Computer Graphics

6 - Projection, Mesh 1

Yoonsang Lee
Spring 2022

Midterm Exam Announcement

- Date & time: **Apr 27**, 09:30 - 10:30 am
- Place: IT.BT, 508
- Scope: Lecture 2 ~ 7

- **You cannot leave the room until the end of the exam** even if you finish the exam earlier.

- Please bring your student ID card to the exam.

- If you are unable to take the offline exam (stay abroad, corona confirmed, etc.), please contact the TA in advance.
  - Chaejun Sohn (손채준 조교), thscowns@gmail.com

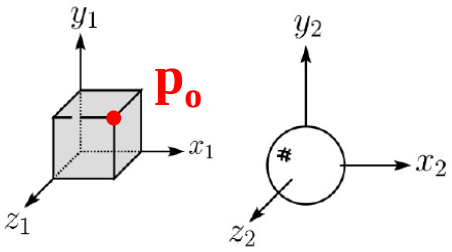
Questions from Last Lecture

- why the order of matrix is MvpMpjMvMm which newer matrix locate left side?

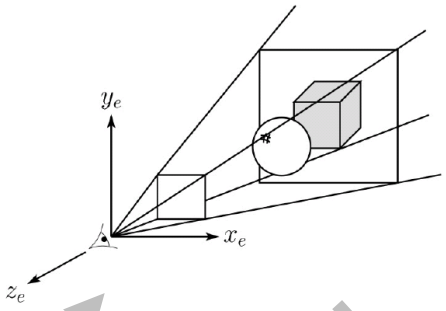
- why vertex processing's multiple order is reversed?

Vertex Processing (Transformation Pipeline)

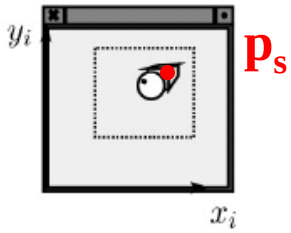
Object space



View space
(Camera space)



Screen space
(Image space)



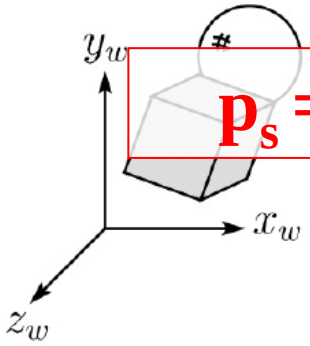
Modeling transformation : M_m

Viewing transformation : M_v

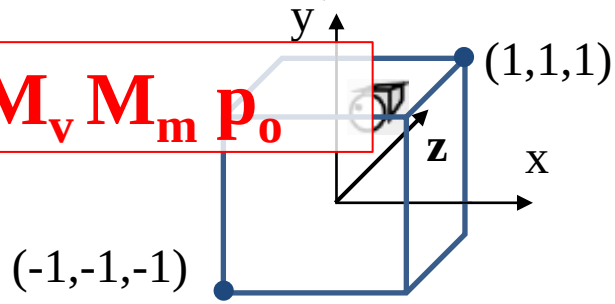
Projection transformation : M_{pj}

Viewport transformation : M_{vp}

$$p_s = M_{vp}\, M_{pj}\, M_v\, M_m\, p_o$$

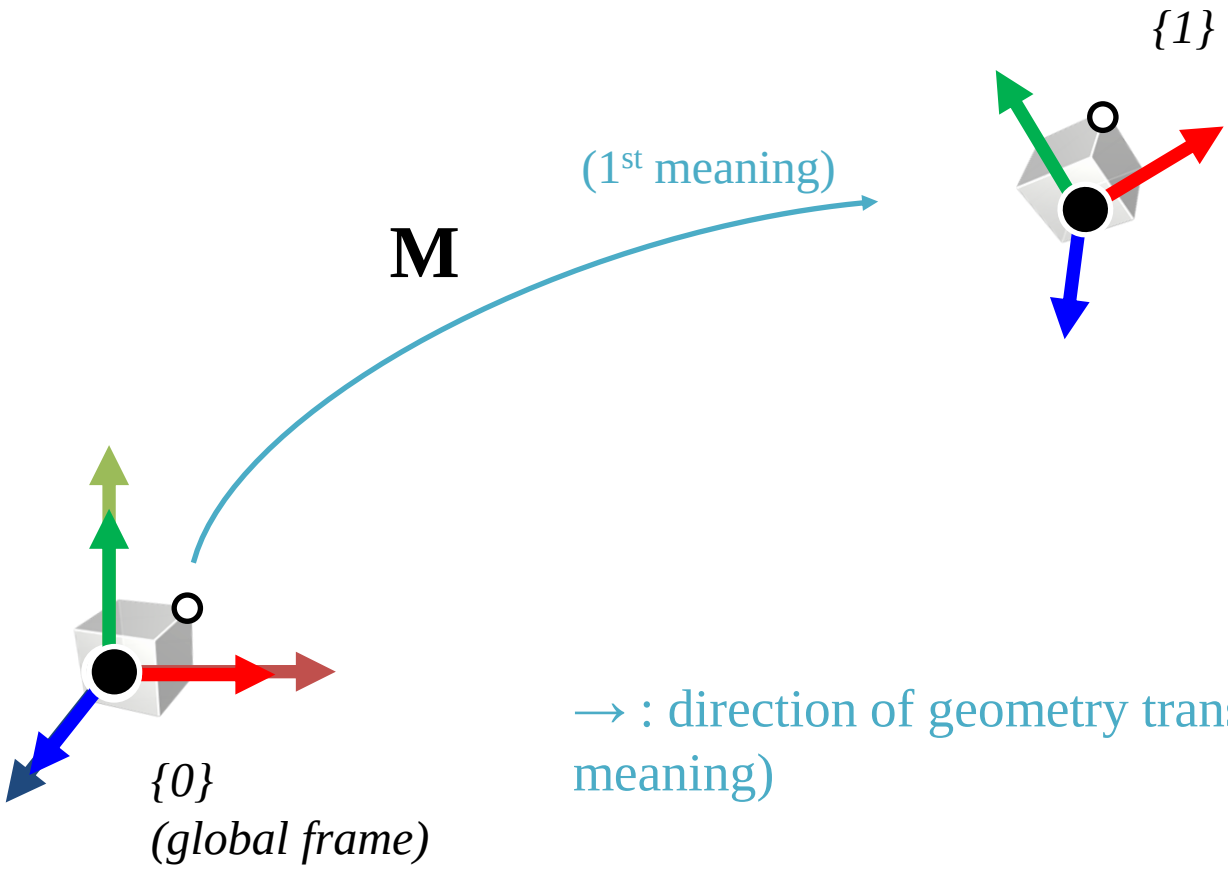


World space



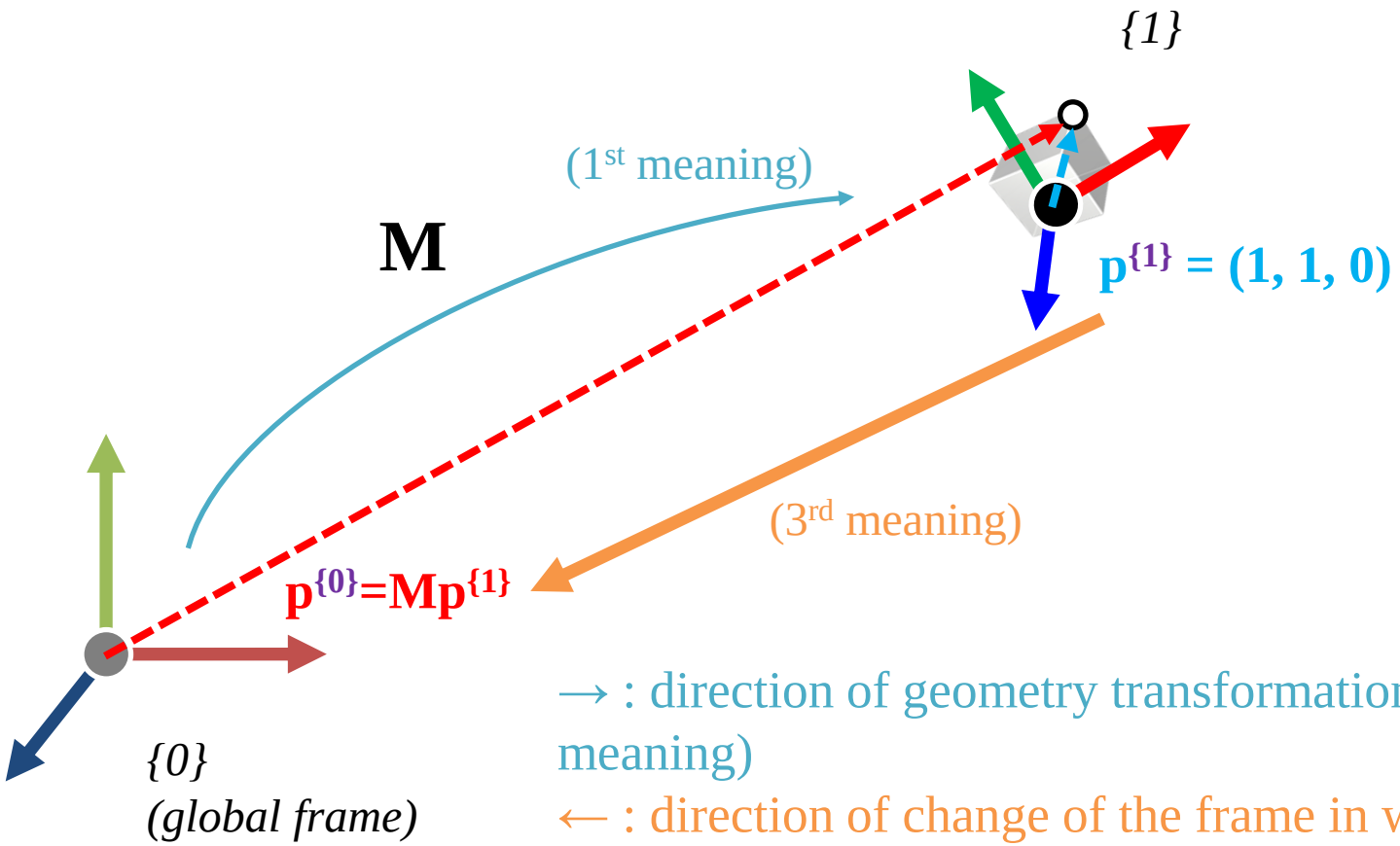
Canonical view volume

Directions of the "arrow"



→ : direction of geometry transformation (1st meaning)

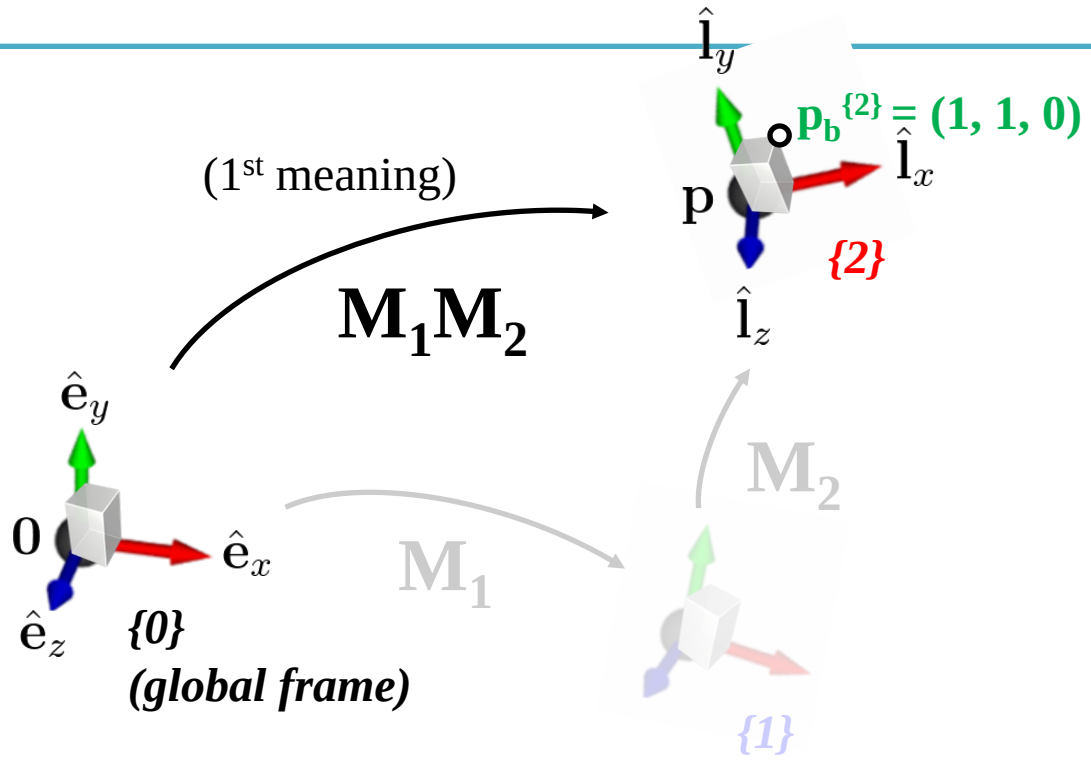
Directions of the "arrow"



