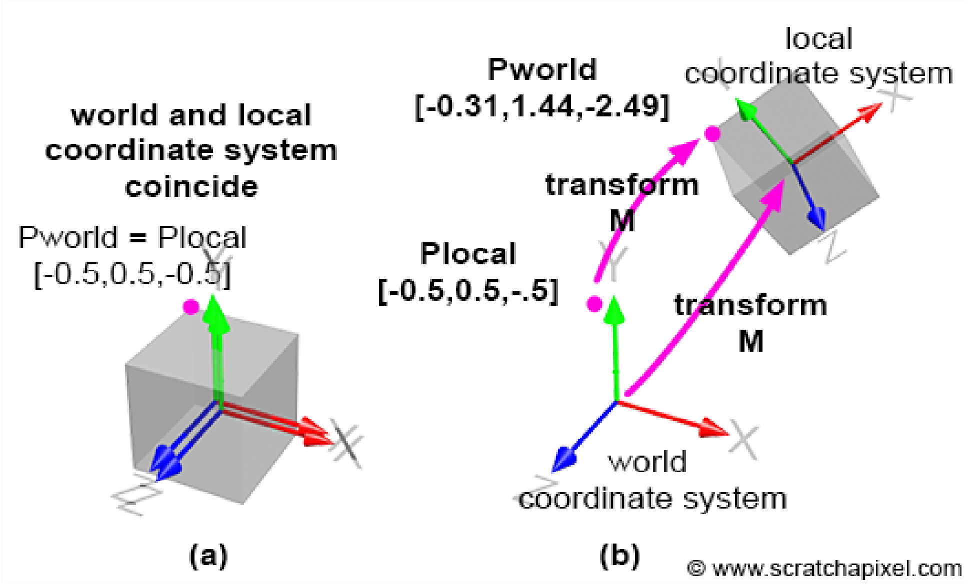
- Two terms are slightly different:
  - **Coordinate system** is a mathematical concept, about a choice of “language” used to describe observations.
  - **Reference frame** is a physical concept related to state of motion.
  - You can think the coordinate system determines the way one describes/observes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

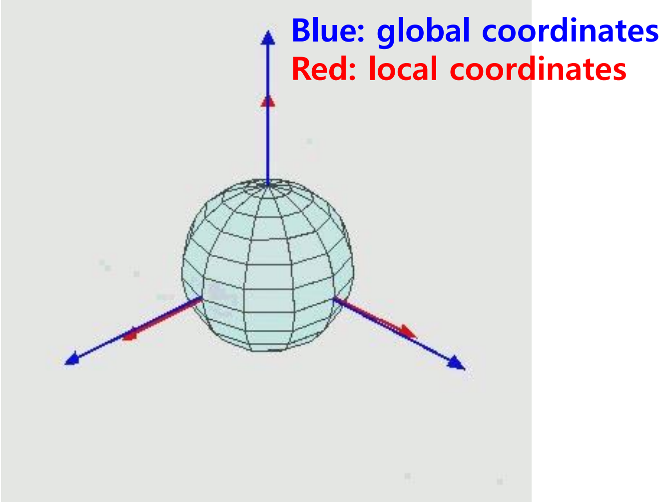
- But these two terms are often mixed.

# Global & Local Coordinate System(or Frame)

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

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




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

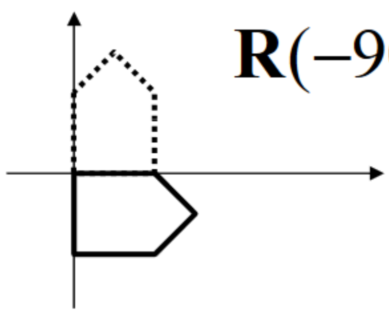
# Interpretation of Composite Transformations #1

- An example transformation:

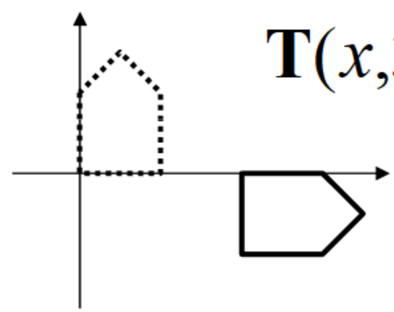
$$T = \mathbf{T}(x,3) \cdot \mathbf{R}(-90°)$$

