# Computer Graphics

## 5 - Affine Transformation Matrix, Rendering Pipeline, Viewing

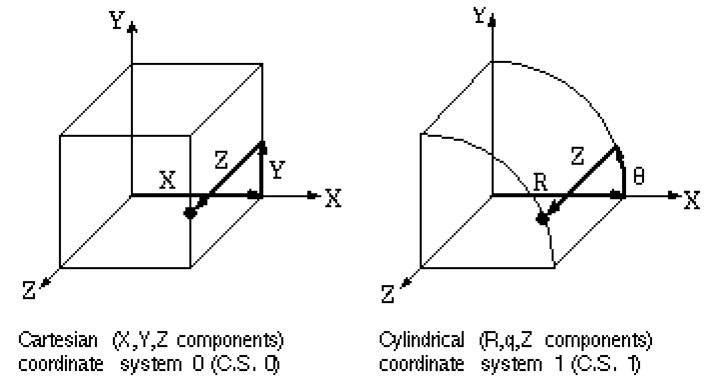
Yoonsang Lee
Spring 2022

# Topics Covered

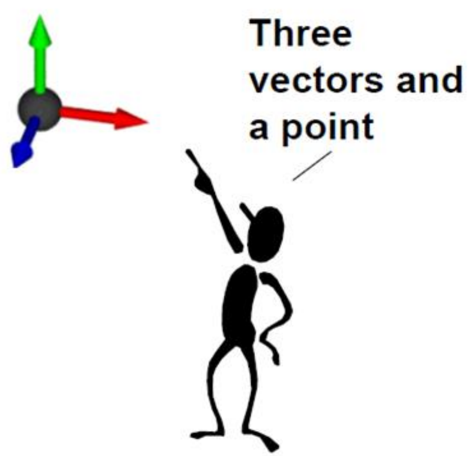
- Coordinate System & Reference Frame
- Affine Transformation Matrix
- Rendering Pipeline & Vertex Processing
- Modeling transformation
- Viewing transformation

# Coordinate System & Reference Frame

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




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



# Coordinate System & Reference Frame

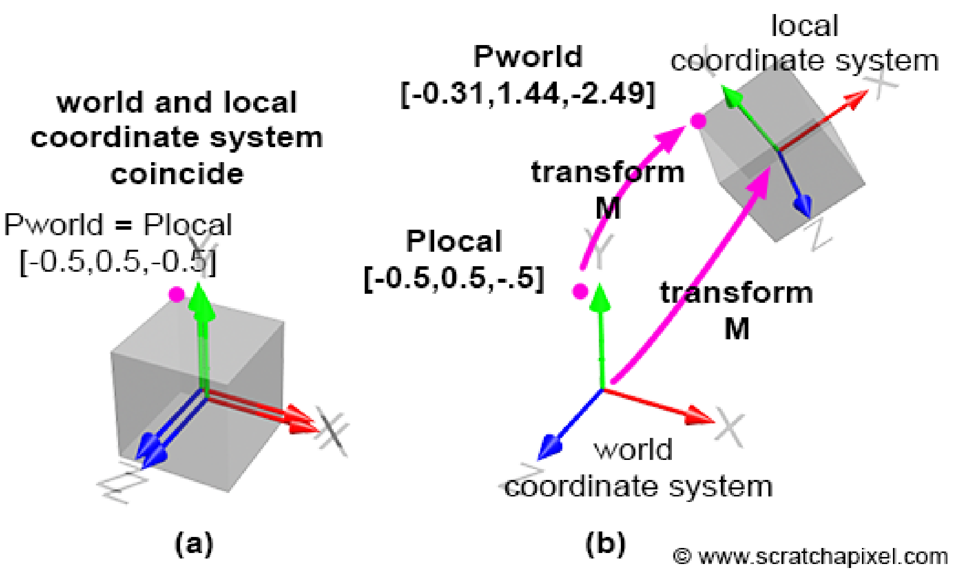
- Two terms are slightly different:
  - **Coordinate system** is a mathematical concept, about a choice of "language" used to describe observations.
  - **Reference frame** is a physical concept related to state of motion.
  - You can think the coordinate system determines the way one describes/observes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

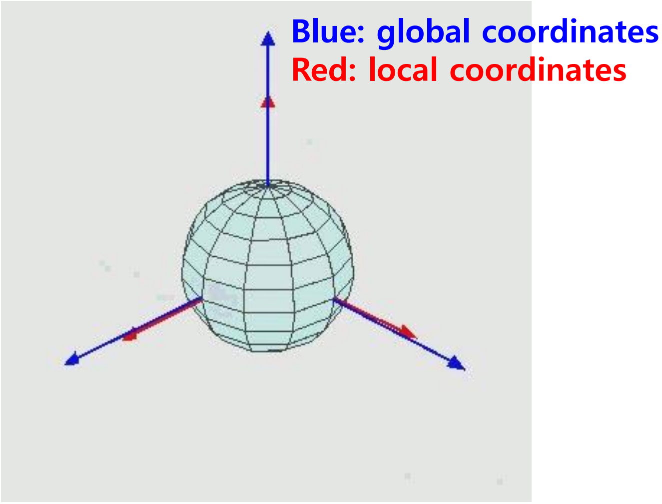
- **But these two terms are often mixed.**

# Global & Local Coordinate System(or Frame)

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

- **Local coordinate system** (or **Local frame**)
  - A coordinate system(or frame) attached to a **moving object.**



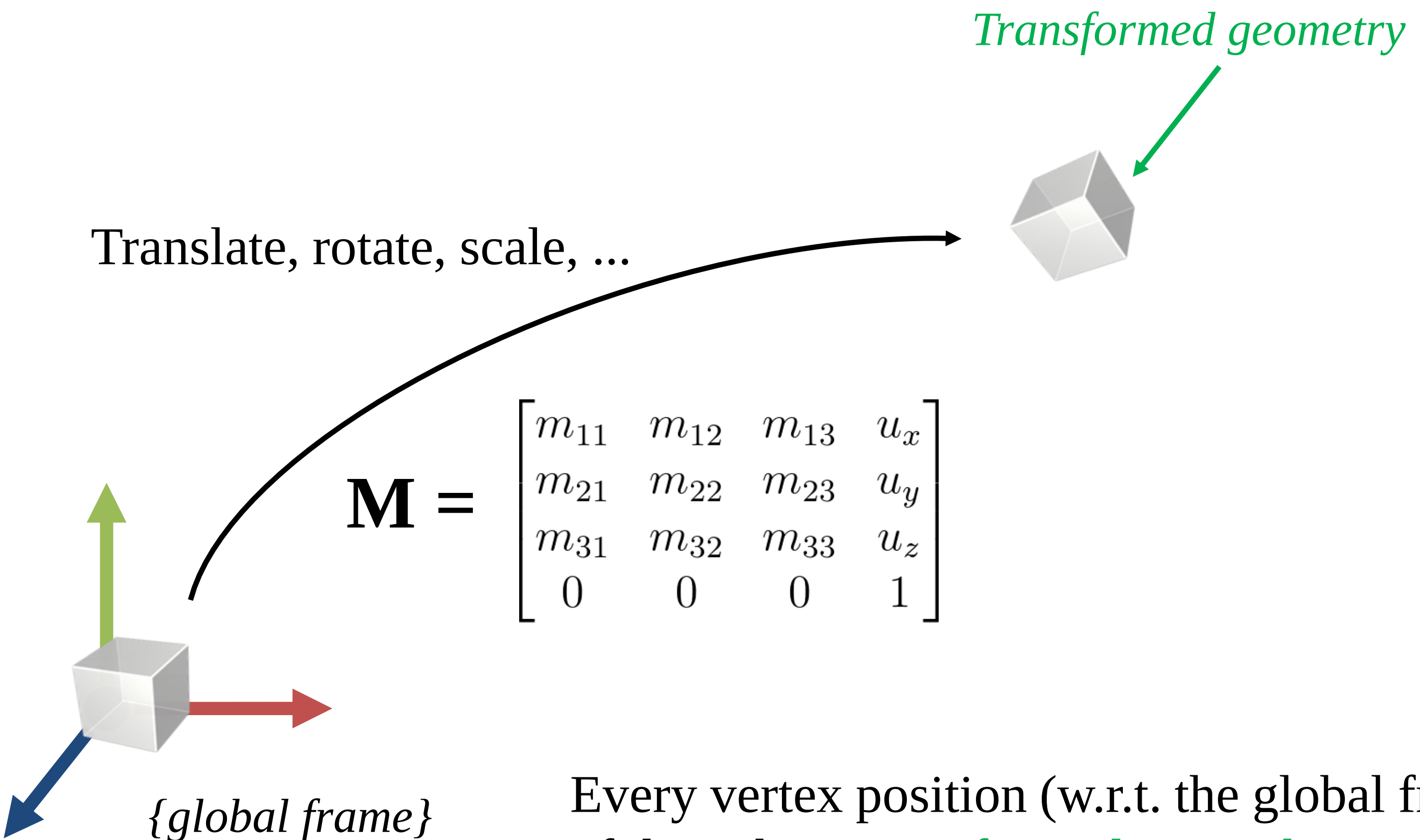

https://commons.wikimedia.org/wiki/File:Euler2a.gif

# Affine Transformation Matrix

# Meanings of Affine Transformation Matrix

- The meaning of the same affine transformation matrix can be described from different perspectives.

# 1) Affine Transformation Matrix **transforms a Geometry** w.r.t. Global Frame

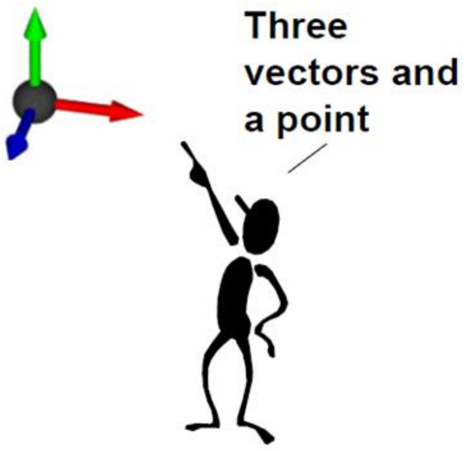



$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & u_x \\ m_{21} & m_{22} & m_{23} & u_y \\ m_{31} & m_{32} & m_{33} & u_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Every vertex position (w.r.t. the global frame) of the cube is transformed to another position (w.r.t. the global frame)

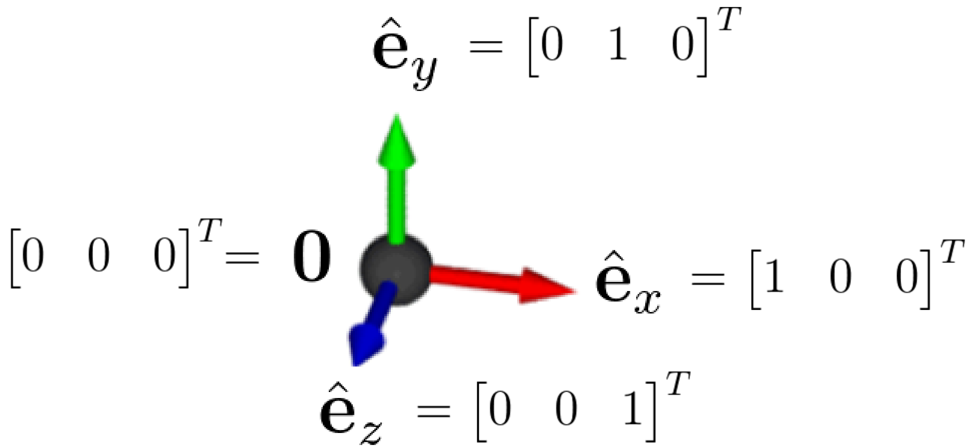
# Review: Affine Frame

- An **affine frame** in 3D space is defined by three vectors and one point
  - Three vectors for x, y, z axes
  - One point for origin



# Global Frame

- A **global frame** is usually represented by
  - Standard basis vectors for axes : $\hat{\mathbf{e}}_x, \hat{\mathbf{e}}_y, \hat{\mathbf{e}}_z$
  - Origin point : $\mathbf{0}$



# Let's transform a "global frame"

- Apply M to this "global frame", that is,
  - Multiply M with the x, y, z axis *vectors* and the origin *point* of the global frame:

x axis *vector*

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & u_x \\ m_{21} & m_{22} & m_{23} & u_y \\ m_{31} & m_{32} & m_{33} & u_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ \boxed{0} \end{bmatrix} = \begin{bmatrix} m_{11} \\ m_{21} \\ m_{31} \\ \boxed{0} \end{bmatrix}$$

y axis *vector*

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & u_x \\ m_{21} & m_{22} & m_{23} & u_y \\ m_{31} & m_{32} & m_{33} & u_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ \boxed{0} \end{bmatrix} = \begin{bmatrix} m_{12} \\ m_{22} \\ m_{32} \\ \boxed{0} \end{bmatrix}$$

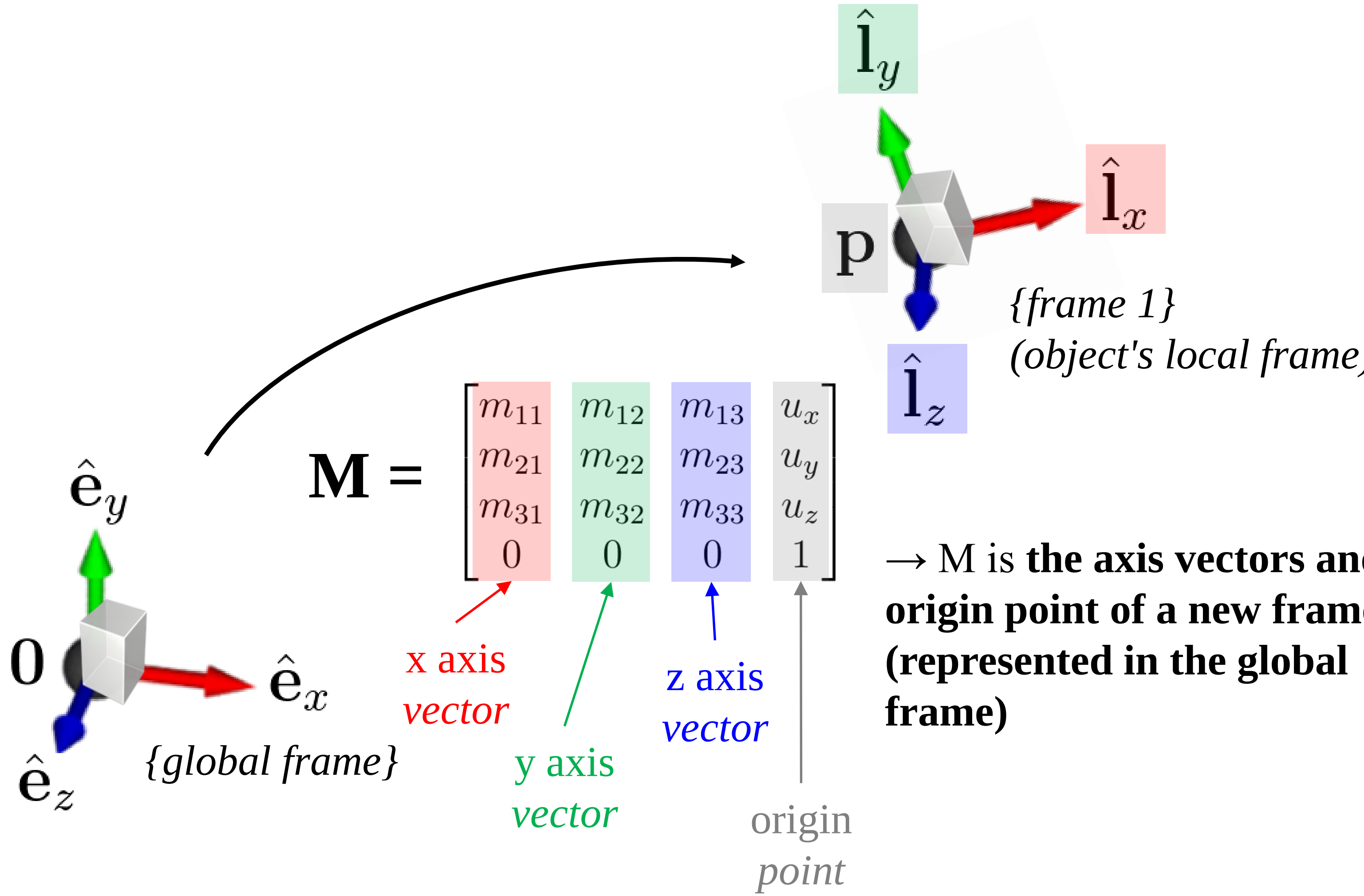
z axis *vector*

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & u_x \\ m_{21} & m_{22} & m_{23} & u_y \\ m_{31} & m_{32} & m_{33} & u_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ \boxed{0} \end{bmatrix} = \begin{bmatrix} m_{13} \\ m_{23} \\ m_{33} \\ \boxed{0} \end{bmatrix}$$

origin *point*

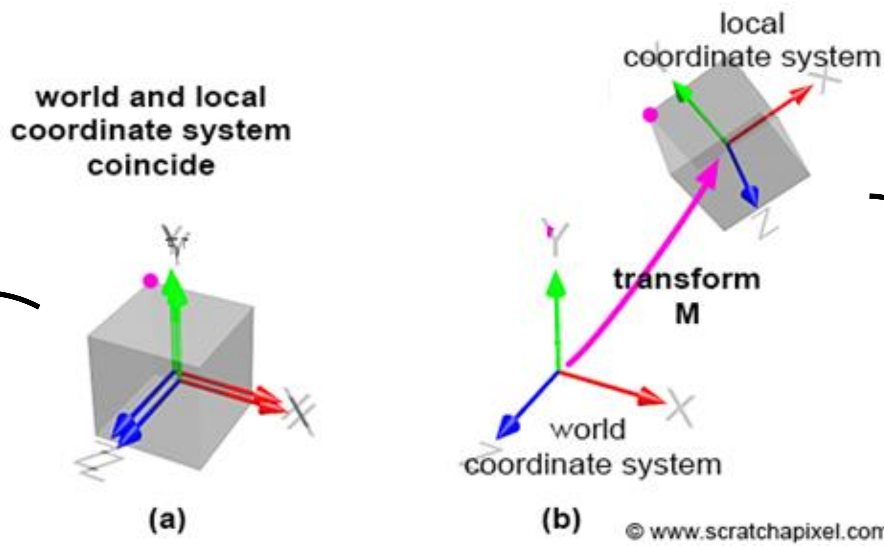
$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & u_x \\ m_{21} & m_{22} & m_{23} & u_y \\ m_{31} & m_{32} & m_{33} & u_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ \boxed{1} \end{bmatrix} = \begin{bmatrix} u_x \\ u_y \\ u_z \\ \boxed{1} \end{bmatrix}$$