→ : direction of geometry transformation (1st meaning)

← : direction of change of the frame in which the point is described (3rd meaning)

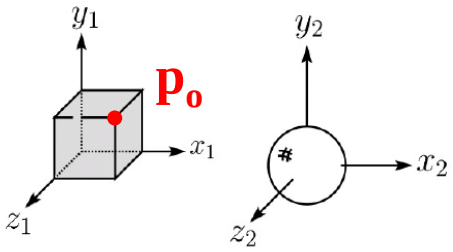
{0} to {2}



- 1) $\mathbf{M_1M_2}$ transforms a geometry (represented in ***{0}***) w.r.t. ***{0}***
- 2) $\mathbf{M_1M_2}$ defines an ***{2}*** w.r.t. ***{0}***
- 3) $\mathbf{M_1M_2}$ transforms a point represented in ***{2}*** to the same point but represented in ***{0}***
  - $\mathbf{p_b^{\{1\}}=M_2p_b^{\{2\}}}$, $\mathbf{p_b^{\{0\}}=M_1p_b^{\{1\}}=M_1M_2p_b^{\{2\}}}$

Modeling Transformation

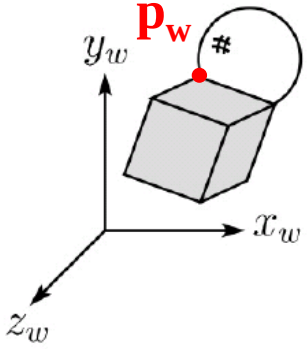
Object space



(3rd meaning)

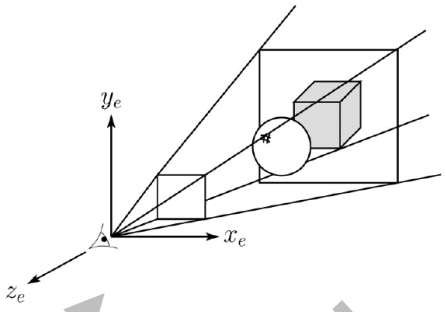
Modeling transformation : M_m

$p_w = M_m p_o$

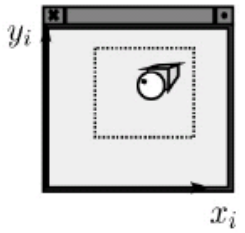


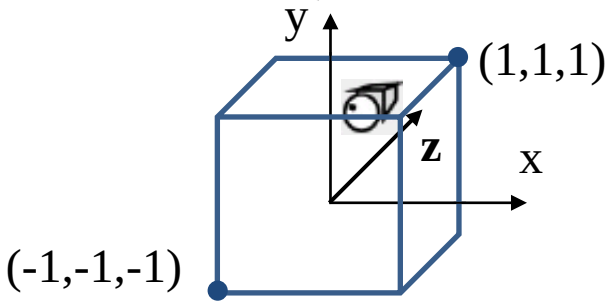
World space

View space (Camera space)



Screen space (Image space)

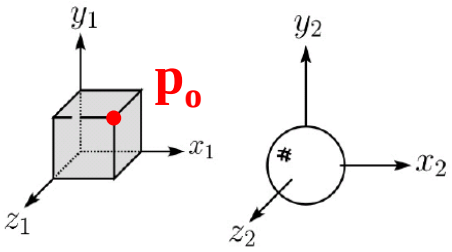




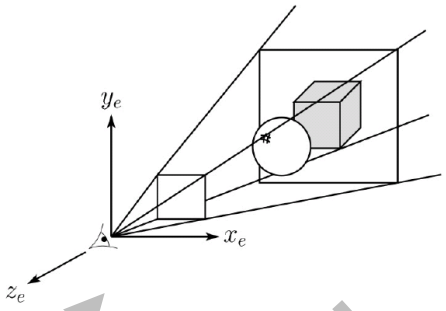
Canonical view volume

Vertex Processing (Transformation Pipeline)

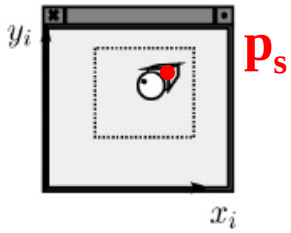
Object space



View space
(Camera space)



Screen space
(Image space)



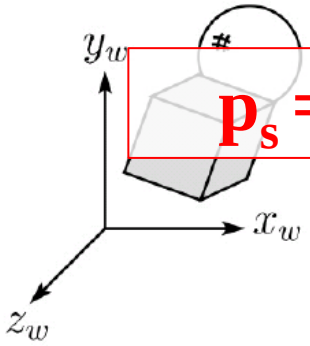
Modeling transformation : M_m

Viewing transformation : M_v

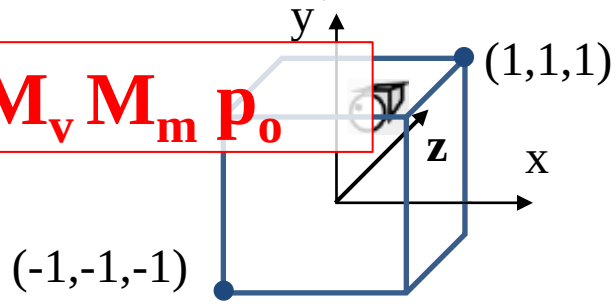
Projection transformation : M_{pj}

Viewport transformation : M_{vp}

$$p_s = M_{vp}\, M_{pj}\, M_v\, M_m\, p_o$$



World space

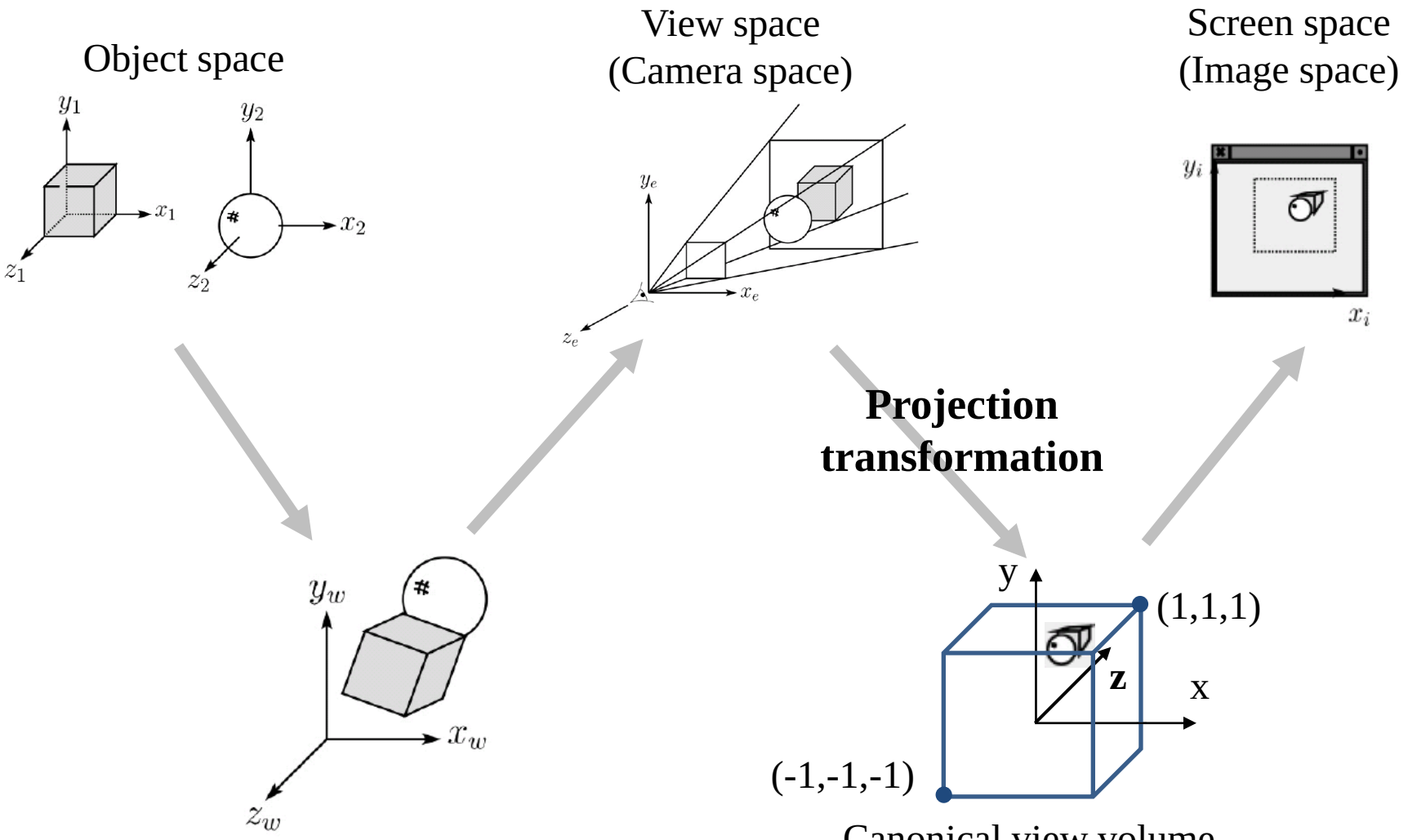


Canonical view volume

Topics Covered

- Projection Transformation
  - Orthographic (Orthogonal) Projection
  - Perspective Projection
- Viewport Transformation
- Mesh
  - Polygon mesh & triangle mesh
  - Representations for triangle meshes - Seperate triangle
  - OpenGL vertex array

Projection Transformation



Recall that...

- 1. Placing objects

→ **Modeling transformation**

- 2. Placing the “camera”

→ **Viewing transformation (covered in the last class)**

- 3. Selecting a “lens”

→ **Projection transformation**

- 4. Displaying on a “cinema screen”

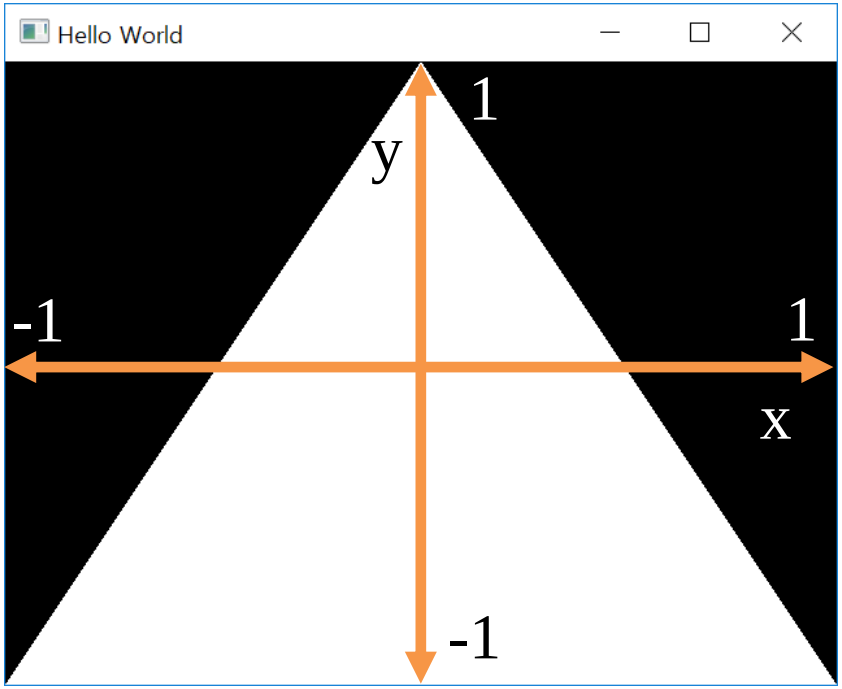
→ **Viewport transformation**

Review:Normalized Device Coordinates

- Remember that you could draw the triangle anywhere in a 2D square ranging from [-1, -1] to [1, 1].

