Introduction to Graphics Programming and its Applications

繪圖程式設計與應用

Transformation

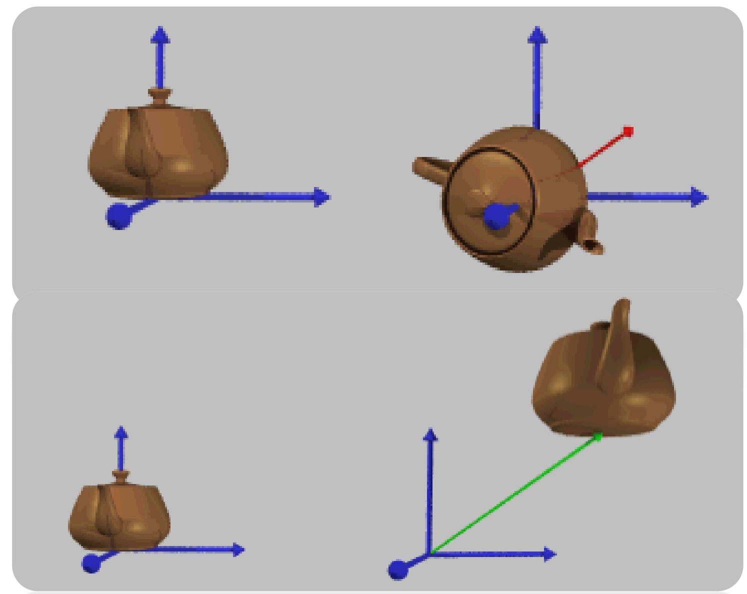
Instructor: Hung-Kuo Chu

Department of Computer Science

National Tsing Hua University

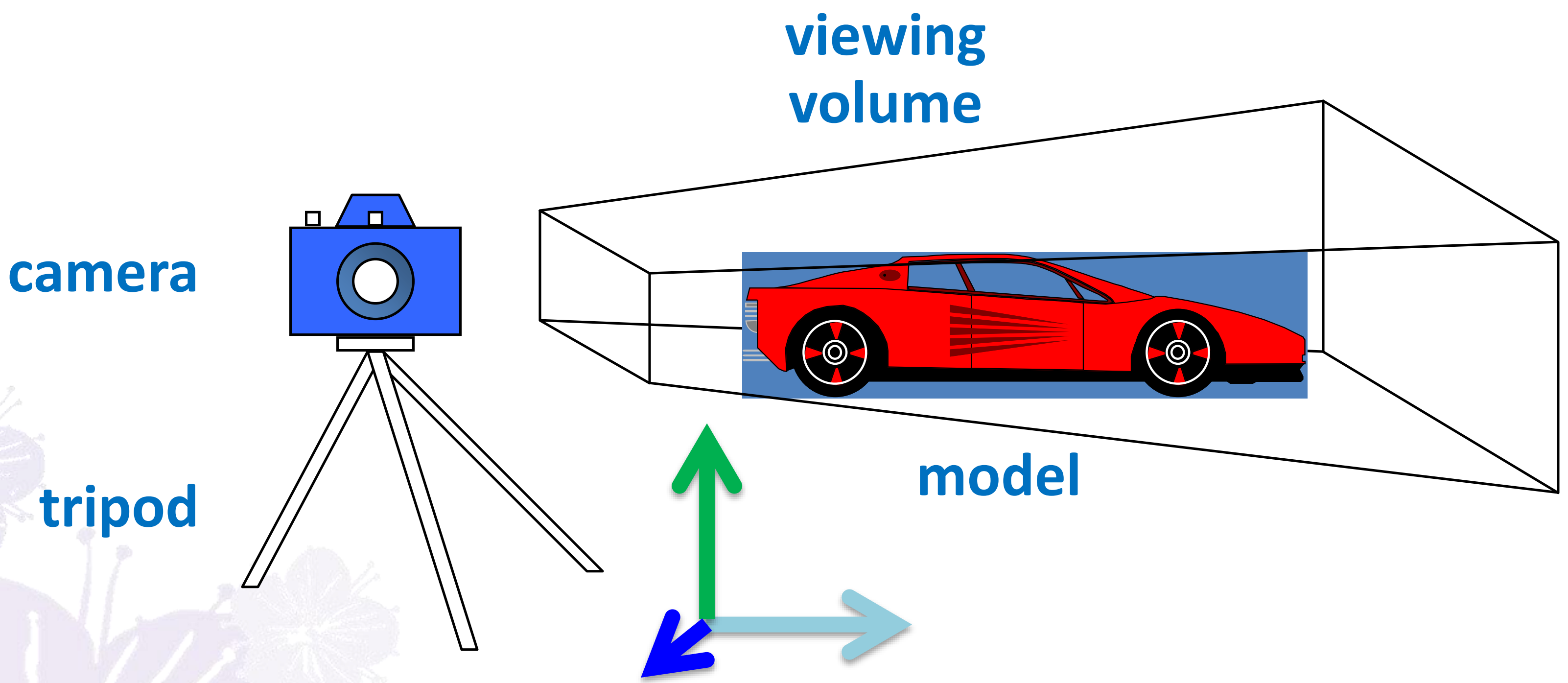
CS4505

Projection & Transformation



Camera Analogy

- 3D rendering is like taking a photograph



Camera Analogy and Transformation

- Modeling transformation
  - Moving the model around the world
- Viewing transformation
  - Tripod : defining how does the camera view the world
- Projection transformation
  - Take the shoot!
- Viewport transformation
  - Manipulate the physical photograph

Coordinate System and Transformation

- Steps toward capturing an image:
  - Specify geometry (world coordinate)
  - Specify camera (camera coordinate)
  - Projection (window coordinate)
