Computer Graphics

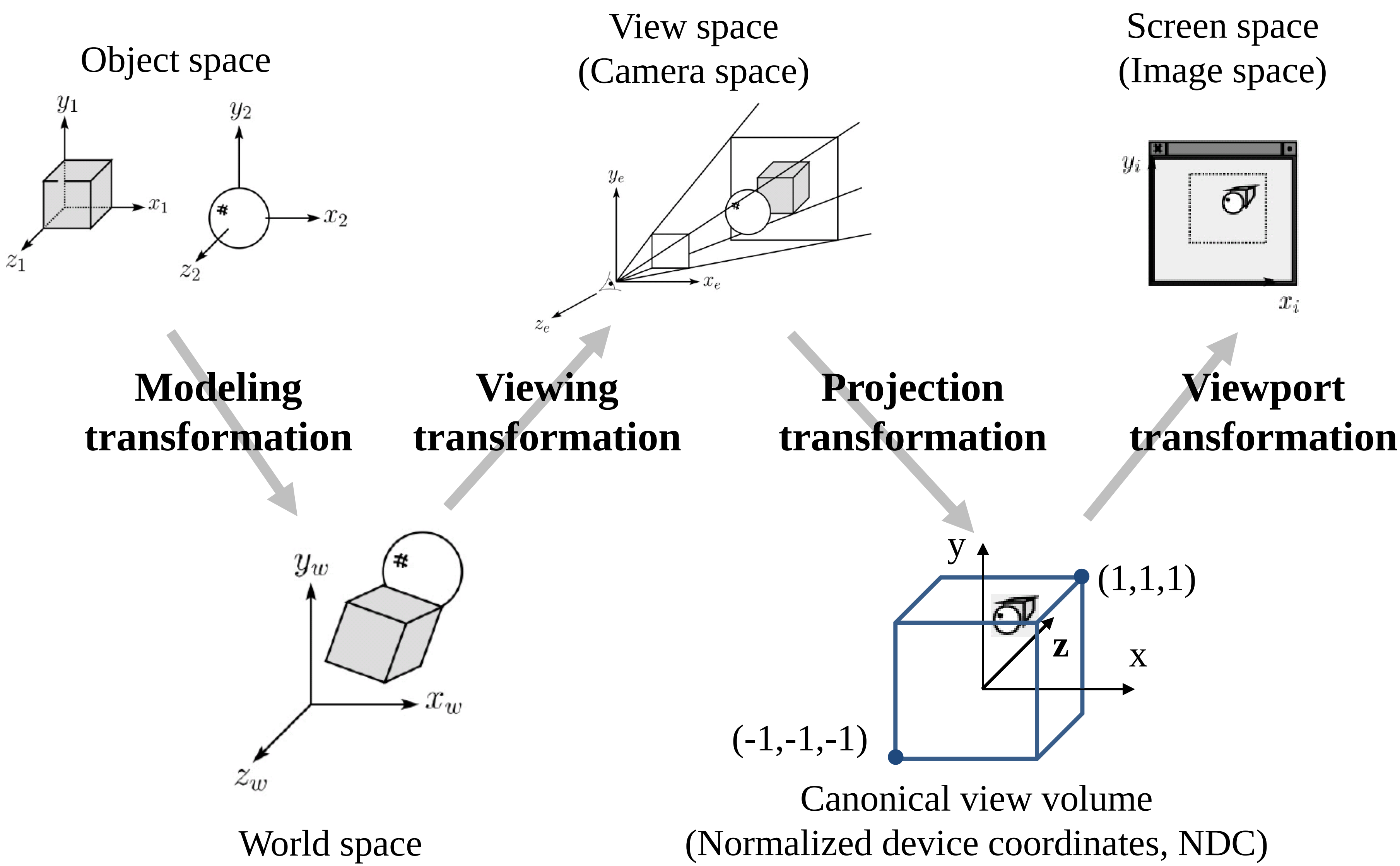
6 - Viewing, Projection

Yoonsang Lee
Spring 2019

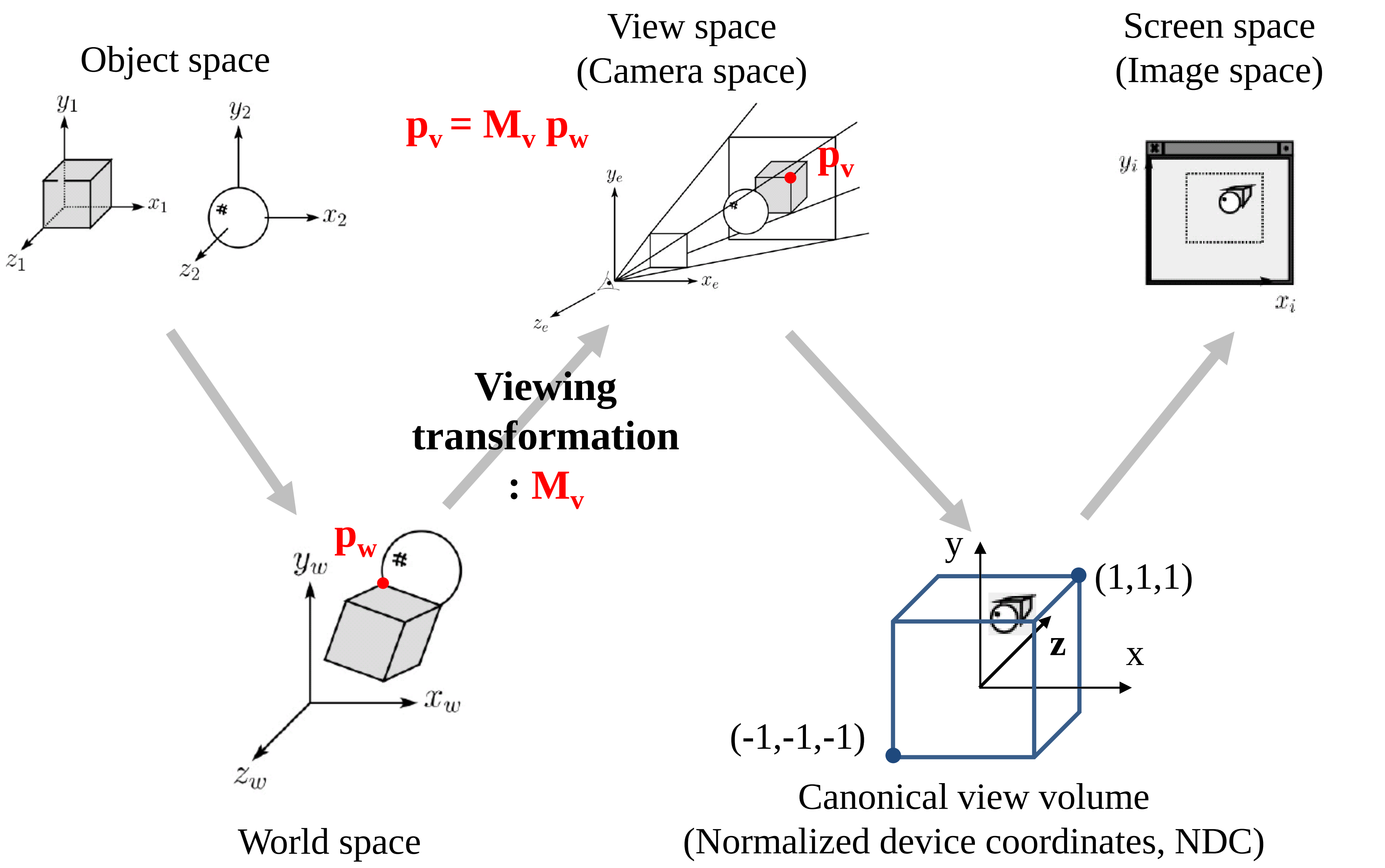
Topics Covered

- Rendering Pipeline
  - Vertex Processing
    - Viewing transformation
    - Projection Transformation
    - Viewport Transformation

Vertex Processing (Transformation Pipeline)



Viewing Transformation



Recall that...

- 1. Placing objects

→ **Modeling transformation**

- 2. Placing the “camera”

→ **Viewing transformation**

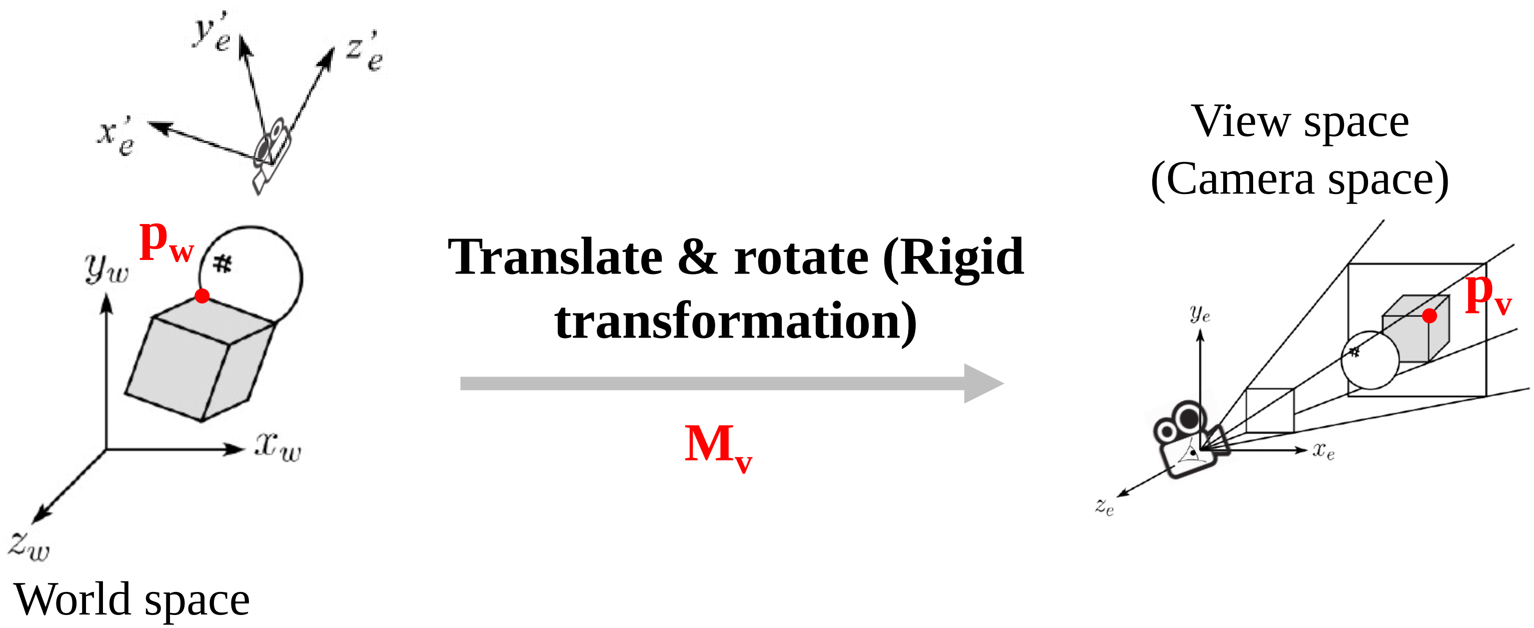
- 3. Selecting a “lens”