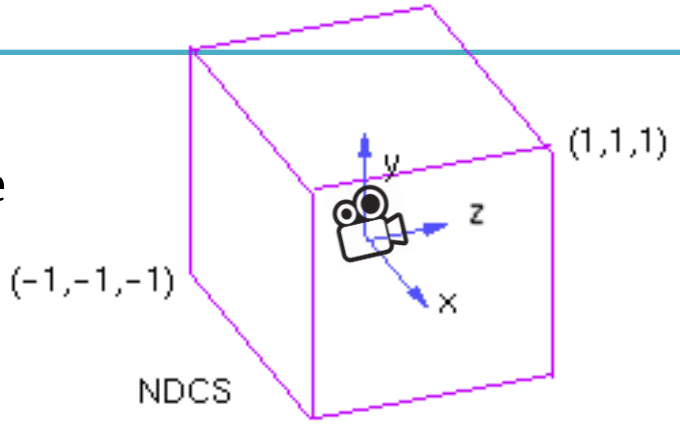
- This coordinate system is called **normalized device coordinates (NDC).**

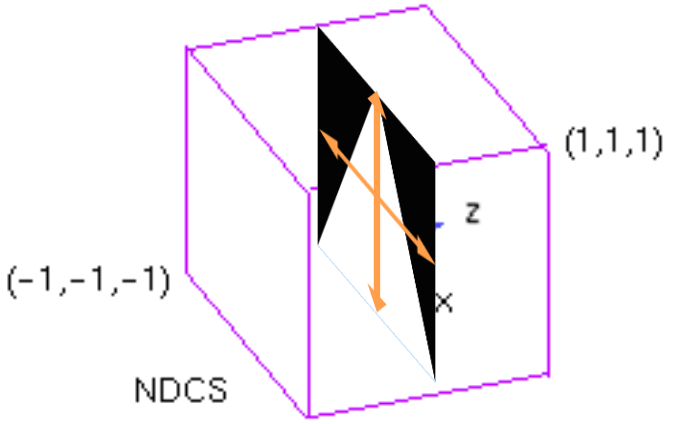
- And the space expressed with NDC is called **canonical view volume.**

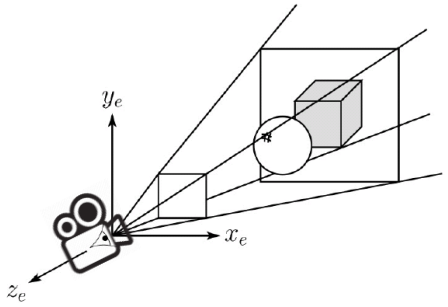


Canonical View “Volume”

- Actually, a canonical view volume is a **3D cube** ranging from [-1,-1,-1] to [1,1,1] in OpenGL.
  - Its coordinate system is NDC.
- Its **xy** plane is a 2D “viewport”.
- Note that NDC in OpenGL is a left-handed coordinate system.
  - Viewing direction in NDC : +z direction
- But OpenGL’s projection functions change the hand-ness – Thus view, world, model spaces use right-handed coordinate system.
  - Viewing direction in view space : -z direction

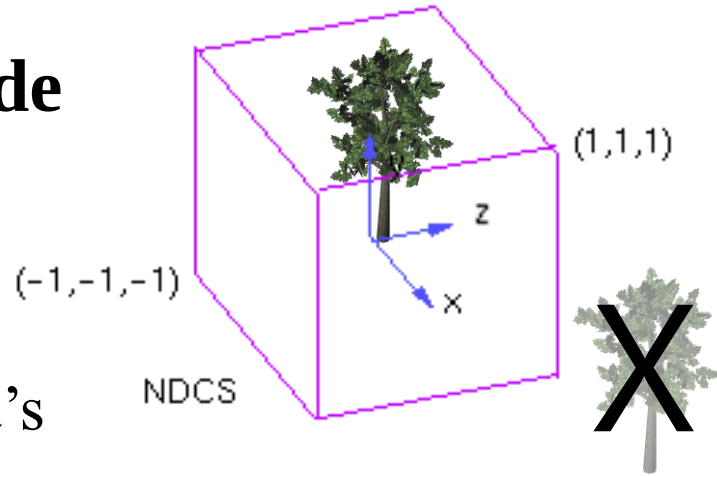






Canonical View Volume

- OpenGL only draws objects **inside** the canonical view volume
  - To draw objects only in the camera's view
  - Not to draw objects too near or too far from the camera



Do we always have to use the cube of size 2 as a view volume?

- No. You can set up a view volume of any size and draw objects in it.
  - Even you can use “frustums” as well as cuboids.

- Then everything in the visible volume is mapped (projected) into the canonical view volume.

- Then 3D points in the canonical view volume are projected onto its xy plane as 2D points.

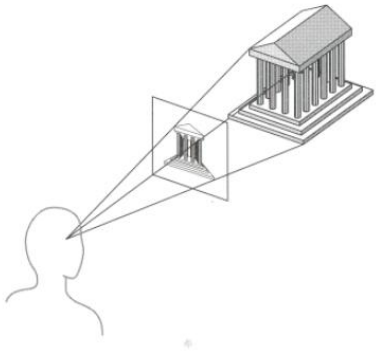
- → **Projection transformation**

Projection in General

- General definition:

- Mapping points in a n-dim space to a m-dim space (m<n).

Projection in Computer Graphics



- Mapping 3D coordinates to 2D screen coordinates.

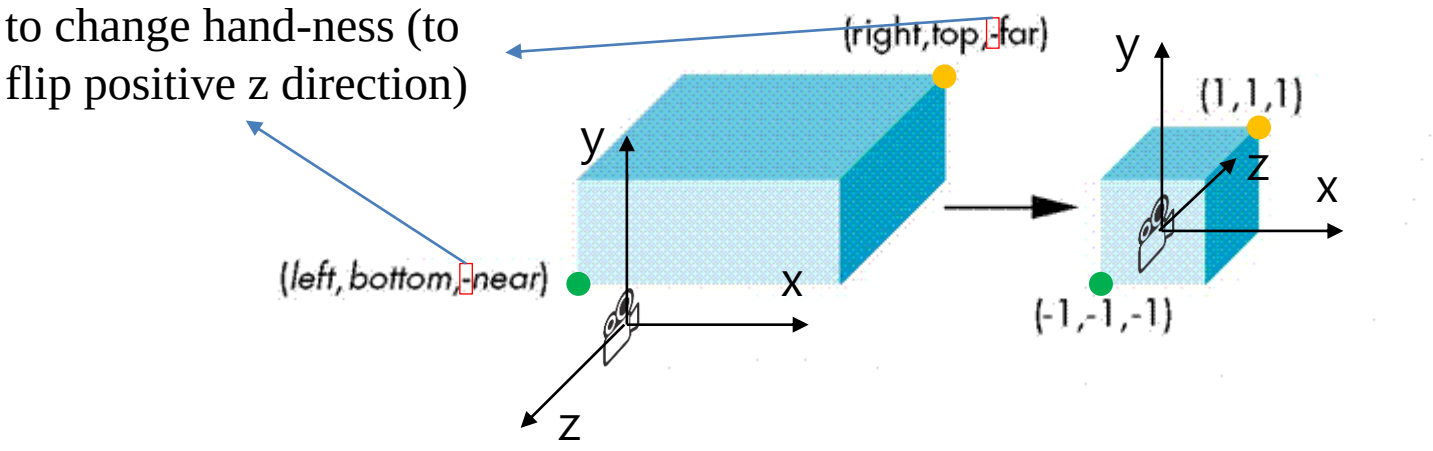
- Two stages:
  - Map an arbitrary view volume to a canonical view volume
  - ~~Map 3D points in the canonical view volume onto its xy plane : But we still need z values of points for *depth test*, so do not consider this second stage~~

- Two common projection methods:
  - Orthographic projection
  - Perspective projection

Orthographic (Orthogonal) Projection

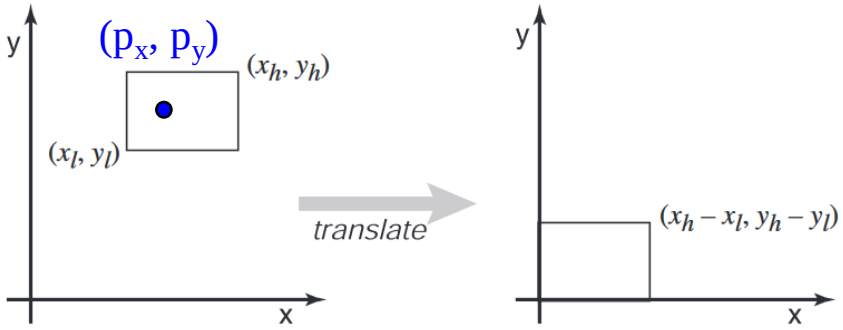
- View volume : Cuboid (직육면체)
- Orthographic projection : Mapping from a cuboid view volume to a canonical view volume
  - Combination of scaling & translation

  → "Windowing" transformation

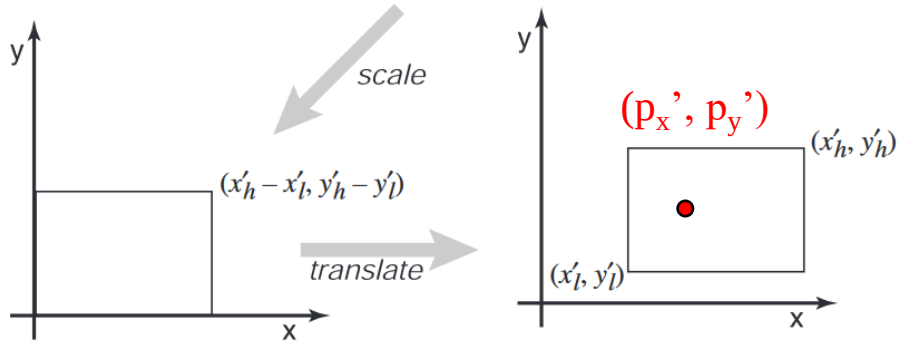


Windowing Transformation

- Transformation that maps a point (p_x, p_y) in a rectangular space from (x_l, y_l) to (x_h, y_h) to a point (p_x', p_y') in a rectangular space from (x_l', y_l') to (x_h', y_h')



$$\begin{pmatrix} p_x' \\ p_y' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & x_l' \\ 0 & 1 & y_l' \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{x_h' - x_l'}{x_h - x_l} & 0 & 0 \\ 0 & \frac{y_h' - y_l'}{y_h - y_l} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_l \\ 0 & 1 & -y_l \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$



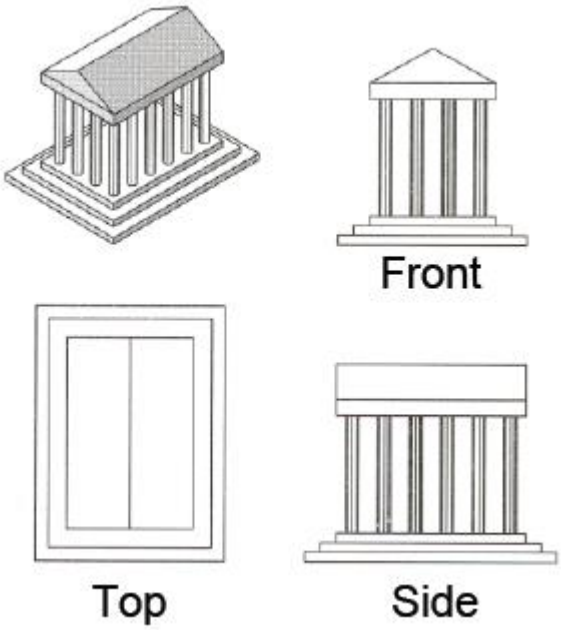
$$\begin{pmatrix} p_x' \\ p_y' \\ 1 \end{pmatrix} = \begin{bmatrix} \frac{x_h' - x_l'}{x_h - x_l} & 0 & \frac{x_l' x_h - x_h' x_l}{x_h - x_l} \\ 0 & \frac{y_h' - y_l'}{y_h - y_l} & \frac{y_l' y_h - y_h' y_l}{y_h - y_l} \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

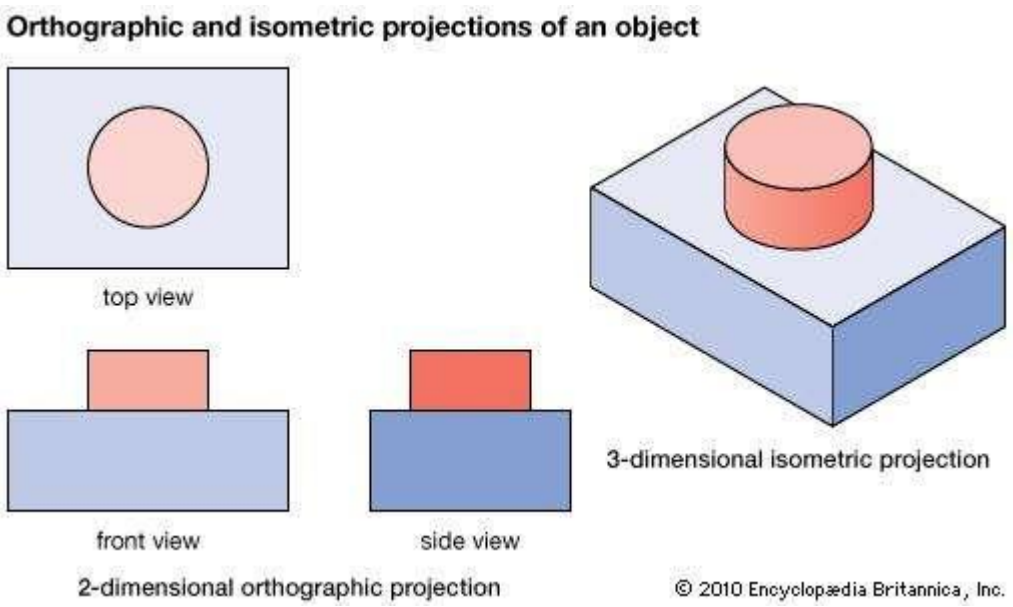
Orthographic Projection Matrix

- By extending the matrix to 3D and substituting
  - x_h=right, x_l=left, x_h'=1, x_l'=-1
  - y_h=top, y_l=bottom, y_h'=1, y_l'=-1
  - z_h=-far, z_l=-near, z_h'=1, z_l'=-1

$$M_{orth} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Examples of Orthographic Projection

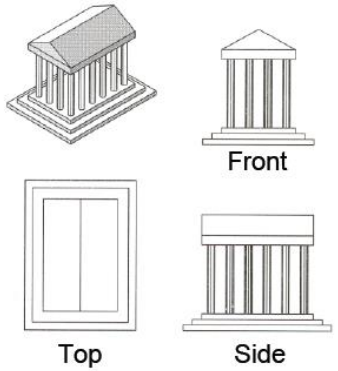




An object always stay the same size, no matter its distance from the viewer.