  - Map to the viewport (screen coordinate)
- Each step uses a transformation
- Every transformation is equivalent to a change in coordinate system

$$\begin{bmatrix} \mathrm{a} & b \\ \mathrm{c} & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

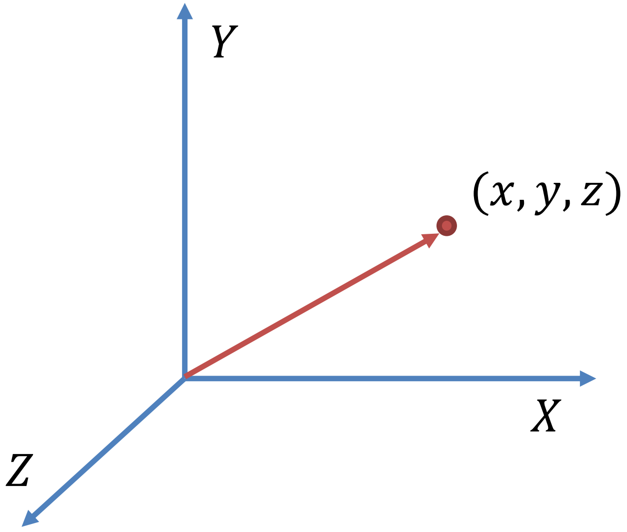
$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \\ 1 \end{bmatrix}$$

VECTOR & MATRIX

Vector

- A vector can be thought of a ***point*** in a space or a ***direction*** from the origin to the point

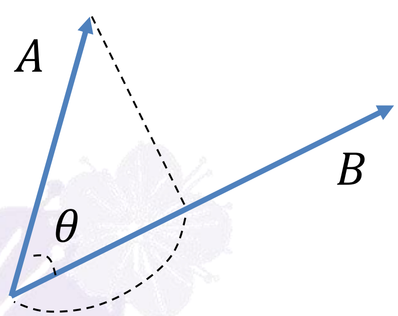


Vector

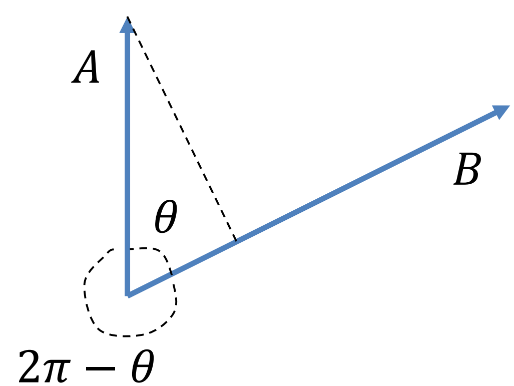
- The length of a vector is the square root of the sum of the squares of each components
- $length(V) = \sqrt{V.x^2 + V.y^2 + V.z^2}$
- ***Euclidean norm*** or ***2-norm***
- V is said to be a ***unit vector*** or a ***normalized vector*** if its length is 1

