Computer Graphics

4 - Transformation 2

Yoonsang Lee
Spring 2022

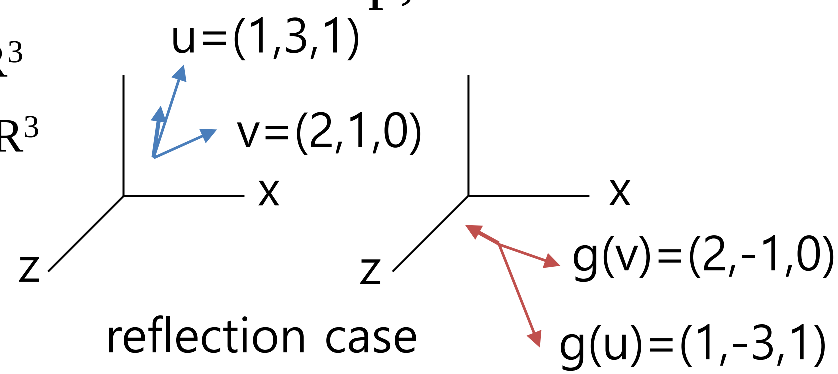
Question from Last Lecture: Rigid Transformation

- Rotation + Translation

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

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

  – Let's say $g : R^3 \rightarrow R^3$ is a rigid transform map, then
    - $\|g(u) - g(v)\| = \|u - v\|, \ \forall u, v \in R^3$
    - $g(u) \times g(v) = g(u \times v), \ \forall u, v \in R^3$



Topics Covered

- 3D Affine Transformation
- OpenGL Transformation Functions
  - OpenGL “Current” Transformation Matrix
  - OpenGL Transformation Functions
  - Composing Transformations using OpenGL Functions
- **Fundamental Idea of Transformation**
- Affine Space & Coordinate-Free Concepts

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):

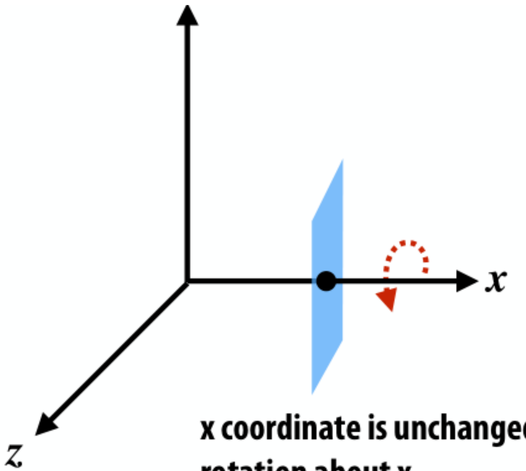
$$\mathbf{H}_{x,\mathbf{d}} = \begin{bmatrix} 1 & \mathbf{d}_y & \mathbf{d}_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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

Linear Transformation in 3D

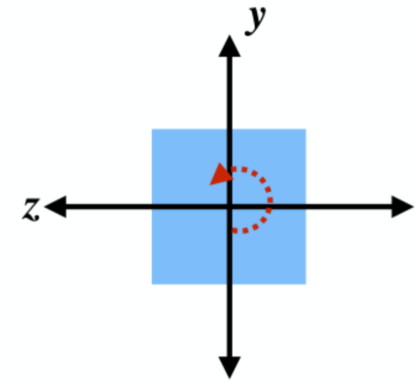
Rotation about x axis:

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



x coordinate is unchanged by rotation about x

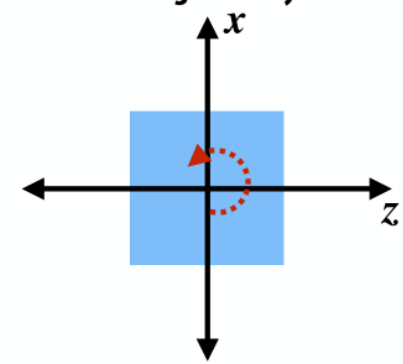
View looking down -x axis:



Rotation about y axis:

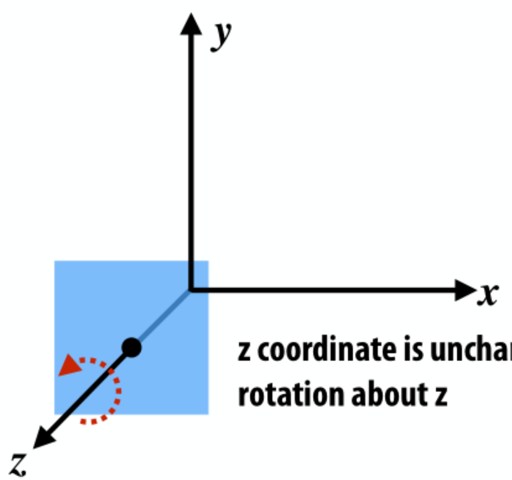
$$\mathbf{R}_{y,\theta} = \begin{bmatrix} \cos\theta & 0 & \boxed{}\sin\theta \\ 0 & 1 & 0 \\ \boxed{-}\sin\theta & 0 & \cos\theta \end{bmatrix}$$

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 system: 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 - p'=Mp
    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()

        # 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.]])
```

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

        glfw.swap_buffers(window)

    glfw.terminate()

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

[Practice] Tips: 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-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”.

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,
- and **C** is the current transformation matrix,

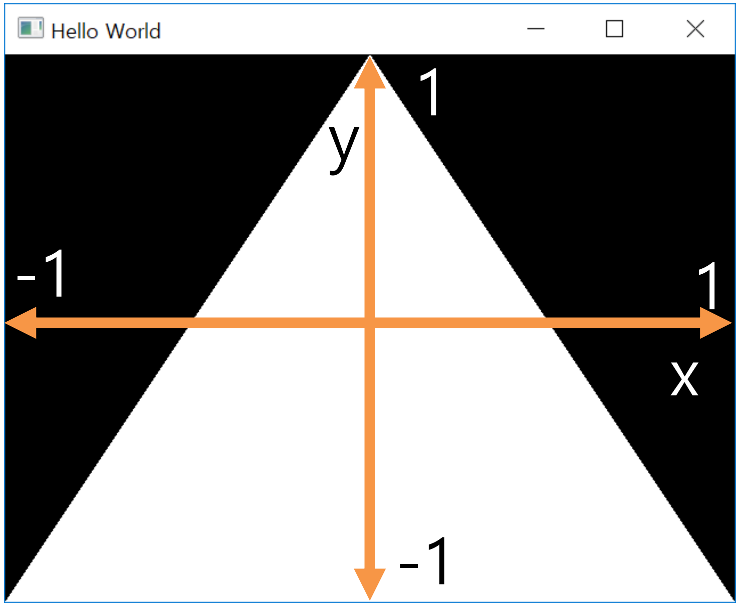
- If you set the vertex position using glVertex3fv(**p**),
- OpenGL will draw the vertex at the position of **Cp**

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,**

- which has been set 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:
- Scale, rotate, translate with parameters
  - **glScale*()**
  - **glRotate*()**
  - **glTranslate*()**
  - OpenGL doesn't provide functions like glShear*() and glReflect*()
- Direct manipulation of the current matrix
  - **glMultMatrix*()**

[Practice] OpenGL Trans. Functions

```python
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
```

```python
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):
    # p1=(0,.5,0), p2=(0,0,0), p3=(.5,0,0)
    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():
    # p1=(0,.5,0), p2=(0,0,0), p3=(.5,0,0)
    glBegin(GL_TRIANGLES)
    glVertex3fv(np.array([.0,.5,0.]))
    glVertex3fv(np.array([.0,.0,0.]))
    glVertex3fv(np.array([.5,.0,0.]))
    glEnd()
```

glScale*()

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

- Let's call the current matrix C
- 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.]) - p'= CSp
    # (C: current transformation matrix at this point)

    # 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*()

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

    # 1)& 2) all draw a triangle with the same transformation
    # (rotate 60 deg about x axis) - p'= CRp
    # (C: current transformation matrix at this point)

    # 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]) - p'= CTp
    # (C: current transformation matrix at this point)

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

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

glMultMatrix*()

- glMultiMatrix*(*m*) - multiply the current transformation matrix with the matrix *m*
  - *m* : 4x4 **column-major** matrix
  - Note that a np.ndarray object stores data in **row-major** order
  - You have to pass the **transpose of np.ndarray** to glMultMatrix()

If this is the memory layout of a stored 4x4 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*()

- Calling glMultMatrix*(M) will update the current matrix as follows:

- C ← C**M** (**right-multiplication by M**)

[Practice] glMultMatrix*()

```python
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 - p`=CRTp
    # (C: current transformation matrix at this moment)