Properties of Orthographic Projection

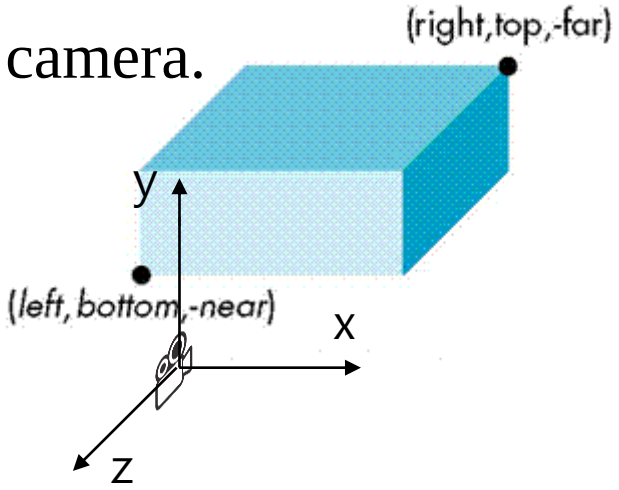
- Not realistic looking



- Good for exact measurement
- Most often used in CAD, architectural drawings, etc. where taking exact measurement is important.
- Affine transformation
  - parallel lines remain parallel
  - ratios are preserved
  - angles are not preserved

glOrtho()

- glOrtho(left, right, bottom, top, zNear, zFar)
- : Creates a orthographic projection matrix and right-multiplies the current transformation matrix by it
- Sign of zNear, zFar:
  - positive value: the plane is in front of the camera
  - negative value: the plane is behind the camera.
- $C \leftarrow CM_{orth}$



[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)
    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()

