Computer Graphics

4 - Transformation 2

Yoonsang Lee and Taesoo Kwon

Spring 2019

Topics Covered

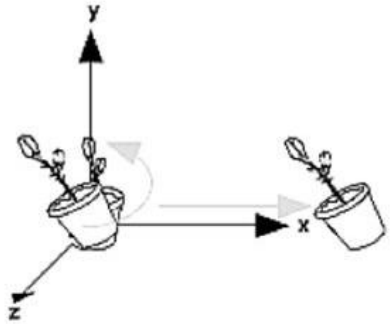
- Reference Frame & Composite Transformations
  - Coordinate System & Reference Frame
  - Global & Local Coordinate System
  - Interpretation of Composite Transformations

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

Reference Frame & Composite Transformations

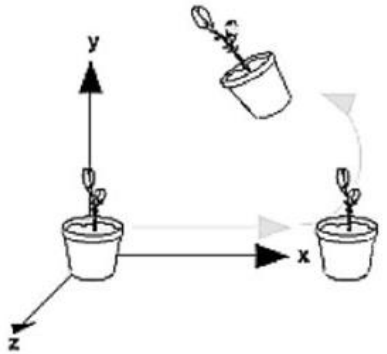
Revisit: Order Matters!

- If T and R are matrices representing affine transformations,



Rotate then Translate

- $\mathbf{p}' = TR\mathbf{p}$
  - First apply transformation R to point **p**, then apply transformation T to transformed point R**p**



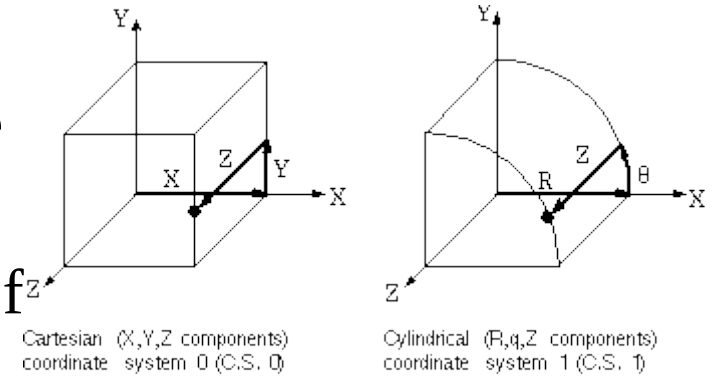
Translate then Rotate

- $\mathbf{p}' = RT\mathbf{p}$
  - First apply transformation T to point **p**, then apply transformation R to transformed point T**p**

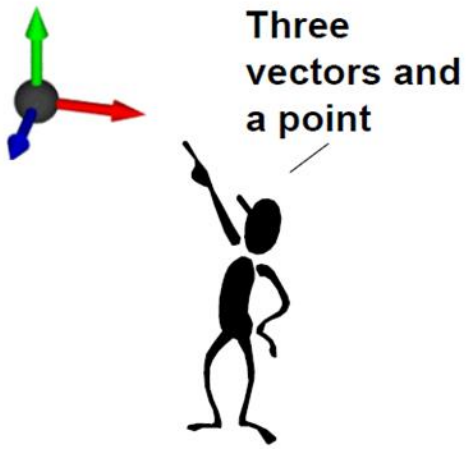
- Note that these are done **w.r.t. global coordinate system**