→ **Projection transformation**

- 4. Displaying on a “cinema screen”

→ **Viewport transformation**

Viewing Transformation



- **Placing the camera** and **expressing all object vertices from the camera's point of view**

- Transform from world to view space is traditionally called the *viewing matrix*, M_v

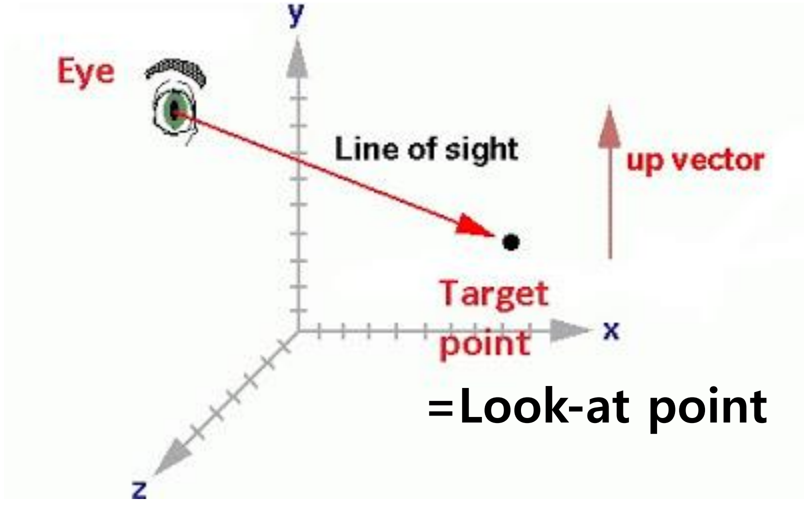
Viewing Transformation

- Placing the camera
- → **How to set the camera’s position & orientation?**

- Expressing all object vertices from the camera's point of view
- → **How to define the camera’s coordinate system (frame)?**

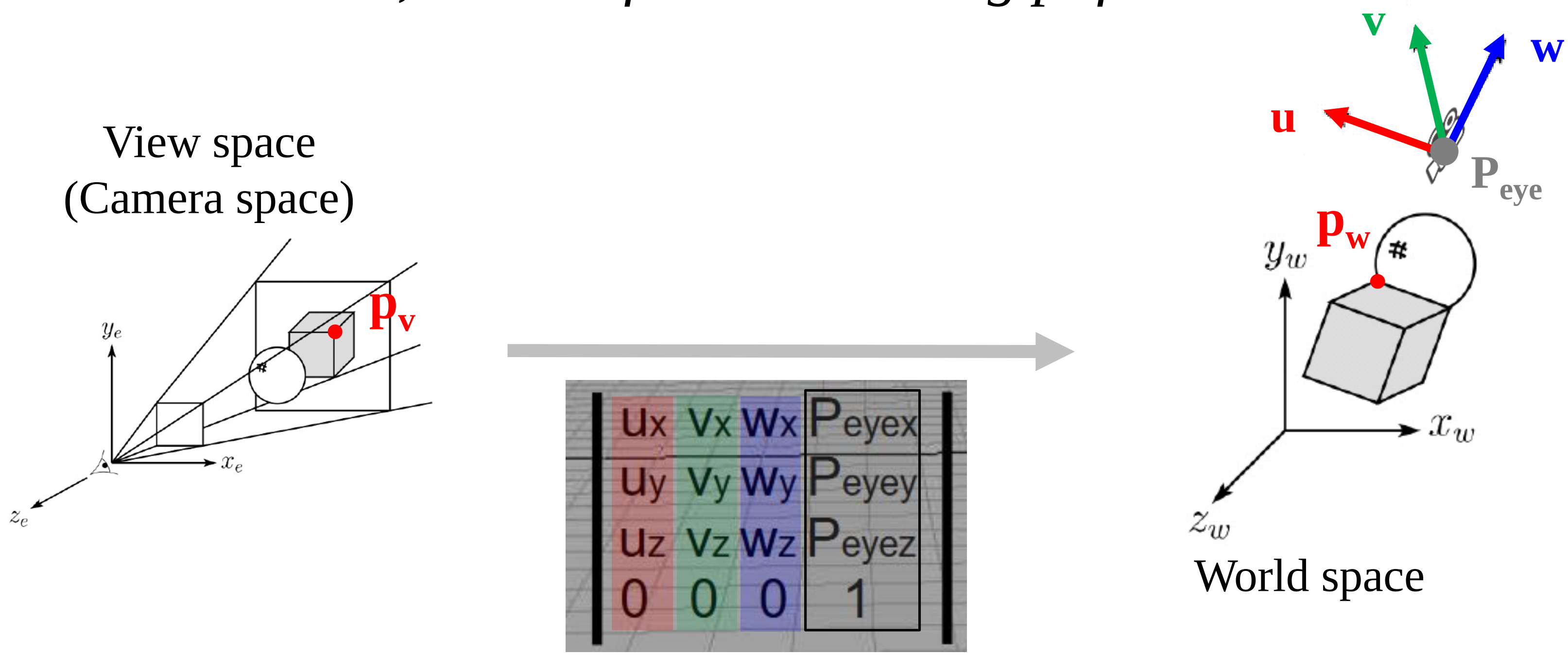
1. Setting Camera's Position & Orientation

- Many ways to do this
- One intuitive way is using:
- **Eye point**
  - Position of the camera
- **Look-at point**
  - The target of the camera
- **Up vector**
  - Roughly defines which direction is *up*



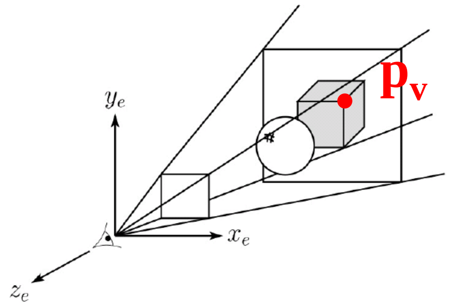
2. Defining Camera's Coordinate System

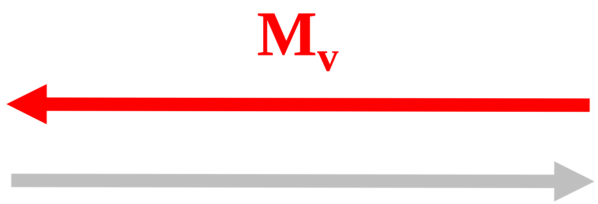
- Given **eye point**, **look-at point**, **up vector**, we can get camera frame ($\mathbf{P}_{\mathbf{eye}}$, **u, v, w**).
  - For details, see *6-reference-viewing.pdf*



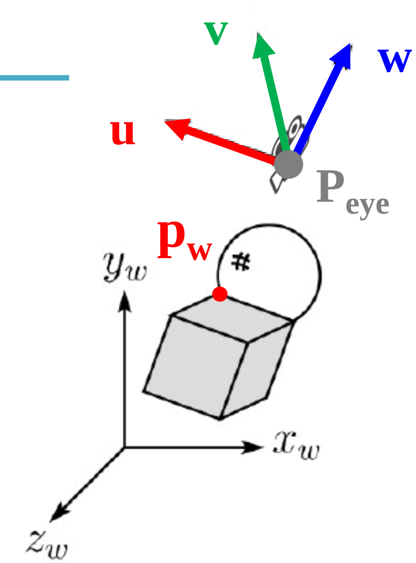
Viewing Transformation is the Opposite Direction

View space
(Camera space)