Vector Dot Product

- Dot product is the projection length of vector A onto vector B multiplied by the length of B
- $dot(A, B) = |A||B|\cos\theta$
- Remember that $\cos\theta = \cos(2\pi - \theta)$
- Note that $dot(A, B) = dot(B, A)$



$$\frac{dot(A, B)}{|\mathrm{B}|} = |A|\cos(\theta)$$



Vector Cross Product

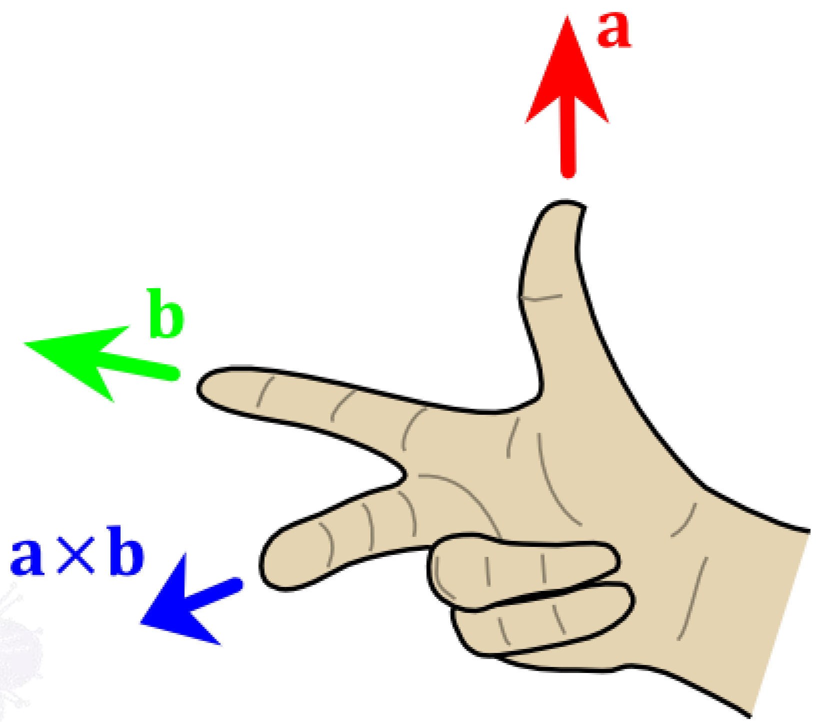
- Cross product of A, B is a vector orthogonal to A and B
- We only discuss the cross product in 3D space
- 2D vector is a 3D vector special case with z = 0
- $cross(A, B) = |A||B| \sin\theta\, n =$

$$\begin{bmatrix} A.y * B.z - A.z * B.y \\ A.z * B.x - A.x * B.z \\ A.x * B.y - A.y * B.x \end{bmatrix}$$

- Note that $cross(A, B) = -cross(B, A)$

Vector Cross Product

- Rule of right hand



Homogeneous Coordinates

- In 3D graphics, we usually present a point by a 4D vector $(x, y, z, 1)$, a direction by $(x, y, z, 0)$
- There is no strong mathematical reason for doing so; but it has many advantages
  - Projective geometry
  - For example, a point subtracting by another point yields a direction
- ***Matrix transformation*** can benefit from this representation too

Homogeneous Coordinates

- To sum up:
  - Each vertex is a column vector
  - w is none-zero for ***points***, zero for ***directions***
  - All operations are multiplication of matrices

$$\vec{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Matrix

- Matrices can be thought of as a table of column vectors, and they are widely used in 3D graphics, we focus on the ***effect*** of a matrix after it is ***multiplied to a vector***, which is often called ***matrix transformation***
- Example: $\begin{bmatrix} x+a \\ y+b \\ z+c \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$

Generic Matrix

- Generally, a 4x4 matrix represents 3 unit axes and an origin of a coordinate system with respect to a basis

$$\begin{bmatrix} X_x & Y_x & Z_x & O_x \\ X_y & Y_y & Z_y & O_y \\ X_z & Y_z & Z_z & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