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

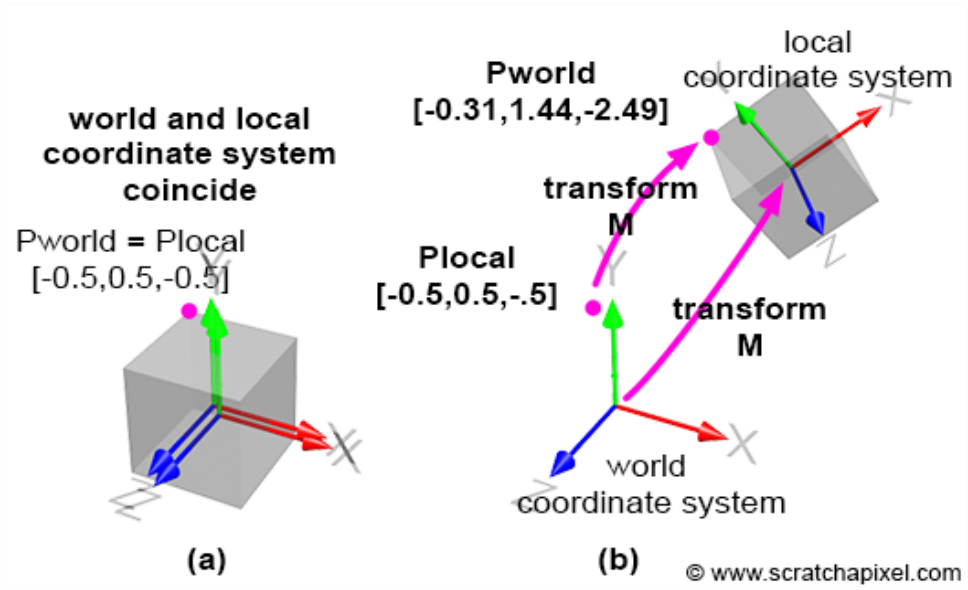
- Two terms are slightly different:
  - **Coordinate system** is a mathematical concept, about a choice of "language" used to describe observations.
  - **Reference frame** is a physical concept related to the state of motion.
  - You can think the coordinate system determines the way one describes/observes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

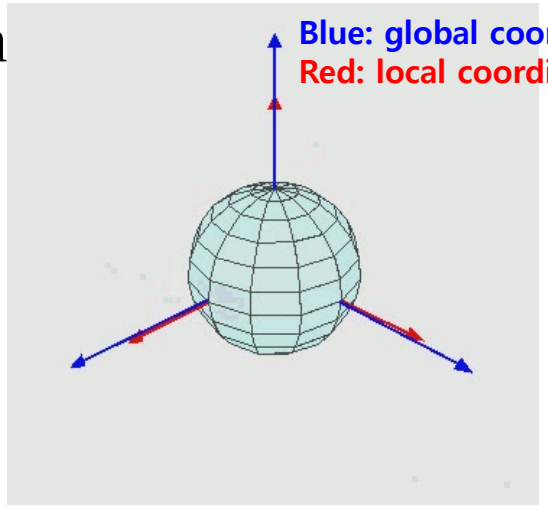
- But these two terms are often mixed.

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**) attach





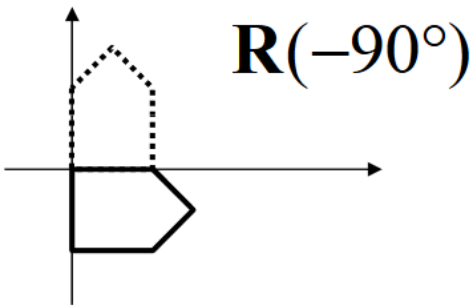
HTTP://commons.wikimedia.org/wiki/File:Euler2a.gif

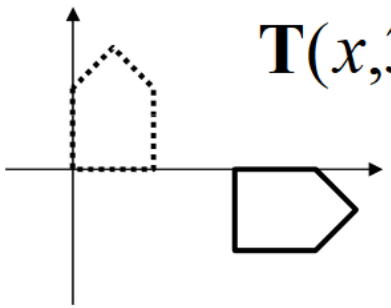
Interpretation of Composite Transformations #1

- 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





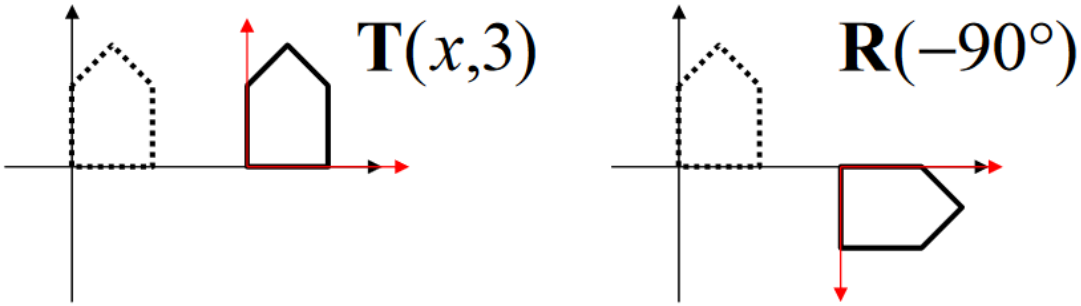
=global coordinate
=world coordinate

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

```
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()
```

```
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)

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