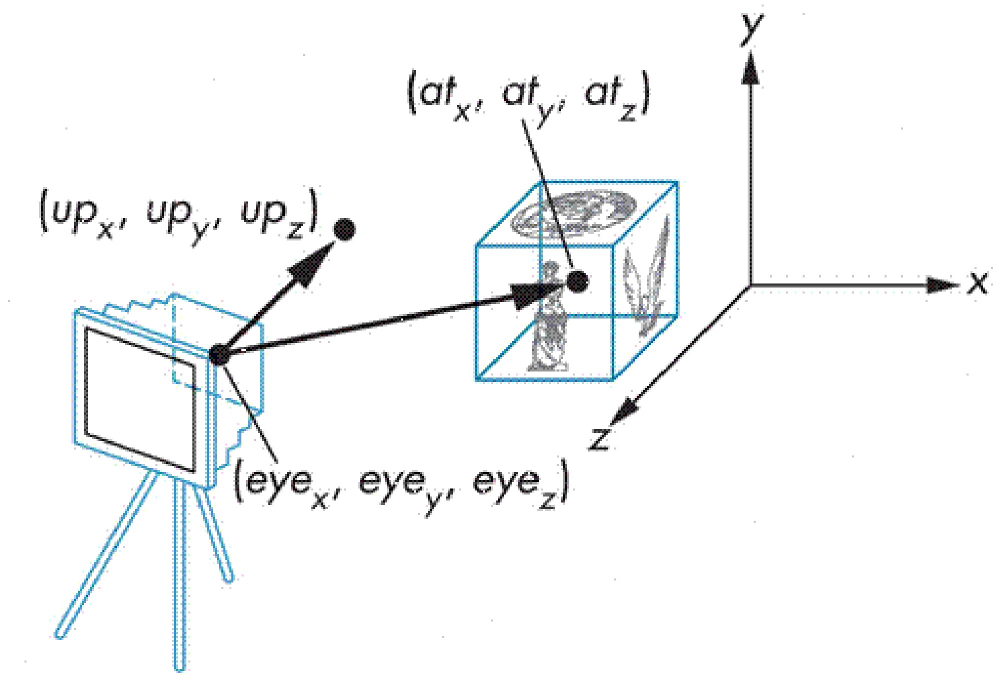
$$\begin{bmatrix} u_x & v_x & w_x & P_{eyex} \\ u_y & v_y & w_y & P_{eyey} \\ u_z & v_z & w_z & P_{eyez} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



World space

$$M_v = \begin{bmatrix} u_x & v_x & w_x & P_{eyex} \\ u_y & v_y & w_y & P_{eyey} \\ u_z & v_z & w_z & P_{eyez} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} u_x & u_y & u_z & -\mathbf{u}\cdot\mathbf{p}_{eye} \\ v_x & v_y & v_z & -\mathbf{v}\cdot\mathbf{p}_{eye} \\ w_x & w_y & w_z & -\mathbf{w}\cdot\mathbf{p}_{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

gluLookAt()



gluLookAt $(eye_x, eye_y, eye_z, at_x, at_y, at_z, up_x, up_y, up_z)$

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

$C \leftarrow CM_v$

[Practice] gluLookAt()