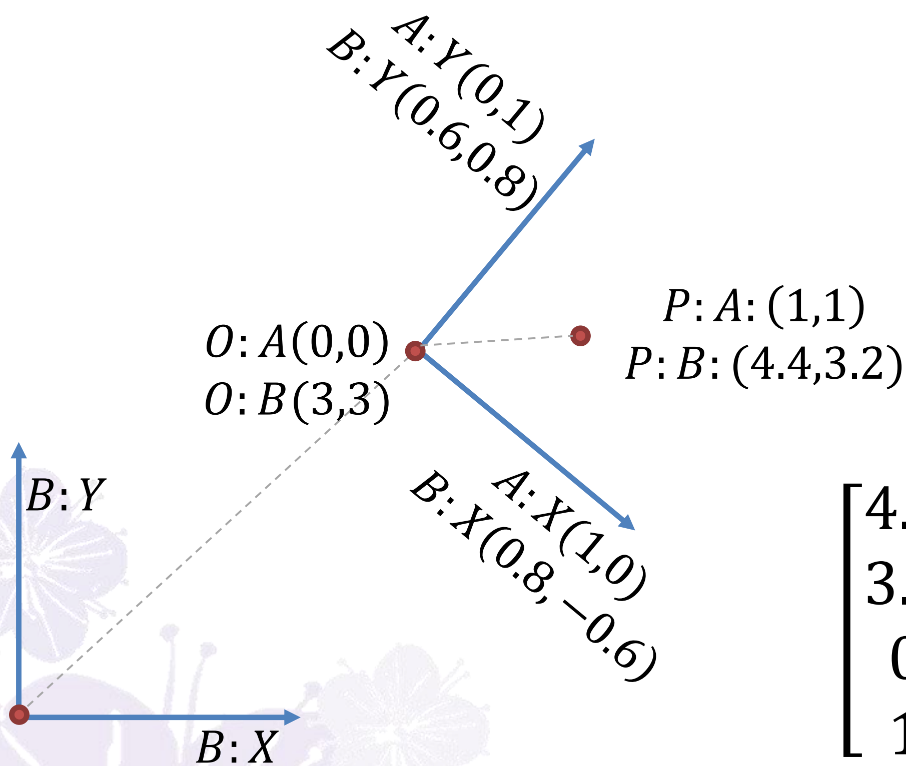
3 axis direction vector

Origin point vector

Generic Matrix

- Multiply a vertex expressed in the identity coordinate system, the result is a new vertex that has been transformed to the new coordinate system
- To transform a vertex in system A to system B, calculate A changed basis to B

Generic Matrix



Transform P from system A to B

Origin

$$\begin{bmatrix}4.4\\3.2\\0\\1\end{bmatrix} = \begin{bmatrix}0.8 & 0.6 & 0 & 3\\-0.6 & 0.8 & 0 & 3\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix} * \begin{bmatrix}1\\1\\0\\1\end{bmatrix}$$

$X\ axis$ $Y\ axis$ $Z\ axis$

Transformation in 3D

- To sum up:
  - A vertex is transformed by a 4 x 4 matrix
  - All matrices are stored in column-major order
  - All operations are multiplication of matrices
  - Matrices are always ***pre-multiplied***

$$GLfloat(GLdouble)\ \mathrm{M}[16]= \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Vertex Matrix Product

$$\mathbf{p}' = \mathbf{M}\mathbf{p}$$

Identity Matrix

- An identity matrix only has 1s in its diagonal components. It has no net effect