    # 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)

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()
```

Quiz #1

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

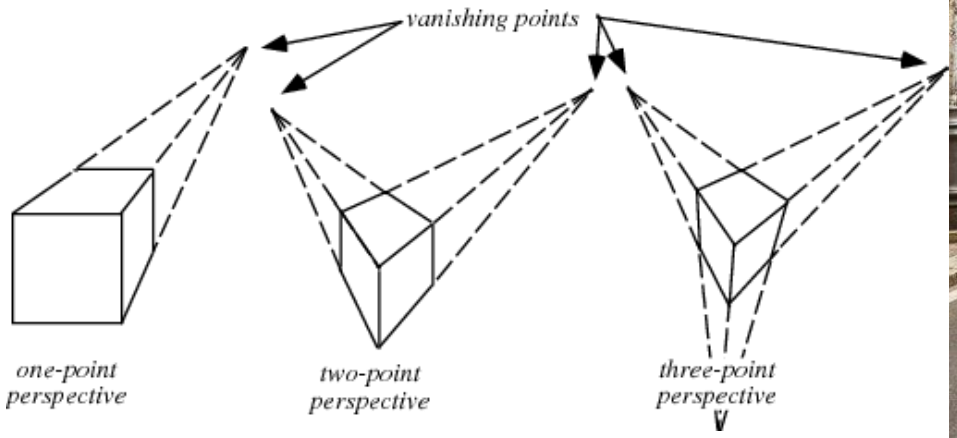
- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

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

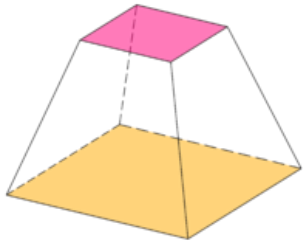
Perspective Effects

- Distant objects become small.

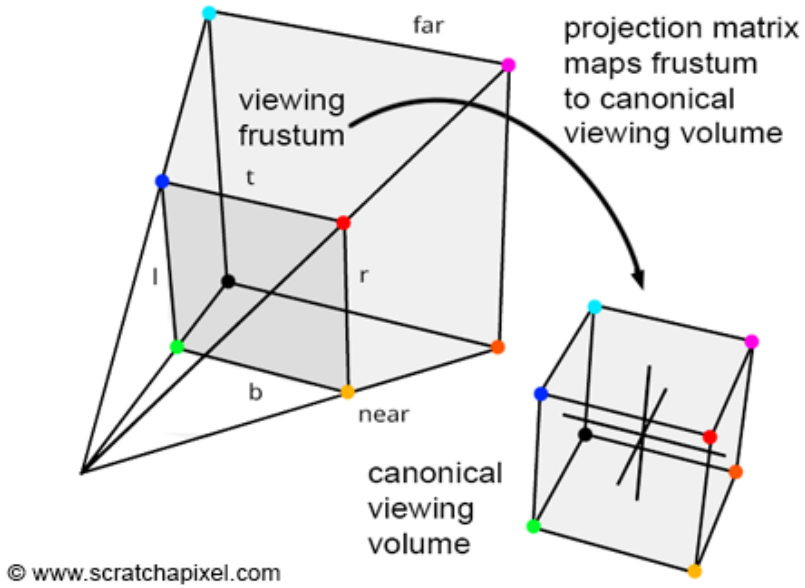
Vanishing point: The point or points to which the extensions of parallel lines appear to converge in a perspective drawing



Perspective Projection



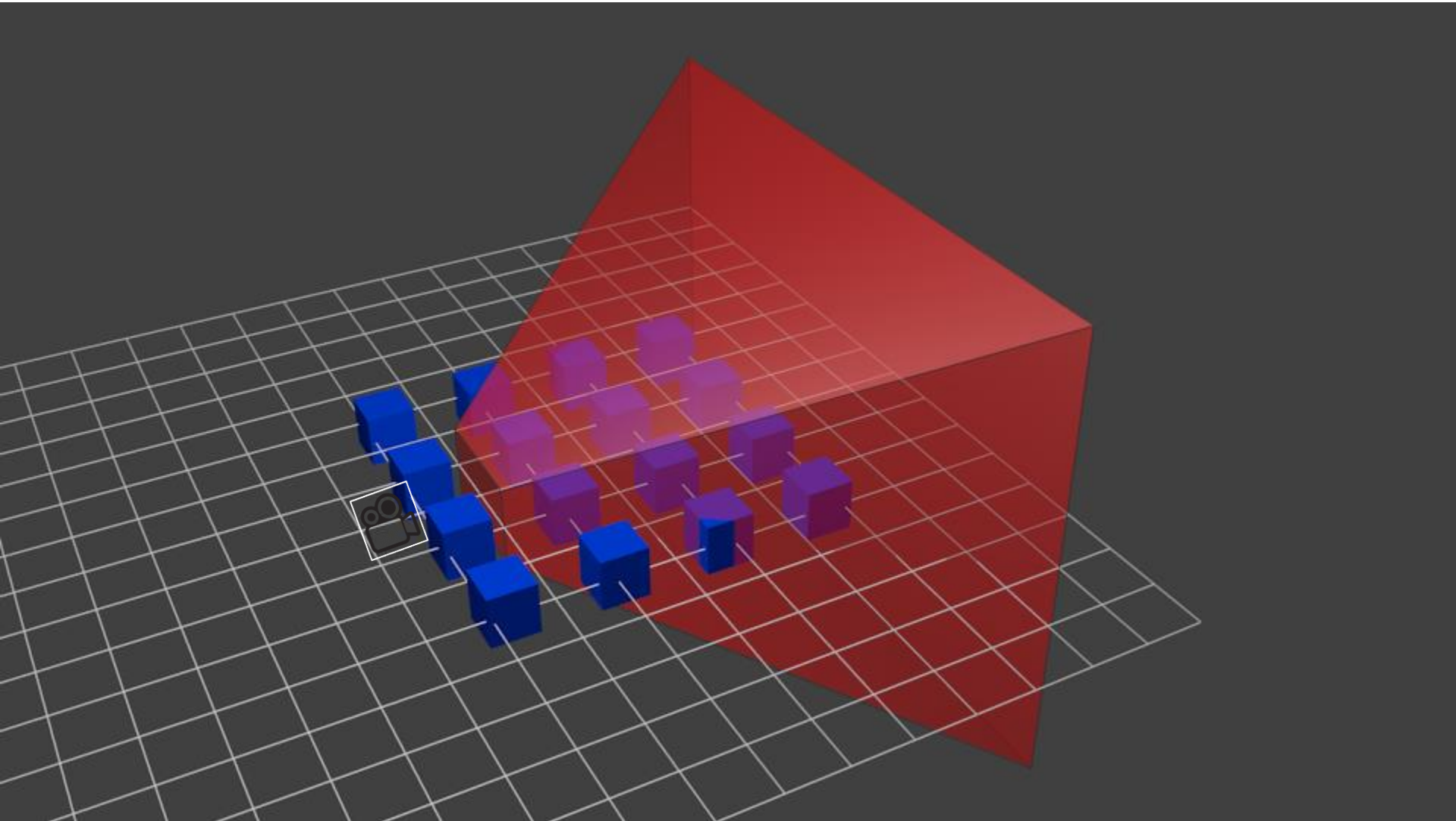
- View volume : Frustum (절두체)
- → “Viewing frustum”
- Perspective projection : Mapping from a viewing frustum to a canonical view volume



Why does this mapping generate a perspective effect?

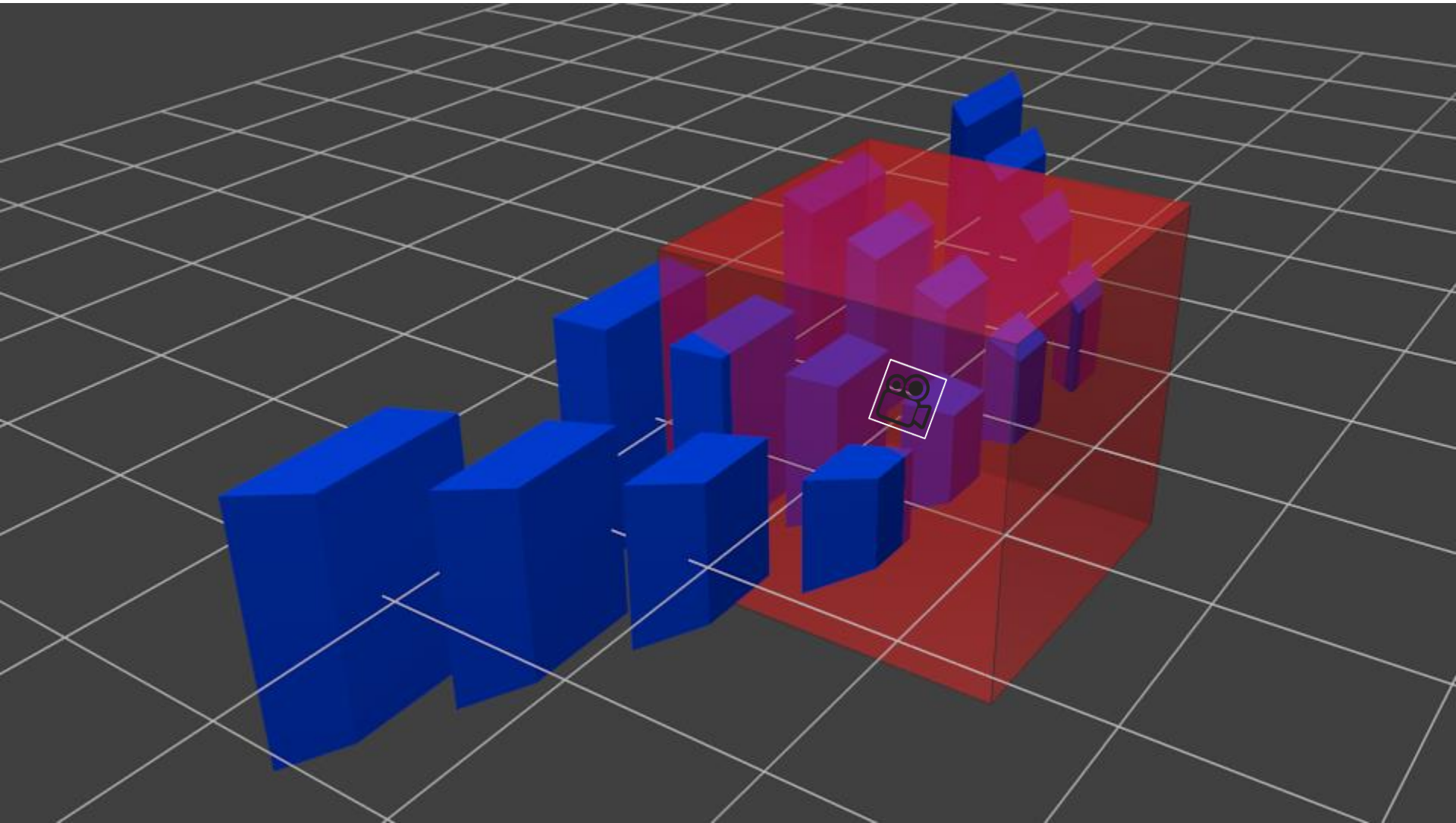
Original 3D scene

Red: viewing frustum, Blue: objects

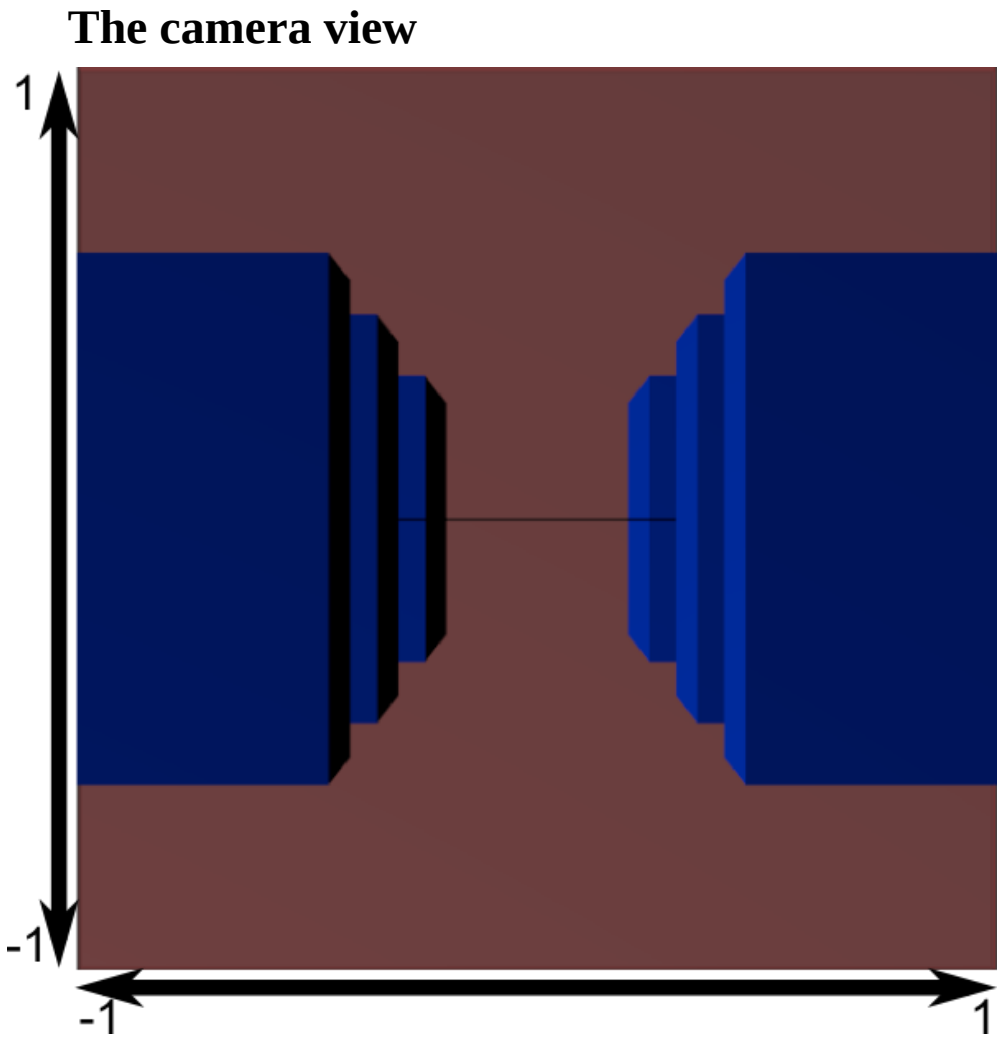


An Example of Perspective Projection

After perspective projection

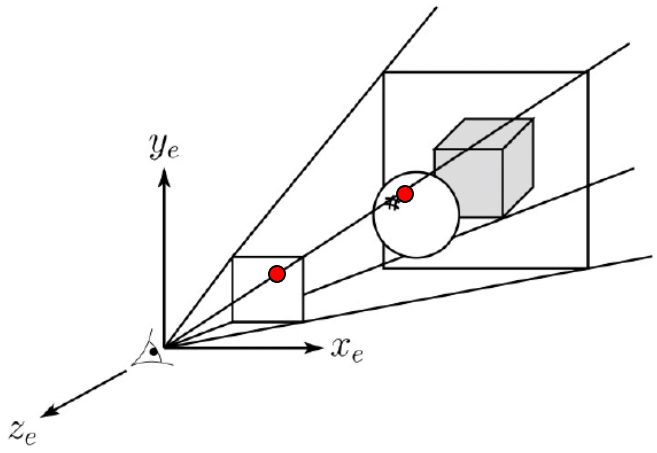


An Example of Perspective Projection



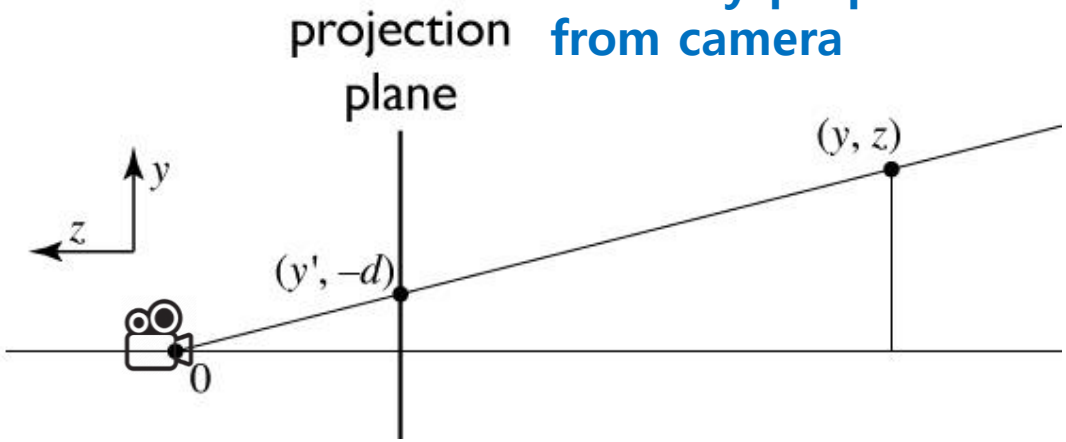
Let's first consider
3D View Frustum→2D Projection Plane

- Consider the projection of a 3D point on the camera plane



Perspective projection

The size of an object on the screen is inversely proportional to its distance from camera



similar triangles:

$$\frac{y'}{d} = \frac{y}{-z}$$

$$y' = -dy/z$$

Homogeneous coordinates revisited

- Perspective requires division
  - that is **not** part of affine transformations
  - in affine, parallel lines stay parallel
    - therefore not vanishing point
    - therefore no rays converging on viewpoint
- "True" purpose of homogeneous coords: projection

Homogeneous coordinates revisited

- Introduced $w = 1$ coordinate as a placeholder

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

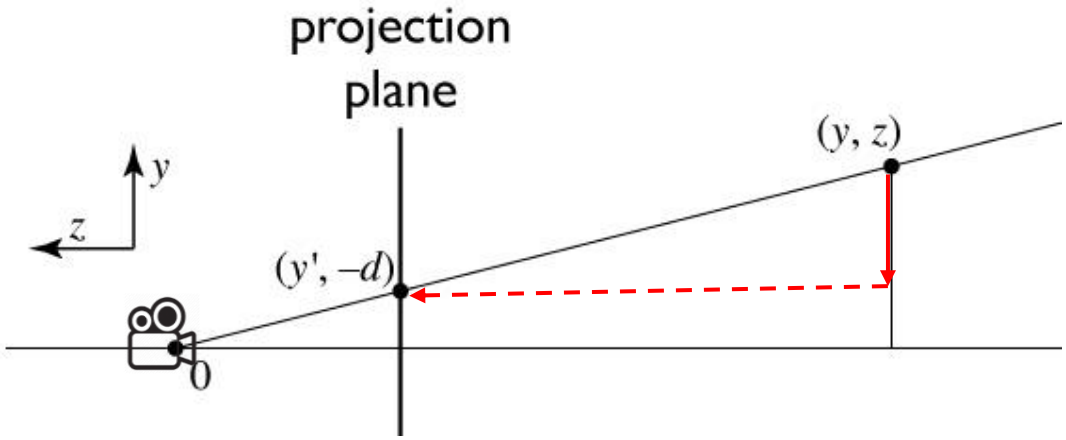
  - used as a convenience for unifying translation with linear transformation

- Can also allow arbitrary w

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \sim \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}$$

All scalar multiples of a 4-vector are equivalent

Perspective projection

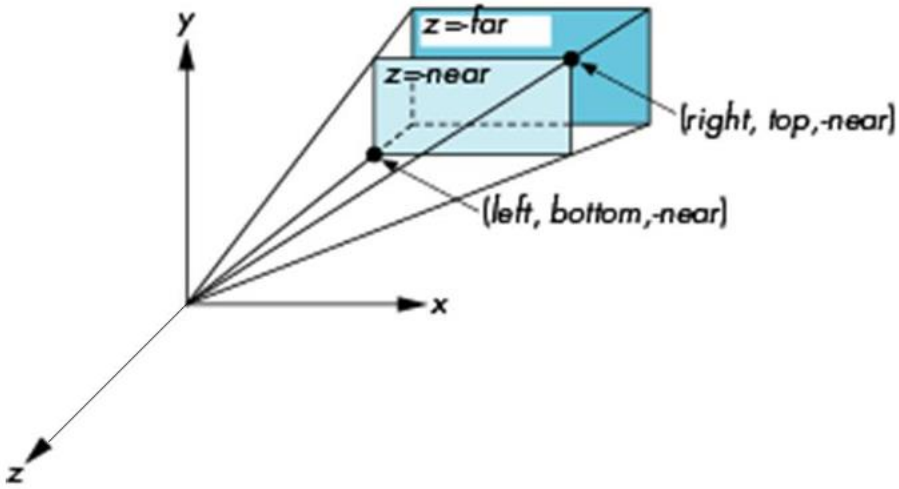


to implement perspective, just move z to w:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -dx/z \\ -dy/z \\ 1 \end{bmatrix} \sim \begin{bmatrix} dx \\ dy \\ -z \end{bmatrix} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Perspective Projection Matrix

- This 3D → 2D projection example gives the basic idea of perspective projection.
- What we really have to do is 3D → 3D, View Frustum → Canonical View Volume.
- For details for this process, see *6 - reference-projection.pdf*
- $M_{pers} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$

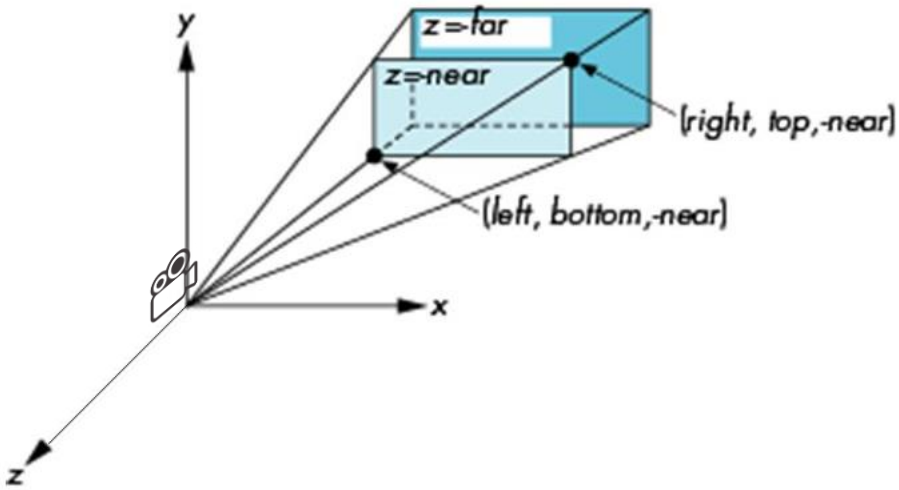


glFrustum()

- glFrustum(left, right, bottom, top, near, far)
  - Note that left, right, bottom, top are those of "near" plane.
- : Creates a perspective projection matrix and right-multiplies the current transformation matrix by it

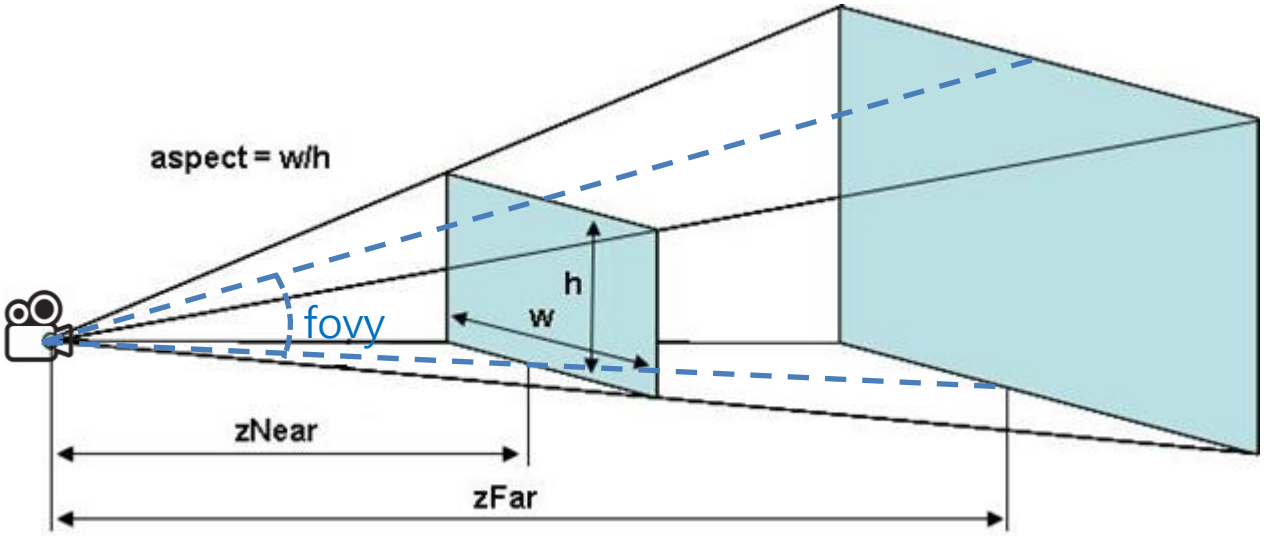
- Sign of near, far:
  - The values for both parameters **must be positive.**

- $C \leftarrow CM_{pers}$



gluPerspective()

- gluPerspective(fovy, aspect, zNear, zFar)
  - fovy: The field of view angle, in degrees, in the y-direction.
  - aspect: The aspect ratio that determines the field of view in the x-direction. The aspect ratio is the ratio of x (width) to y (height).
- : Creates a perspective projection matrix and right-multiplies the current transformation matrix by it
- $C \leftarrow CM_{pers}$



[Practice] glFrustum(), gluPerspective()