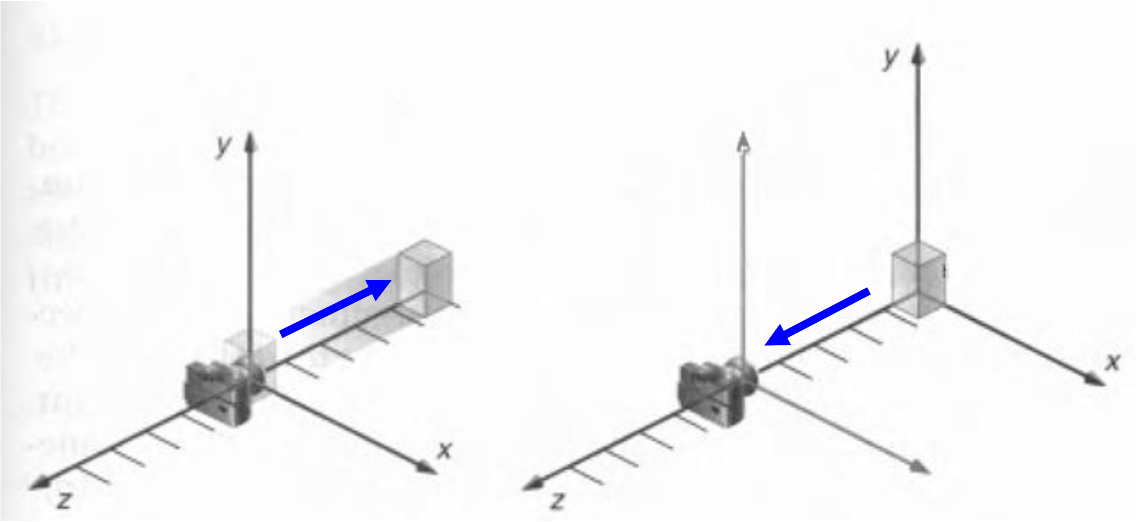
```
import glfw
from OpenGL.GL import *
from OpenGL.GLU import *
import numpy as np

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()
```

```python
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
```

```python
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

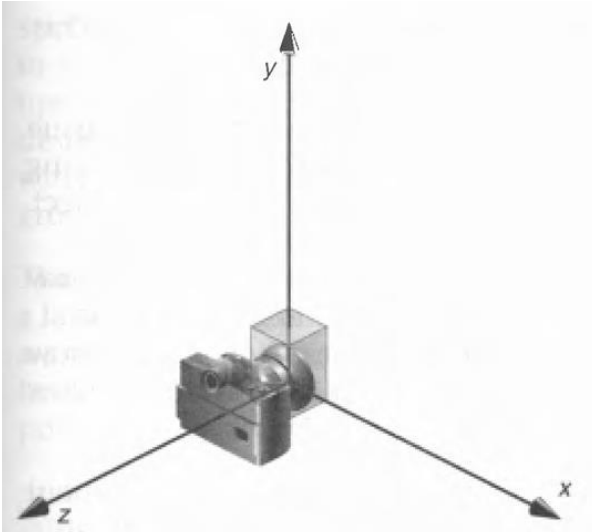


Moving Camera vs. Moving World

- Thus **you also can use glRotate*() or glTranslate*() to manipulate the camera!**

- Using gluLookAt() is just one option of many other choices to manipulate the camera.

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



Modelview Matrix

- As we've just seen, moving camera & moving world are equivalent operations.

- That's why OpenGL combines a *viewing matrix* M_v and a *modeling matrix* M_m into a ***modelview*** *matrix* $M=M_vM_m$

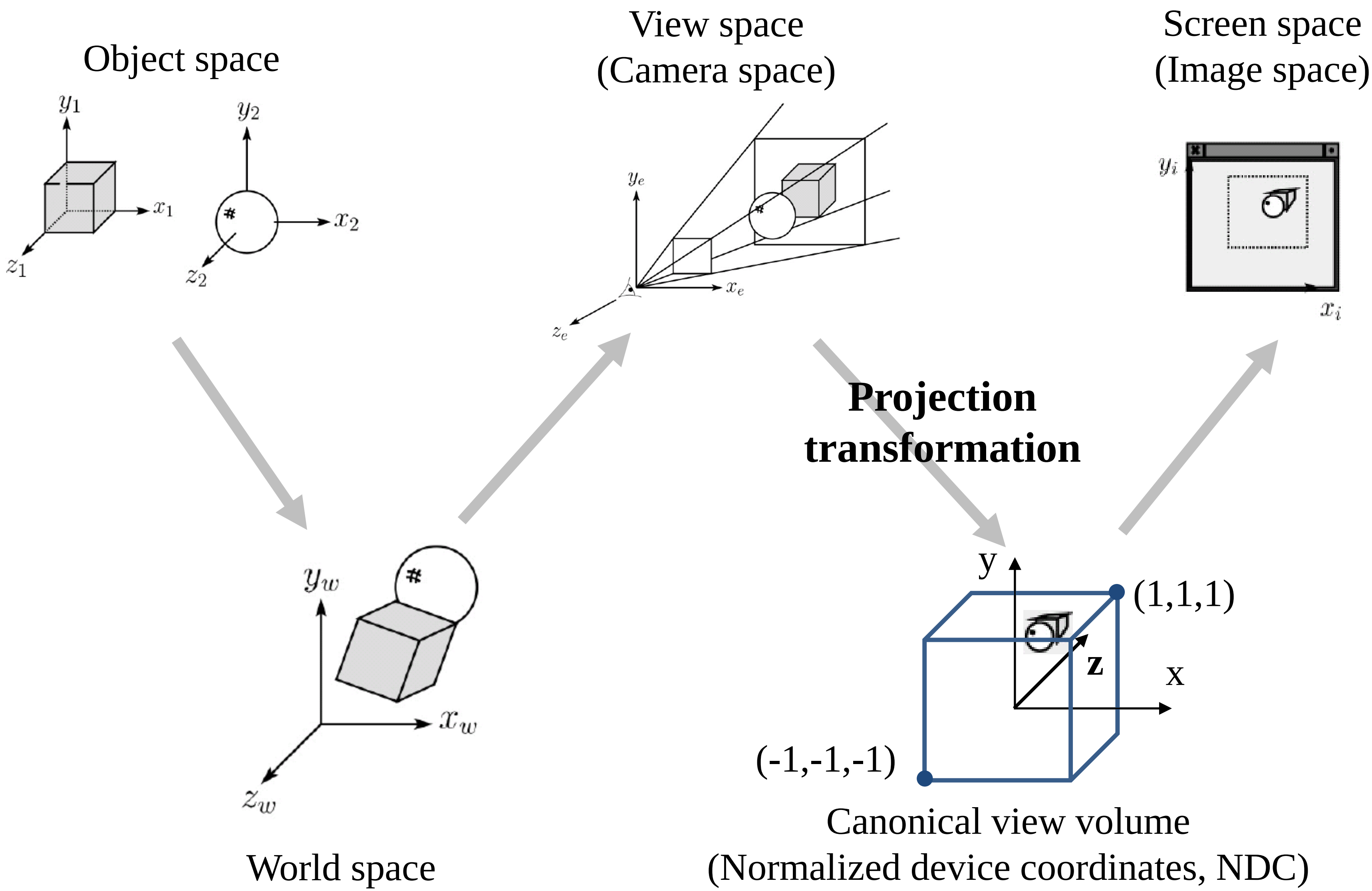
Quiz #1

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

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

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

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”

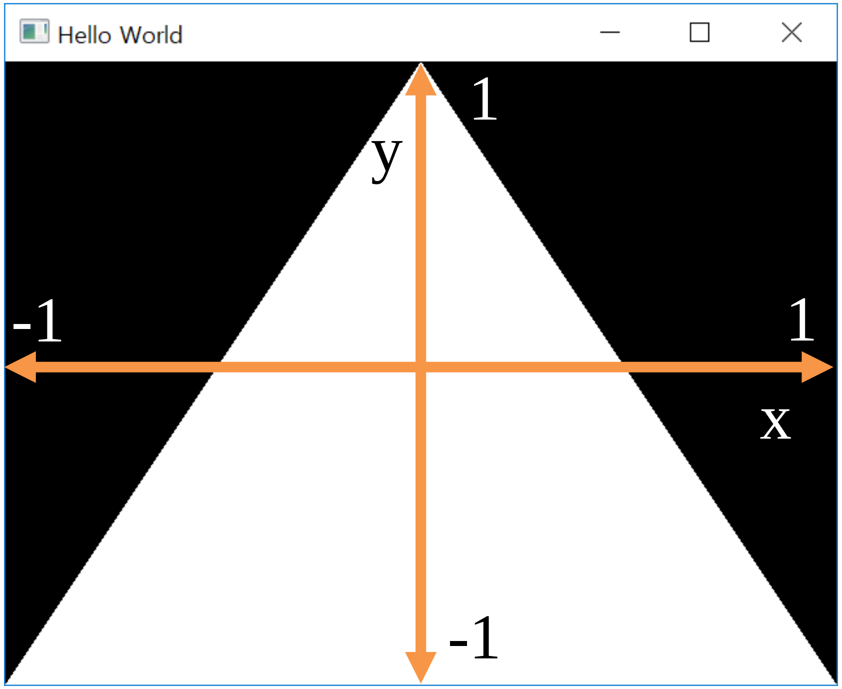
→ **Viewport transformation**

Review:Normalized Device Coordinates

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

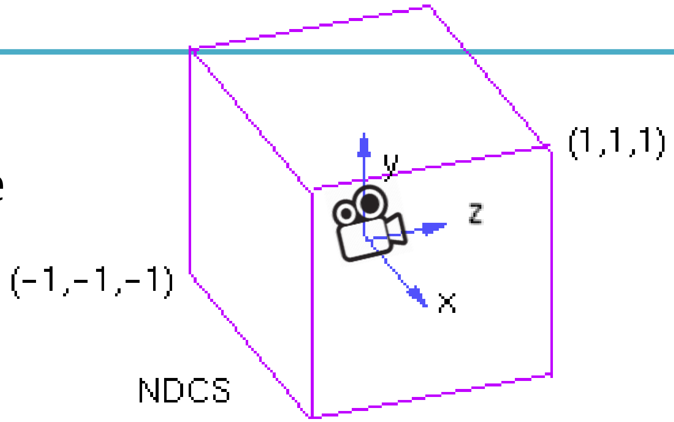
- Called **normalized device coordinates (NDC)**

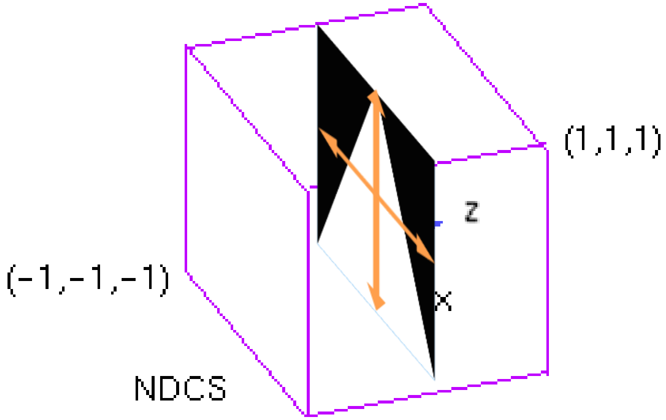