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

=global coordinate
=world coordinate



$\mathbf{R}(-90°)$



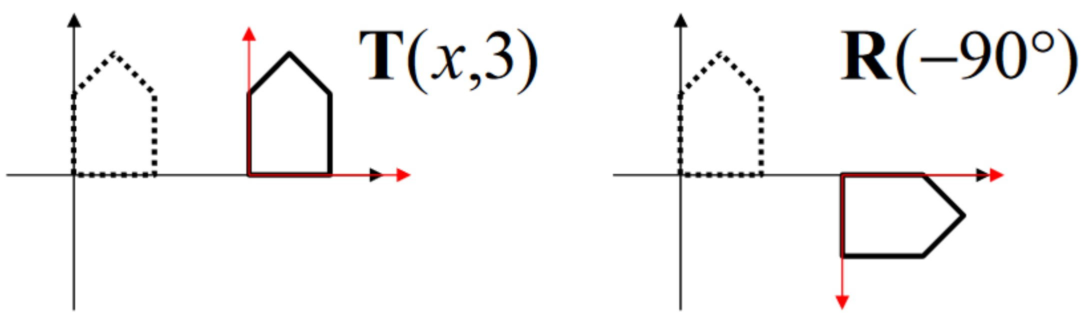
$\mathbf{T}(x,3)$

# 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]

- Use the “[Review] 3D Transformations” practice code and try to interpret again:
- `render(T @ R)`
- `render(R @ T)`

# 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,
- C is the current transformation matrix

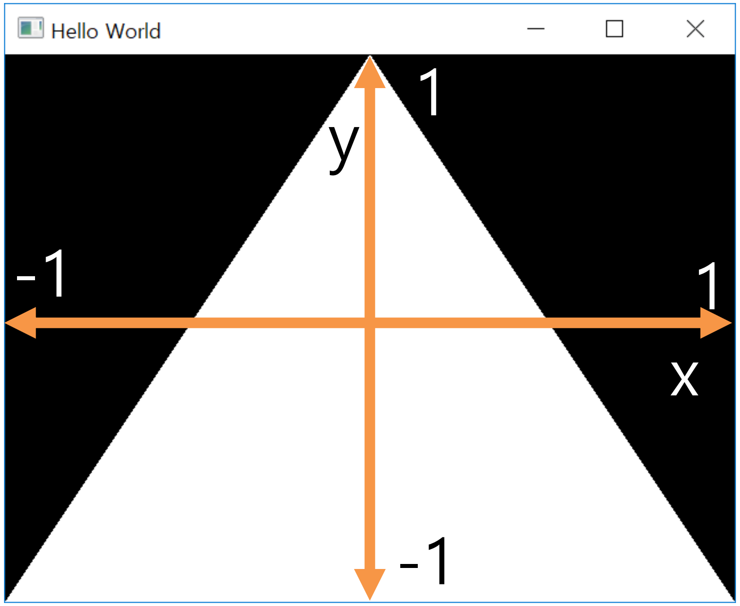
- If you set the vertex position using glVertex3fv(**p**),
- OpenGL will draw the vertex at the position of C**p**

# OpenGL “Current” Transformation Matrix

- Except the “3D Transformation” example (which uses 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] …and add two 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(camAng):
    # ...
    # 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:
- $C \leftarrow C\mathbf{S}$ (**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(camAng):
    # ...
    # 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(camAng):
    # ...
    # 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

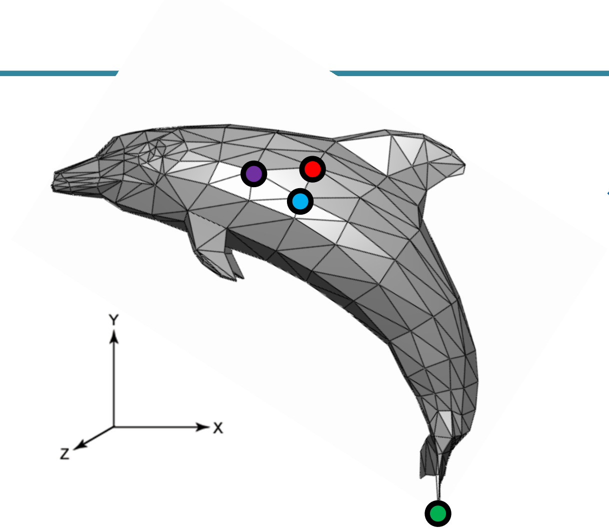
- Calling glTranslate*(*x*, *y*, *z*) will update the current matrix as follows:
- C ← C**T** (**right-multiplication by T**)