```
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)
    glPolygonMode( GL_FRONT_AND_BACK, GL_LINE )

    glLoadIdentity()

    # test other parameter values
    glFrustum(-1,1, -1,1, .1,10)
    # glFrustum(-1,1, -1,1, 1,10)

    # test other parameter values
    # gluPerspective(45, 1, 1,10)

    # test with this line
gluLookAt(5*np.sin(gCamAng),gCamHeight,5*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,'glFrustum()',
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()
```

Quiz #2

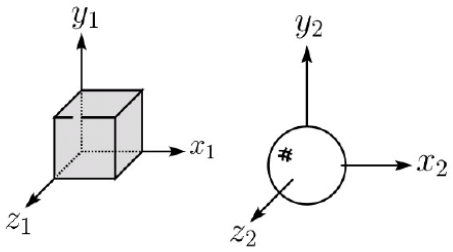
- Go to https://www.slido.com/
- Join #**cg-ys**
- Click “Polls”

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

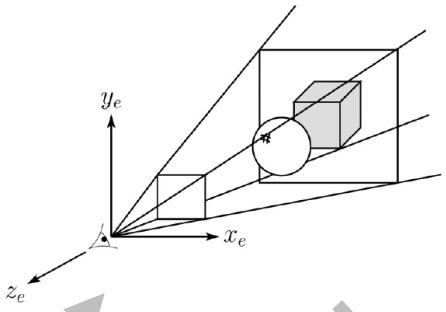
- Note that you must submit all quiz answers in the above format to be checked for “attendance”.

Viewport Transformation

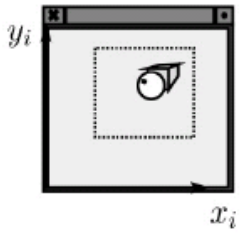
Object space



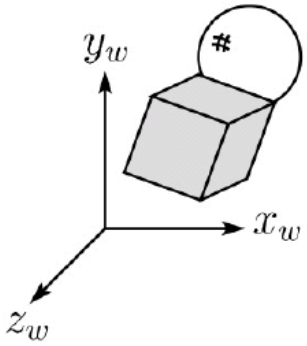
View space
(Camera space)



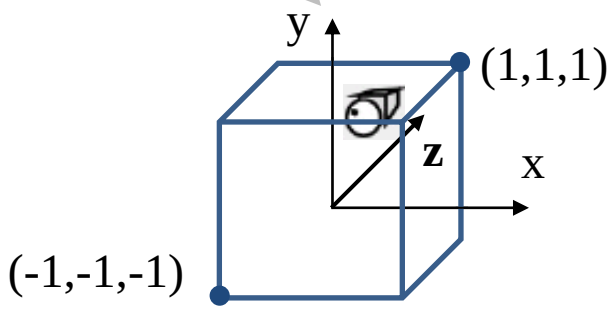
Screen space
(Image space)



Viewport transformation



World space



Canonical view volume
(Normalized device coordinates, NDC)

Recall that...

- 1. Placing objects

→ **Modeling transformation**

- 2. Placing the “camera”

→ **Viewing transformation**

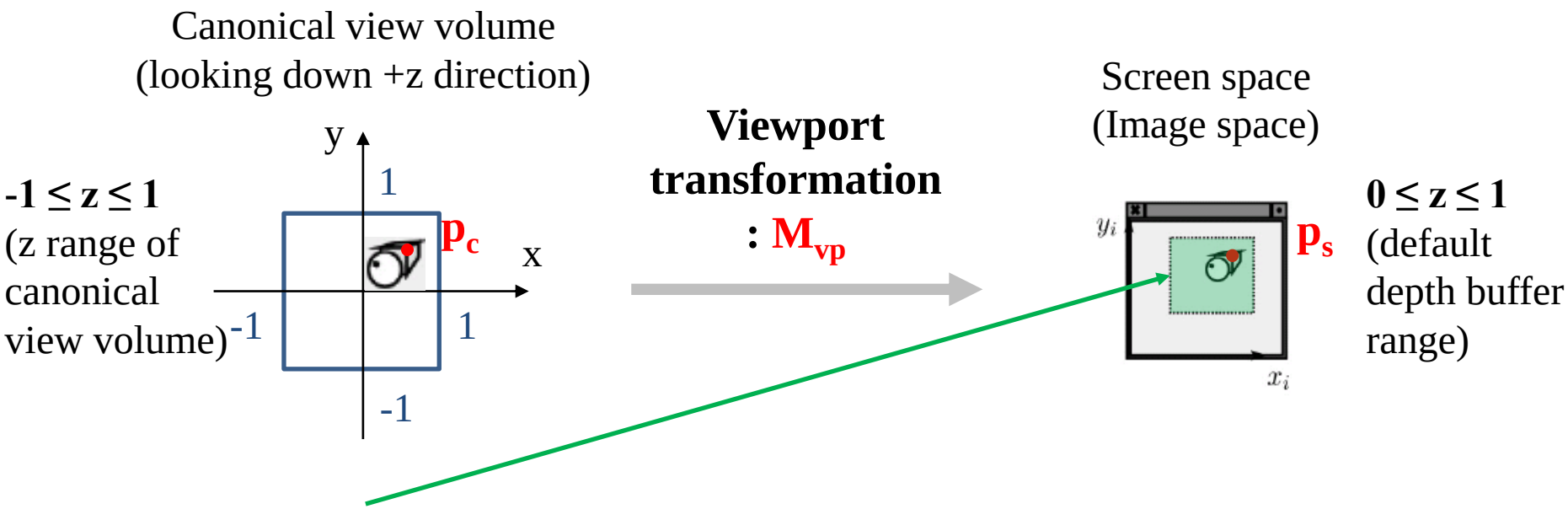
- 3. Selecting a “lens”

→ **Projection transformation**

- 4. Displaying on a “cinema screen”

→ **Viewport transformation**

Viewport Transformation



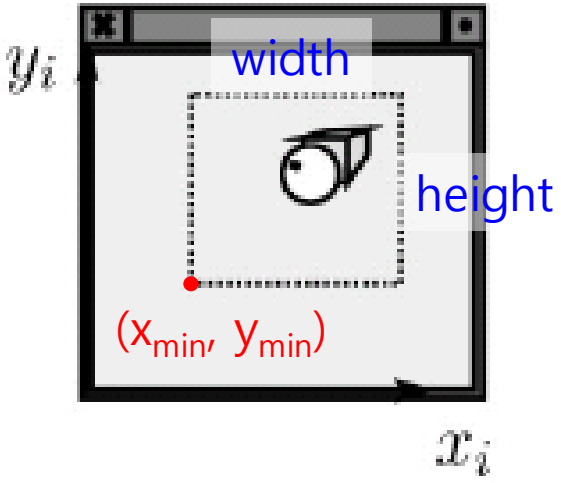
- Viewport: a rectangular viewing region of screen

- So, viewport transformation is also a kind of windowing transformation.

Viewport Transformation Matrix

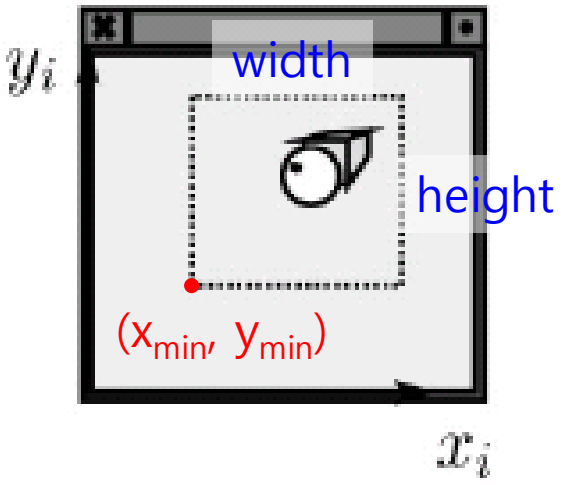
- In the windowing transformation matrix,
- By substituting x_h, x_l, x_h', ... with corresponding variables in viewport transformation,

$$M_{vp} = \begin{bmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} + x_{min} \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} + y_{min} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



glViewport()

- glViewport(xmin, ymin, width, height)
  - xmin, ymin, width, height: specified **in pixels**



- : Sets the viewport
  - This function does NOT explicitly multiply a viewport matrix with the current matrix.
  - Viewport transformation is internally done in OpenGL, so you can apply transformation matrices **starting from a canonical view volume**, not a screen space.

- Default viewport setting for (xmin, ymin, width, height) is **(0, 0, window width, window height).**
  - If you do not call glViewport(), OpenGL uses this default viewport setting.

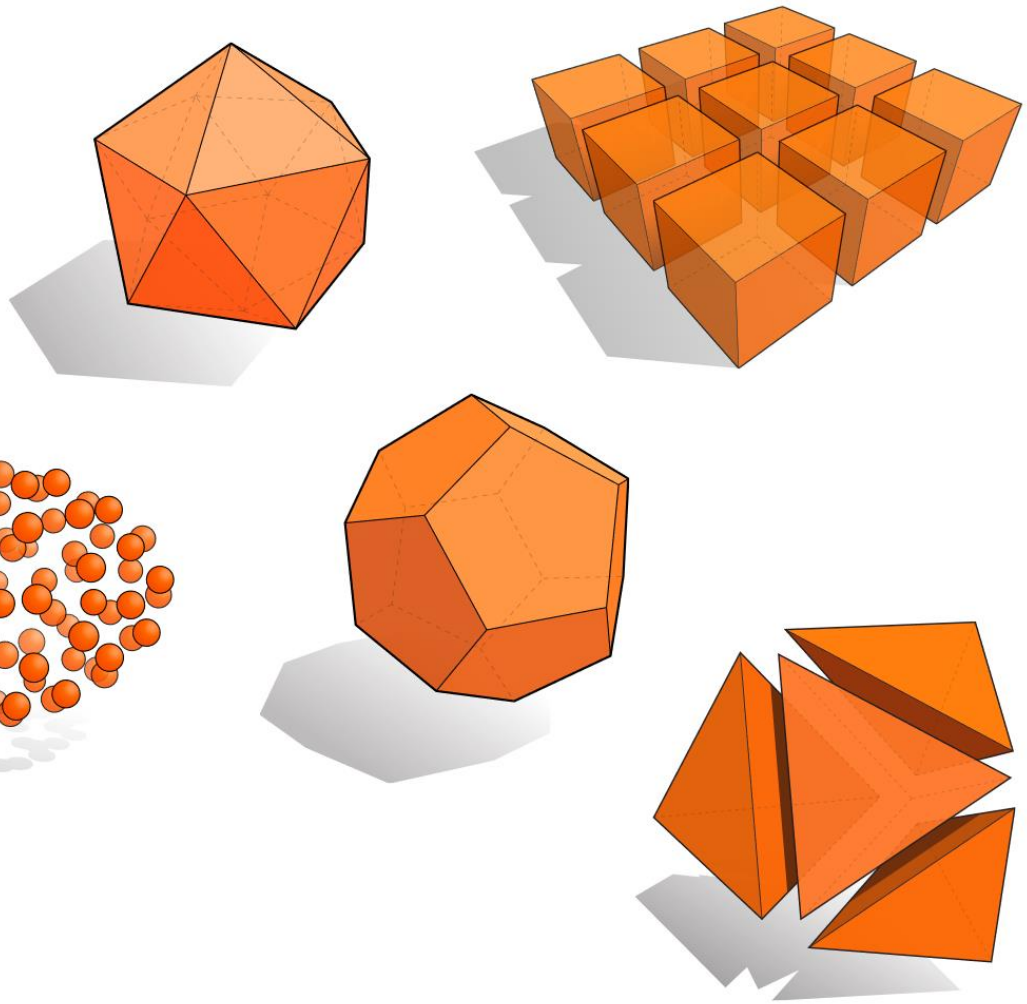
[Practice] glViewport()