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

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

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

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** (**p'=TRp**)
- → **p** is first rotated by **R**, then translated by **T**.
  ```

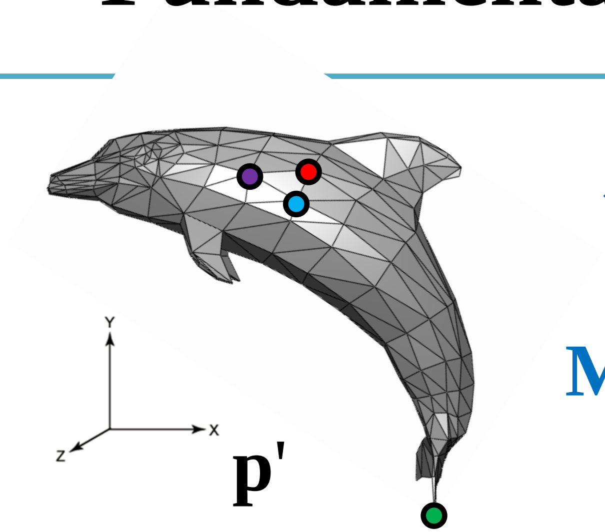
Quiz #2

- 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”.

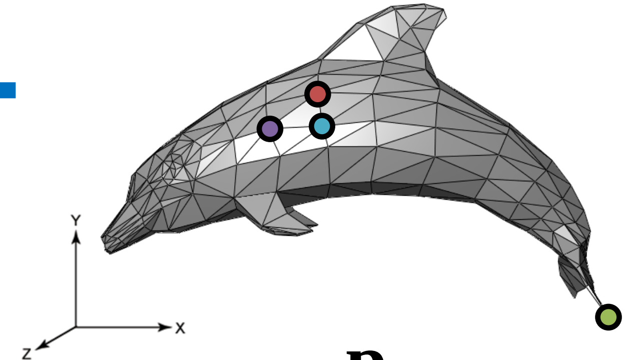
Fundamental Idea of Transformation



p'

Affine transformation

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



p

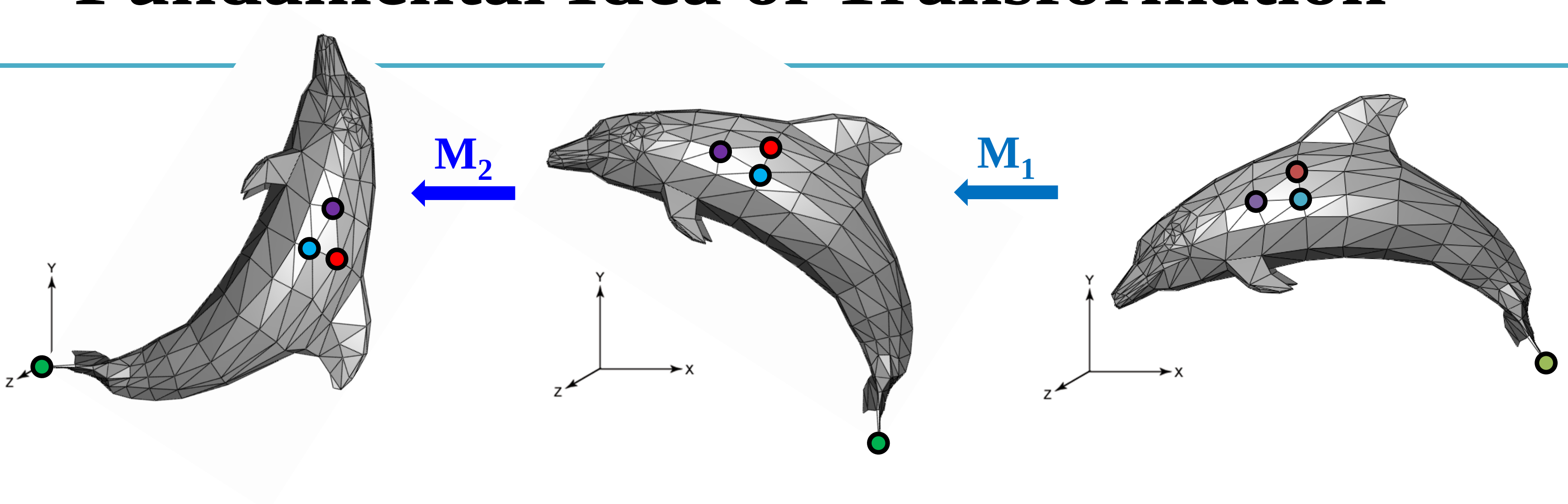
$$\mathbf{p}_1' \leftarrow \mathbf{M}\,\mathbf{p}_1$$
$$\mathbf{p}_2' \leftarrow \mathbf{M}\,\mathbf{p}_2$$
$$\mathbf{p}_3' \leftarrow \mathbf{M}\,\mathbf{p}_3$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$\mathbf{p}_N' \leftarrow \mathbf{M}\,\mathbf{p}_N$$

Fundamental idea	Implementation 1: Using numpy matrix multiplication	Implementation 2: Using OpenGL transformation functions
$p_1' \leftarrow M\ p_1$ $p_2' \leftarrow M\ p_2$ $p_3' \leftarrow M\ p_3$ $\vdots$ $p_N' \leftarrow M\ p_N$	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(M)** *(M.T for numpy array)* 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. (We'll cover it later)		

Fundamental idea	Implementation 1: Using numpy matrix multiplication	Implementation 2: Using OpenGL transformation functions