```
        # 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]

        camAng = t
        # render(R, camAng)
        # render(T, camAng)

        # try to interpret below two lines
        render(T @ R, camAng)
        # render(R @ T, camAng)

        glfw.swap_buffers(window)

    glfw.terminate()

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

OpenGL Transformation Functions

OpenGL "Current" Transformation Matrix

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

- An OpenGL context keeps the "current" transformation matrix somewhere

OpenGL "Current" Transformation Matrix

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

- Let's assume that **p** is the position of a vertex in an object represented locally to the object,
- and **C** is the current transformation matrix,

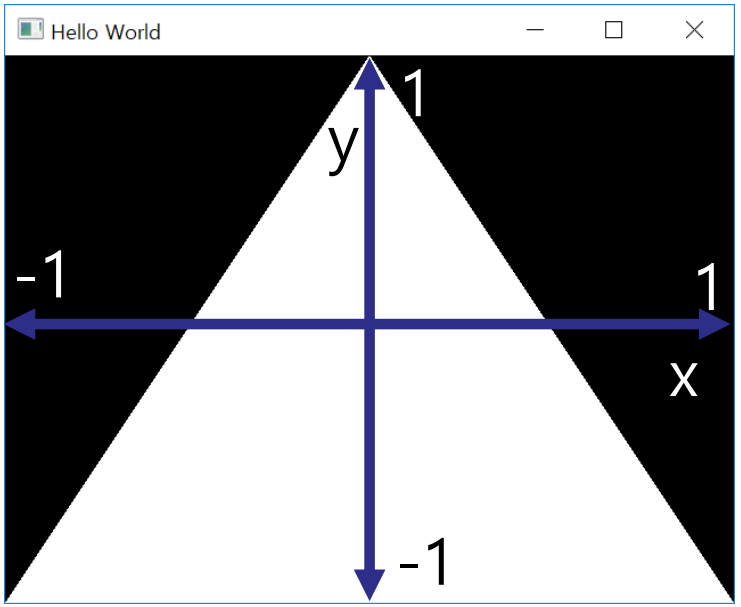
- If you set the vertex position using glVertex3fv(**p**),
  OpenGL will draw the vertex at the location **C p**

OpenGL "Current" Transformation Matrix

All the previous examples so far used the **identity matrix** as the current model-view 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.

- Whenever you want to change the current transformation matrix, first set the current matrix to the identity matrix using **glLoadIdentity().**

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

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

glMultMatrix*()

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

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(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 ← 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(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

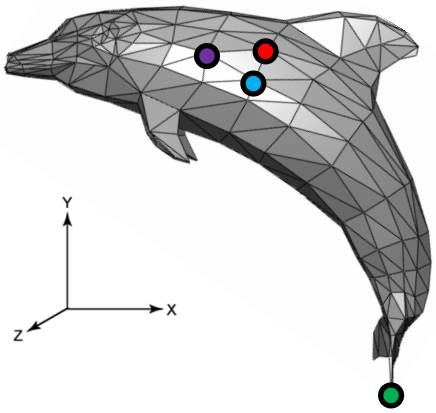
- Calling glTranslate*(*x*, *y*, *z*) will update the current matrix as follows:
- C ← C**T** (**right-multiplication by T**)

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

[Practice] glTranslate*()

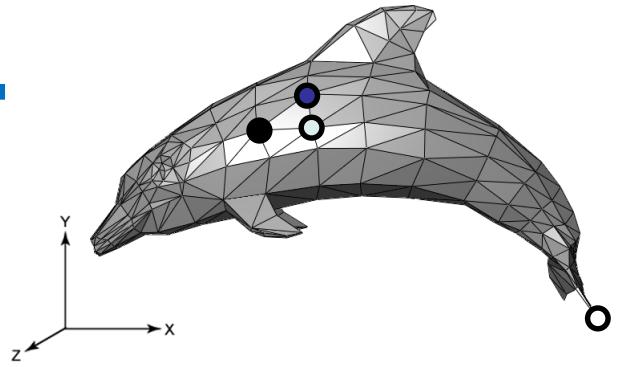
```
def render(camAng):
    # ...
    # edit here
    glColor3ub(255, 255, 255)

    # 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)
```