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

# [Practice] glTranslate*()

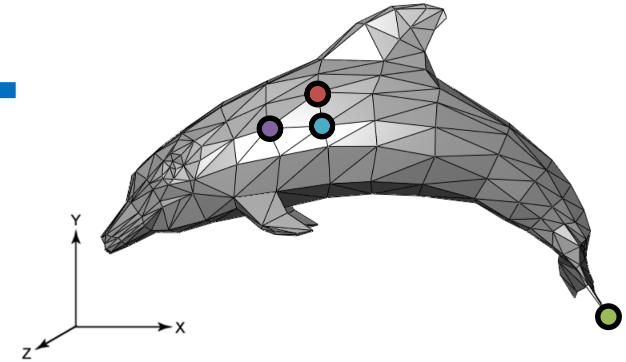
```python
def render(camAng):
    # ...
    # edit here
    glColor3ub(255, 255, 255)

    # 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 Concept of Transformation




Affine transformation

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

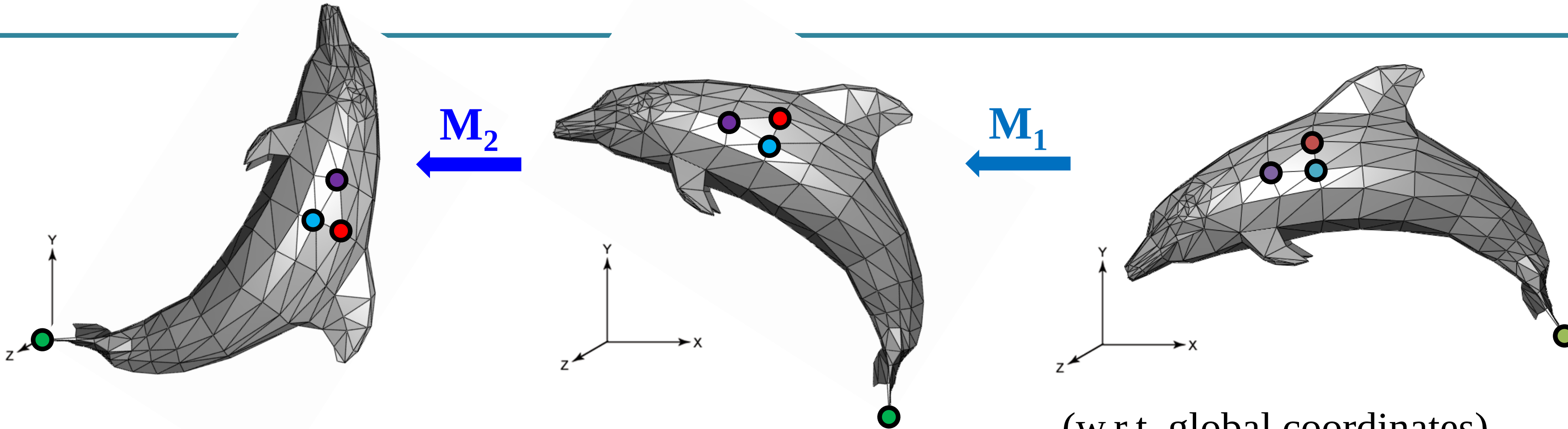



$$\mathbf{p}_1' \leftarrow \mathbf{M}\,\mathbf{p}_1$$
$$\mathbf{p}_2' \leftarrow \mathbf{M}\,\mathbf{p}_2$$
$$\mathbf{p}_3' \leftarrow \mathbf{M}\,\mathbf{p}_3$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$\mathbf{p}_N' \leftarrow \mathbf{M}\,\mathbf{p}_N$$

| **Fundamental concept**<br>(What we have to do) | Using numpy matrix multiplication<br>(What we've used so far) | Using OpenGL transformation functions<br>(What we've learned today) |
| --- | --- | --- |
| $\mathbf{p}_1' \leftarrow \mathbf{M}\ \mathbf{p}_1$<br>$\mathbf{p}_2' \leftarrow \mathbf{M}\ \mathbf{p}_2$<br>$\mathbf{p}_3' \leftarrow \mathbf{M}\ \mathbf{p}_3$<br>$\vdots$<br>$\mathbf{p}_N' \leftarrow \mathbf{M}\ \mathbf{p}_N$ | glVertex3fv($\mathbf{M}\mathbf{p}_1$)<br>glVertex3fv($\mathbf{M}\mathbf{p}_2$)<br>glVertex3fv($\mathbf{M}\mathbf{p}_3$)<br>.<br>.<br>glVertex3fv($\mathbf{M}\mathbf{p}_N$)<br>(slicing is omitted) | **glMultMatrixf($\mathbf{M}^T$)**<br>glVertex3fv($\mathbf{p}_1$)<br>glVertex3fv($\mathbf{p}_2$)<br>glVertex3fv($\mathbf{p}_3$)<br>.<br>.<br>glVertex3fv($\mathbf{p}_N$)<br>(or you can use **glScalef(x,y,z)**, **glRotatef(ang,x,y,z)**, **glTranslatef(x,y,z)**) |
| An array that stores all vertex data.<br>This enables very fast drawing. | | |

| **Fundamental concept**<br>(What we have to do) | Using numpy matrix multiplication<br>(What we've used so far) | Using OpenGL transformation functions<br>(What we've learned today) |
|---|---|---|
| $\mathbf{p}_1' \leftarrow \mathbf{M}\ \mathbf{p}_1$<br>$\mathbf{p}_2' \leftarrow \mathbf{M}\ \mathbf{p}_2$<br>$\mathbf{p}_3' \leftarrow \mathbf{M}\ \mathbf{p}_3$<br>$\vdots$<br>$\mathbf{p}_N' \leftarrow \mathbf{M}\ \mathbf{p}_N$ | glVertex3fv($\mathbf{Mp}_1$)<br>glVertex3fv($\mathbf{Mp}_2$)<br>glVertex3fv($\mathbf{Mp}_3$)<br>.<br>.<br>glVertex3fv($\mathbf{Mp}_N$)<br>(slicing is omitted) | **glMultMatrixf($\mathbf{M}^T$)**<br>glVertex3fv($\mathbf{p}_1$)<br>glVertex3fv($\mathbf{p}_2$)<br>glVertex3fv($\mathbf{p}_3$)<br>.<br>.<br>glVertex3fv($\mathbf{p}_N$)<br>(or you can use **glScalef(x,y,z), glRotatef(ang,x,y,z), glTranslatef(x,y,z)**) |