$p_1' \leftarrow \mathbf{M}\ p_1$ $p_2' \leftarrow \mathbf{M}\ p_2$ $p_3' \leftarrow \mathbf{M}\ p_3$ $\vdots \qquad \vdots \quad \vdots$ $p_N' \leftarrow \mathbf{M}\ p_N$	glVertex3fv($\mathbf{M}p_1$) glVertex3fv($\mathbf{M}p_2$) glVertex3fv($\mathbf{M}p_3$) $\vdots$ glVertex3fv($\mathbf{M}p_N$) (slicing is omitted)	**glMultMatrixf(M)** *(M.T for numpy array)* glVertex3fv(p_1) glVertex3fv(p_2) glVertex3fv(p_3) $\vdots$ glVertex3fv(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. (We'll cover it later)	• 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



$$p_1' \leftarrow M_2\, M_1\, p_1$$
$$p_2' \leftarrow M_2\, M_1\, p_2$$
$$p_3' \leftarrow M_2\, M_1\, p_3$$
$$\vdots \qquad \vdots \quad \vdots \quad \vdots$$
$$p_N' \leftarrow M_2\, M_1\, p_N$$

Fundamental idea	Implementation 1: Using numpy matrix multiplication	Implementation 2: Using OpenGL transformation functions
$p_1' \leftarrow M_2\ M_1\ p_1$ $p_2' \leftarrow M_2\ M_1\ p_2$ $p_3' \leftarrow M_2\ M_1\ p_3$ . . . . $p_N' \leftarrow M_2\ M_1\ p_N$	glVertex3fv($M_2M_1p_1$) glVertex3fv($M_2M_1p_2$) glVertex3fv($M_2M_1p_3$) . . glVertex3fv($M_2M_1p_N$) (slicing is omitted)	**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)**)

Fundamental Idea is Most Important!

- If you see the term "transformation", what you have to think of 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 Idea is Most Important!

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

- For example, 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.