# 2) Affine Transformation Matrix defines an Affine Frame w.r.t. Global Frame




→ M is **the axis vectors and origin point of a new frame (represented in the global frame)**

# Examples




The object's local frame is defined by:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

x axis *vector* — y axis *vector* — z axis *vector* — origin *point*

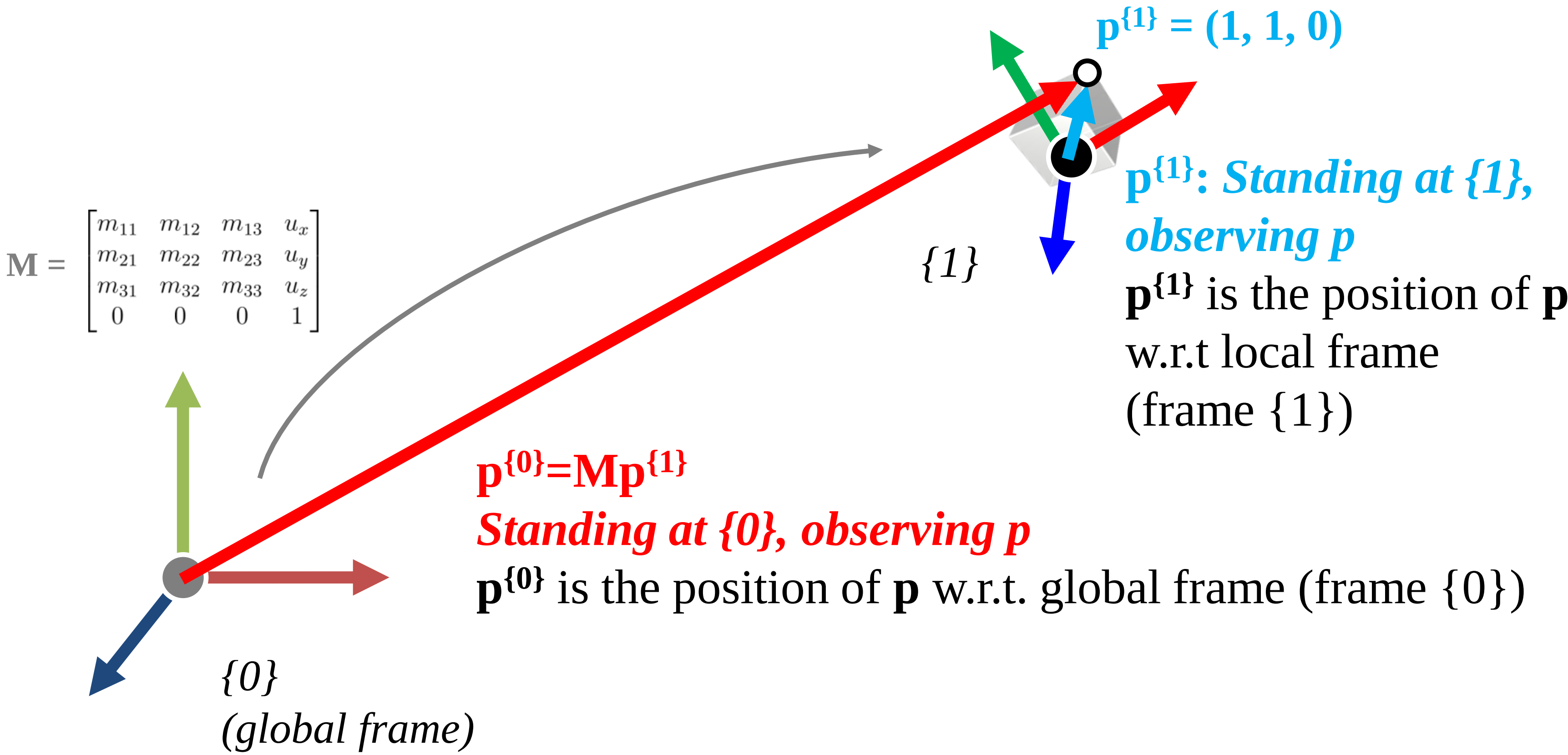
*of the local frame* ***represented in the global frame***

The object's local frame is defined by:

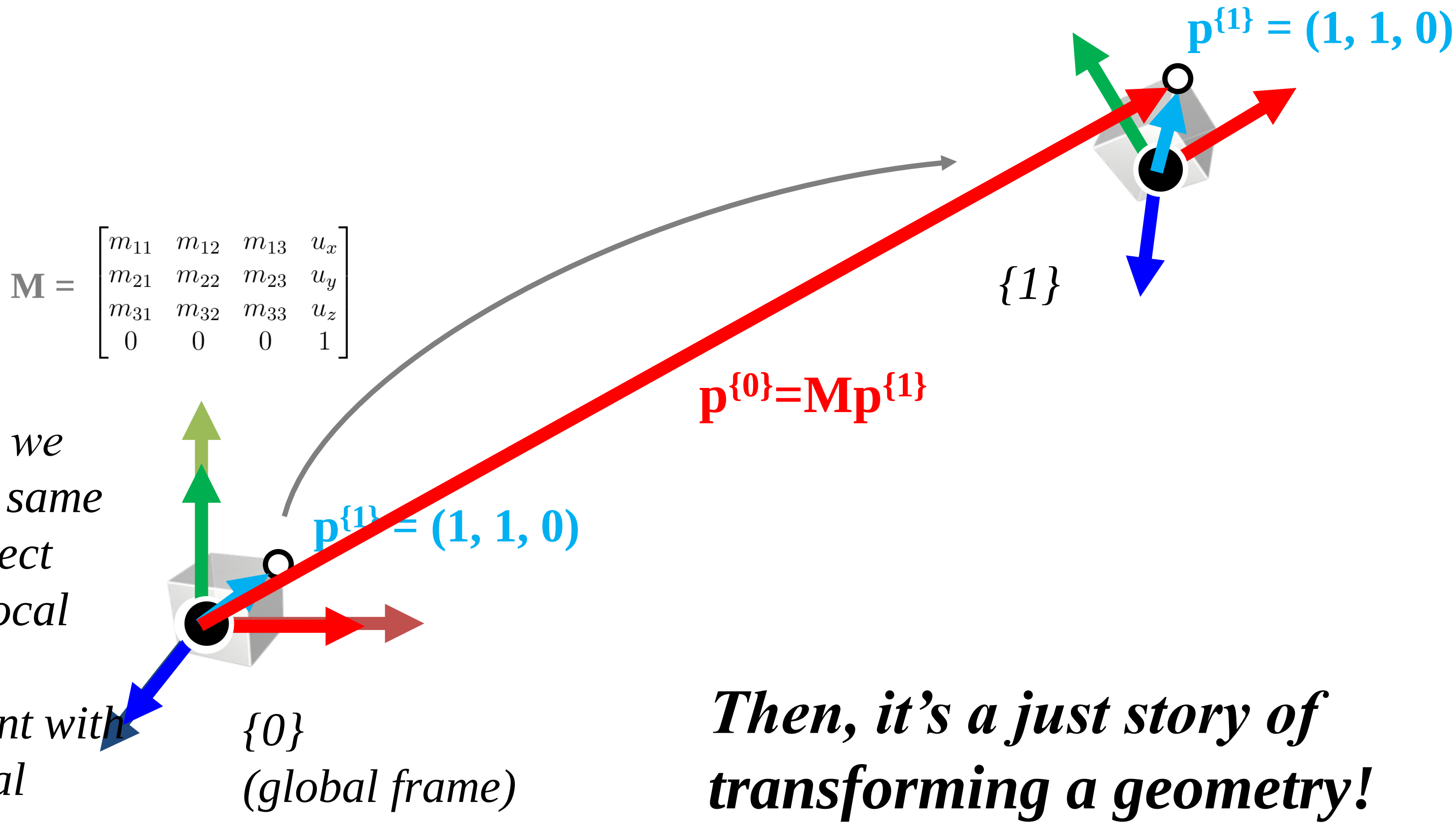
$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & u_1 \\ m_{21} & m_{22} & m_{23} & u_2 \\ m_{31} & m_{32} & m_{33} & u_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

x axis *vector* — y axis *vector* — z axis *vector* — origin *point*

# 3) Affine Transformation Matrix transforms a Point Represented in an Affine Frame to (the same) Point (but) Represented in Global Frame



## 3) Affine Transformation Matrix transforms a Point Represented in an Affine Frame to (the same) Point (but) Represented in Global Frame Because...




*Let's say we have the same cube object and its local frame coincident with the global frame*

***Then, it's a just story of transforming a geometry!***

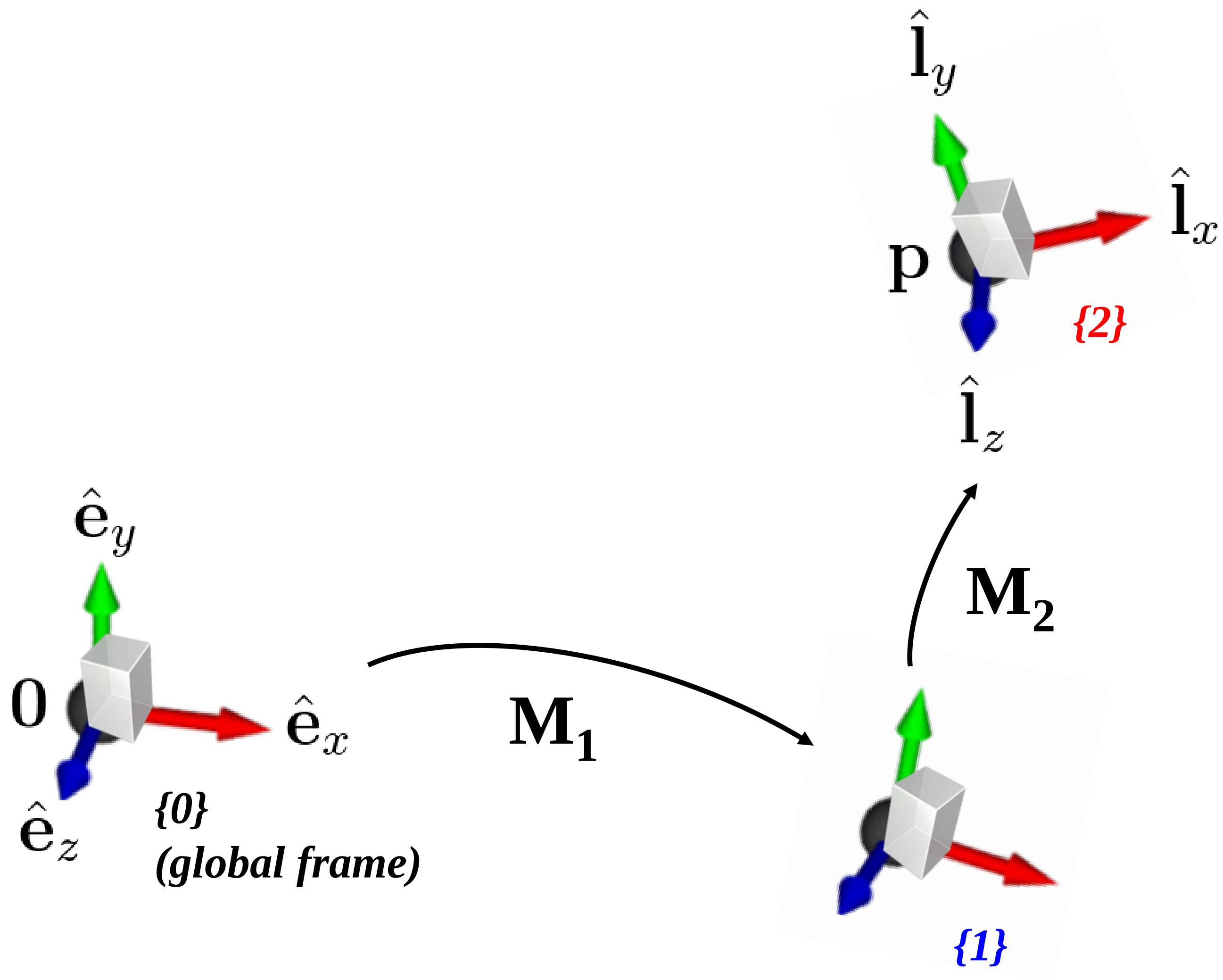
# Quiz #1

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

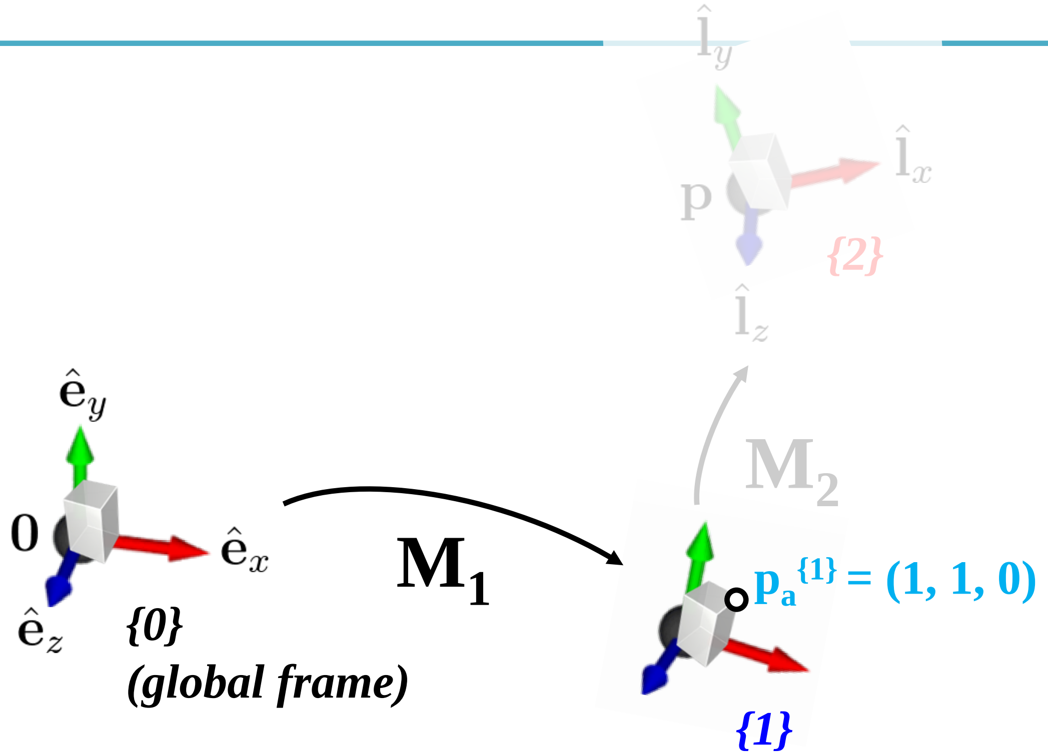
- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