- Also known as **canonical view volume**

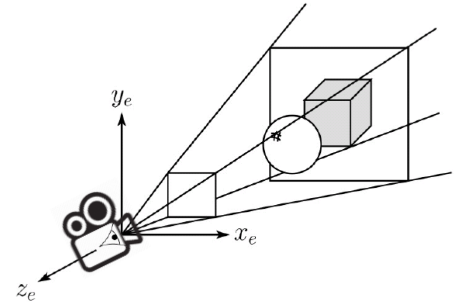


Canonical View "Volume"

- 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

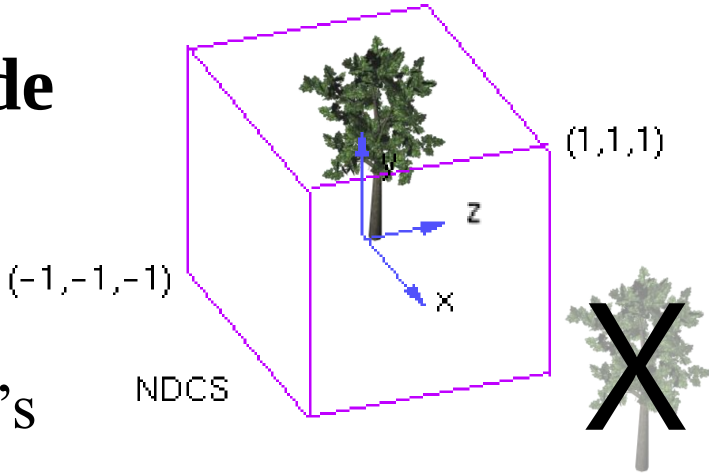






Canonical View Volume

- 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 any size visible volume and draw objects inside 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.

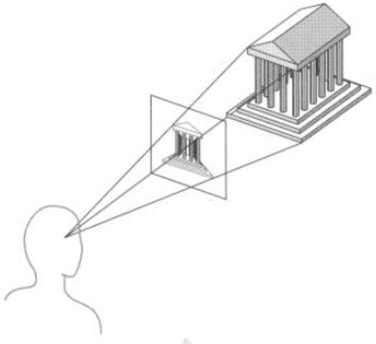
- → **Projection transformation**

Projection in General

- General definition:

- Transforming points in n-space to m-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~~

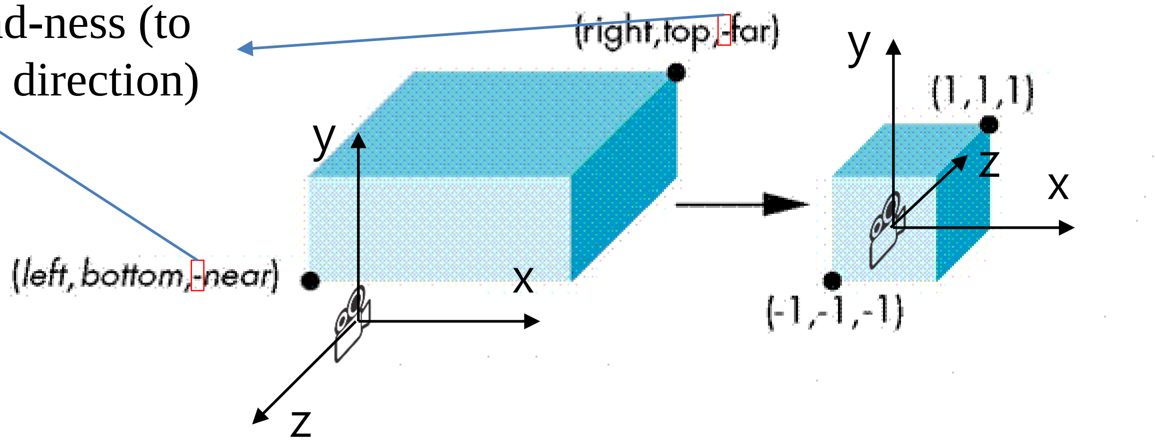
- Two common projection methods
  - Orthographic projection
  - Perspective projection

Orthographic(Orthogonal) Projection

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

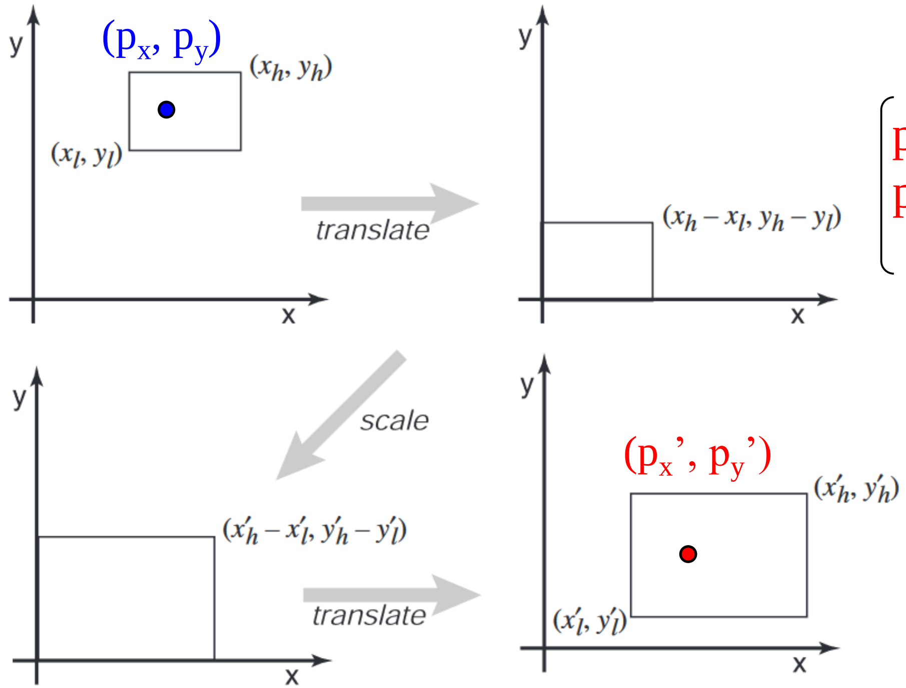
  → "Windowing" transformation

to change hand-ness (to flip positive z direction)



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')



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

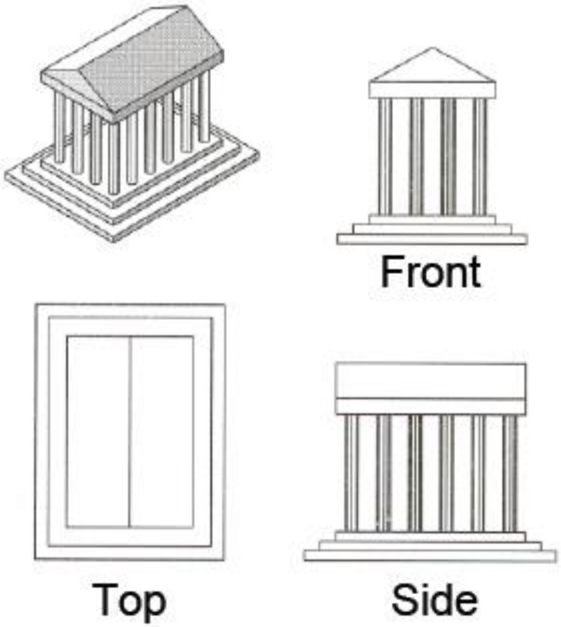
$$\begin{pmatrix} p_x' \\ p_y' \\ 1 \end{pmatrix} = \begin{bmatrix} \frac{x_h' - x_l'}{x_h - x_l} & 0 & \frac{x_l' x_h - x_h' x_l}{x_h - x_l} \\ 0 & \frac{y_h' - y_l'}{y_h - y_l} & \frac{y_l' y_h - y_h' y_l}{y_h - y_l} \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

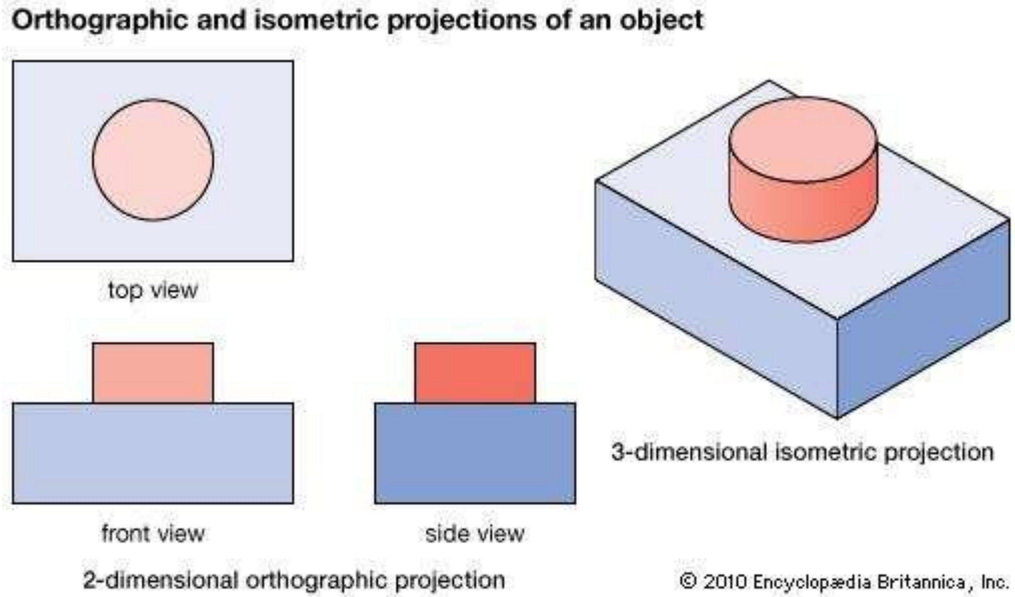