  - They provide much faster computation.

Affine Space & Coordinate-Free Concepts

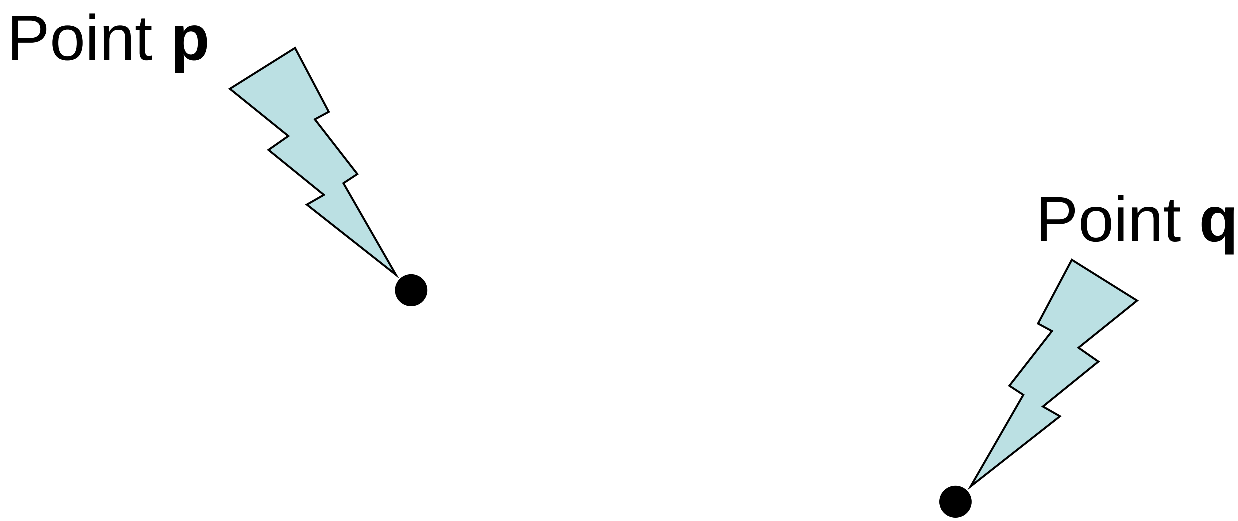
Coordinate-invariant (Coordinate-free)

- Traditionally, computer graphics packages are implemented using *homogeneous coordinates*.

- We will see *affine space* and *coordinate-invariant geometric programming* concepts and their relationship with the homogeneous coordinates.

- Because of historical reasons, it has been called *"coordinate-free"* geometric programming.

Points



- What is the “sum” of these two "points" ?

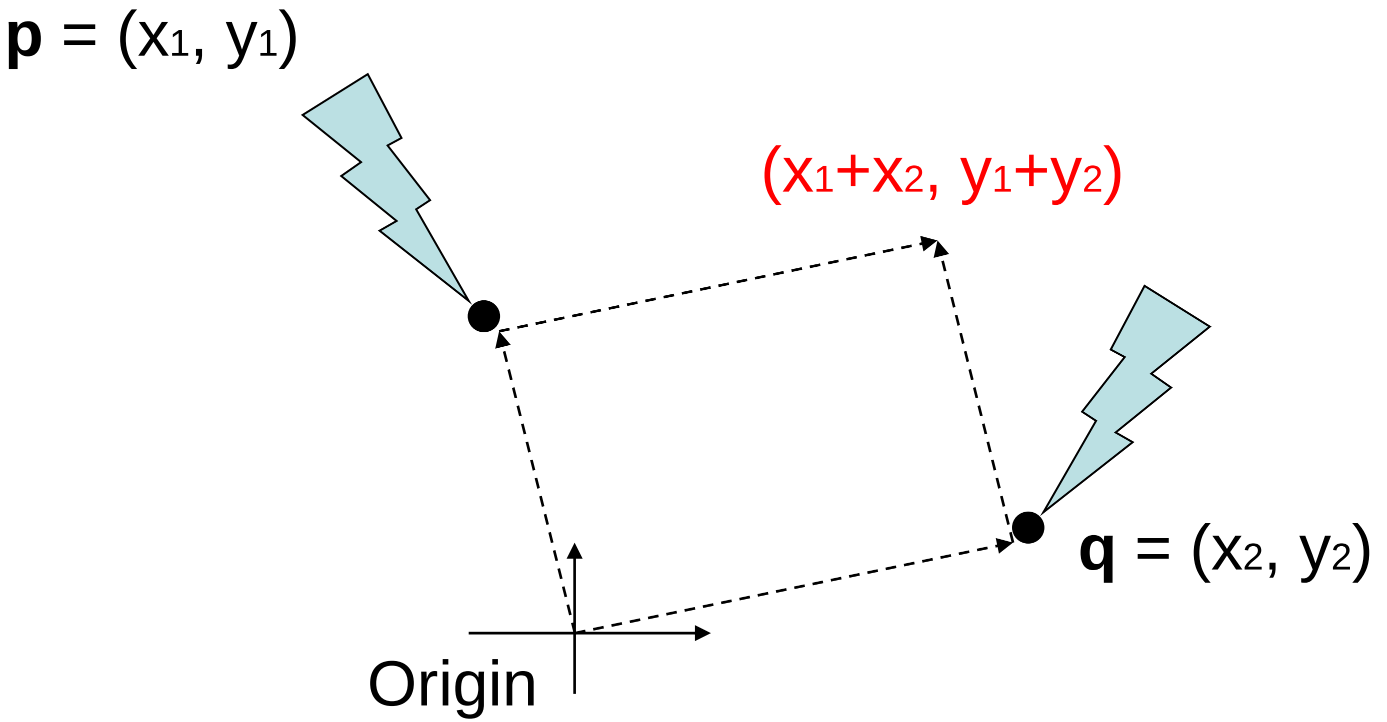
If you assume coordinates, …

p = (x_1, y_1)

q = (x_2, y_2)

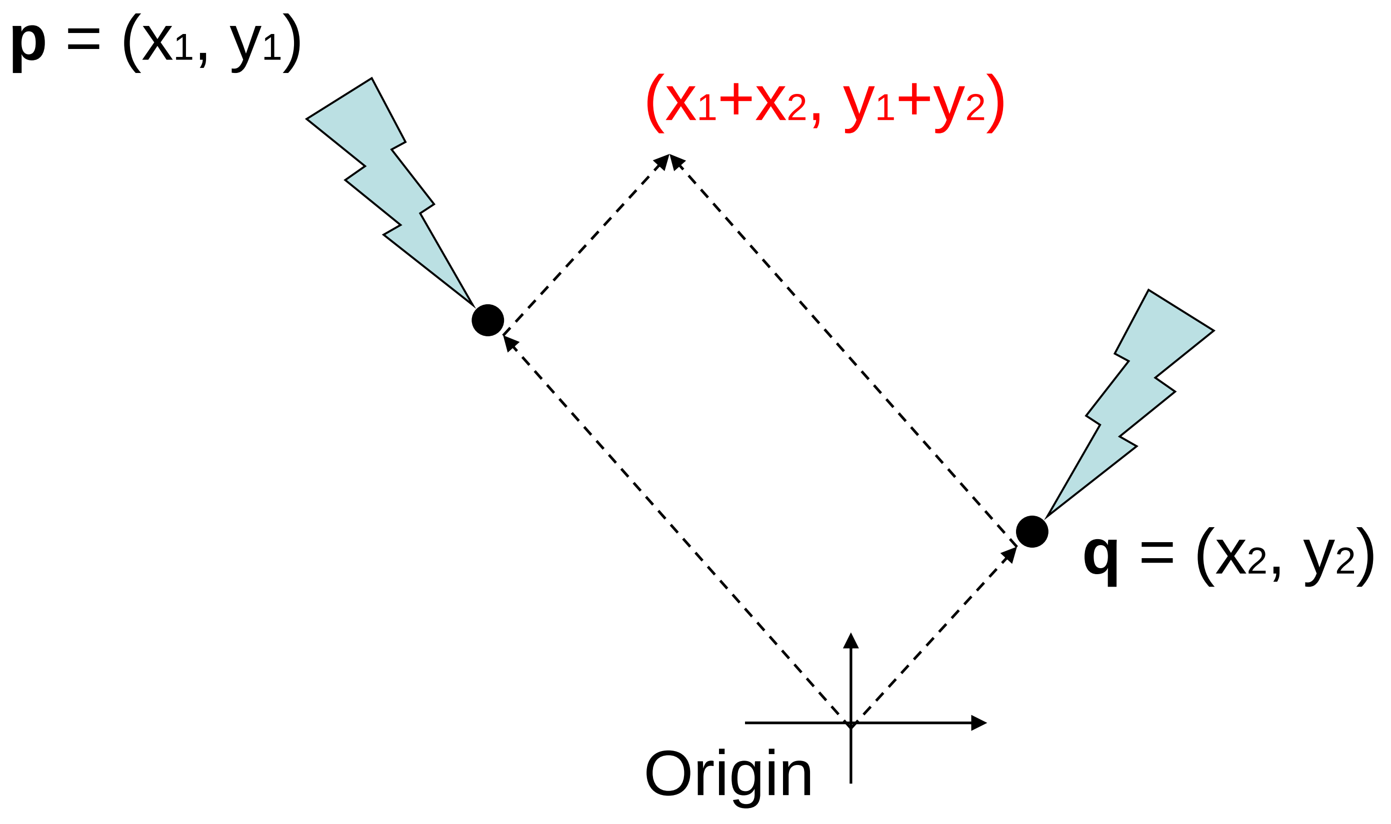
- The sum is (x_1+x_2, y_1+y_2)
  - Is it correct ?
  - Is it geometrically meaningful ?

If you assume coordinates, …



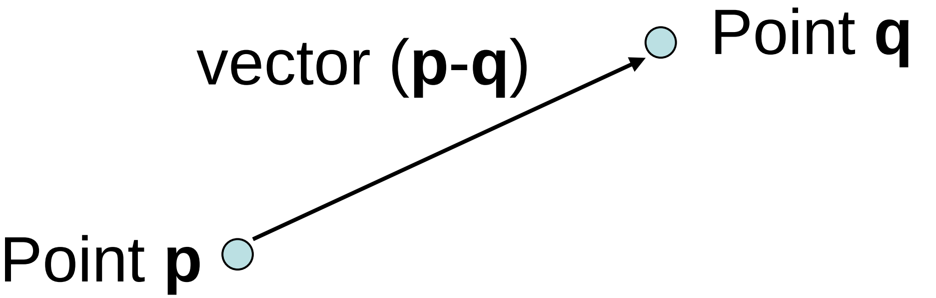
- Vector sum
  - (x_1, y_1) and (x_2, y_2) are considered as vectors from the origin to **p** and **q**, respectively.

If you select a different origin, …



- If you choose a different coordinate frame, you will get a different result

Points and Vectors



- A ***point*** is a position specified with coordinate values.
- A ***vector*** is specified as the difference between two points.
- If an ***origin*** is specified, then a **point** can be represented by a **vector from the origin.**
- But, a point is still not a vector in ***coordinate-free*** concepts.

Points & Vectors are Different!