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

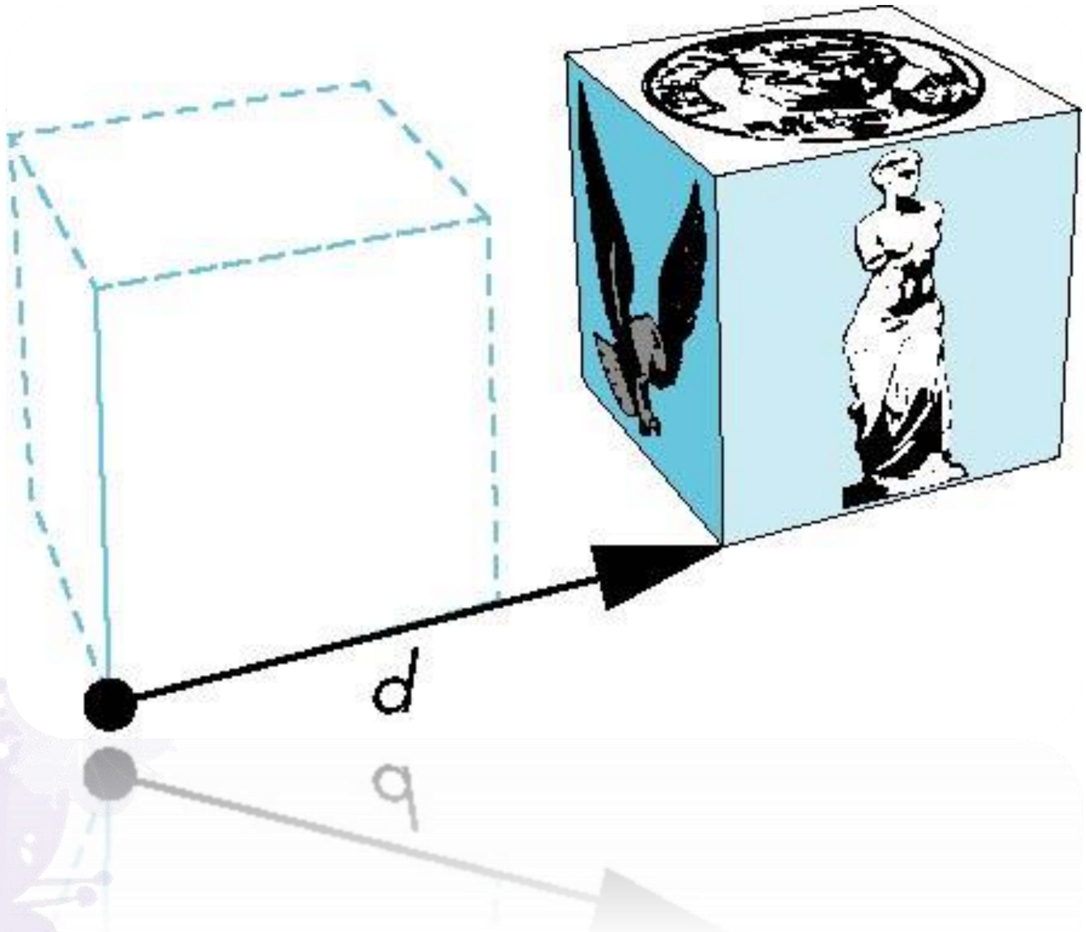
Translation Matrix

- A translation matrix is used to position an object within 3D space without rotating in any way
- Direction vectors ($w = 0$) are not effected

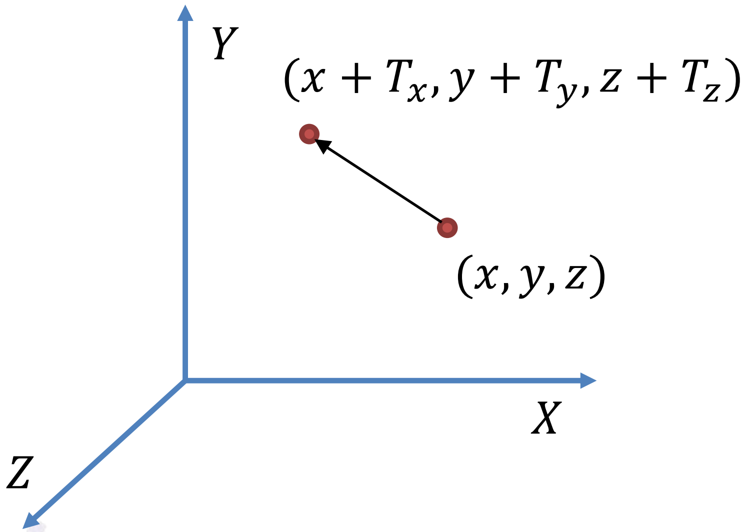
$$\begin{bmatrix} x + T_x w \\ y + T_y w \\ z + T_z w \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