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

# All these concepts works even if the starting frame is not global frame!

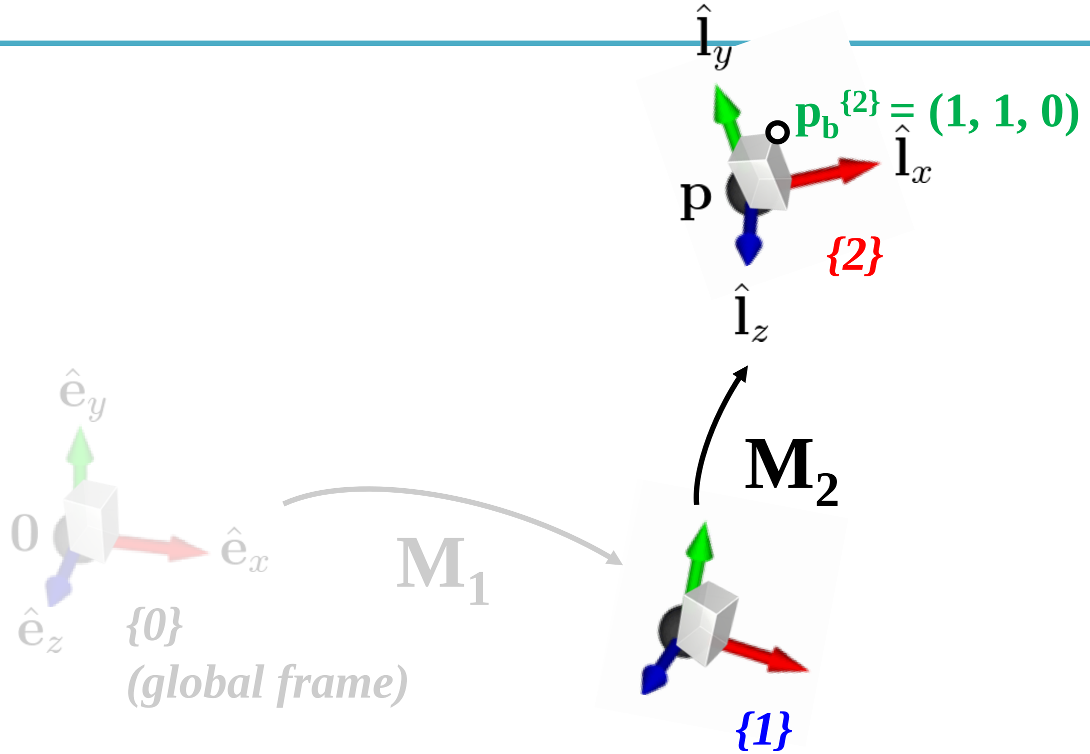


# {0} to {1}



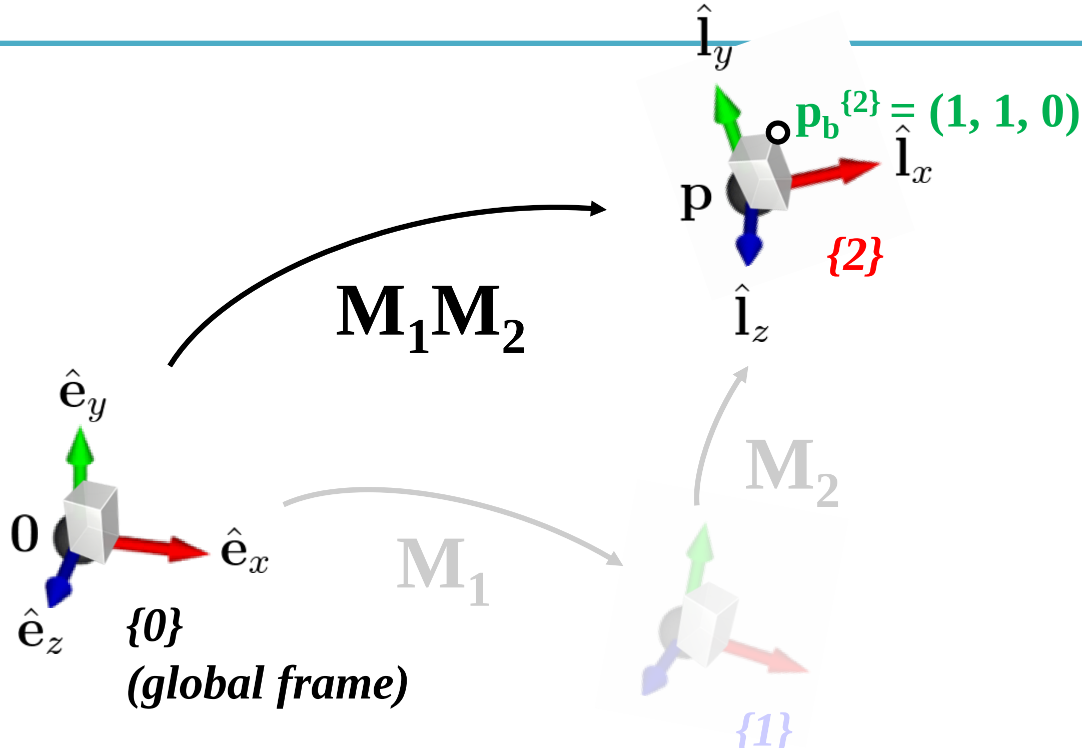

- 1) $\mathbf{M}_1$ transforms a geometry (represented in ***{0}***) w.r.t. ***{0}***
- 2) $\mathbf{M}_1$ defines an ***{1}*** w.r.t. ***{0}***
- 3) $\mathbf{M}_1$ transforms a point represented in ***{1}*** to the same point but represented in ***{0}***
  - $\mathbf{p}_a^{\{0\}}=\mathbf{M}_1\mathbf{p}_a^{\{1\}}$

# {1} to {2}




- 1) $\mathbf{M_2}$ transforms a geometry (represented in ***{1}***) w.r.t. ***{1}***
- 2) $\mathbf{M_2}$ defines an ***{2}*** w.r.t. ***{1}***
- 3) $\mathbf{M_2}$ transforms a point represented in ***{2}*** to the same point but represented in ***{1}***
  - $\mathbf{p_b}^{\{1\}} = \mathbf{M_2 p_b}^{\{2\}}$

# {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}***
  - $\mathbf{p_b^{\{1\}}=M_2p_b^{\{2\}}}$, $\mathbf{p_b^{\{0\}}=M_1p_b^{\{1\}}=M_1M_2p_b^{\{2\}}}$

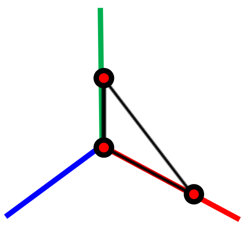
# Rendering Pipeline

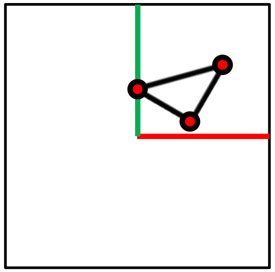
# Rendering Pipeline

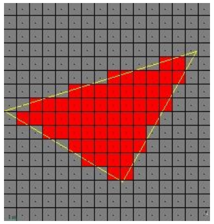
- A conceptual model that describes what steps a graphics system needs to perform to render a 3D scene to a 2D image.

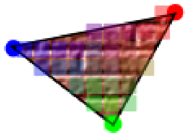
- Also known as **graphics pipeline**.

# Rendering Pipeline









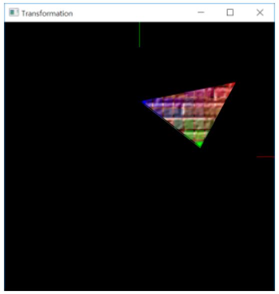
**vertex processing** → **rasterization** → **fragment processing** → output merging

**vertex processing**: performs a sequence of vertex transformations

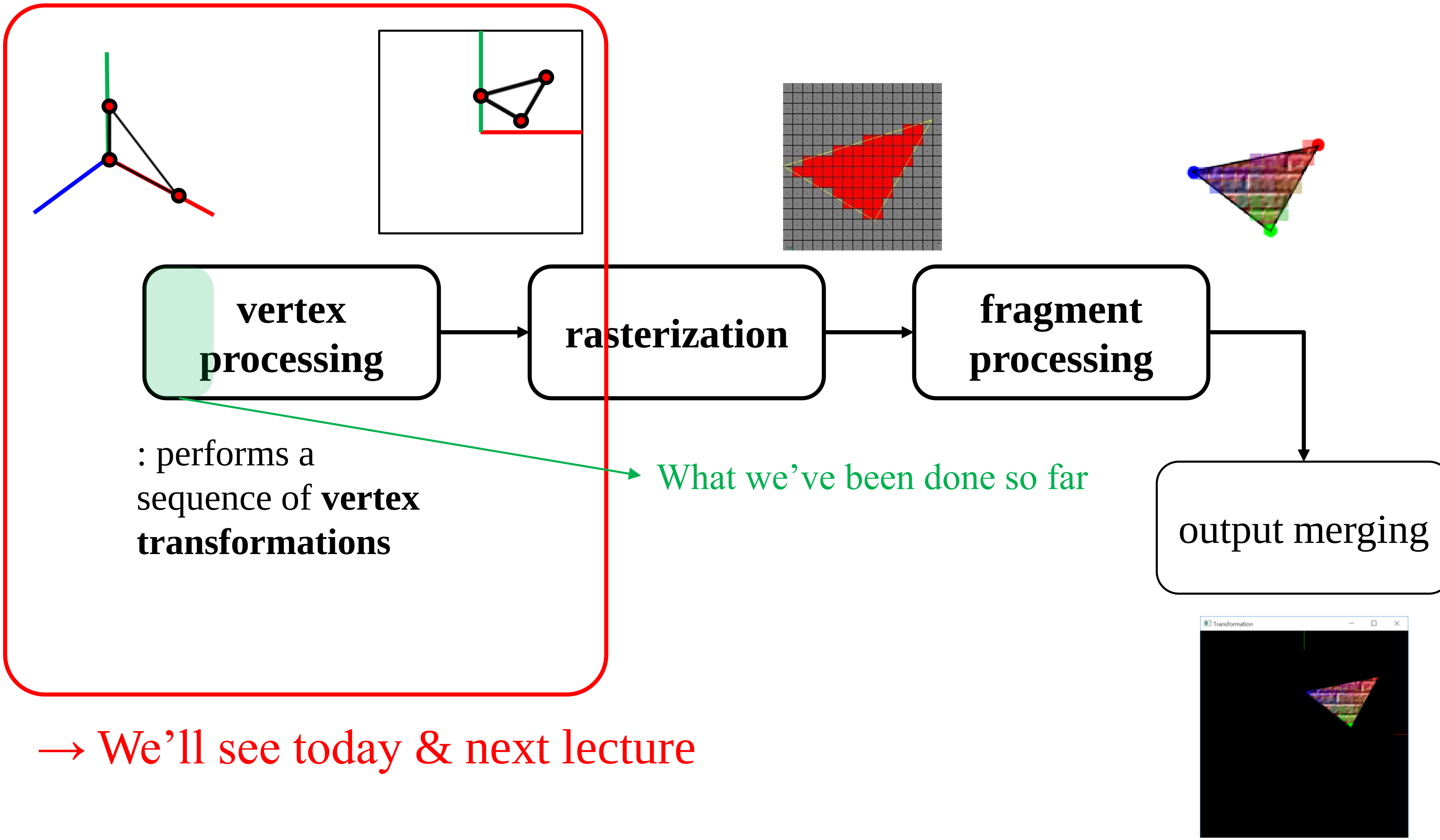
**rasterization**: assembles polygons & converts each polygon into a set of fragments (pixels)

**fragment processing**: determines color of each fragment with light & texture

output merging



# Rendering Pipeline




→ We'll see today & next lecture

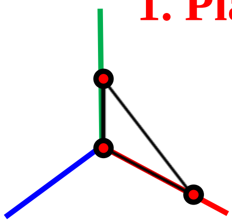
# Vertex Processing

*Set vertex positions*

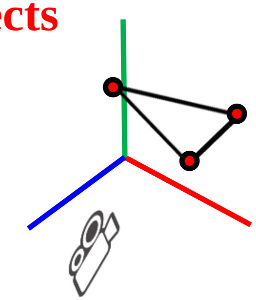
*Transformed vertices*

*Vertex positions in 2D viewport*

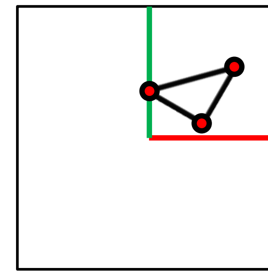
**1. Placing objects**



**M** →



**?**



**We have to somehow set the "camera" that is watching the "scene".**

glVertex3fv($\mathbf{p}_1$)
glVertex3fv($\mathbf{p}_2$)
glVertex3fv($\mathbf{p}_3$)

**glMultMatrixf($\mathbf{M}^T$)**
glVertex3fv($\mathbf{p}_1$)
glVertex3fv($\mathbf{p}_2$)
glVertex3fv($\mathbf{p}_3$)

…or

glVertex3fv($\mathbf{Mp}_1$)
glVertex3fv($\mathbf{Mp}_2$)
glVertex3fv($\mathbf{Mp}_3$)

**Then what we have to do are…**

**2. Placing the "camera"**
**3. Selecting a "lens"**
**4. Displaying on a "cinema screen"**

# In Terms of CG Transformation,

- 1. Placing objects

→ **Modeling transformation**

- 2. Placing the “camera”

→ **Viewing transformation**

- 3. Selecting a “lens”

→ **Projection transformation**

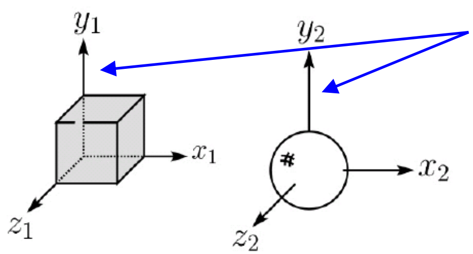
- 4. Displaying on a “cinema screen”

→ **Viewport transformation**

- All these transformations just work by **matrix multiplications**!

# Vertex Processing (Transformation Pipeline)

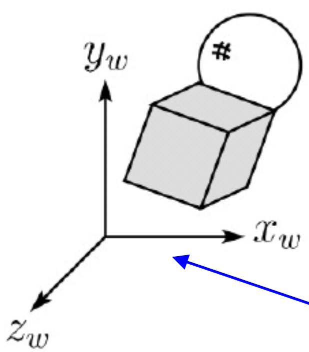
Object space




local coordinate system

Translate, scale, rotate, ... any affine transformations

**(What we've already covered in prev. lectures)**

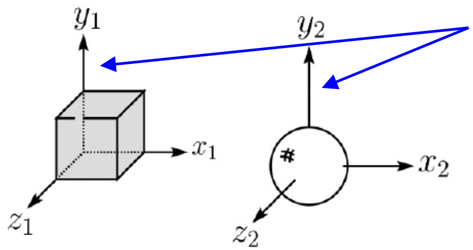



global coordinate system

World space

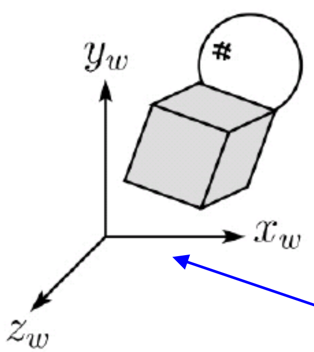
# Vertex Processing (Transformation Pipeline)