Orthographic Projection Matrix

- 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

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

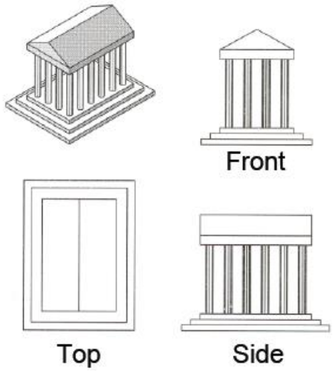
Examples of Orthographic Projection





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

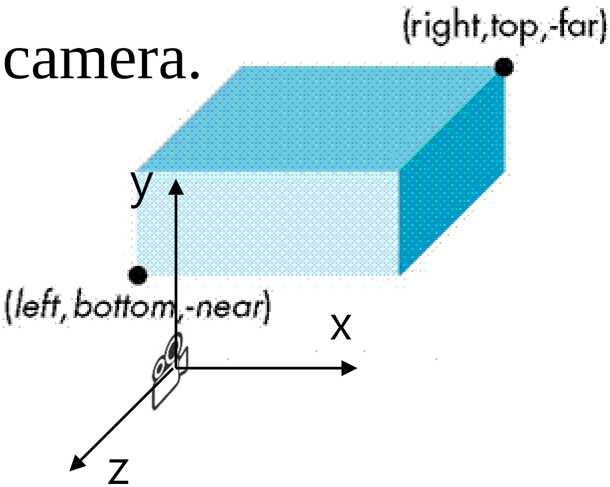
  - parallel lines remain parallel

  - ratios are preserved

  - angles are often 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)
```

```
    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()
```

```python
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()
```

```python
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 #2

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

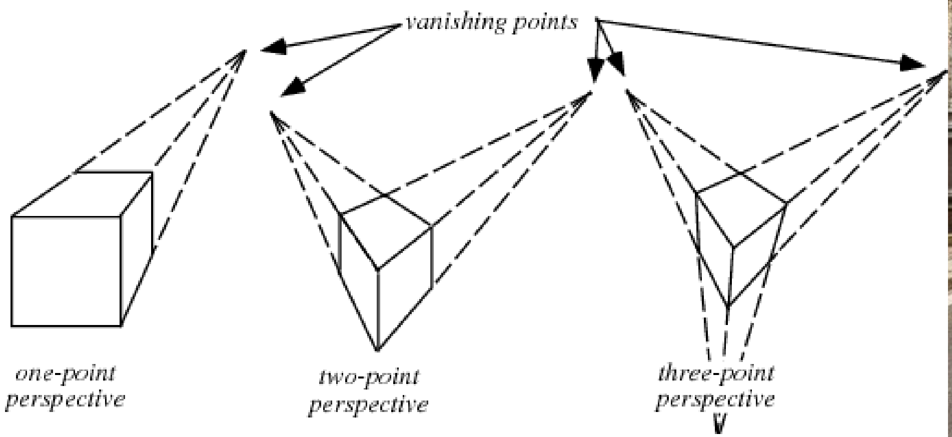
- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

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

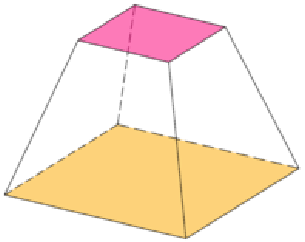
Perspective Effects

- Distant objects become small.

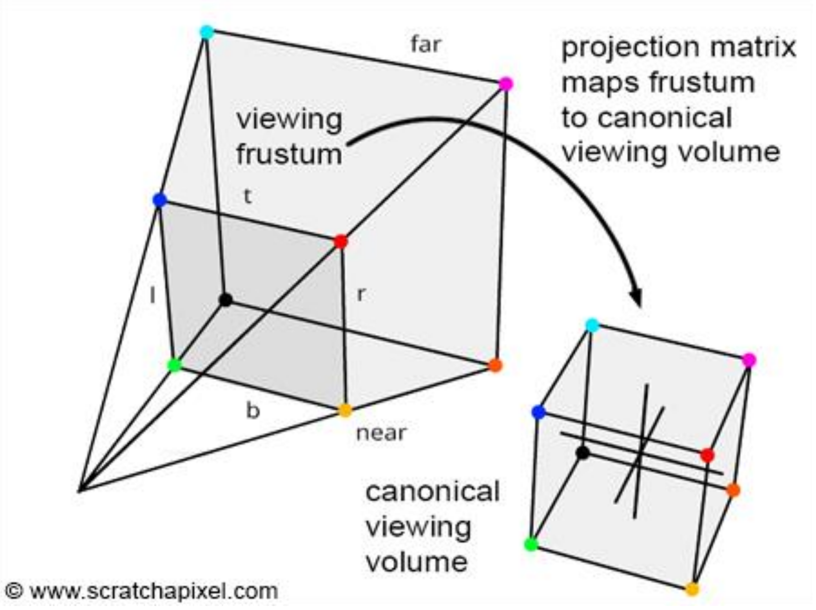
Vanishing point: The point or points to which the extensions of parallel lines appear to converge in a perspective drawing



Perspective Projection



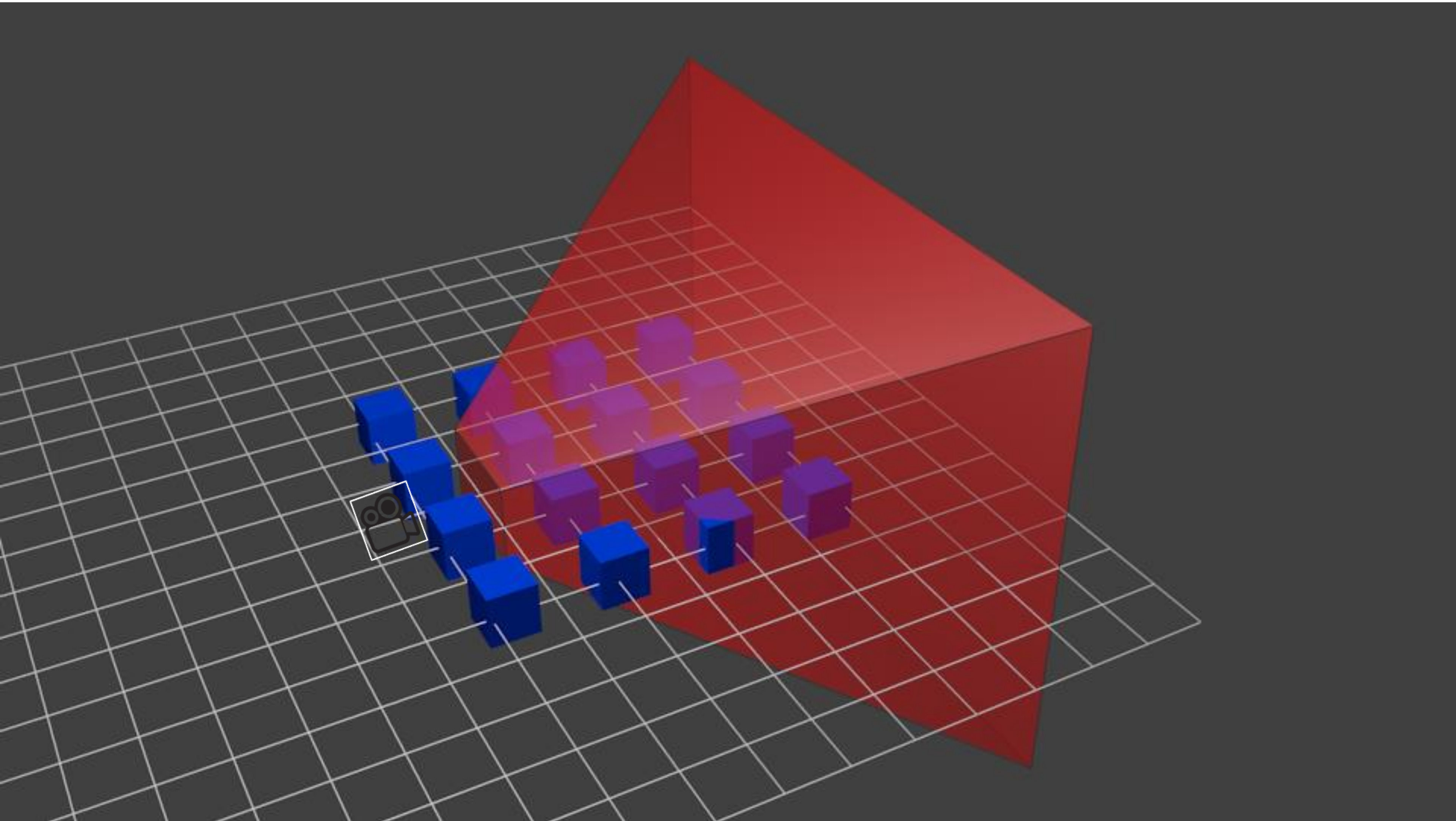
- View volume : Frustum (절두체)
- → "Viewing frustum"
- Perspective projection : Mapping from a viewing frustum to a canonical view volume



Why this mapping make “perspective”?

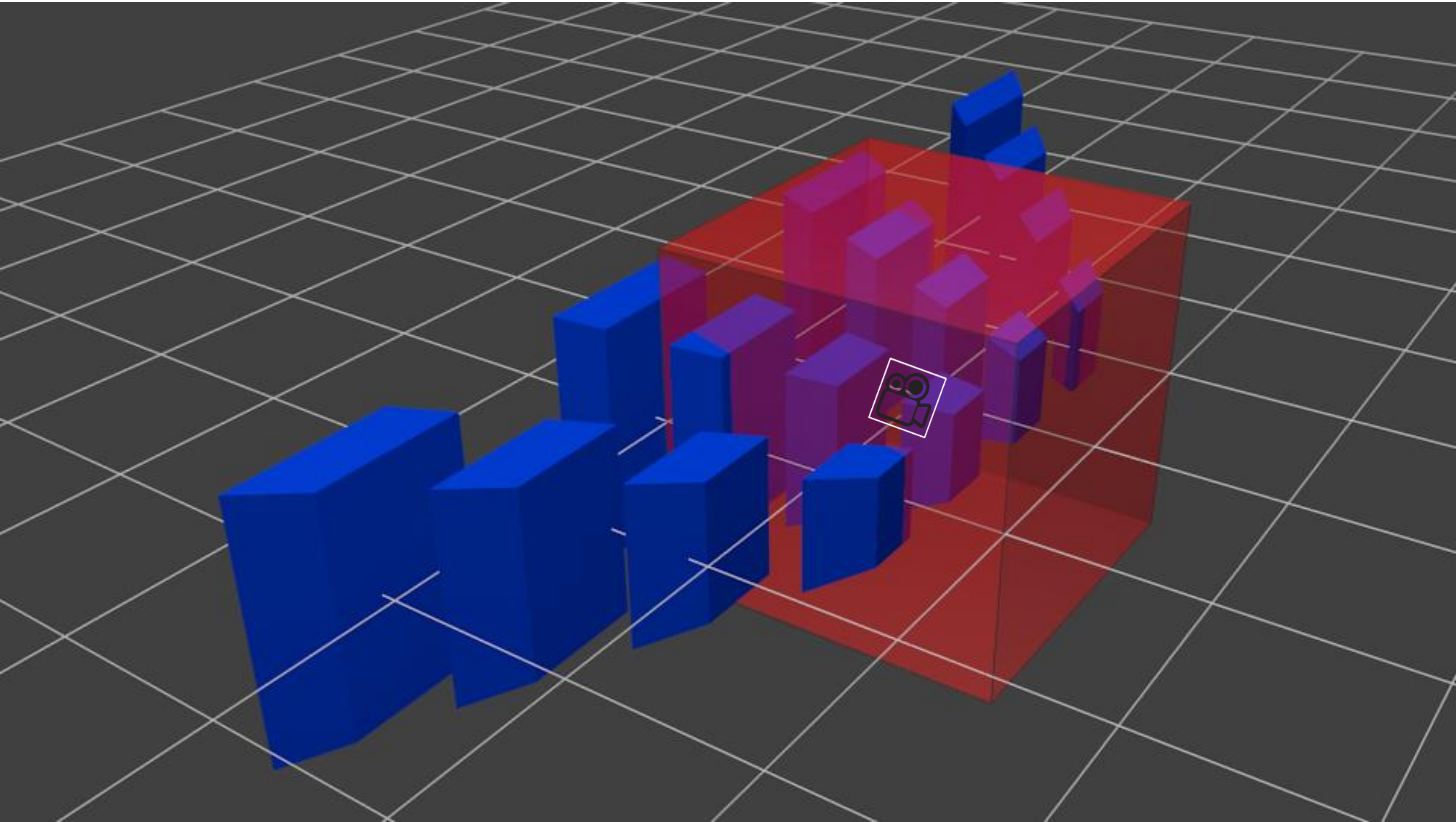