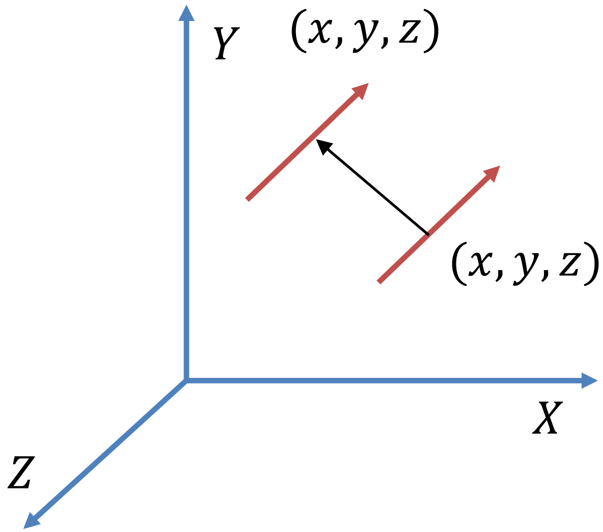
- The LHS equals $(x, y, z, 0)$ when $w = 0$

Translation Matrix



Translation Matrix





A direction is not changed by a translation matrix

Scaling Matrix

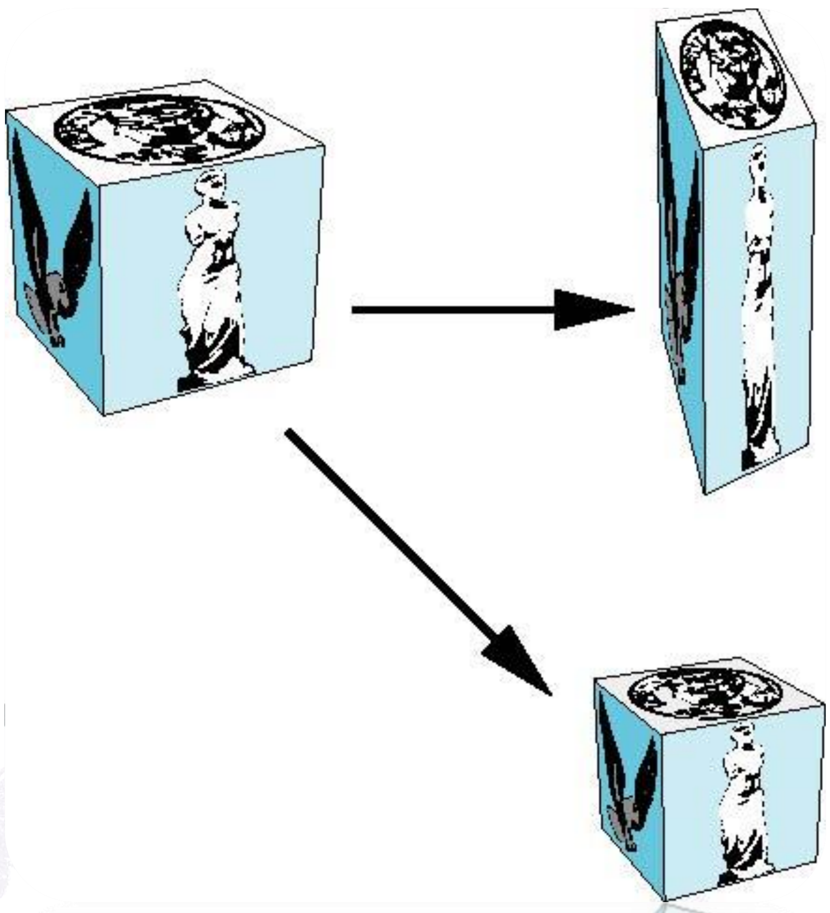
- A scaling matrix is used to enlarge or shrink the size of a 3D object
- ***Uniform scaling***: $S_x = S_y = S_z$

$$\begin{bmatrix} S_x x \\ S_y y \\ S_z z \\ w \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