Object space




local coordinate system

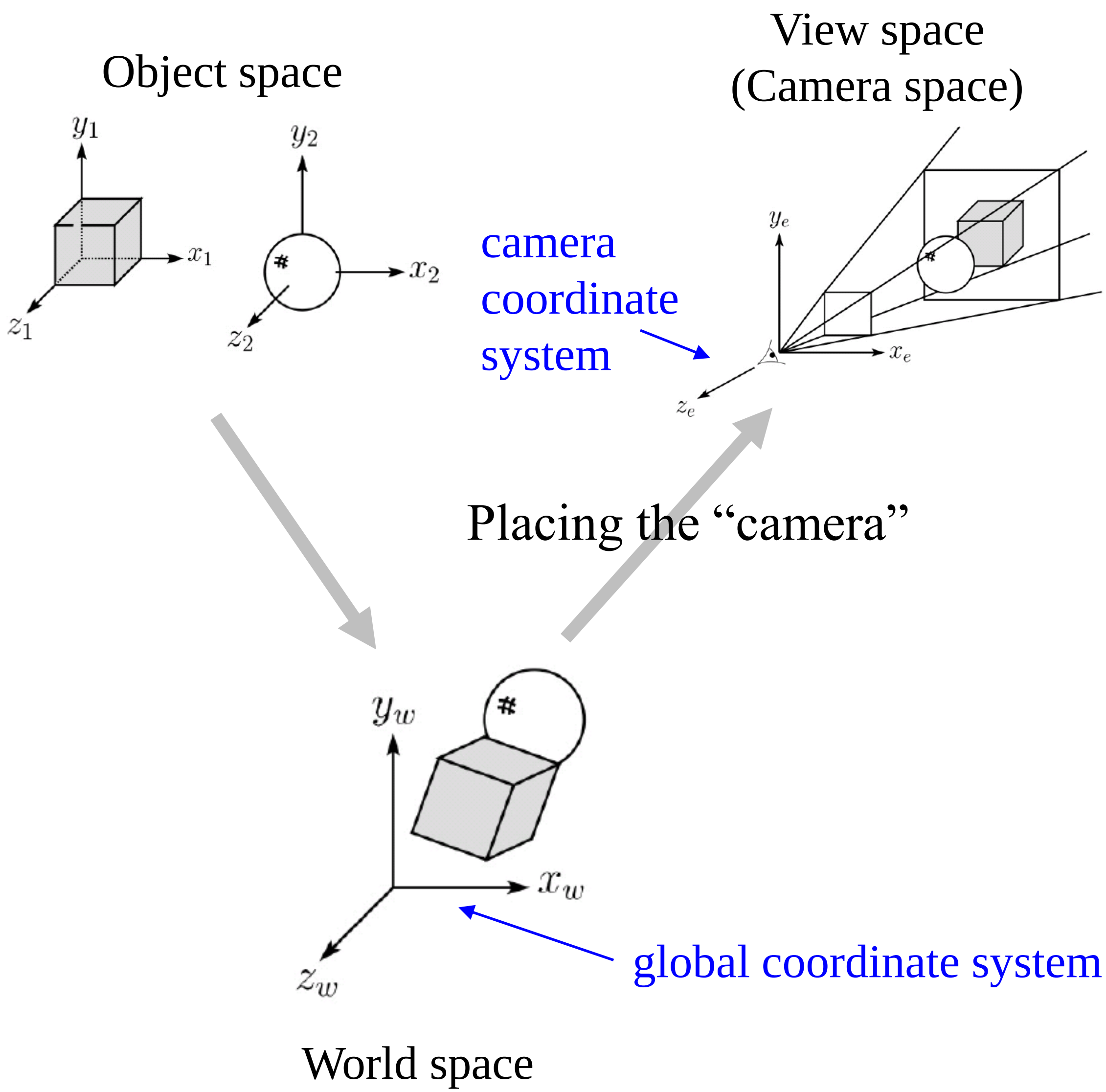
**Modeling transformation**



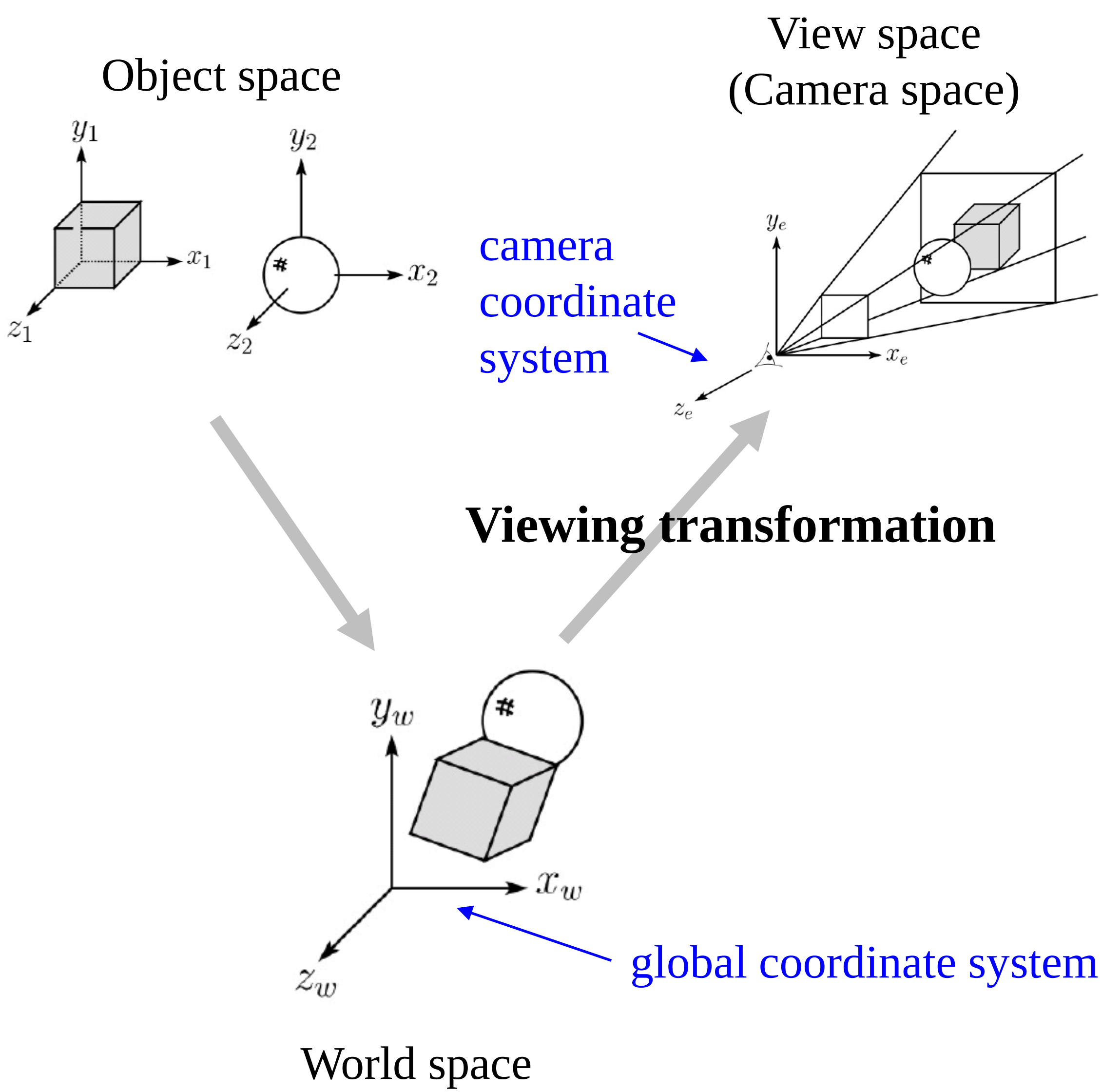

global coordinate system

World space

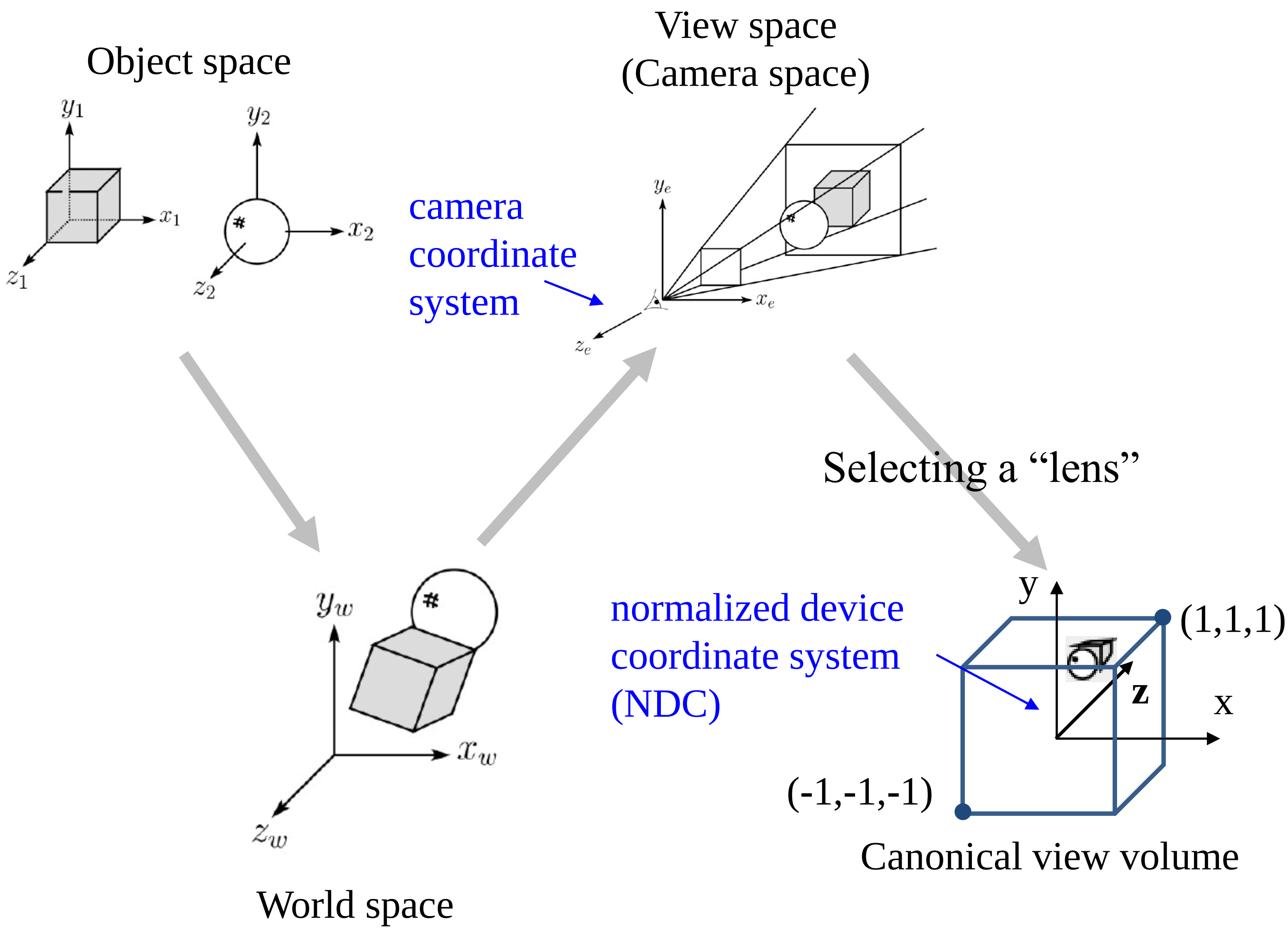
# Vertex Processing (Transformation Pipeline)



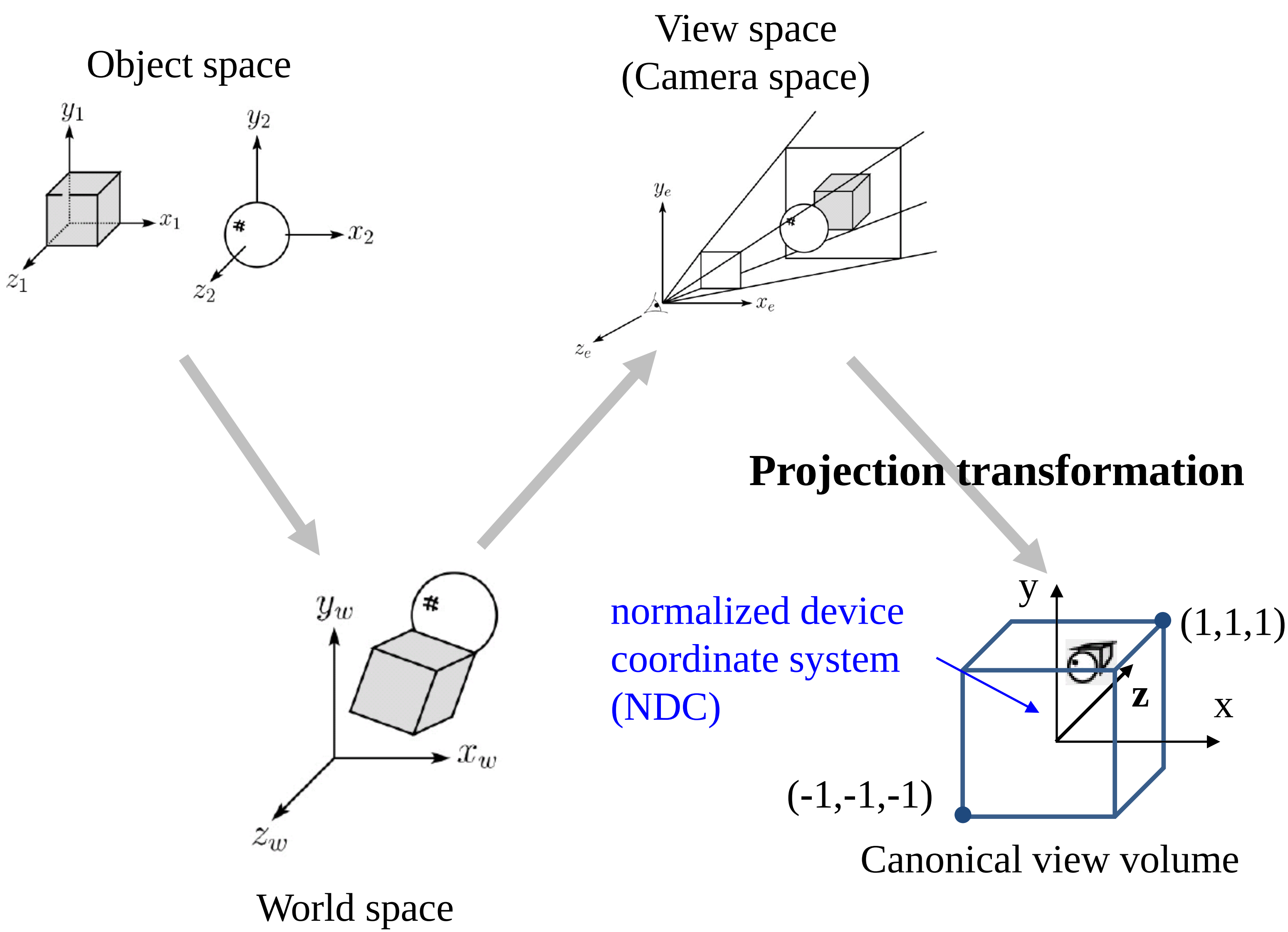
# Vertex Processing (Transformation Pipeline)



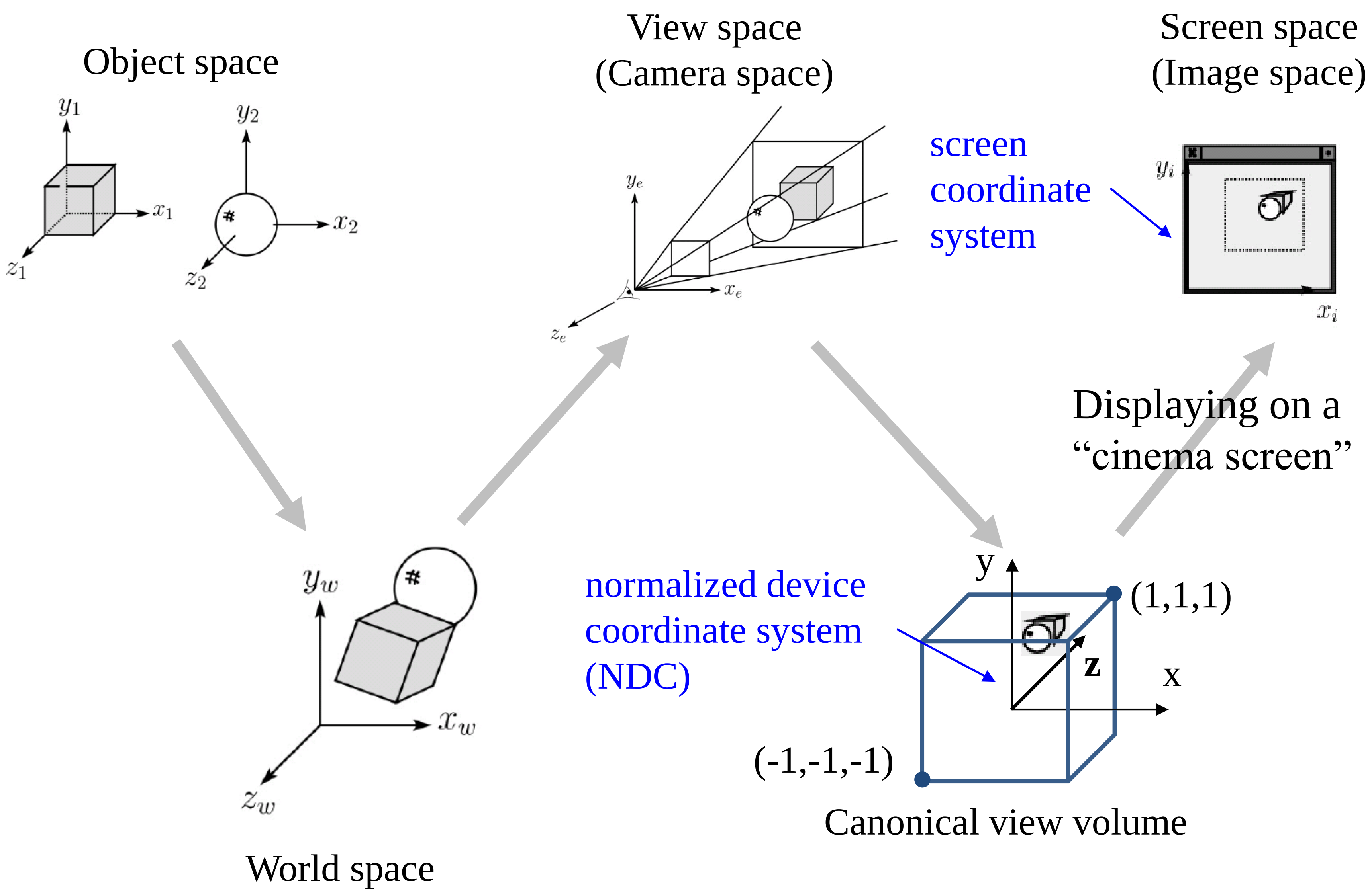
# Vertex Processing (Transformation Pipeline)



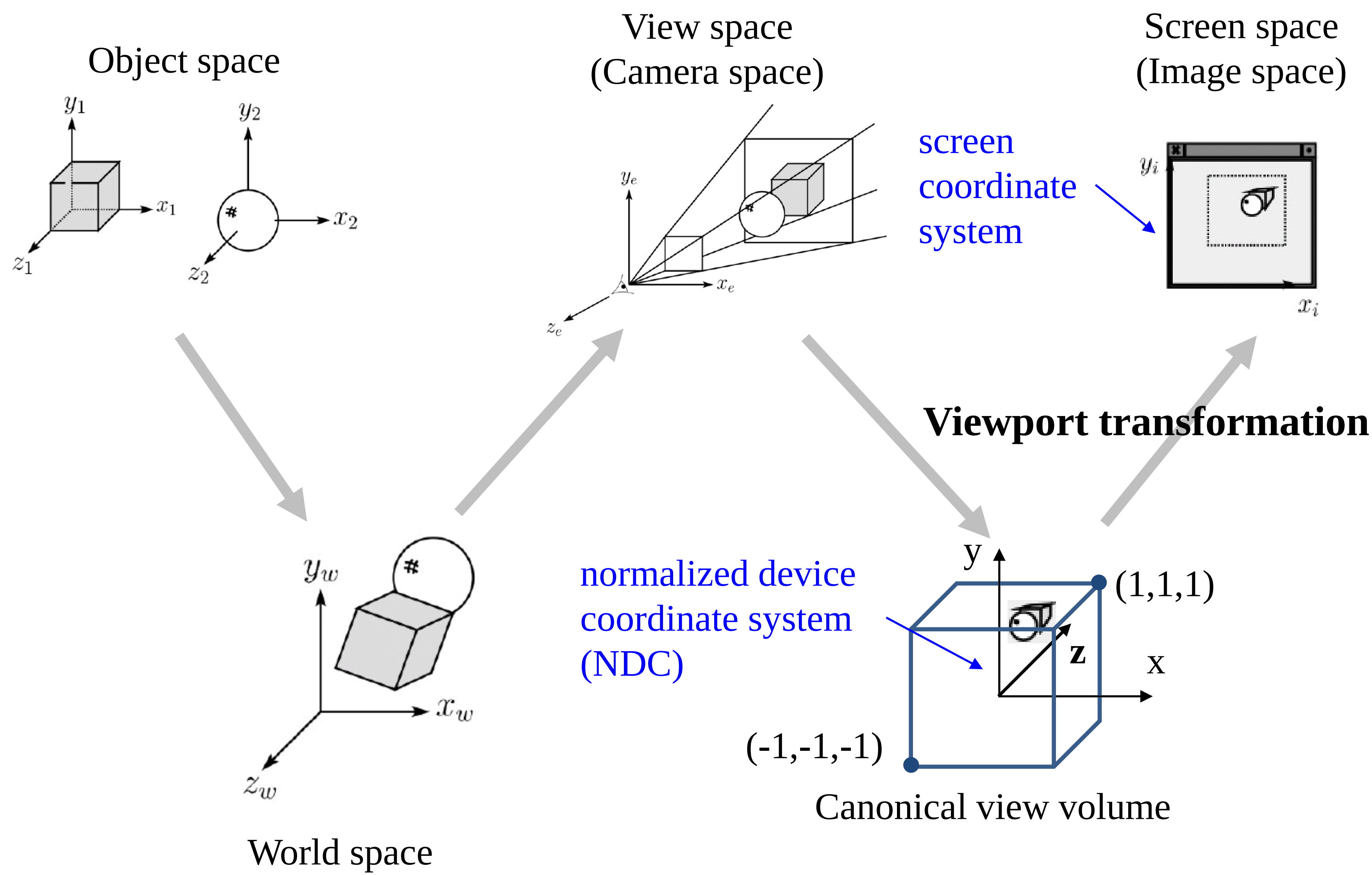
# Vertex Processing (Transformation Pipeline)