Transformation



Affine transformation

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

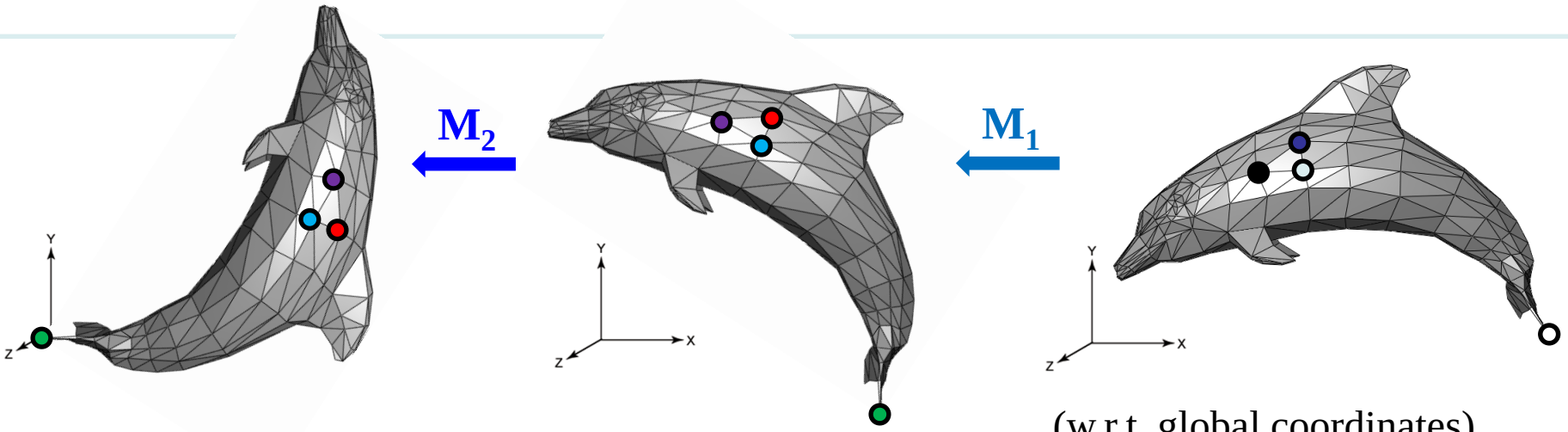


$$\mathbf{p}_1' \leftarrow \mathbf{M}\,\mathbf{p}_1$$
$$\mathbf{p}_2' \leftarrow \mathbf{M}\,\mathbf{p}_2$$
$$\mathbf{p}_3' \leftarrow \mathbf{M}\,\mathbf{p}_3$$
$$\vdots$$
$$\mathbf{p}_N' \leftarrow \mathbf{M}\,\mathbf{p}_N$$

Transformation	Using numpy matrix multiplication (What we've used so far)	Using OpenGL transformation functions (What we've learned today)
$p_1' \leftarrow M\ p_1$ $p_2' \leftarrow M\ p_2$ $p_3' \leftarrow M\ p_3$ . . . $p_N' \leftarrow M\ p_N$	glVertex3fv(Mp_1) glVertex3fv(Mp_2) glVertex3fv(Mp_3) . . glVertex3fv(Mp_N)	**glMultMatrixf(M)** glVertex3fv(p_1) glVertex3fv(p_2) glVertex3fv(p_3) . . glVertex3fv(p_N) (or you can use **glScalef(x,y,z), glRotatef(ang,x,y,z), glTranslatef(x,y,z)**)
	• CPU performs all matrix multiplications	• This is the **usual legacy OpenGL way** • Can be used with *vertex array* • Faster than the left method because GPU performs matrix multiplications

An array that stores all vertex data.
This enables very fast drawing.