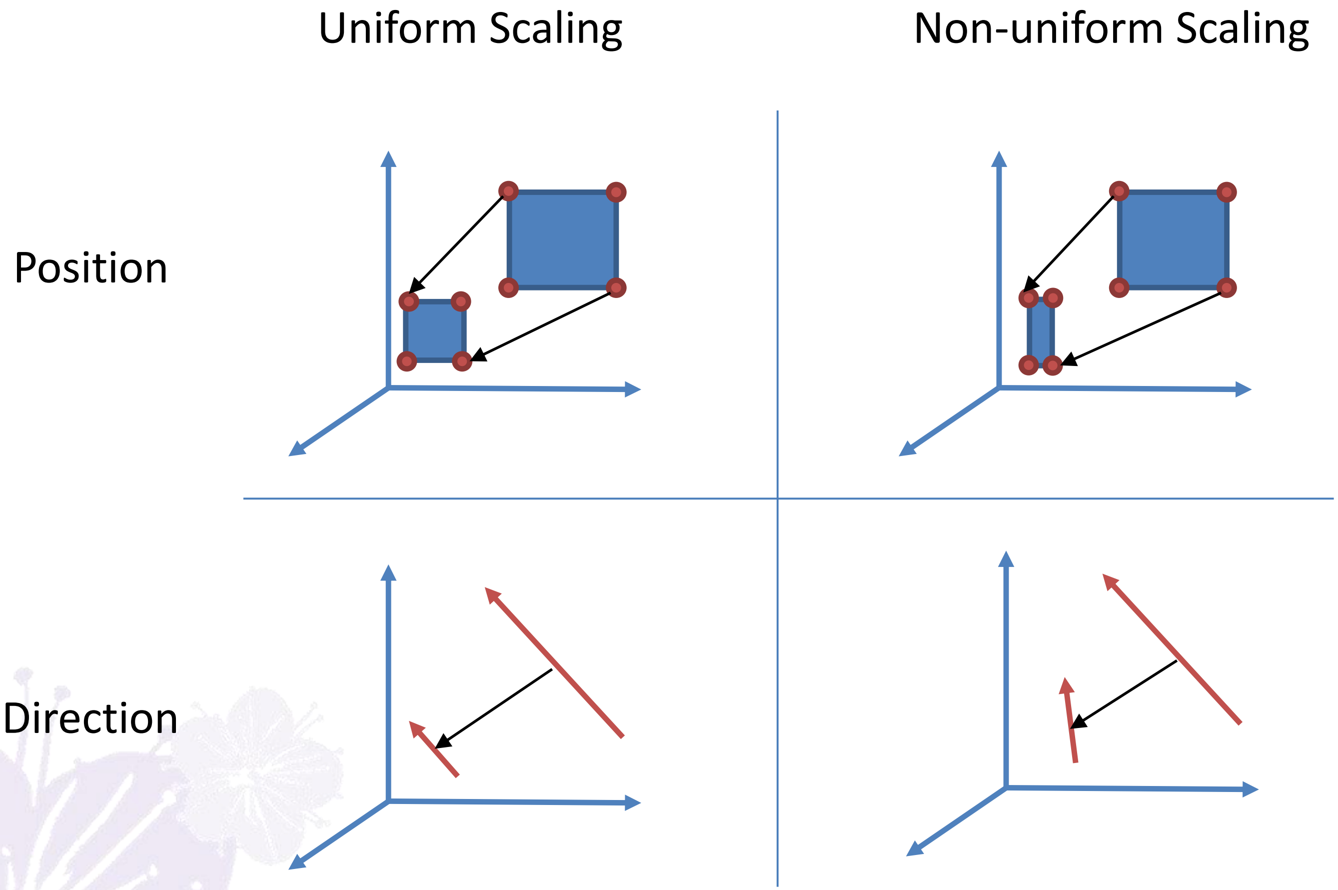
Scaling Matrix

- Point vectors
  - Point is scaled
- Direction vectors
  - Uniform: length scaled, direction may be unchanged or negated
  - Non-uniform: both length and direction may change
- Usually, scaling matrix won't be applied to direction vectors