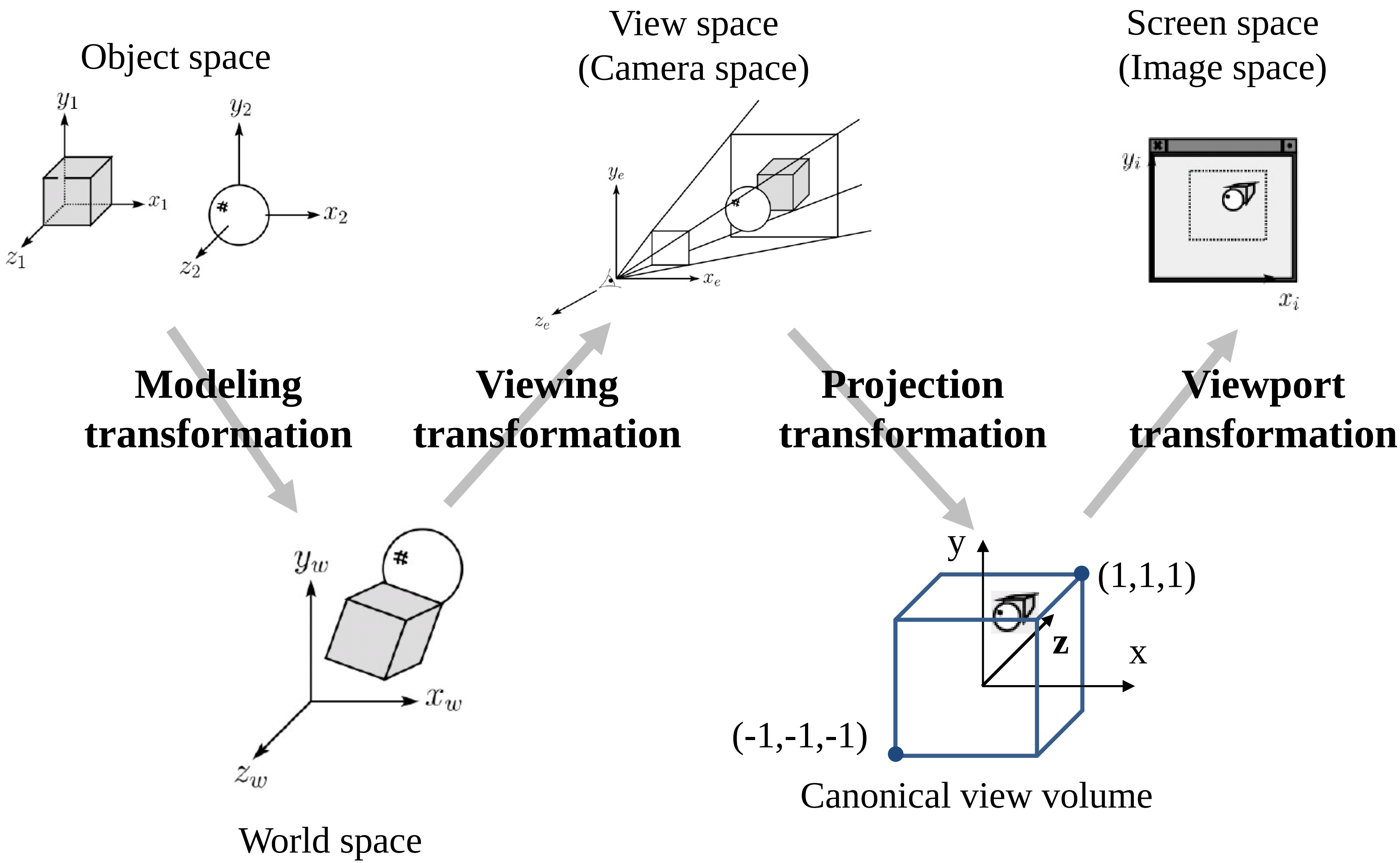
# Vertex Processing (Transformation Pipeline)



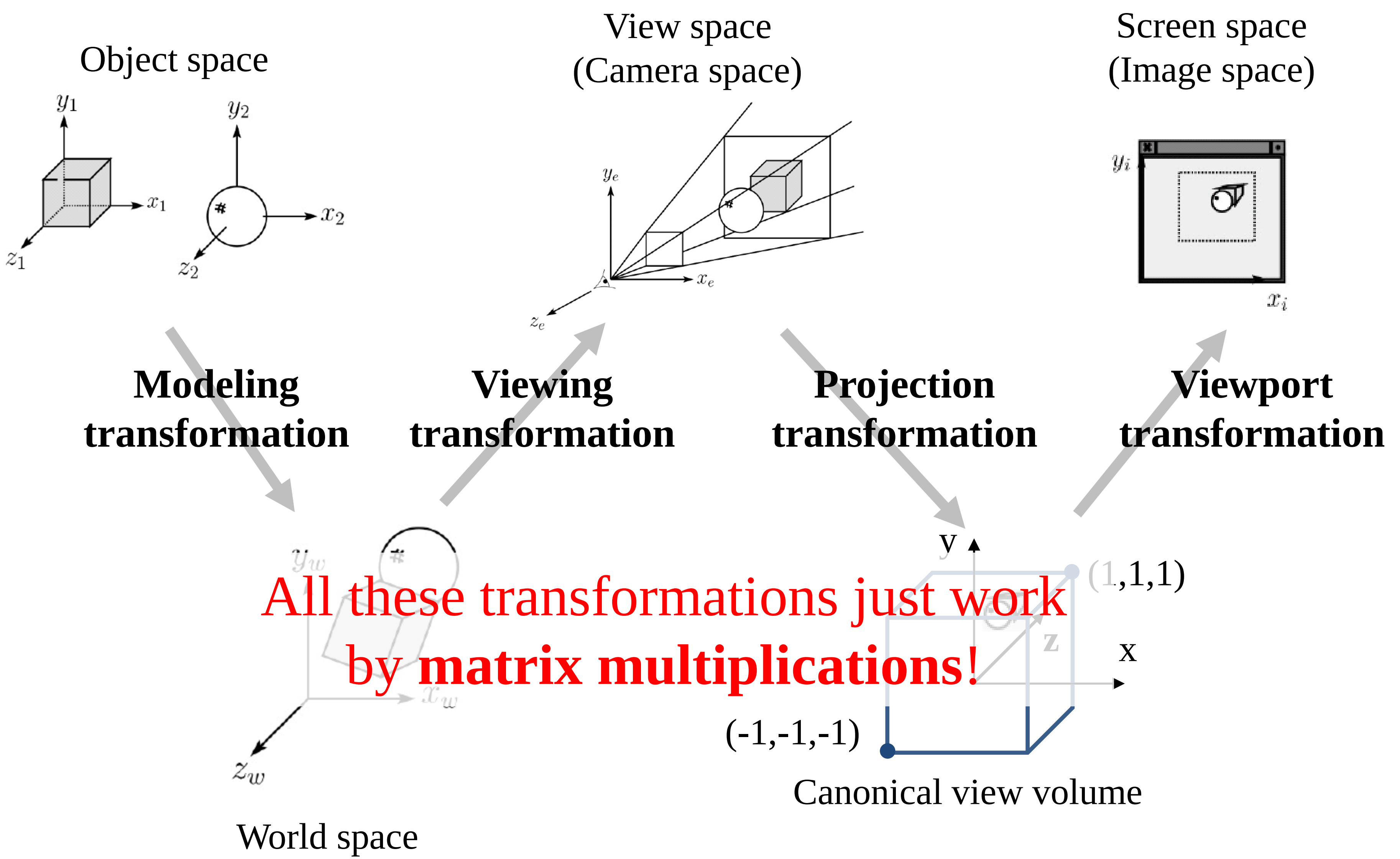
# Vertex Processing (Transformation Pipeline)