Composite Transformation



(w.r.t. global coordinates)

$$p_1' \leftarrow M_2 M_1 p_1$$
$$p_2' \leftarrow M_2 M_1 p_2$$
$$p_3' \leftarrow M_2 M_1 p_3$$
$$\vdots$$
$$p_N' \leftarrow M_2 M_1 p_N$$

Composite Transformation	Using numpy matrix multiplication (What we've used so far)	Using OpenGL transformation functions (What we've learned today)
$p_1' \leftarrow M_2 M_1 p_1$ $p_2' \leftarrow M_2 M_1 p_2$ $p_3' \leftarrow M_2 M_1 p_3$ . . . $p_N' \leftarrow M_2 M_1 p_N$	glVertex3fv($M_2M_1p_1$) glVertex3fv($M_2M_1p_2$) glVertex3fv($M_2M_1p_3$) . . glVertex3fv($M_2M_1p_N$)	**glMultMatrixf(M_2)** **glMultMatrixf(M_1)** …or… **glMultMatrixf(M_2M_1)** glVertex3fv(p_1) glVertex3fv(p_2) glVertex3fv(p_3) . . glVertex3fv(p_N) (or you can use combination of **glScalef(x,y,z), glRotatef(ang,x,y,z), glTranslatef(x,y,z)**) (don't forget to transpose the input matrix when using a row-major np.ndarray)

Composing Transformations using OpenGL Functions

- Let's suppose that 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:

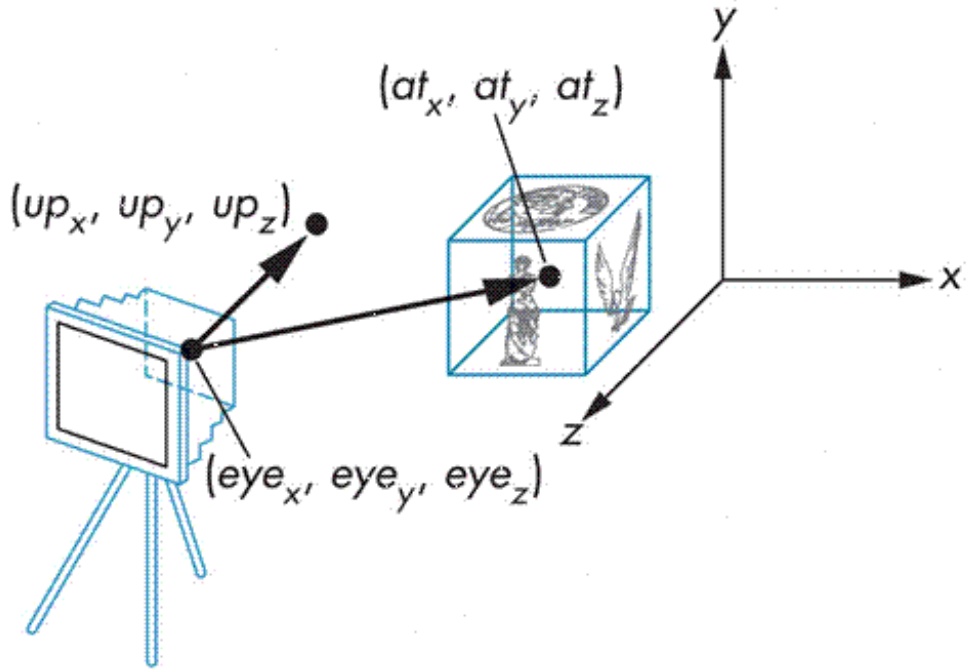
1) Rotate the triangles first by R, then translate by T **w.r.t. global coordinates** or,