Original 3D scene

Red: viewing frustum, Blue: objects

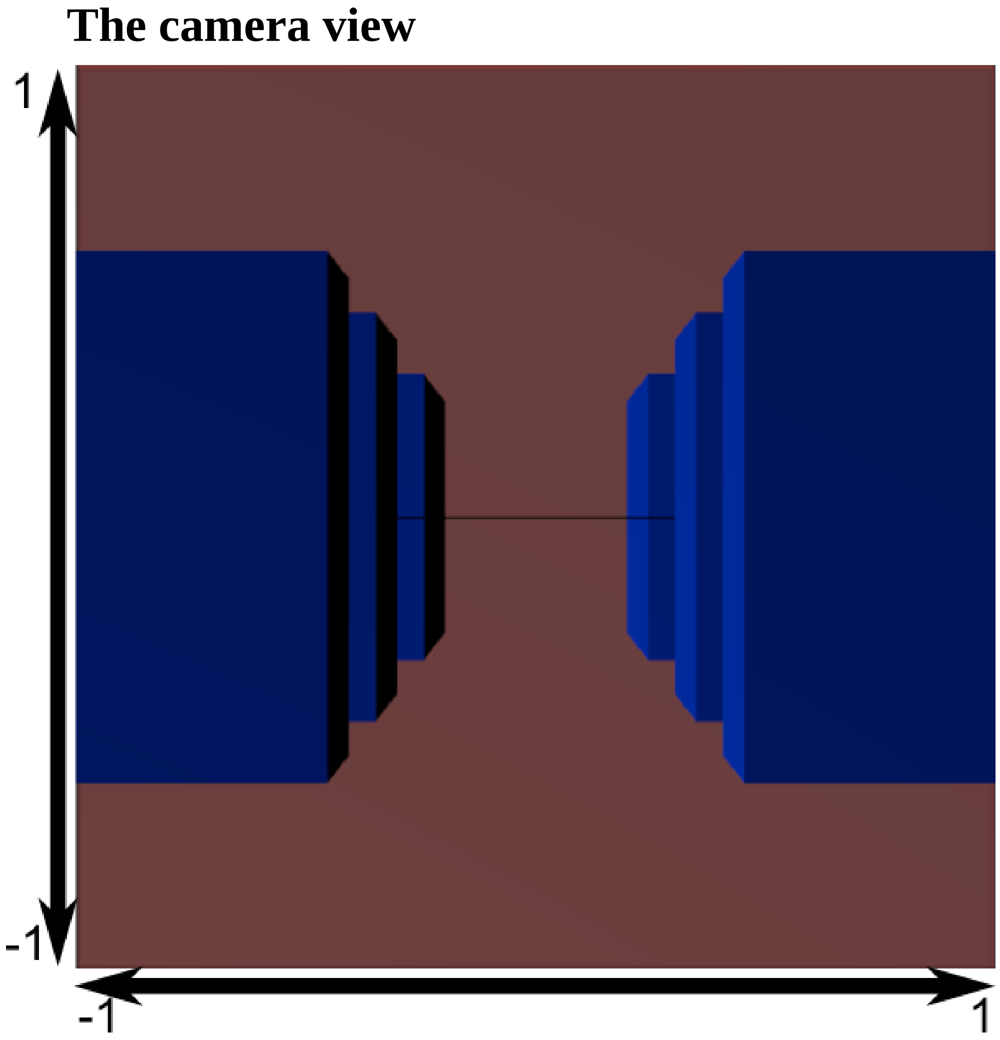


An Example of Perspective Projection

After perspective projection

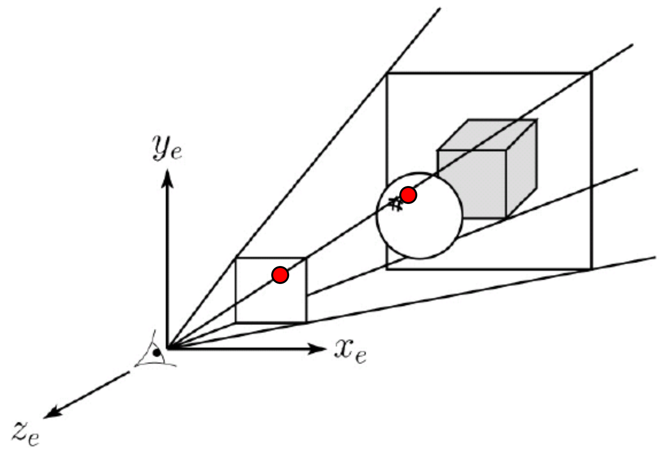


An Example of Perspective Projection



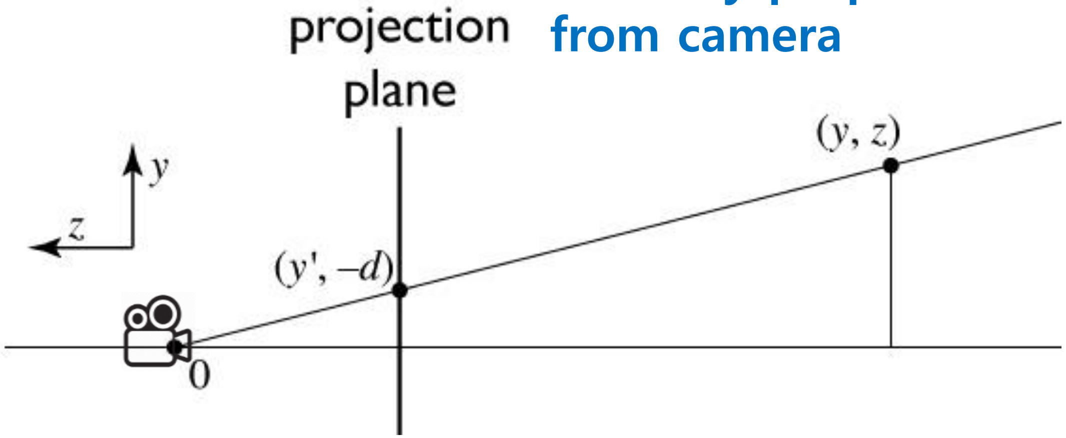
Let's first consider 3D View Frustum→2D Projection Plane

- 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

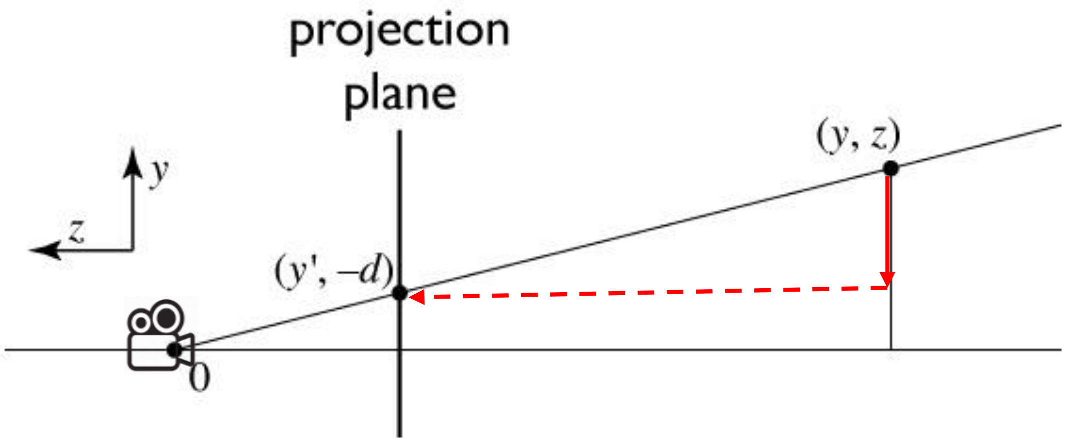
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

  – used as a convenience for unifying translation with linear

- Can also allow arbitrary *w*

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

Perspective projection

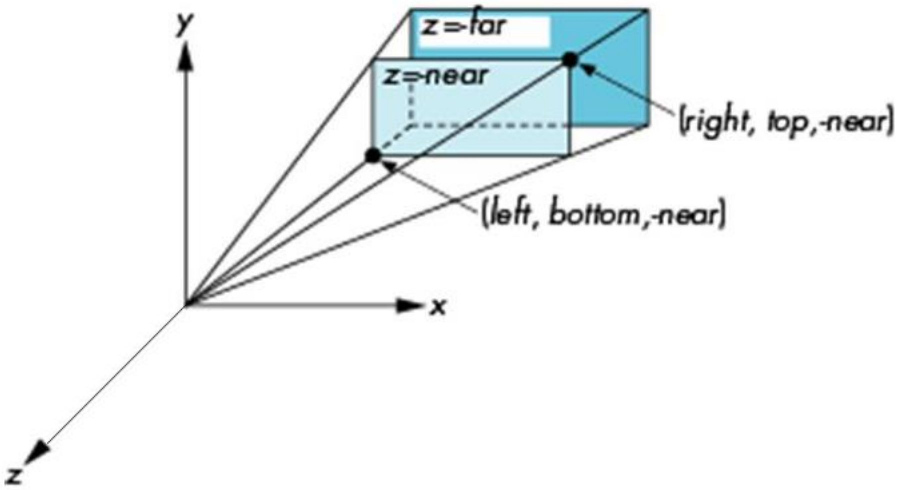


to implement perspective, just move z to w:

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

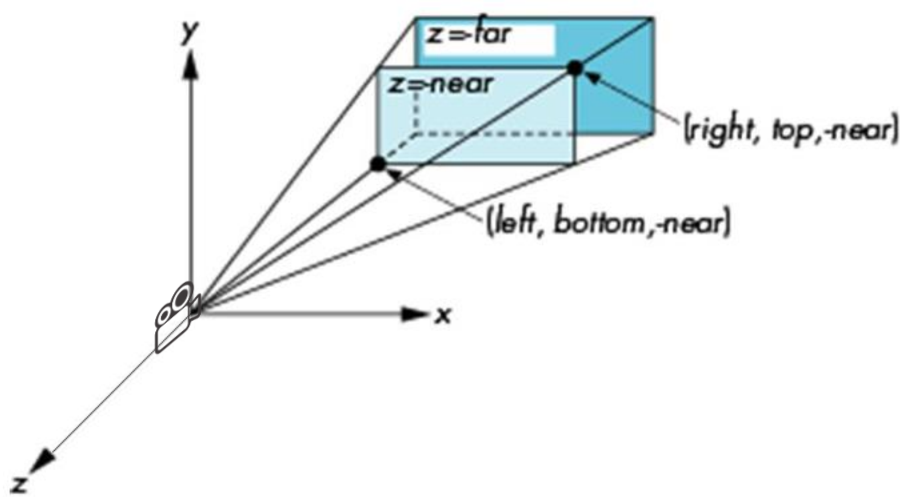
Perspective Projection Matrix

- 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*
- $M_{pers}=$ $$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



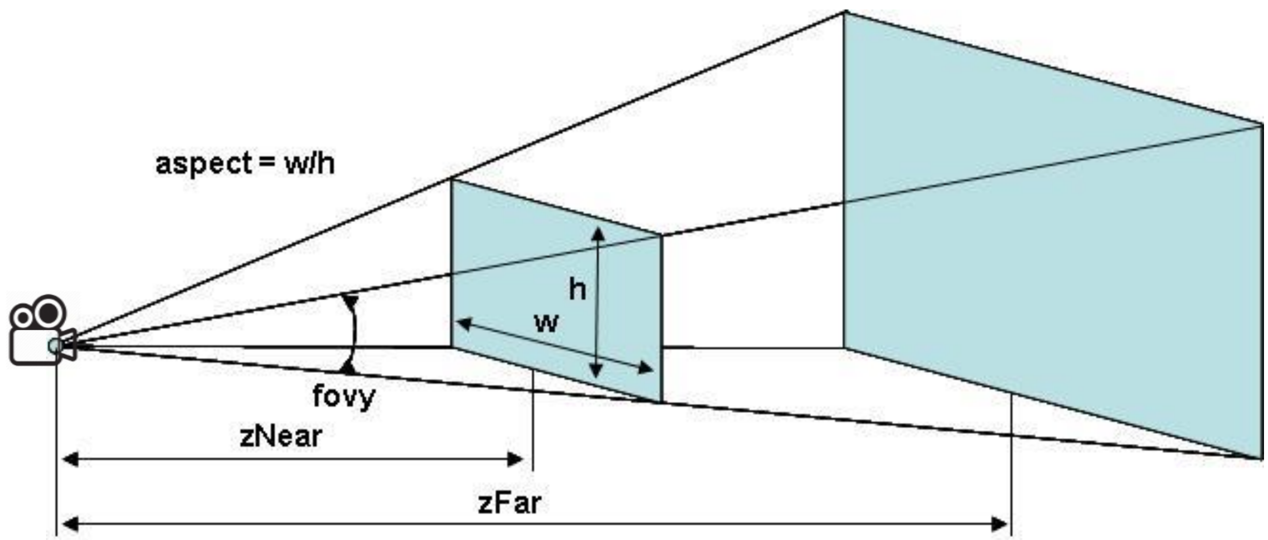
glFrustum()

- glFrustum(left, right, bottom, top, near, far)
  - near, far: The distances to the near and far depth clipping planes. **Both distances must be positive.**
- : Creates a perspective projection matrix and right-multiplies the current transformation matrix by it
- $C \leftarrow CM_{pers}$



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()