- Mathematically (and physically),
- *Points* are **locations in space**.
- *Vectors* are **displacements in space**.

- An analogy with time:
- *Times* (or datetimes) are **locations in time**.
- *Durations* are **displacements in time**.

Vector and Affine Spaces

- ***Vector space***
  - Includes vectors and related operations
  - No points

- ***Affine space***
  - Superset of vector space
  - Includes vectors, points, and related operations

Vector spaces

- A ***vector space*** consists of
  - Set of vectors, together with
  - Two operations: addition of vectors and multiplication of vectors by scalar numbers

- A ***linear combination*** of vectors is also a vector

$$\mathbf{u}_0, \mathbf{u}_1, \cdots, \mathbf{u}_N \in V \quad \Rightarrow \quad c_0\mathbf{u}_0 + c_1\mathbf{u}_1 + \cdots + c_N\mathbf{u}_N \in V$$

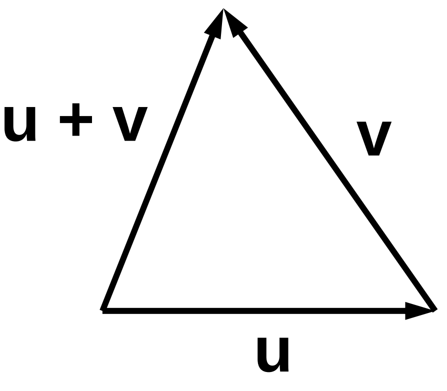
Affine Spaces

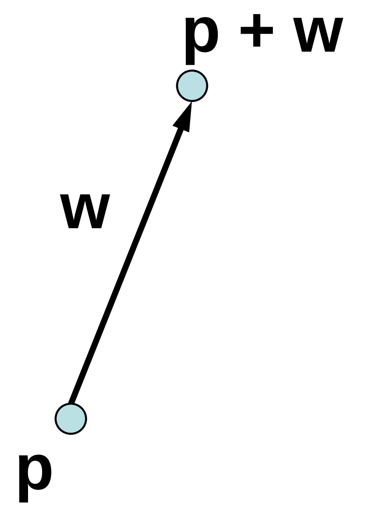
- An ***affine space*** consists of
  - Set of points, an associated vector space, and
  - Two operations: the difference between two points and the addition of a vector to a point

Coordinate-Free Geometric Operations

- Addition
- Subtraction
- Scalar multiplication

Addition



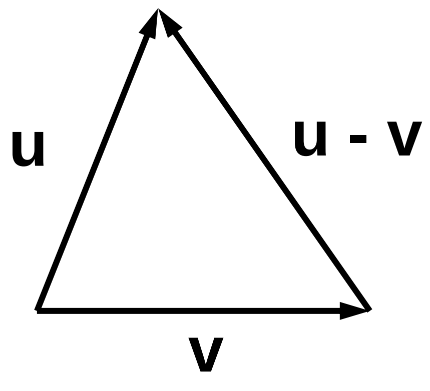


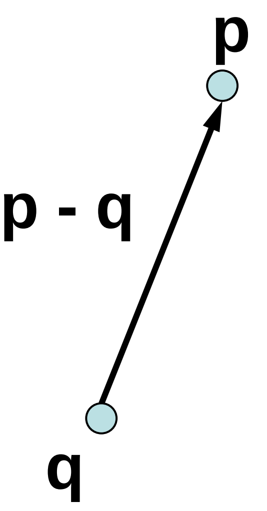
u + **v** is a vector

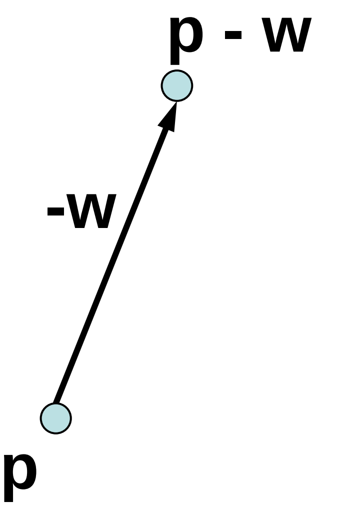
p + **w** is a point

u, v, w : vectors
p, q : points

Subtraction







u - **v** is a vector