2) Transform the local coordinate frame first by T then 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()
```

gluLookAt()



gluLookAt (eyex,eyey,eyez,atx,aty,atz,upx, upy,upz)

: creates a viewing matrix and right-multiplies the current transformation matrix by it

C ← CMv

[Practice] gluLookAt()

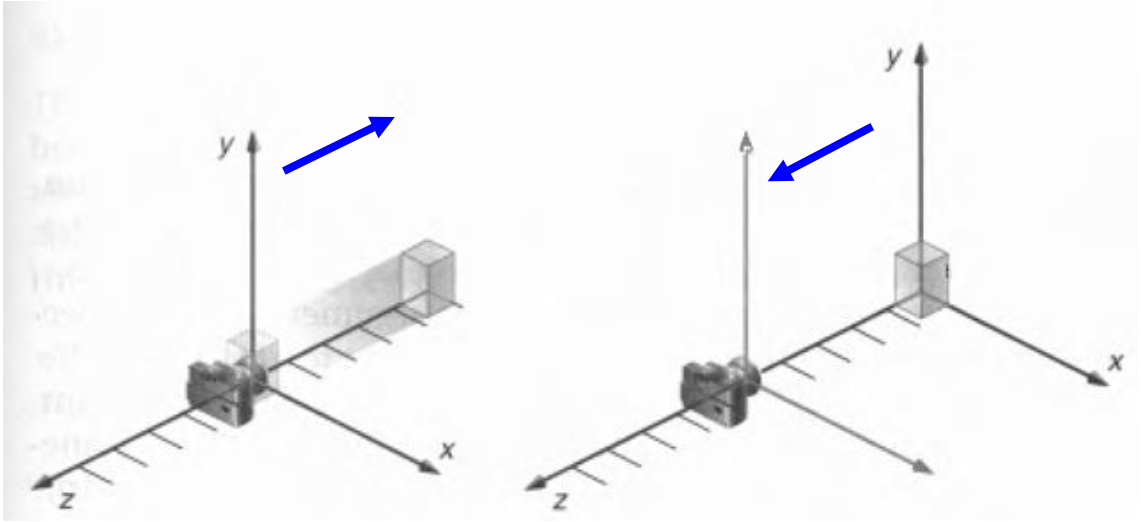
```
import glfw
from OpenGL.GL import *
from OpenGL.GLU import *
import numpy as np

gCamAng = 0.
gCamHeight = .1

def render():
    # 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 (right-multiply the current matrix by viewing
matrix)
    # try to change parameters
    gluLookAt(.1*np.sin(gCamAng),gCamHeight,.1*np.cos(gCamAng), 0,0,0, 0,1,0)

    drawFrame()

    glColor3ub(255, 255, 255)
    drawTriangle()
```

```
def drawFrame():
    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()

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

def key_callback(window, key, scancode, action,
mods):
    global gCamAng, gCamHeight
    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)
        elif key==glfw.KEY_2:
            gCamHeight += .1
        elif key==glfw.KEY_W:
            gCamHeight += -.1
```

```
def main():
    if not glfw.init():
        return
    window =
glfw.create_window(640,640,'gluLookAt()',
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()
        glfw.swap_buffers(window)

    glfw.terminate()

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

Moving Camera vs. Moving World

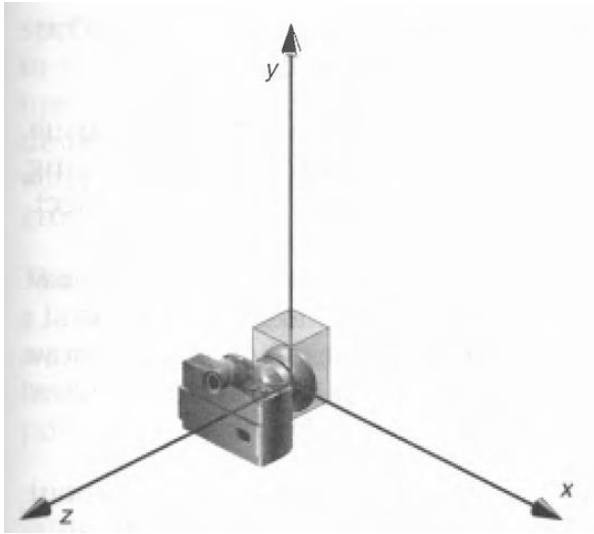
- Actually, these are two **equivalent operations**
- Translate camera by (1, 0, 2) == Translate world by (-1, 0, -2)
- Rotate camera by 60° about y ==Rotate world by -60° about y



Moving Camera vs. Moving World

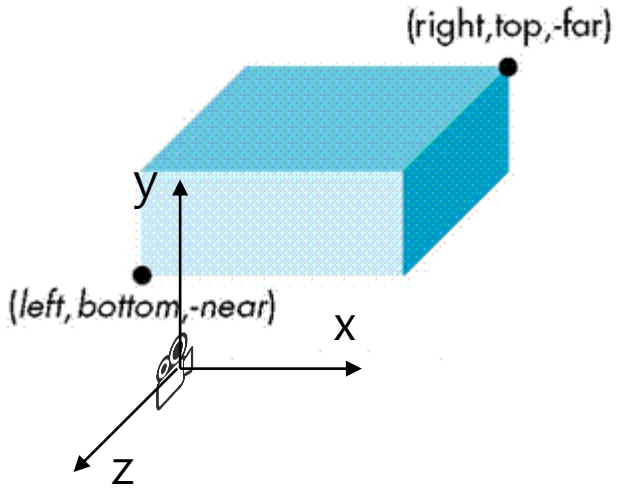
- Thus **you also can use glRotate*() or glTranslate*() to manipulate the camera!**
- Using gluLookAt() is just one of many other options to manipulate the camera