# Vertex Processing (Transformation Pipeline)

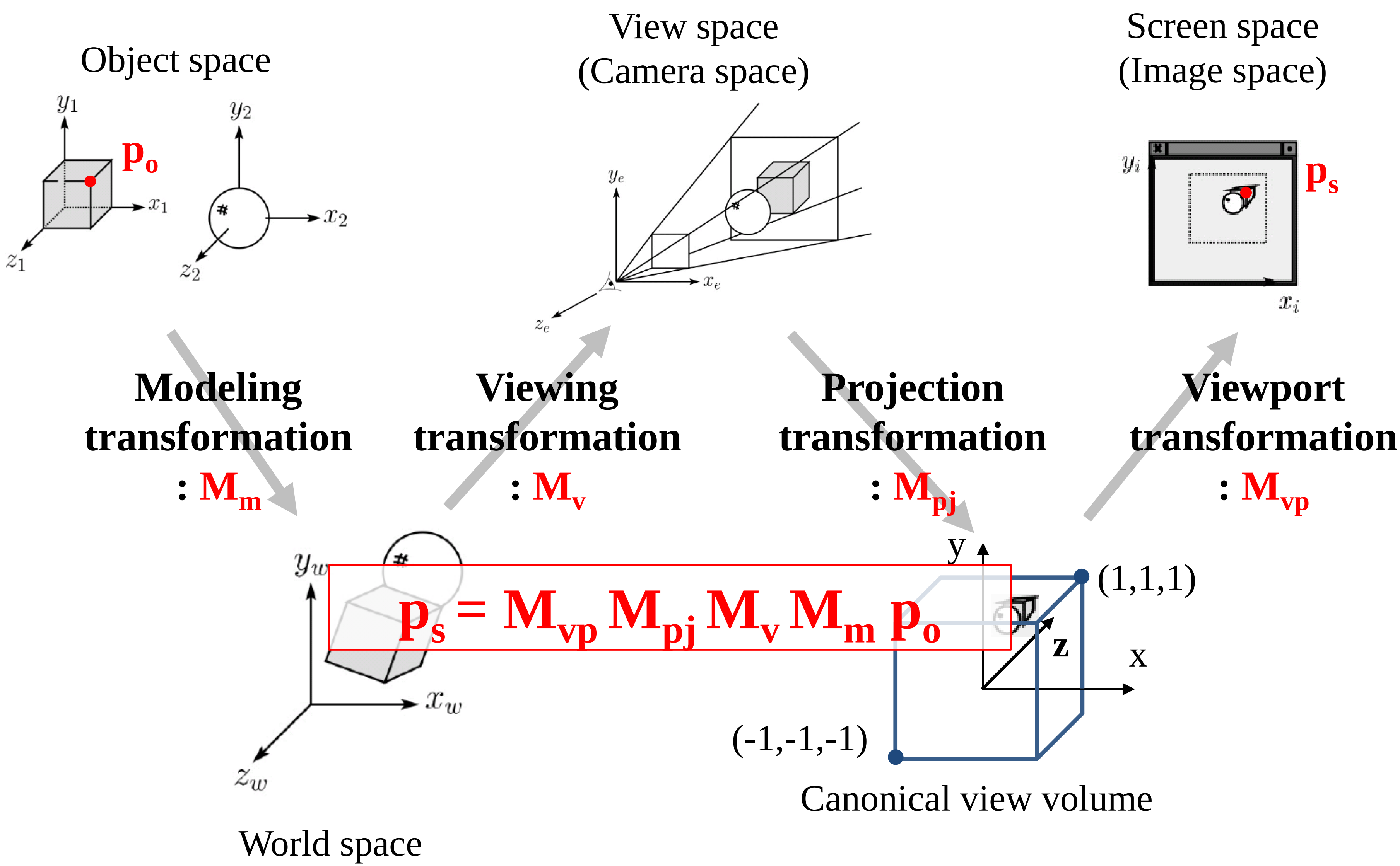


# Vertex Processing (Transformation Pipeline)

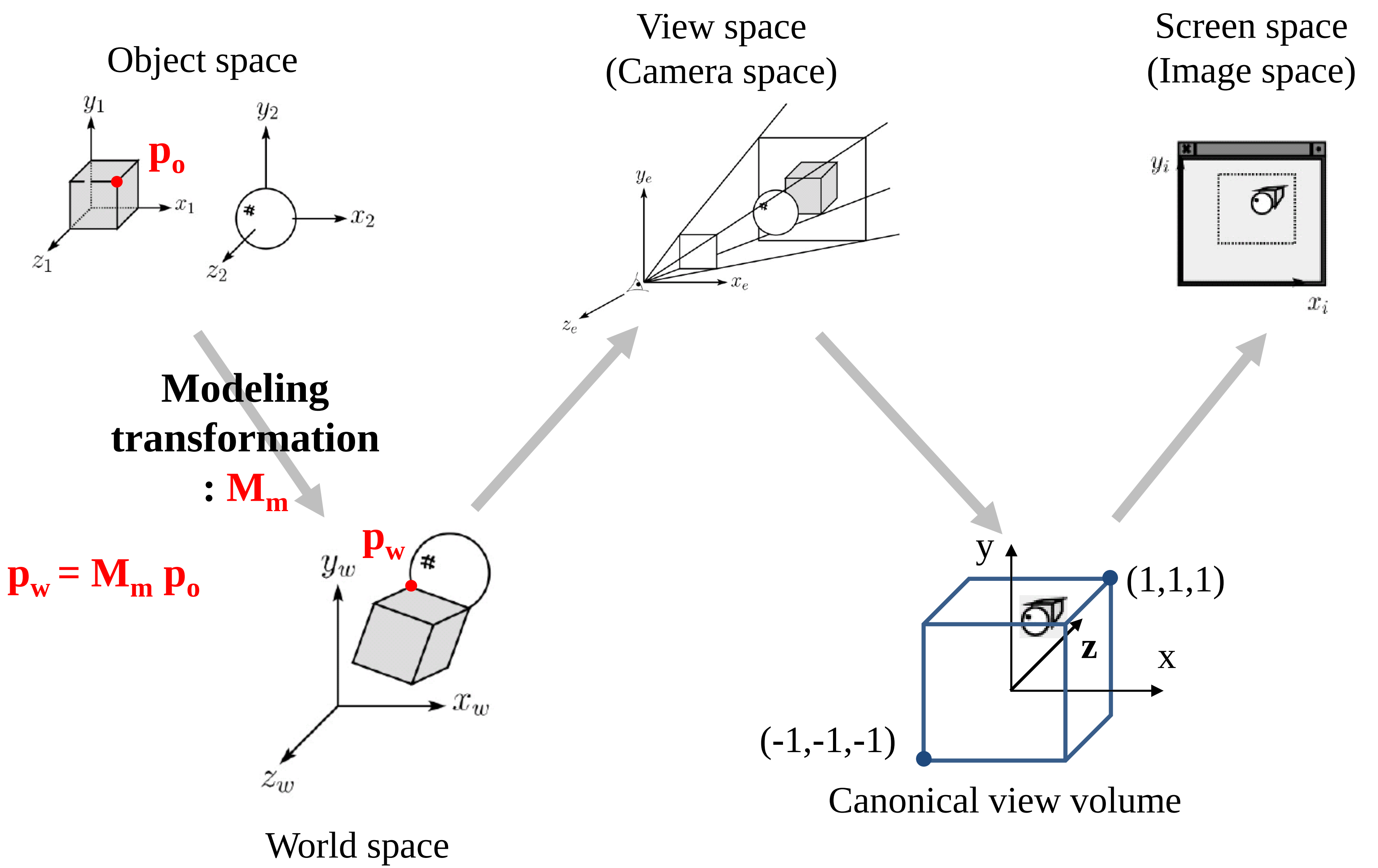



All these transformations just work by **matrix multiplications!**

# Vertex Processing (Transformation Pipeline)

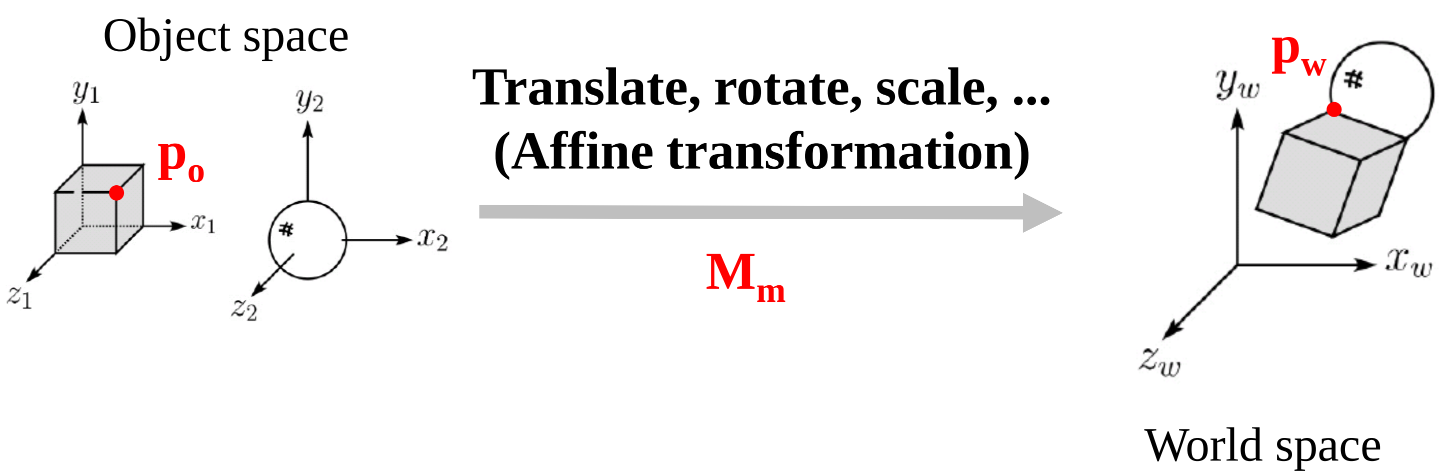


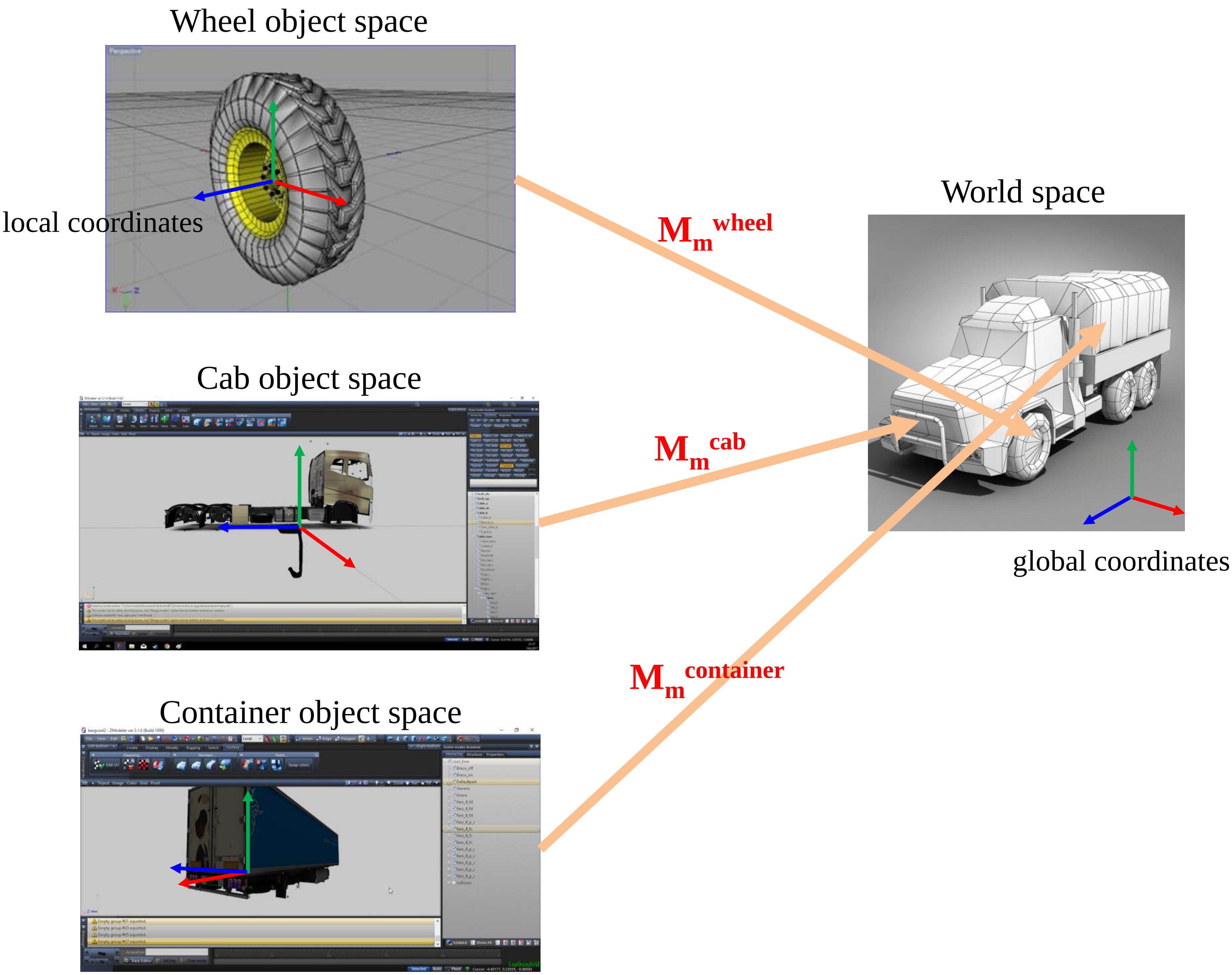
# Modeling Transformation



# Modeling Transformation

- Geometry would originally have been in the **object's local coordinates**.
- Transform into world coordinates is called the *modeling matrix*, $M_m$ .
- Composite affine transformations
- (What we've covered so far!)




Wheel object space
local coordinates
World space
$M_m^{wheel}$
Cab object space
$M_m^{cab}$
global coordinates
$M_m^{container}$
Container object space

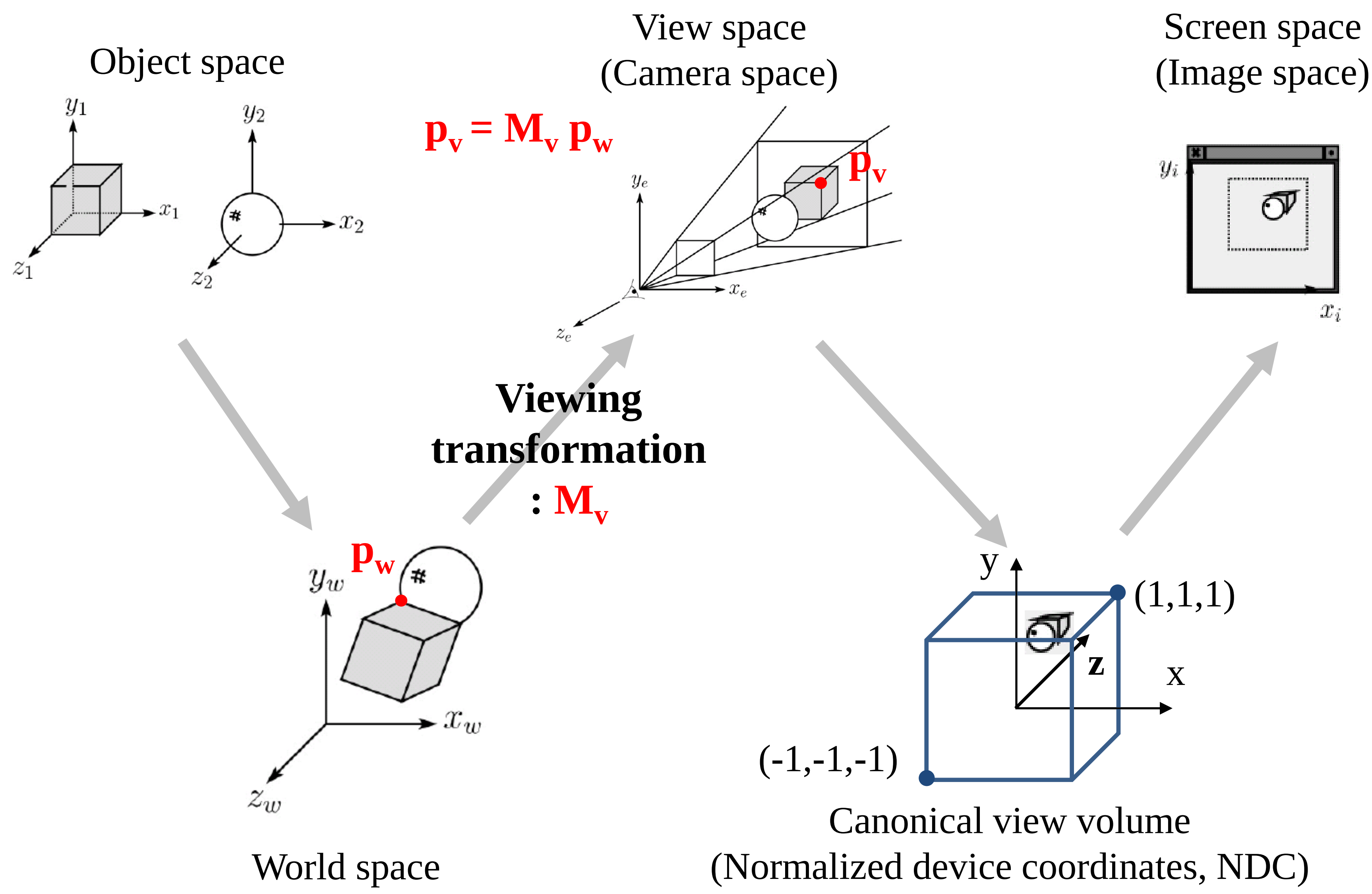
# Quiz #2

- 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”.

# Viewing Transformation



# Recall that...

- 1. Placing objects

→ **Modeling transformation**

- 2. Placing the "camera"

→ **Viewing transformation**

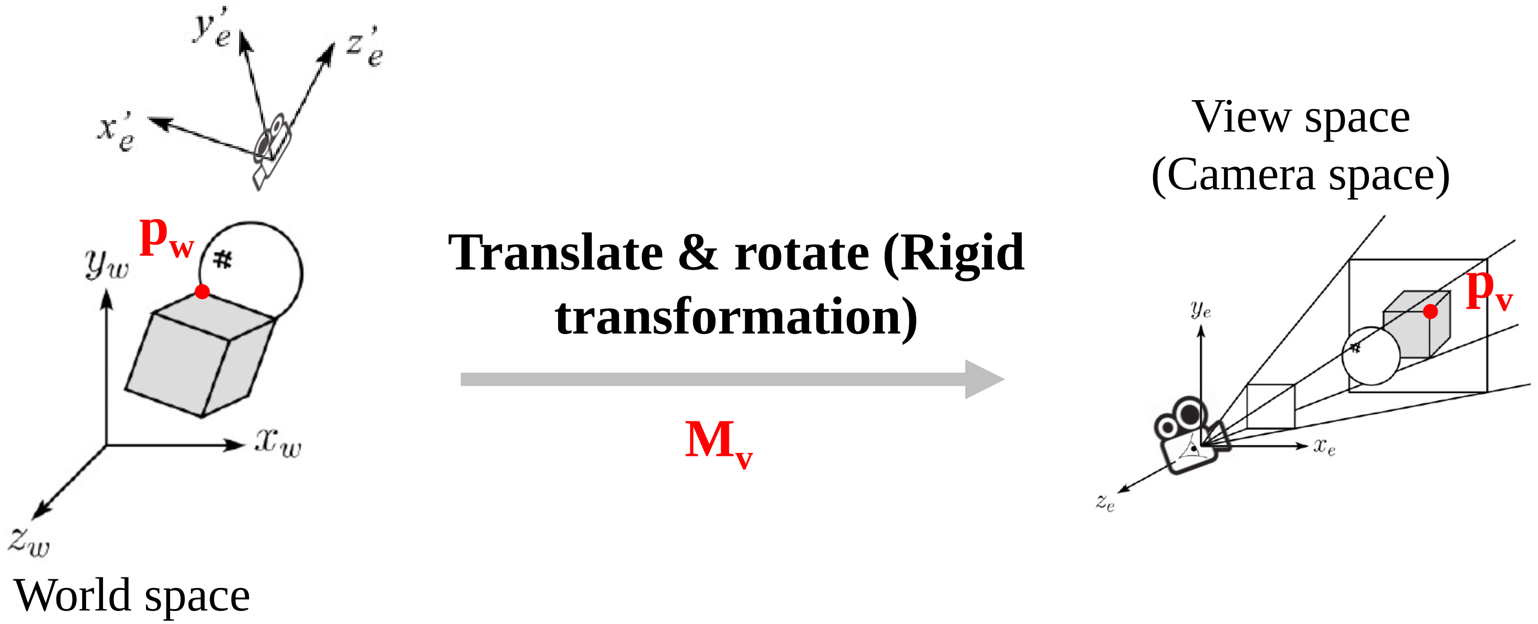
- 3. Selecting a "lens"

→ **Projection transformation**

- 4. Displaying on a "cinema screen"

→ **Viewport transformation**

# Viewing Transformation




- Transformation 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

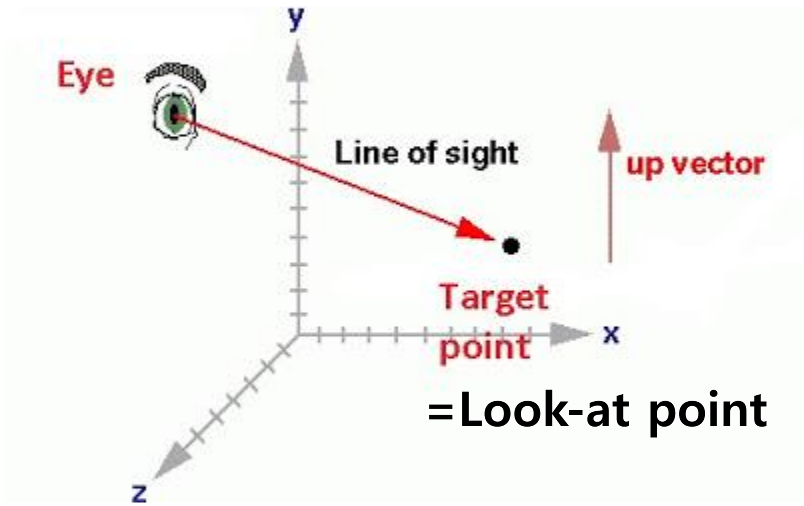
- Many ways to do this

- I'd like to introduce an intuitive way using:

- **Eye point**
  - Position of the camera

- **Look-at point**
  - The target of the camera

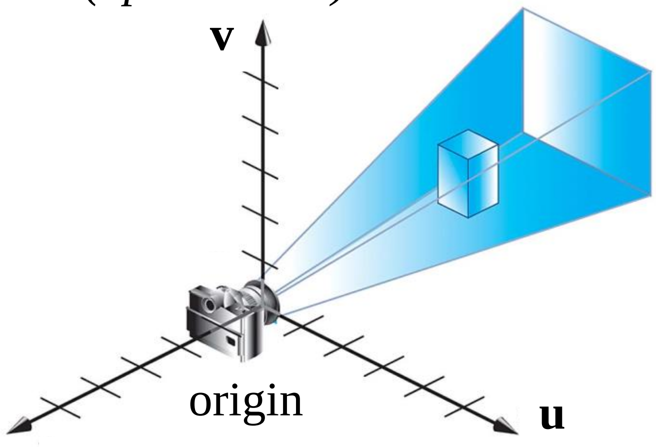
- **Up vector**
  - Roughly defines which direction is *up*



# 2. Defining Camera’s Coordinate System

- From the given **eye point**, **look-at point**, **up vector**, we can compute the **camera frame**.

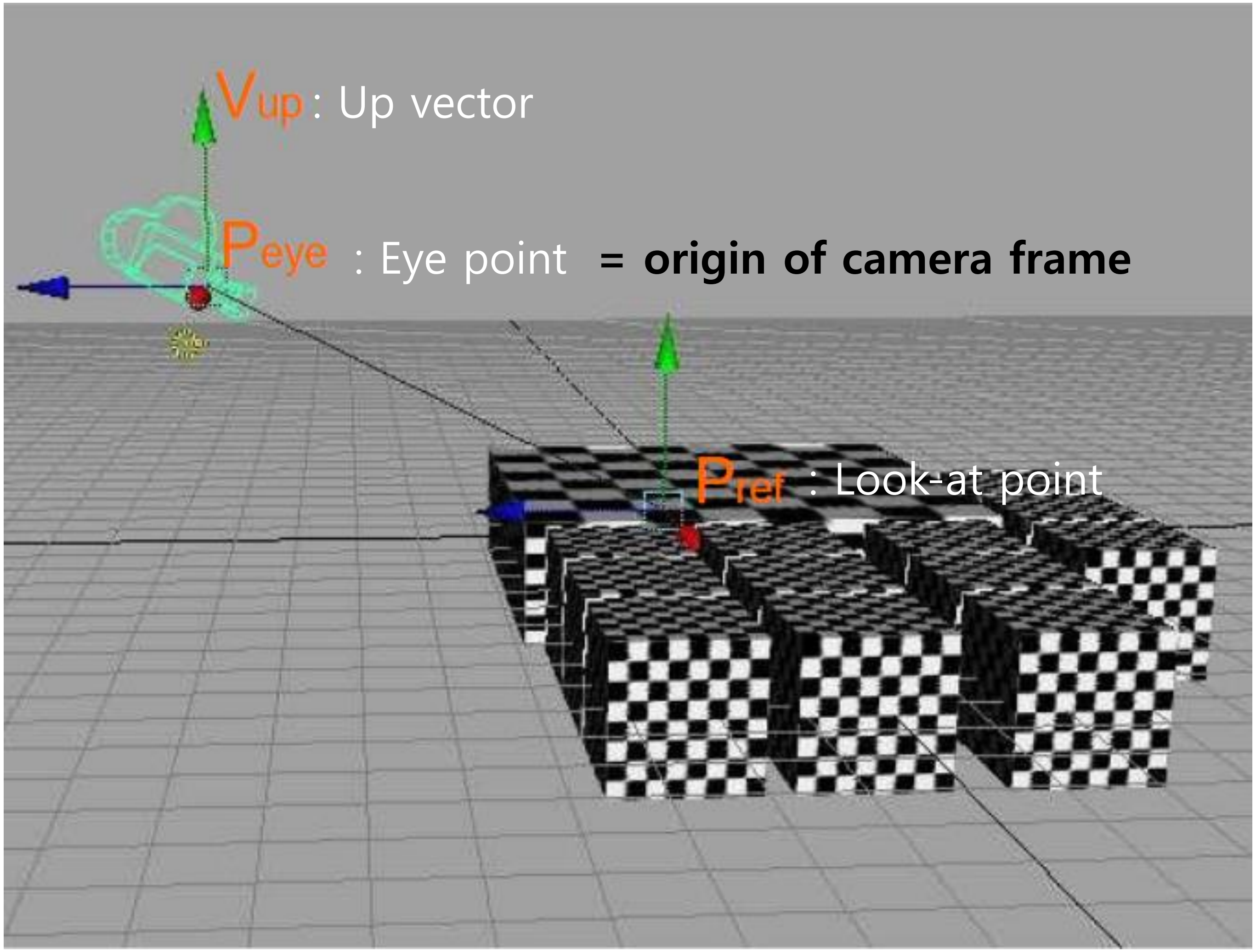
- **u, v, w** are commonly used for camera coordinates axes instead of x, y, z.