| An array that stores all vertex data.<br>This enables very fast drawing. | • Not applicable to serious OpenGL programs (Because they do not use glVertex3f()! Instead they use *vertex array*)<br>• CPU performs all matrix multiplications | • This is the **usual legacy OpenGL way**<br>• Can be used with *vertex array*<br>• Faster than the left method because GPU performs matrix multiplications |

# Fundamental Concept of Transformation




(w.r.t. global coordinates)

$$
\begin{array}{lcl}
p_1' & \leftarrow & \mathbf{M_2}\,\mathbf{M_1}\,p_1 \\
p_2' & \leftarrow & \mathbf{M_2}\,\mathbf{M_1}\,p_2 \\
p_3' & \leftarrow & \mathbf{M_2}\,\mathbf{M_1}\,p_3 \\
\vdots & & \vdots \quad \vdots \quad \vdots \\
p_N' & \leftarrow & \mathbf{M_2}\,\mathbf{M_1}\,p_N
\end{array}
$$

| **Fundamental concept** (What we have to do) | Using numpy matrix multiplication (What we've used so far) | Using OpenGL transformation functions (What we've learned today) |
|---|---|---|
| $p_1' \leftarrow M_2 M_1 p_1$<br>$p_2' \leftarrow M_2 M_1 p_2$<br>$p_3' \leftarrow M_2 M_1 p_3$<br>.<br>.<br>.<br>$p_N' \leftarrow M_2 M_1 p_N$ | glVertex3fv($M_2M_1p_1$)<br>glVertex3fv($M_2M_1p_2$)<br>glVertex3fv($M_2M_1p_3$)<br>.<br>.<br>glVertex3fv($M_2M_1p_N$)<br><br>(slicing is omitted) | **glMultMatrixf($M_2^T$)**<br>**glMultMatrixf($M_1^T$)**<br>…or…<br>**glMultMatrixf($(M_2M_1)^T$)**<br>glVertex3fv($p_1$)<br>glVertex3fv($p_2$)<br>glVertex3fv($p_3$)<br>.<br>.<br>glVertex3fv($p_N$)<br><br>(or you can use combination of **glScalef(x,y,z), glRotatef(ang,x,y,z), glTranslatef(x,y,z)**) |

# Fundamental Concept is Important!

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

$$
\begin{array}{lcll}
p_1' & \leftarrow & M & p_1 \\
p_2' & \leftarrow & M & p_2 \\
p_3' & \leftarrow & M & p_3 \\
\vdots & & \vdots & \vdots \\
p_N' & \leftarrow & M & p_N
\end{array}
$$

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

- Not this one:

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

# Fundamental Concept is Important!

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

- 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 matrix multiplication, not OpenGL transform functions.
  - The fundamental concept 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 TR**p**

- 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(camAng):
    # ...
    # 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()
```

# Next Time

- Affine Matrix, Hierarchical Modeling


- Acknowledgement: Some materials come from the lecture slides of
  - Prof. Jehee Lee, SNU, http://mrl.snu.ac.kr/courses/CourseGraphics/index_2017spring.html