```
def main():
    # ...
    glfw.make_context_current(window)
    glViewport(100,100,200,200)
    # ...
```

Mesh

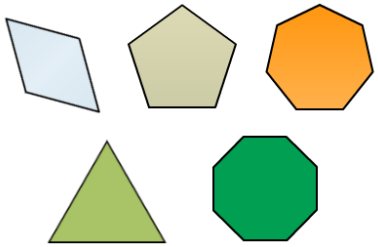
Many ways to digitally encode geometry

- **EXPLICIT**
  - point cloud
  - polygon mesh
  - subdivision, NURBS
  - L-systems
  - ...
- **IMPLICIT**
  - level set
  - algebraic surface
  - ...
- **Each choice best suited to a different task/type of geometry**



The Most Popular Representation : Polygon Mesh

- Because this can model any arbitrary complex shapes with relatively simple representations and can be rendered fast.

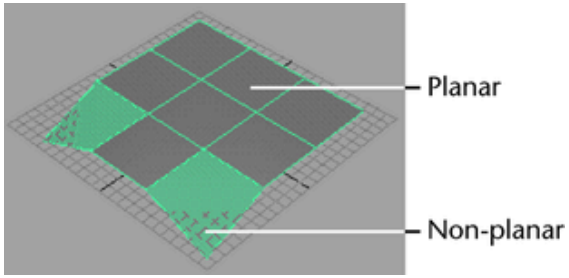


- **Polygon**: a “closed” shape with straight sides

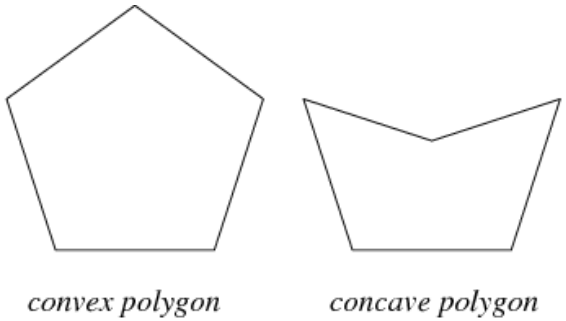
- **Polygon mesh**: a bunch of polygons in 3D space that are connected together to form a surface
  - Usually use *triangles* or *quads* (4 side polygon)

Triangle Mesh

- A general N-polygon can be
  - Non-planar
  - Non-convex
- , which are not desirable for fast rendering.



- A triangle does not have such problems. It's always planar & convex.
- and N-polygons can be composed of multiple triangles.

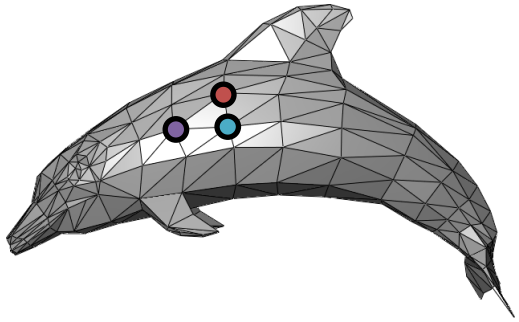


- That's why modern GPUs draw everything as a set of triangles.

- So, we'll focus on triangle meshes.

Representation for Triangle Mesh

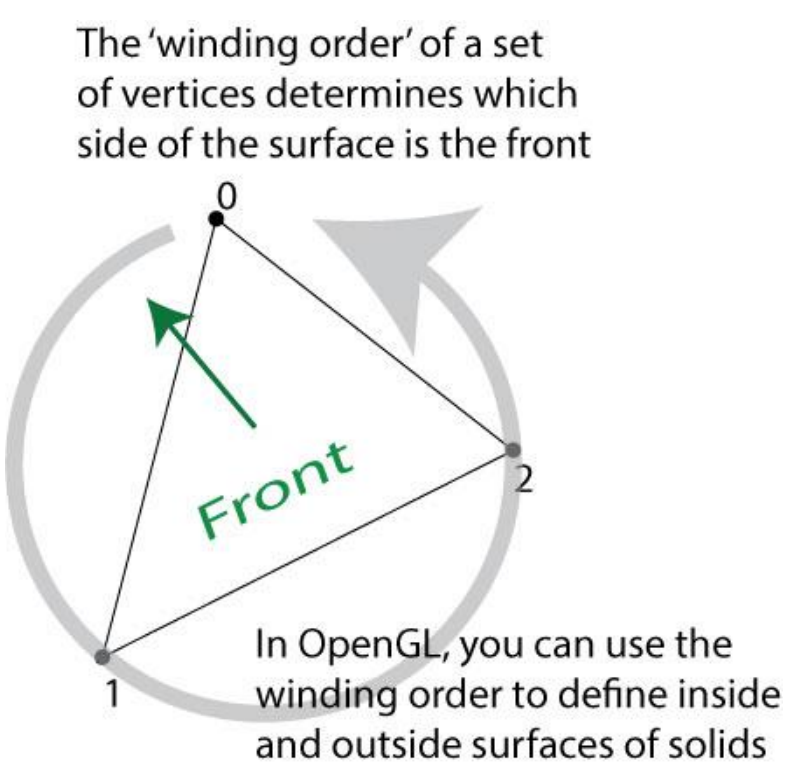
- It’s about how to store
  - vertex positions
  - relationship between vertices (to make triangles)
- on memory.

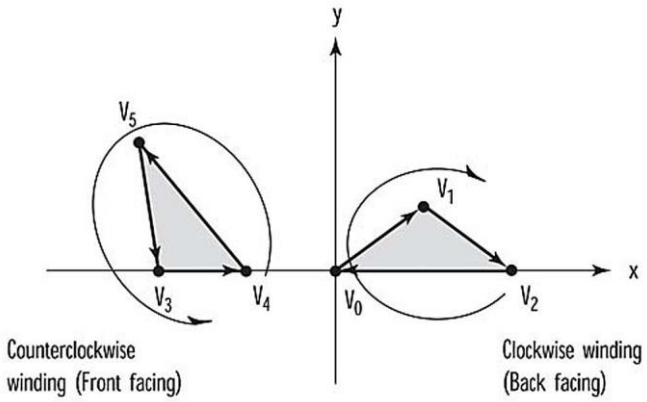


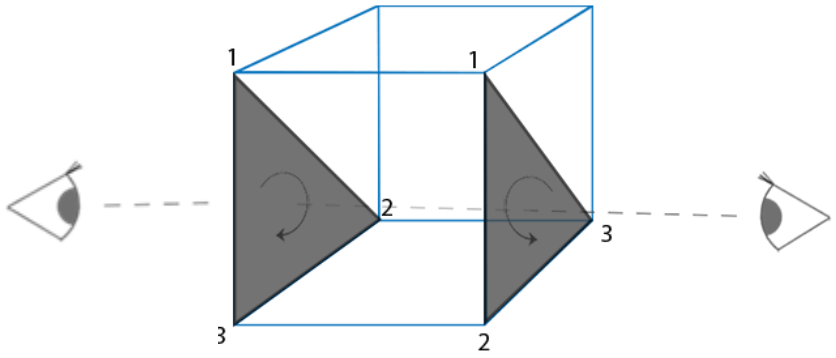
- We'll see
  - Separate triangles (today)
  - Indexed triangle set (next lecture)

Vertex Winding Order

- In OpenGL, by default, polygons whose vertices appear in **counterclockwise** order on the screen is front-facing

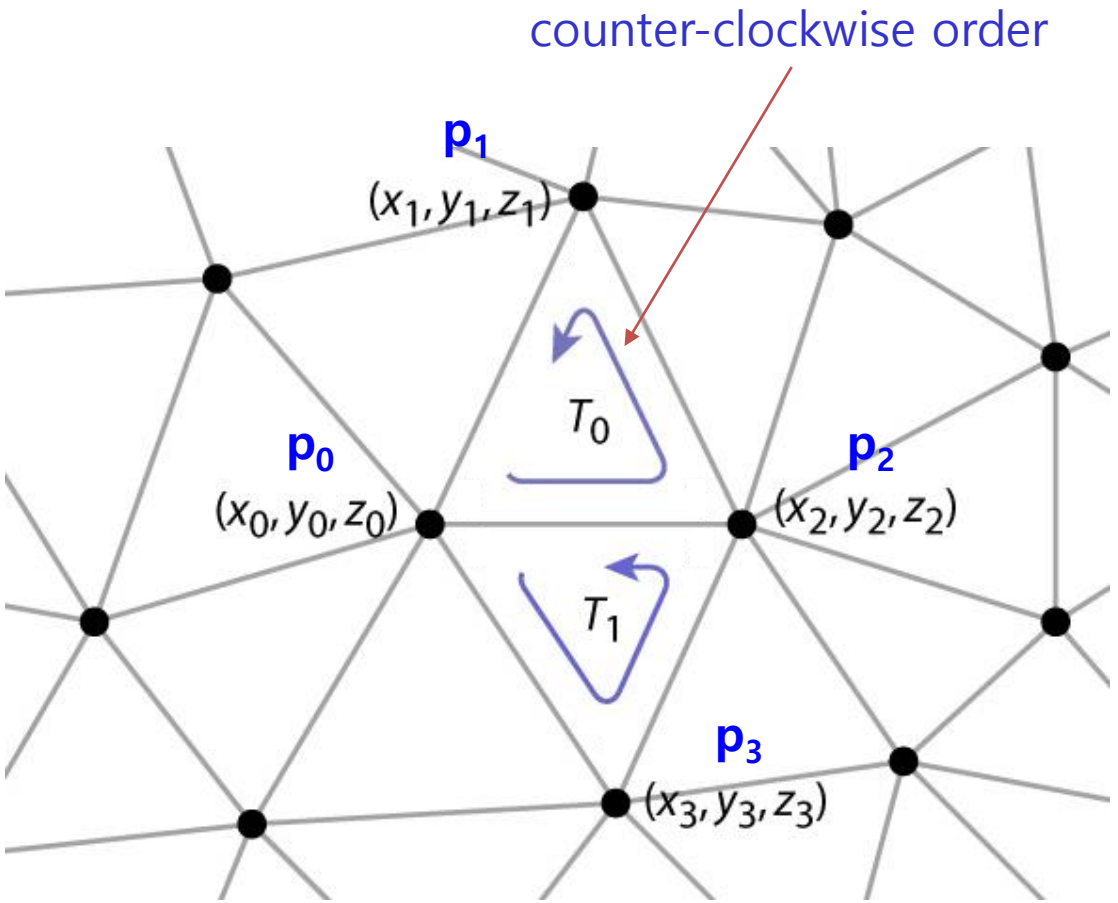






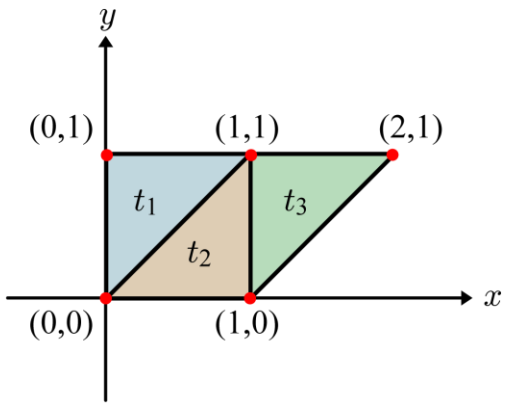
Separate triangles

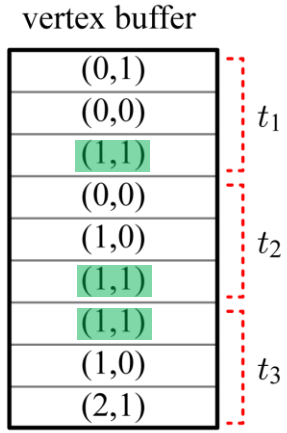
	[0]	[1]	[2]
tris[0]	x_0, y_0, z_0	x_2, y_2, z_2	x_1, y_1, z_1
tris[1]	x_0, y_0, z_0	x_3, y_3, z_3	x_2, y_2, z_2
	⋮	⋮	⋮

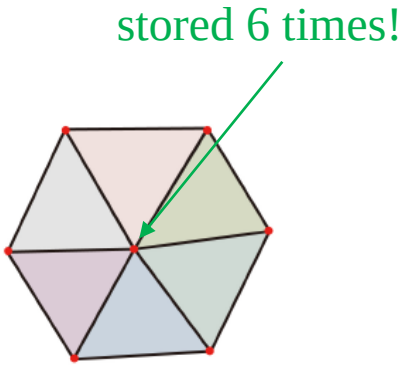


Separate Triangles

- Various problems
  - Wastes space
  - Cracks due to roundoff
  - Difficulty of finding neighbors
    - If you want find "neighbor" triangles of t2, you have to find all "zero-distance" vertices from t2's each vertex.

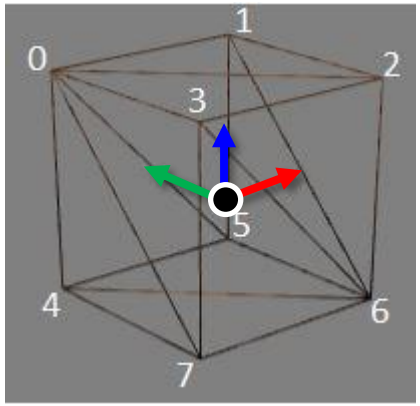






(1,1) is stored 3 times!

Example: a cube of length 2

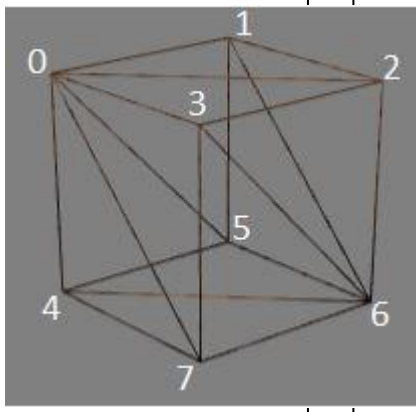


vertex index	position
0	(-1 , 1 , 1)
1	(1 , 1 , 1)
2	(1 , -1 , 1)
3	(-1 , -1 , 1)
4	(-1 , 1 , -1)
5	(1 , 1 , -1)
6	(1 , -1 , -1)
7	(-1 , -1 , -1)

Drawing Separate Triangles using glVertex*()

- You can use glVertex*() like this:

```
def drawCube_glVertex():
    glBegin(GL_TRIANGLES)
    glVertex3f( -1 ,  1 ,  1 ) # v0
    glVertex3f(  1 , -1 ,  1 ) # v2
    glVertex3f(  1 ,  1 ,  1 ) # v1

    glVertex3f( -1 ,  1 ,  1 ) # v0
    glVertex3f( -1 , -1 ,  1 ) # v3
    glVertex3f(  1 , -1 ,  1 ) # v2

    glVertex3f( -1 ,  1 , -1 ) # v4
    glVertex3f(  1 ,  1 , -1 ) # v5
    glVertex3f(  1 , -1 , -1 ) # v6

    glVertex3f( -1 ,  1 , -1 ) # v4
    glVertex3f(  1 , -1 , -1 ) # v6
    glVertex3f( -1 , -1 , -1 ) # v7

    glVertex3f( -1 ,  1 ,  1 ) # v0
    glVertex3f(  1 ,  1 ,  1 ) # v1
    glVertex3f(  1 ,  1 , -1 ) # v5

    glVertex3f( -1 ,  1 ,  1 ) # v0
    glVertex3f(  1 ,  1 , -1 ) # v5
    glVertex3f( -1 ,  1 , -1 ) # v4
```




```
    glVertex3f( -1 , -1 ,  1 ) # v3
    glVertex3f(  1 , -1 , -1 ) # v6
    glVertex3f(  1 , -1 ,  1 ) # v2

    glVertex3f( -1 , -1 ,  1 ) # v3
    glVertex3f( -1 , -1 , -1 ) # v7
    glVertex3f(  1 , -1 , -1 ) # v6

    glVertex3f(  1 ,  1 ,  1 ) # v1
    glVertex3f(  1 , -1 ,  1 ) # v2
    glVertex3f(  1 , -1 , -1 ) # v6

    glVertex3f(  1 ,  1 ,  1 ) # v1
    glVertex3f(  1 , -1 , -1 ) # v6
    glVertex3f(  1 ,  1 , -1 ) # v5

    glVertex3f( -1 ,  1 ,  1 ) # v0
    glVertex3f( -1 , -1 , -1 ) # v7
    glVertex3f( -1 , -1 ,  1 ) # v3

    glVertex3f( -1 ,  1 ,  1 ) # v0
    glVertex3f( -1 ,  1 , -1 ) # v4
    glVertex3f( -1 , -1 , -1 ) # v7
    glEnd()
```

Vertex Array

- But from now on, let's use a more advanced method to draw polygons: *Vertex array*

- **Vertex array**: an array of vertex data including vertex positions, normals, texture coordinates and color information
  - For now, consider vertex positions only

- By using a vertex array, you can draw a whole mesh just by calling a OpenGL function **once**! (instead of a huge number of glVertex*() calls!)
- → Tremendous increase in rendering performance!

Drawing Separate Triangles using Vertex Array

- 1. Create a vertex array for your mesh
  - Using numpy.ndarray or python list
- 2. Specify “pointer” to this vertex array
  - Using glVertexPointer()
- 3. Render the mesh using the specified “pointer”
  - Using glDrawArrays()

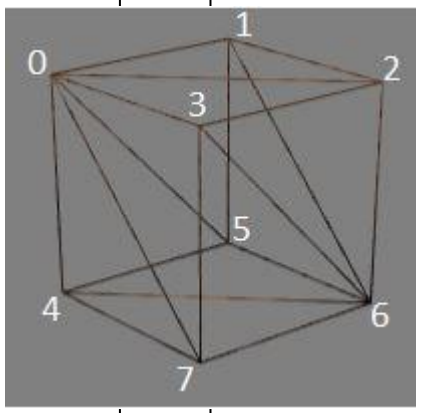
glVertexPointer() & glDrawArrays()

- **glVertexPointer(size, type, stride, pointer)**
- : specifies the location and data format of a vertex array
  - **size**: The number of vertex coordinates, 2 for 2D points, 3 for 3D points
  - **type**: The data type of each coordinate value in the array. GL_FLOAT, GL_SHORT, GL_INT or GL_DOUBLE.
  - **stride**: The byte offset to the next vertex
  - **pointer**: The pointer to the first coordinate of the first vertex in the array

- **glDrawArrays(mode , first , count)**
- : render primitives from the vertex array specified by glVertexPointer()
  - **mode**: The primitive type to render. GL_POINTS, GL_TRIANGLES, ...
  - **first**: The starting index in the array specified by glVertexPointer()
  - **count**: The number of vertices to be rendered (duplicate vertices also should be counted separately)

[Practice] Drawing Separate Triangles using Vertex Array

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

gCamAng = 0
gCamHeight = 1.

def createVertexArraySeparate():
    varr = np.array([
            ( -1 ,  1 ,  1 ), # v0
            (  1 , -1 ,  1 ), # v2
            (  1 ,  1 ,  1 ), # v1

            ( -1 ,  1 ,  1 ), # v0
            ( -1 , -1 ,  1 ), # v3
            (  1 , -1 ,  1 ), # v2

            ( -1 ,  1 , -1 ), # v4
            (  1 ,  1 , -1 ), # v5
            (  1 , -1 , -1 ), # v6

            ( -1 ,  1 , -1 ), # v4
            (  1 , -1 , -1 ), # v6
            ( -1 , -1 , -1 ), # v7

            ( -1 ,  1 ,  1 ), # v0
            (  1 ,  1 ,  1 ), # v1
            (  1 ,  1 , -1 ), # v5
```




```
            ( -1 ,  1 ,  1 ), # v0
            (  1 , -1 ,  1 ), # v5
            ( -1 ,  1 , -1 ), # v4

            ( -1 , -1 ,  1 ), # v3
            (  1 , -1 , -1 ), # v6
            (  1 , -1 ,  1 ), # v2

            ( -1 , -1 ,  1 ), # v3
            ( -1 , -1 , -1 ), # v7
            (  1 , -1 , -1 ), # v6

            (  1 ,  1 ,  1 ), # v1
            (  1 , -1 ,  1 ), # v2
            (  1 , -1 , -1 ), # v6

            (  1 ,  1 ,  1 ), # v1
            (  1 , -1 , -1 ), # v6
            (  1 ,  1 , -1 ), # v5

            ( -1 ,  1 ,  1 ), # v0
            ( -1 , -1 , -1 ), # v7
            ( -1 , -1 ,  1 ), # v3

            ( -1 ,  1 ,  1 ), # v0
            ( -1 ,  1 , -1 ), # v4
            ( -1 , -1 , -1 ), # v7
            ], 'float32')
    return varr
```

```
def render():
    global gCamAng, gCamHeight
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glEnable(GL_DEPTH_TEST)
    glPolygonMode( GL_FRONT_AND_BACK, GL_LINE )

    glLoadIdentity()
    gluPerspective(45, 1, 1,10)
    gluLookAt(5*np.sin(gCamAng),gCamHeight,5*np.cos(gCamAng), 0,0,0, 0,1,0)

    drawFrame()
    glColor3ub(255, 255, 255)

    # drawCube_glVertex()
    drawCube_glDrawArrays()


def drawCube_glDrawArrays():
    global gVertexArraySeparate
    varr = gVertexArraySeparate
    glEnableClientState(GL_VERTEX_ARRAY)  # Enable it to use vertex array
    glVertexPointer(3, GL_FLOAT, 3*varr.itemsize, varr)
    glDrawArrays(GL_TRIANGLES, 0, int(varr.size/3))
```

```
gVertexArraySeparate = None
def main():
    global gVertexArraySeparate

    if not glfw.init():
        return
    window = glfw.create_window(640,640,'Lecture10', None,None)
    if not window:
        glfw.terminate()
        return
    glfw.make_context_current(window)
    glfw.set_key_callback(window, key_callback)

    gVertexArraySeparate = createVertexArraySeparate()

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

    glfw.terminate()

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

```
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 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
```

Quiz #3

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

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

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

Next Time

- Lab for this lecture (next Monday):
  - Lab assignment 6

- Next lecture:
  - 7 - Mesh 2, Lighting & Shading 1

- Acknowledgement: Some materials come from the lecture slides of
  - Prof. Jinxiang Chai, Texas A&M Univ., http://faculty.cs.tamu.edu/jchai/csce441_2016spring/lectures.html
  - Prof. Taesoo Kwon, Hanyang Univ., http://calab.hanyang.ac.kr/cgi-bin/cg.cgi
  - Prof. Steve Marschner, Cornell Univ., http://www.cs.cornell.edu/courses/cs4620/2014fa/index.shtml