- What we have to do is to define the coordinate system:

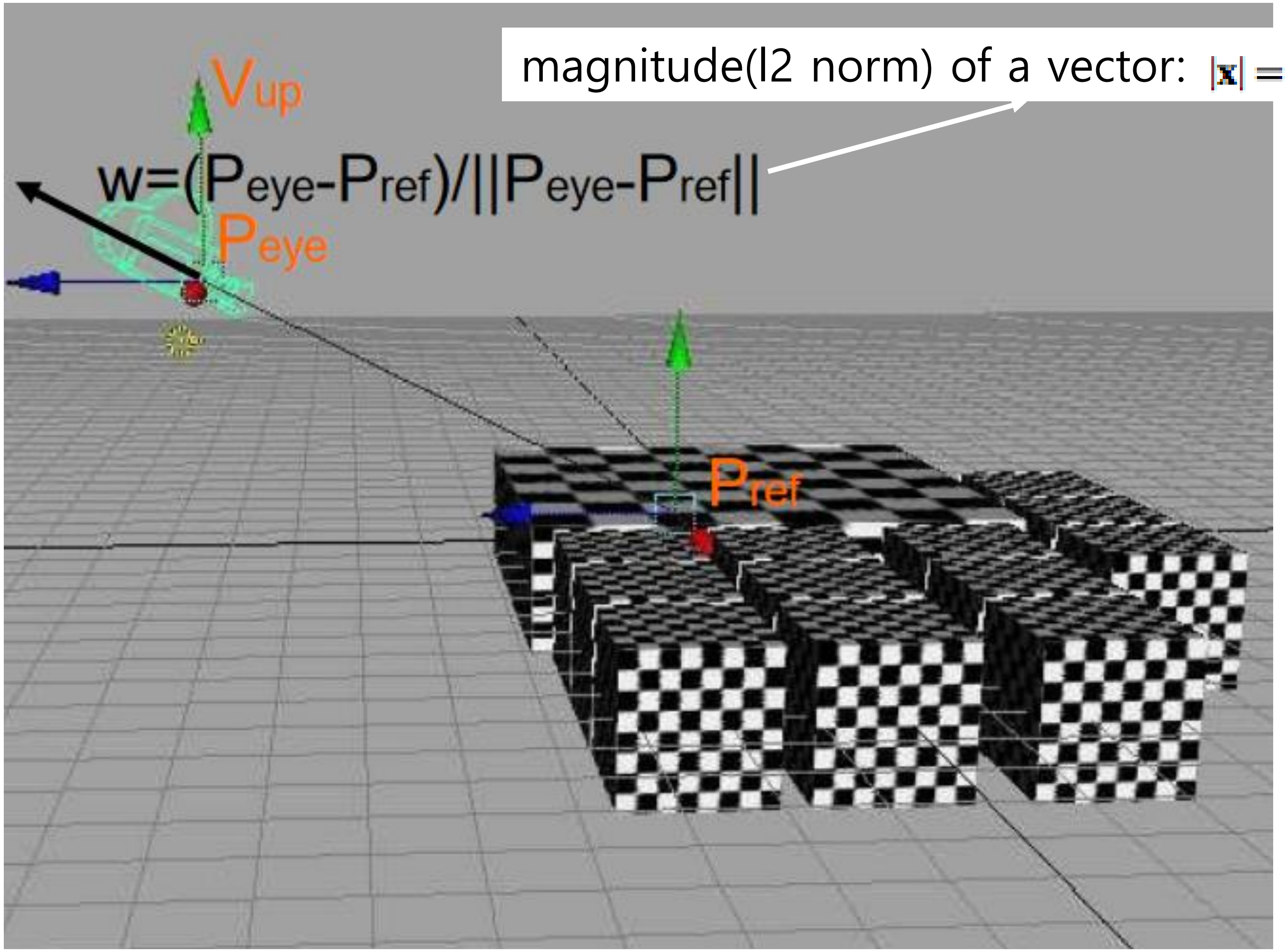
- Finding **u, v, w** vectors
- Finding the **origin** point

# Given Eye point, Look-at point, Up vector,

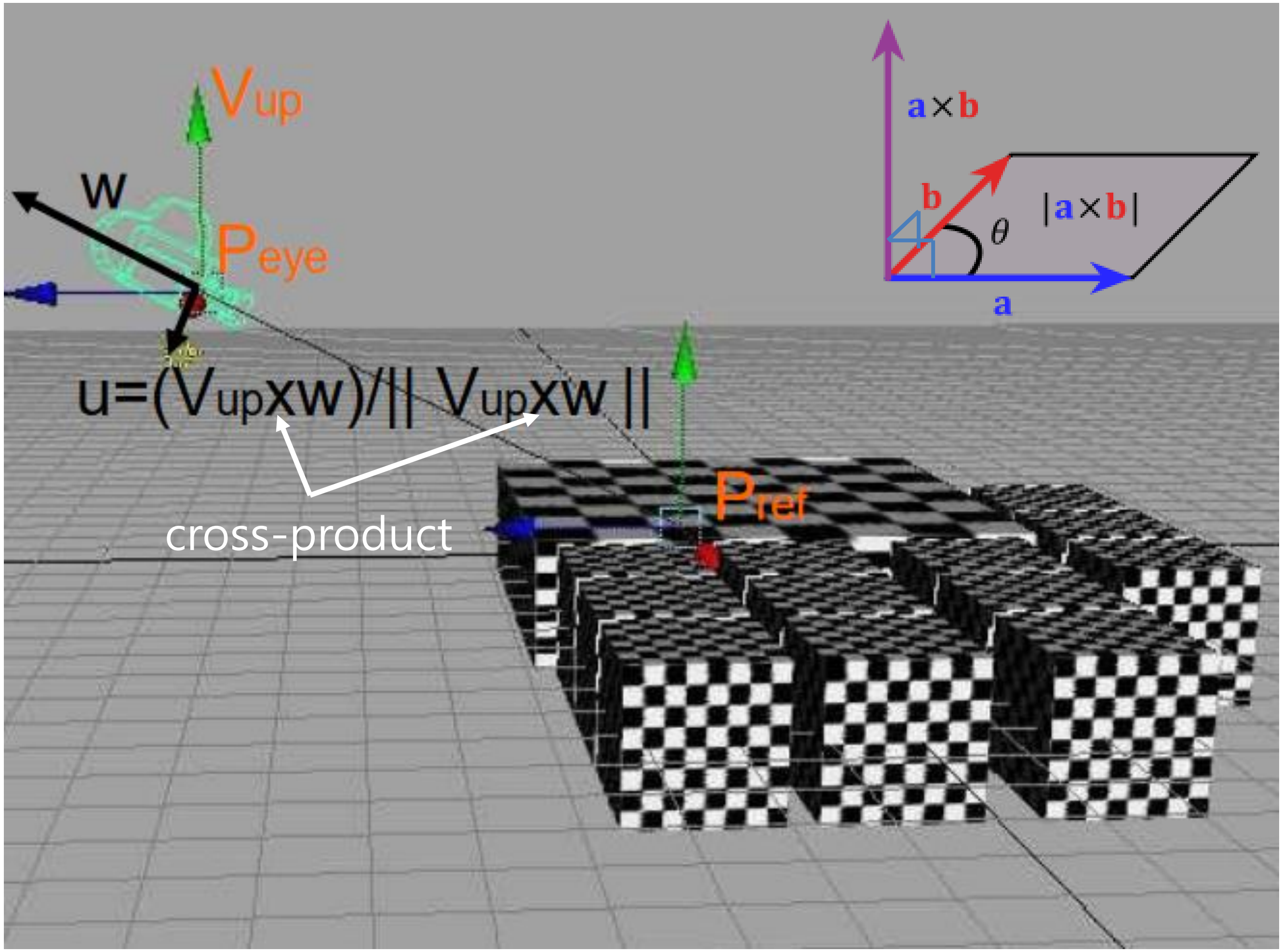


# Getting “w” axis vector

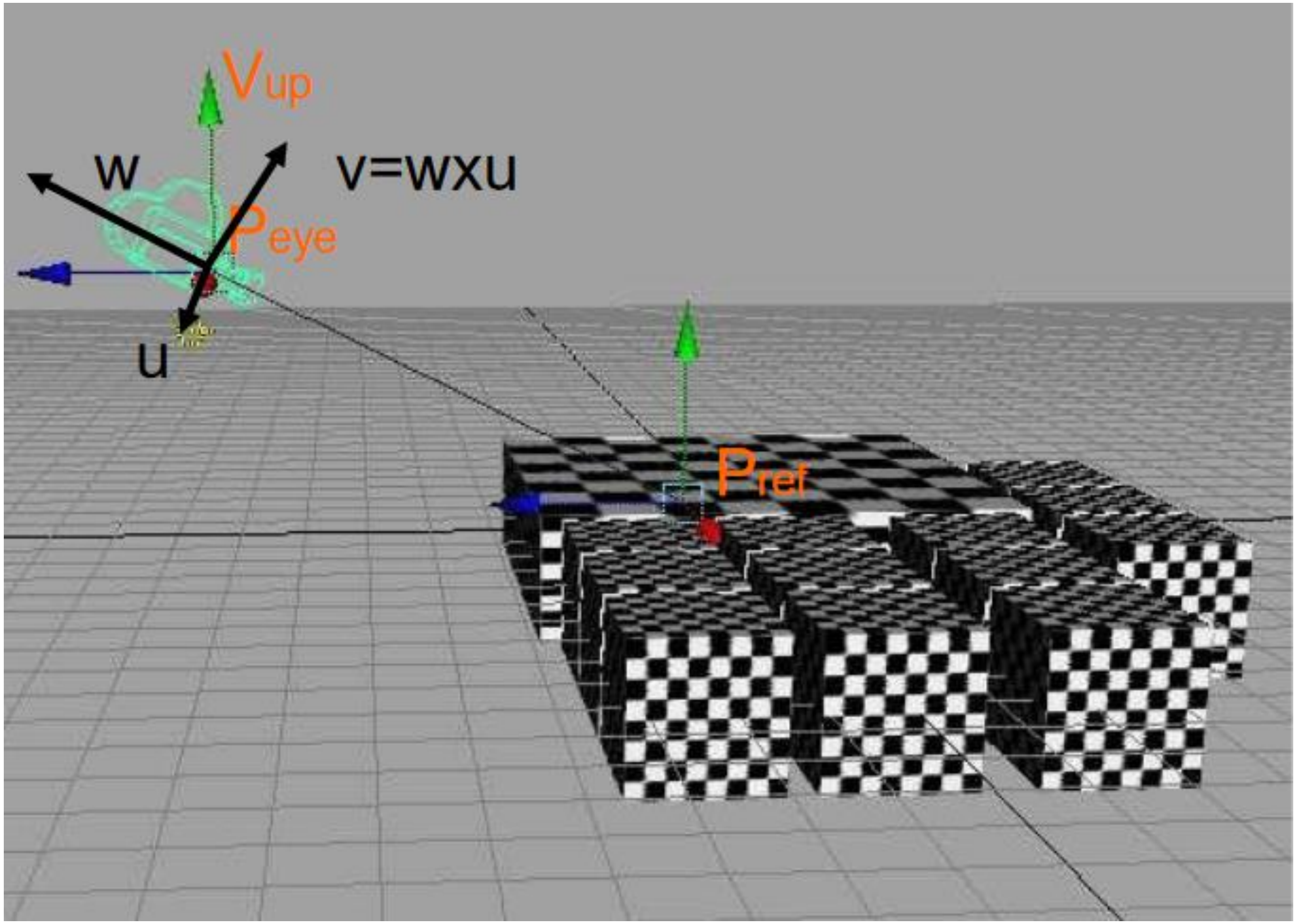
magnitude(l2 norm) of a vector: $|\mathbf{x}| = \sqrt{x_1^2 + x_2^2 + x_3^2}$ .