p - **q** is a vector

p - **w** is a point

u, v, w : vectors
p, q : points

Scalar Multiplication

scalar • vector = vector

1 • point = point

0 • point = vector

c • point = (undefined)   if ($c \neq 0,1$)

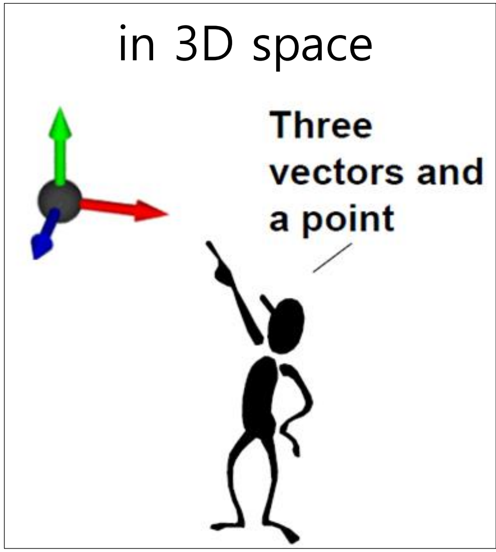
Affine Frame

- A ***frame*** is defined as a set of vectors $\{\mathbf{v}_i \mid i=1, \ldots, N\}$ and a point **o**
  - Set of vectors $\{\mathbf{v}_i\}$ are bases of the associate vector space
  - **o** is an origin of the frame
  - *N* is the dimension of the affine space
  - Any point **p** can be written as

$$\mathbf{p} = \mathbf{o} + c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_N\mathbf{v}_N$$

  - Any vector **v** can be written as

$$\mathbf{v} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_N\mathbf{v}_N$$



Summary

- In an affine space,

point + point = undefined		
point − point = vector		
point ± vector = point		
vector ± vector = vector		
scalar • vector = vector		
scalar • point = point		iff scalar = 1
	= vector	iff scalar = 0
	= undefined	otherwise

Points & Vectors in Homogeneous Coordinates

- In 3D spaces,
- A **point** is represented: (x, y, z, **1**)
- A **vector** can be represented: (x, y, z, **0**)

$(x_1, y_1, z_1, 1) + (x_2, y_2, z_2, 1) = (x_1+x_2, y_1+y_2, z_1+z_2, 2)$

point ***point*** ***undefined***

$(x_1, y_1, z_1, 1) - (x_2, y_2, z_2, 1) = (x_1-x_2, y_1-y_2, z_1-z_2, 0)$

point ***point*** ***vector***