```
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(gCam
Ang), 0,0,0, 0,1,0)

    drawFrame()
    glColor3ub(255, 255, 255)

    drawUnitCube()

    # test
    # drawCubeArray()
```

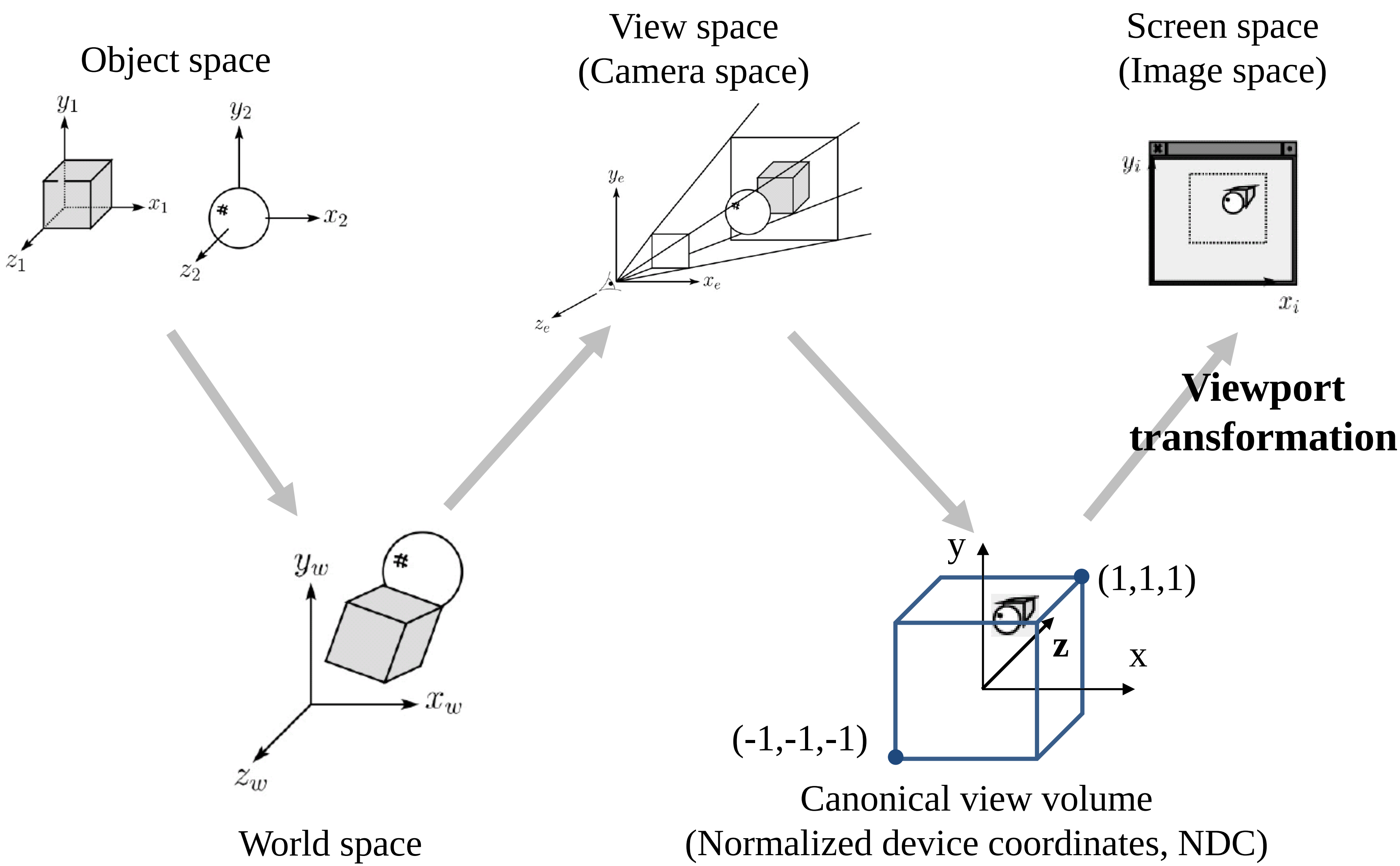
Quiz #3

- 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”.

Viewport Transformation



Recall that...

- 1. Placing objects

→ **Modeling transformation**

- 2. Placing the “camera”

→ **Viewing transformation**

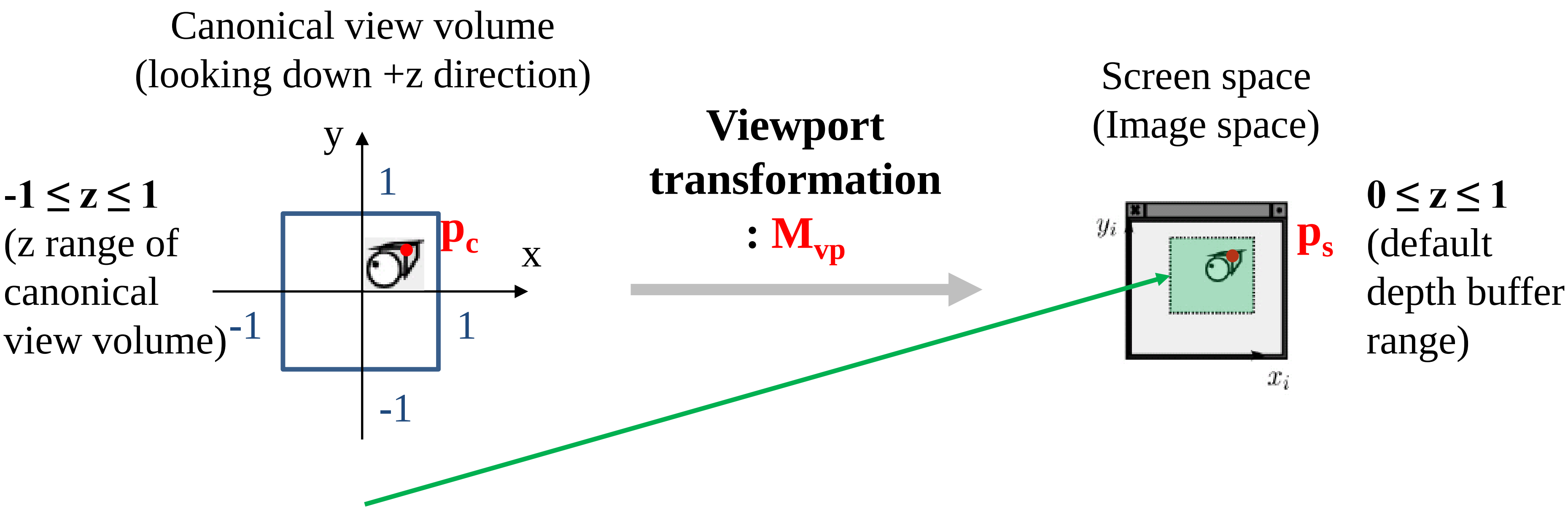
- 3. Selecting a “lens”

→ **Projection transformation**

- 4. Displaying on a “cinema screen”

→ **Viewport transformation**

Viewport Transformation



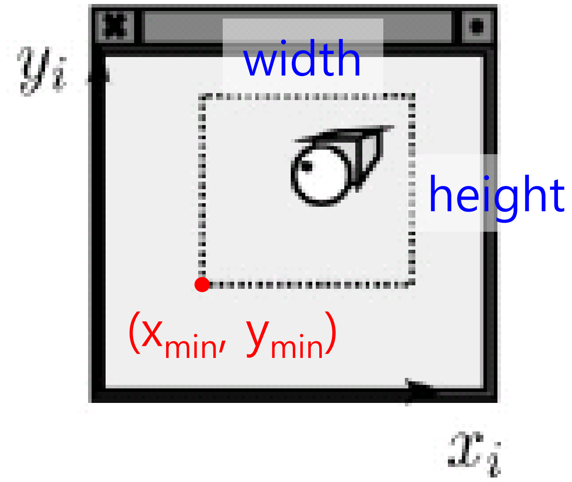
- Viewport: a rectangular viewing region of screen

- 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,

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

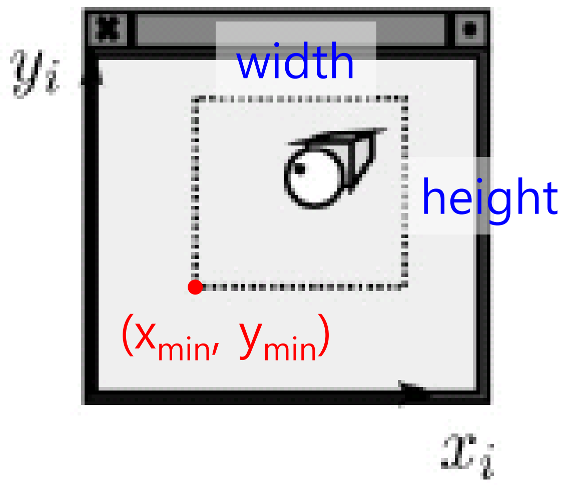


glViewport()

- 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)
    # ...
```

Next Time

- Lab in this week:
  - Lab assignment 6

- Next lecture:
  - 7 - Hierarchical Modeling, Mesh

- **Class Assignment #1**
  - **Due: 23:59, May 3, 2019**

- **Midterm Exam Notice**

- 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