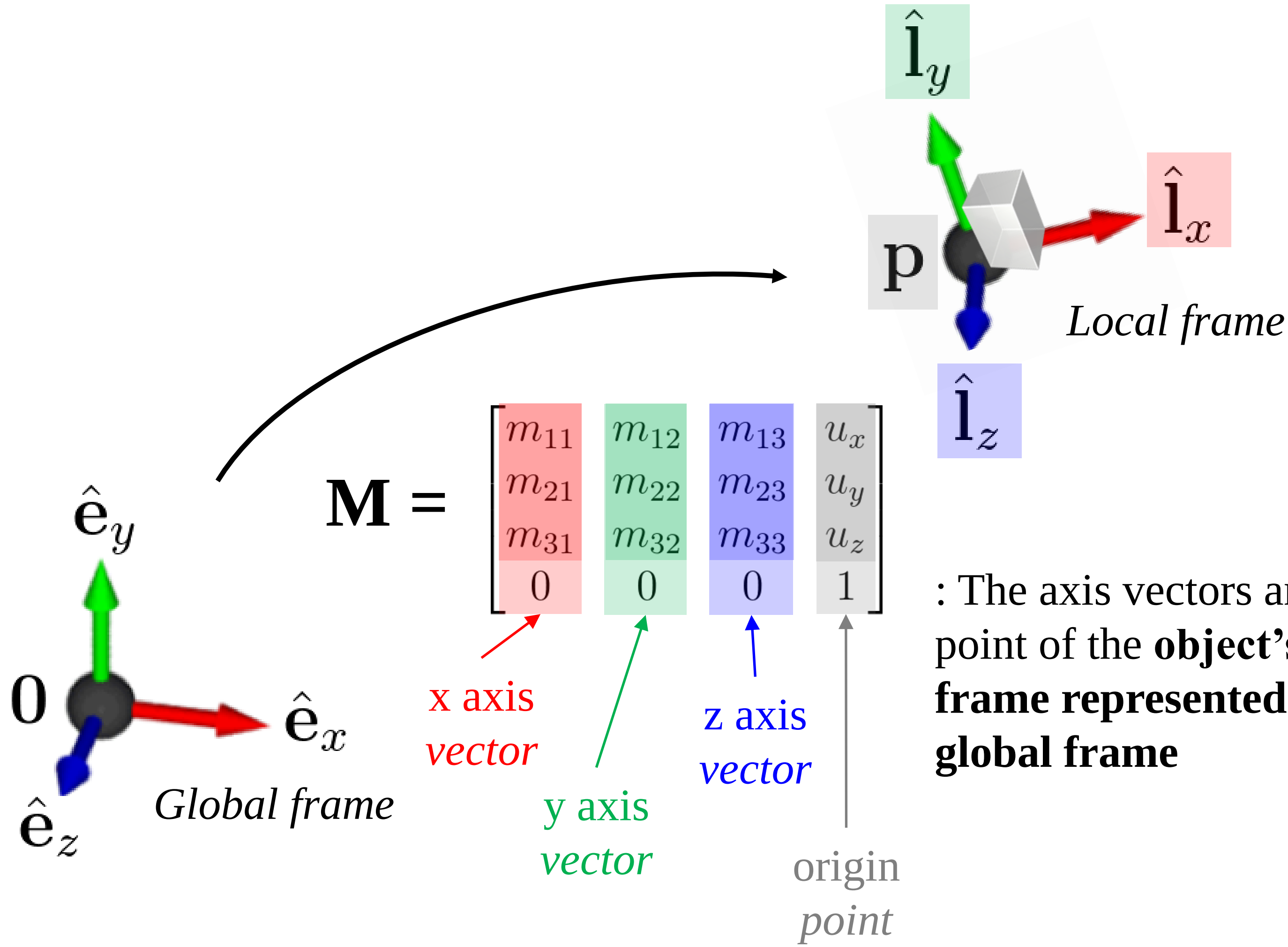
# Getting “u” axis vector



# Getting “v” axis vector

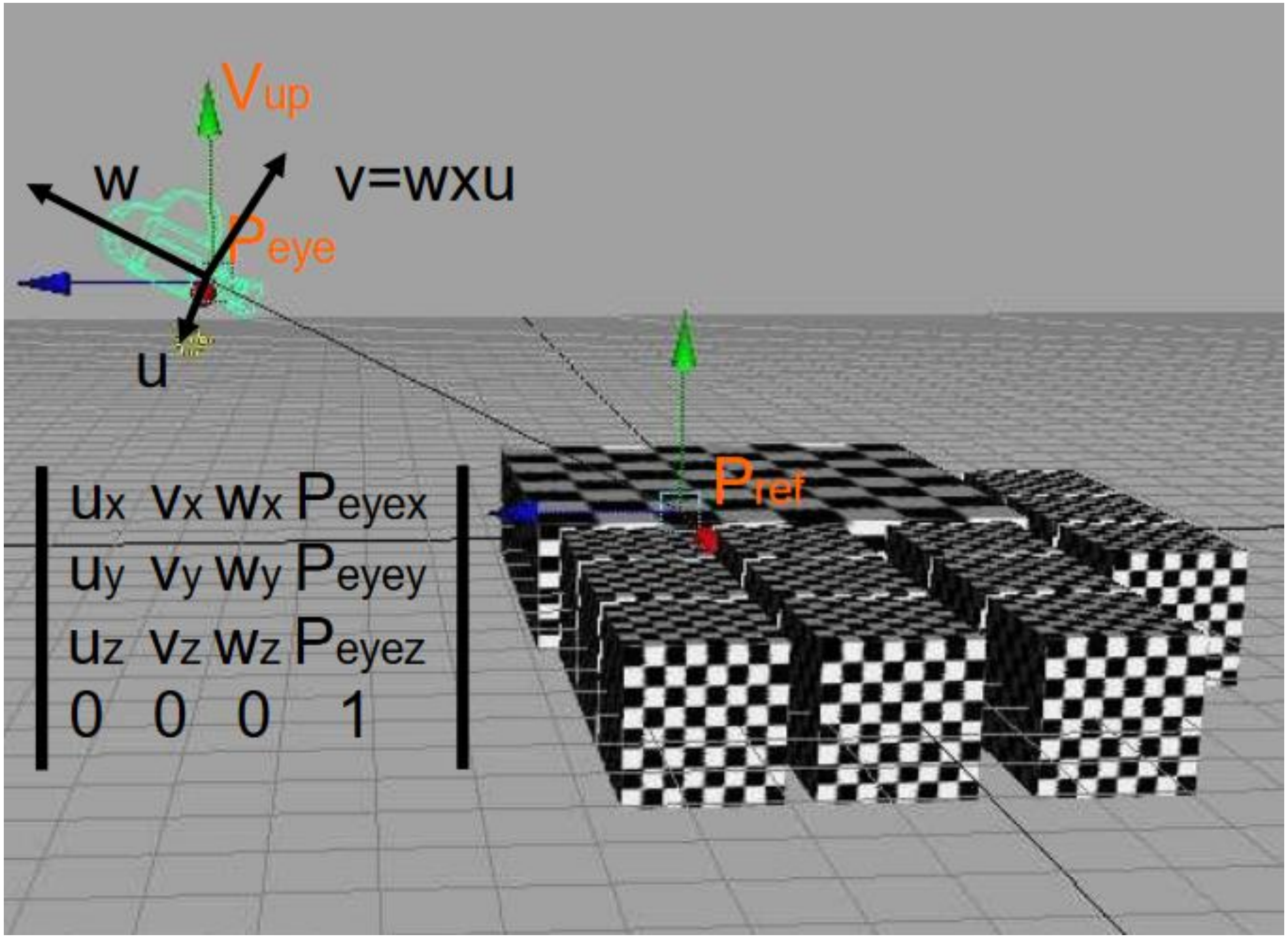


# 2) Affine Transformation Matrix **defines an Affine Frame w.r.t. Global Frame**



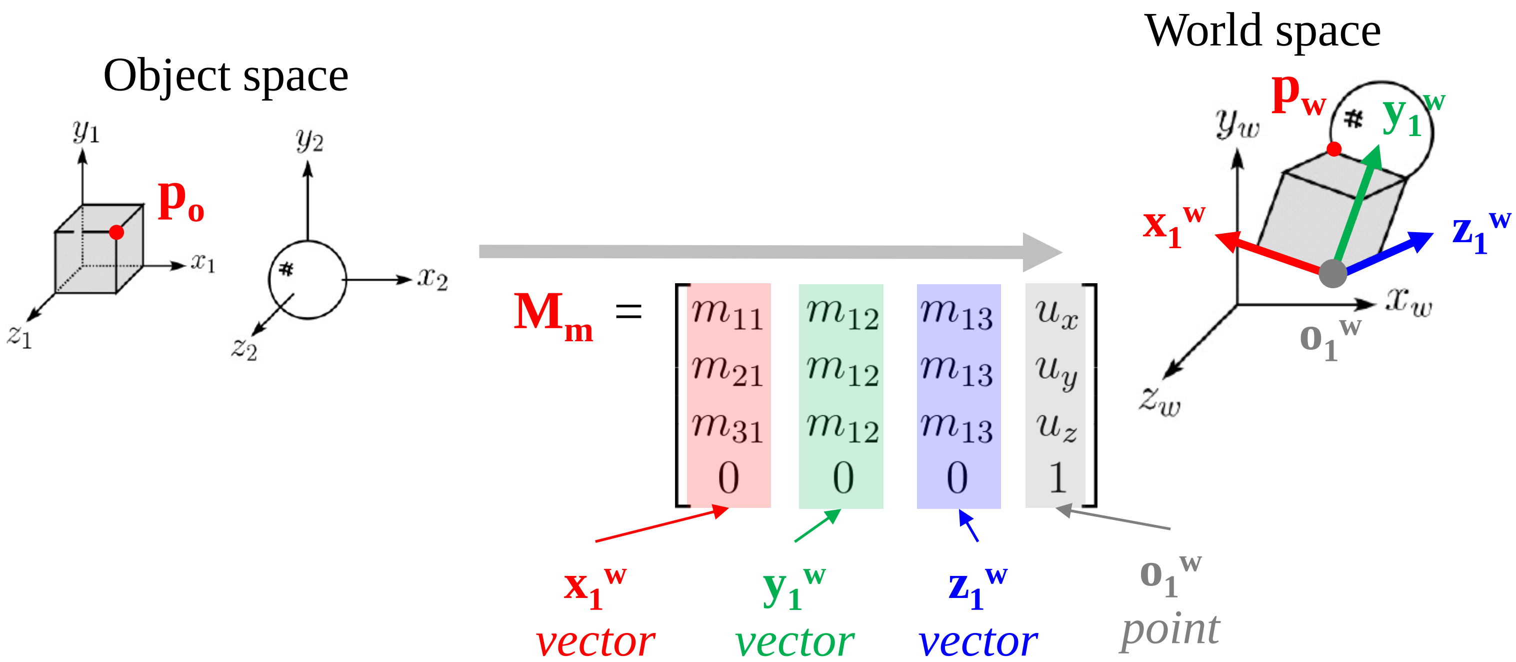

: The axis vectors and origin point of the **object's local frame represented in the global frame**

# Thus, the Camera Frame is defined by



# How can we get viewing matrix $M_v$ from this camera frame?

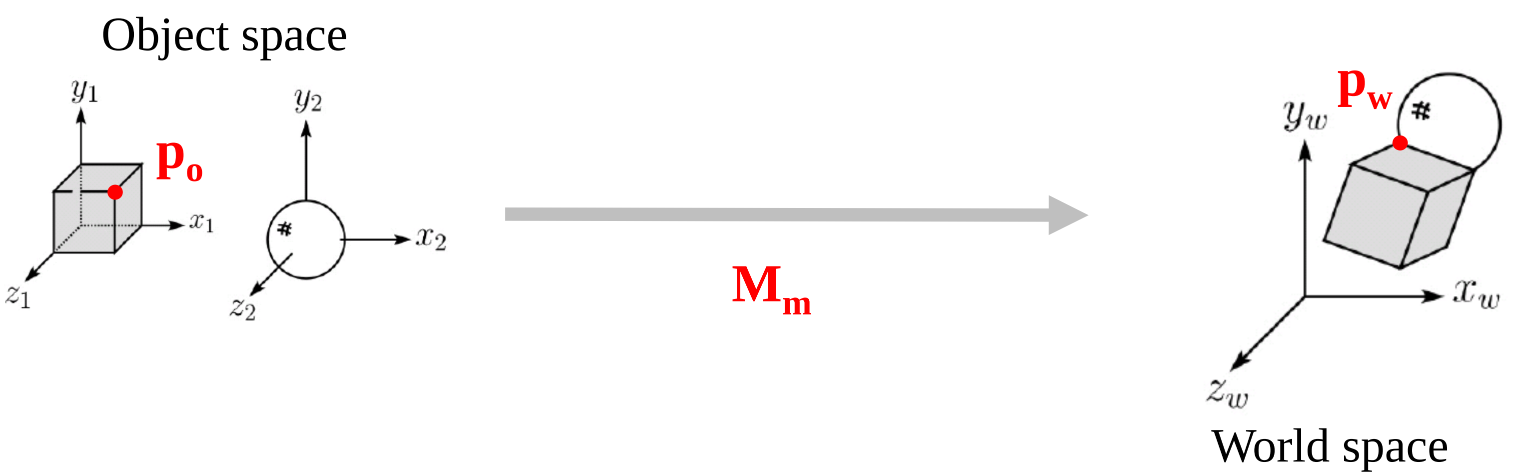
- Recall the modeling transformation:




: The axis vectors and origin point of the **object's local frame represented in the global frame**

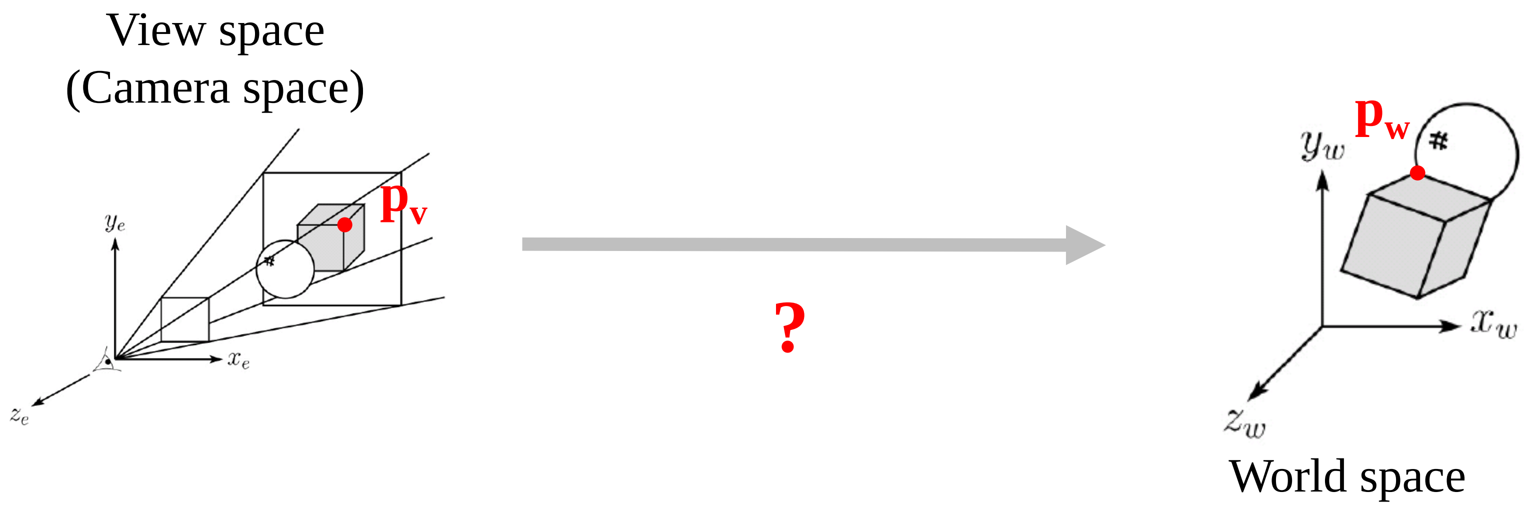
# How can we get viewing matrix $M_v$ from the camera frame?

- If we replace *object space* to *camera space*, what should be the transformation matrix?



# How can we get viewing matrix $M_v$ from the camera frame?

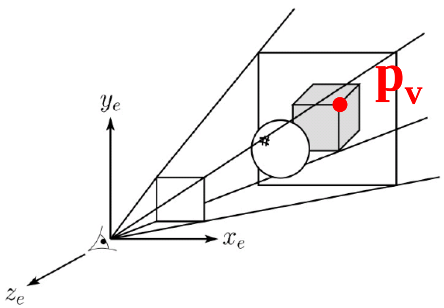
- If we replace *object space* to *camera space*, what should be the transformation matrix?



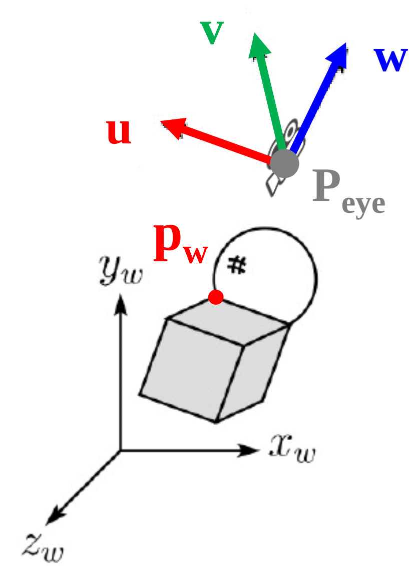
# How can we get viewing matrix $M_v$ from the camera frame?

- If we replace *object space* to *camera space*, what should be the transformation matrix?

View space
(Camera space)




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

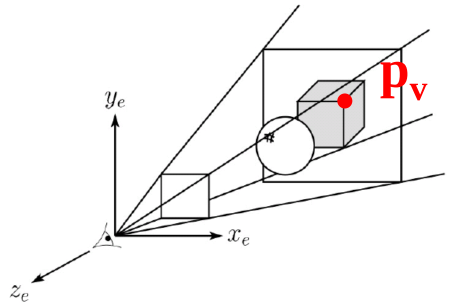



World space

: The axis vectors and origin point of the **camera frame represented in the global frame**

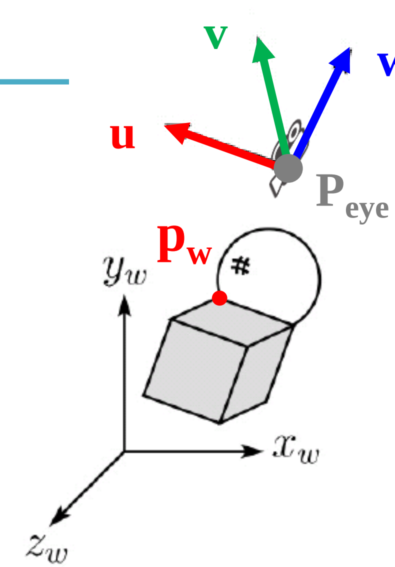
# Viewing Transformation is the Opposite Direction

View space
(Camera space)




$\mathbf{M_v}$

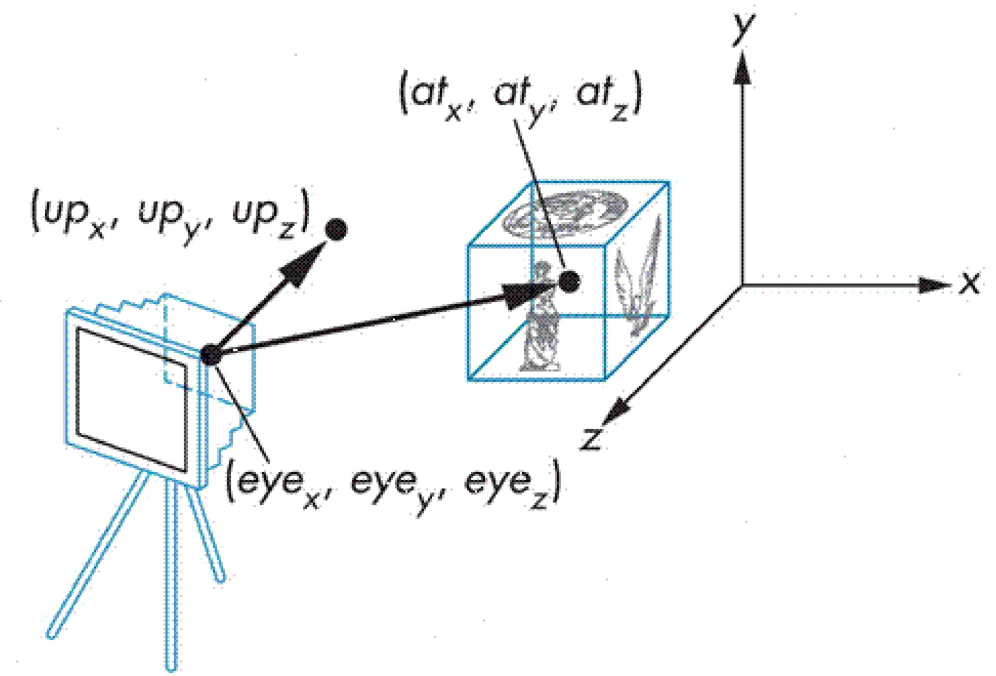
$$\begin{bmatrix} u_x & v_x & w_x & P_{eyex} \\ u_y & v_y & w_y & P_{eyey} \\ u_z & v_z & w_z & P_{eyez} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$




World space

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

# 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()

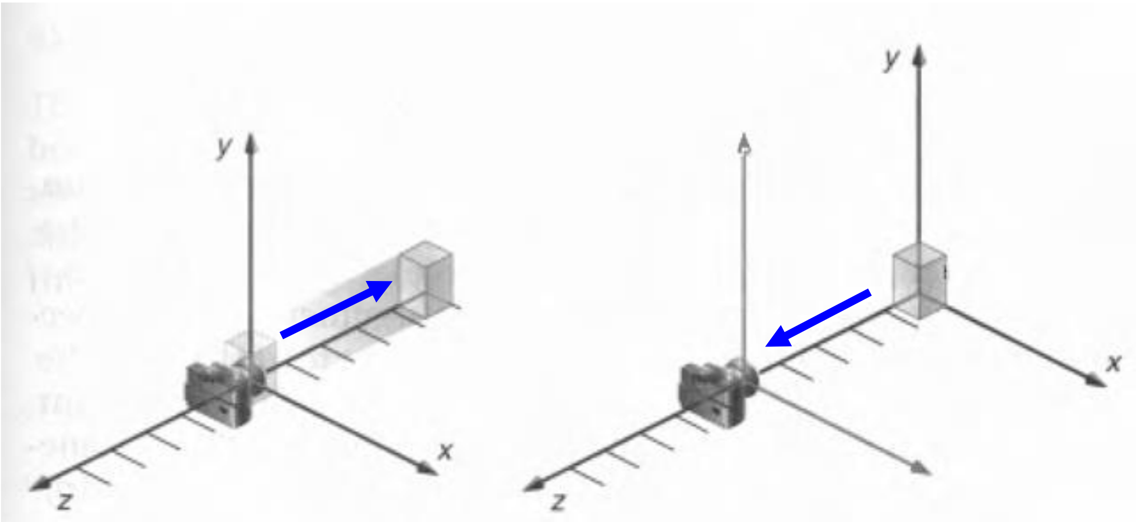
```
import glfw
from OpenGL.GL import *
from OpenGL.GLU import *
import numpy as np

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()
```

```
def drawFrame():
    glBegin(GL_LINES)
    glColor3ub(255, 0, 0)
    glVertex3fv(np.array([0.,0.,0.]))
    glVertex3fv(np.array([1.,0.,0.]))
    glColor3ub(0, 255, 0)
    glVertex3fv(np.array([0.,0.,0.]))
    glVertex3fv(np.array([0.,1.,0.]))
    glColor3ub(0, 0, 255)
    glVertex3fv(np.array([0.,0.,0]))
    glVertex3fv(np.array([0.,0.,1.]))
    glEnd()

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

def key_callback(window, key, scancode, action,
mods):
    global gCamAng, gCamHeight
    if action==glfw.PRESS or action==glfw.REPEAT:
        if key==glfw.KEY_1:
            gCamAng += np.radians(-10)
        elif key==glfw.KEY_3:
            gCamAng += np.radians(10)
        elif key==glfw.KEY_2:
            gCamHeight += .1
        elif key==glfw.KEY_W:
            gCamHeight += -.1
```

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

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

    glfw.terminate()

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

# Moving Camera vs. Moving World

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

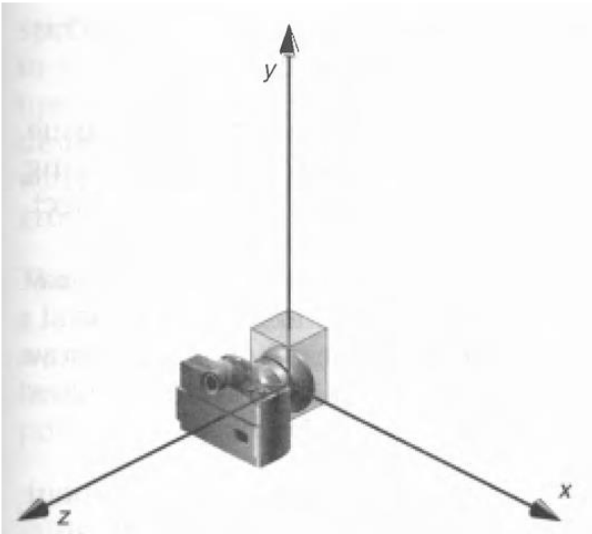


# Moving Camera vs. Moving World

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

- Note that **gluLookAt() is NOT the only way to manipulate the camera.**

- The **default OpenGL camera** is:
- located at the **origin**
- looking in **negative z direction**
- its up direction is **positive y**



# 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 #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 5

- Next lecture:
  – 6 - Projection, Mesh 1

- **Class Assignment #1**
  – **Due: 23:59, April 19, 2022**


- 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. Karan Singh http://www.dgp.toronto.edu/~karan/courses/418/