$(x_1, y_1, z_1, 1) + (x_2, y_2, z_2, 0) = (x_1+x_2, y_1+y_2, z_1+z_2, 1)$

point ***vector*** ***point***

A Consistent Model

- **Behavior of affine frame coordinates is completely consistent with our intuition**
  - **Subtracting two points yields a vector**
  - **Adding a vector to a point produces a point**
  - **If you multiply a vector by a scalar you still get a vector**
  - **Scaling points gives a nonsense 4th coordinate element in most cases**

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 - b_1 \\ a_2 - b_2 \\ a_3 - b_3 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix} = \begin{bmatrix} a_1 + v_1 \\ a_2 + v_2 \\ a_3 + v_3 \\ 1 \end{bmatrix}$$

KAIST

Points & Vectors in Homogeneous Coordinates

- Multiplying affine transformation matrix to a point and a vector:

$$\begin{bmatrix} M & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ \boxed{1} \end{bmatrix} = \begin{bmatrix} M\mathbf{p} + \mathbf{t} \\ \boxed{1} \end{bmatrix} \qquad \begin{bmatrix} M & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \boxed{0} \end{bmatrix} = \begin{bmatrix} \boxed{M\mathbf{v}} \\ \boxed{0} \end{bmatrix}$$

point $\longrightarrow$ point

vector $\longrightarrow$ vector

- Note that translation is not applied to a vector!

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 for this lecture (next Monday):
  - Lab assignment 4

- Next lecture:
  - 5 - Rendering Pipeline, Viewing & Projection 1

- 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/