- By default, OpenGL places a camera at the origin pointing in **negative z direction**.



glOrtho()

- glOrtho(left, right, bottom, top, zNear, zFar)
- : Creates an orthographic projection matrix and
- right-multiplies the current transformation matrix
- by it
  - zNear, zFar: These values are negative if the plane is to be behind the viewer.
- $C \leftarrow CM_{orth}$



[Practice] glOrtho

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

gCamAng = 0.
gCamHeight = 1.

# draw a cube of side 1, centered at the origin.
def drawUnitCube():
    glBegin(GL_QUADS)
    glVertex3f( 0.5, 0.5,-0.5)
    glVertex3f(-0.5, 0.5,-0.5)
    glVertex3f(-0.5, 0.5, 0.5)
    glVertex3f( 0.5, 0.5, 0.5)

    glVertex3f( 0.5,-0.5, 0.5)
    glVertex3f(-0.5,-0.5, 0.5)
    glVertex3f(-0.5,-0.5,-0.5)
    glVertex3f( 0.5,-0.5,-0.5)

    glVertex3f( 0.5, 0.5, 0.5)
    glVertex3f(-0.5, 0.5, 0.5)
    glVertex3f(-0.5,-0.5, 0.5)
    glVertex3f( 0.5,-0.5, 0.5)

    glVertex3f( 0.5,-0.5,-0.5)
    glVertex3f(-0.5,-0.5,-0.5)
    glVertex3f(-0.5, 0.5,-0.5)
    glVertex3f( 0.5, 0.5,-0.5)
```

```
    glVertex3f(-0.5, 0.5, 0.5)
    glVertex3f(-0.5, 0.5,-0.5)
    glVertex3f(-0.5,-0.5,-0.5)
    glVertex3f(-0.5,-0.5, 0.5)

    glVertex3f( 0.5, 0.5,-0.5)
    glVertex3f( 0.5, 0.5, 0.5)
    glVertex3f( 0.5,-0.5, 0.5)
    glVertex3f( 0.5,-0.5,-0.5)
    glEnd()

def drawCubeArray():
    for i in range(5):
        for j in range(5):
            for k in range(5):
                glPushMatrix()
                glTranslatef(i,j,-k-1)
                glScalef(.5,.5,.5)
                drawUnitCube()
                glPopMatrix()

def drawFrame():
    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()
```

```
def render():
    global gCamAng, gCamHeight

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glEnable(GL_DEPTH_TEST)

    # draw polygons only with boundary edges
    glPolygonMode( GL_FRONT_AND_BACK, GL_LINE )

    glLoadIdentity()

    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()

# test other parameter values
    # near plane: 10 units behind the camera
    # far plane: 10 units in front of
 the camera
    glOrtho(-5,5, -5,5, -10,10)

    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

gluLookAt(1*np.sin(gCamAng),gCamHeight,1*np.cos(
gCamAng), 0,0,0, 0,1,0)

    drawFrame()
    glColor3ub(255, 255, 255)

    drawUnitCube()

    # test
    # drawCubeArray()
```

```
def key_callback(window, key, scancode, action,
mods):
    global gCamAng, gCamHeight
    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)
        elif key==glfw.KEY_2:
            gCamHeight += .1
        elif key==glfw.KEY_W:
            gCamHeight += -.1

def main():
    if not glfw.init():
        return
    window =
glfw.create_window(640,640,'glOrtho()',
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()
        glfw.swap_buffers(window)

    glfw.terminate()

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

For debugging the model-view matrix, use:

```
model = glGetDoublev(GL_MODELVIEW_MATRIX).T

print(model)
```

Now,

- Lab in this week:
  - Lab assignment 3