Scaling Matrix



Scaling Matrix

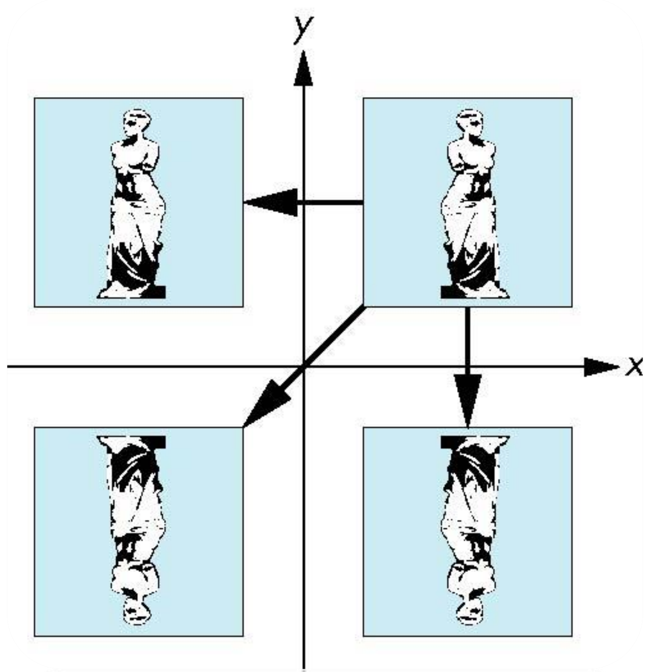


Reflection Matrix

- Using negative scale factors

$s_x = -1 \; s_y = 1$

original



$s_x = -1 \; s_y = -1$

$s_x = 1 \; s_y = -1$

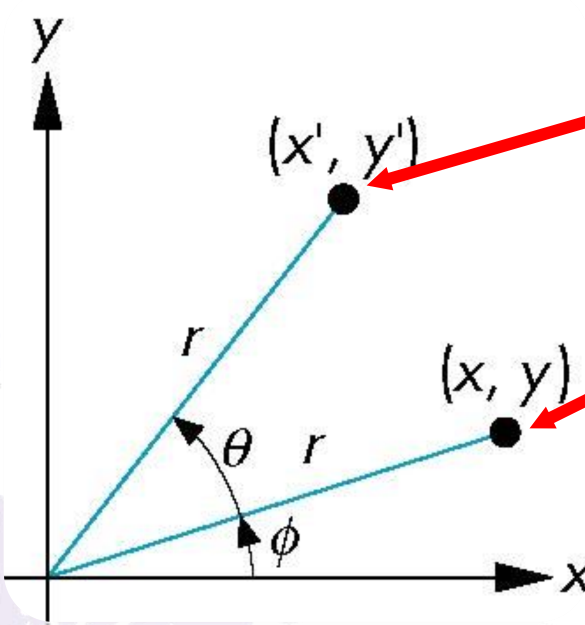
Rotation Matrix

- A rotation matrix is used to rotate a set of points within a coordinate system.
- While the individual points are assigned new coordinates, their relative distances do not change

1. Rotation around x, y, z axis
2. Rotation based on Euler angles
3. Rotation around given axis $(\mathrm{R}_x, \mathrm{R}_y, \mathrm{R}_z)$

Rotation in 2D

- Consider rotating about the **origin** by **q** degrees.
  - radius stays the same, angle increases by q.



$$x' = r\cos(\phi + \theta)$$
$$y' = r\sin(\phi + \theta)$$

$+$

$$x = r\cos\phi$$
$$y = r\sin\phi$$

$\Downarrow$

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

Rotation around Axis

- Given x, y or z axis and an angle θ

x Axis $$M_{rot\_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

y Axis $$M_{rot\_y} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

z Axis $$M_{rot\_z} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Euler Angles Rotation

- Extension of axis rotations
- Given a Euler angles $(\theta_x, \theta_y, \theta_z)$ representing rotation angles of x, y, z axis, apply axis rotation to each axis to rotate the object
- **Gimbal-lock**: The final rotation matrix depends on the ***order of multiplication***, it is sometimes the case that the rotation in one axis will be mapped onto another rotation axis

Rotation around Given Axis

- First, we compute a ***quaternion*** $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{w})$ from the given axis $\left(\mathrm{R}_x, \mathrm{R}_y, \mathrm{R}_z\right)$ and angle θ
- **Quaternion**: a 4D vector representing a rotation, it extends the concept of rotation in 3D to rotation in 4D. This avoids the problem of gimbal-lock and allows for the implementation of smooth and continuous rotation

Rotation around Given Axis

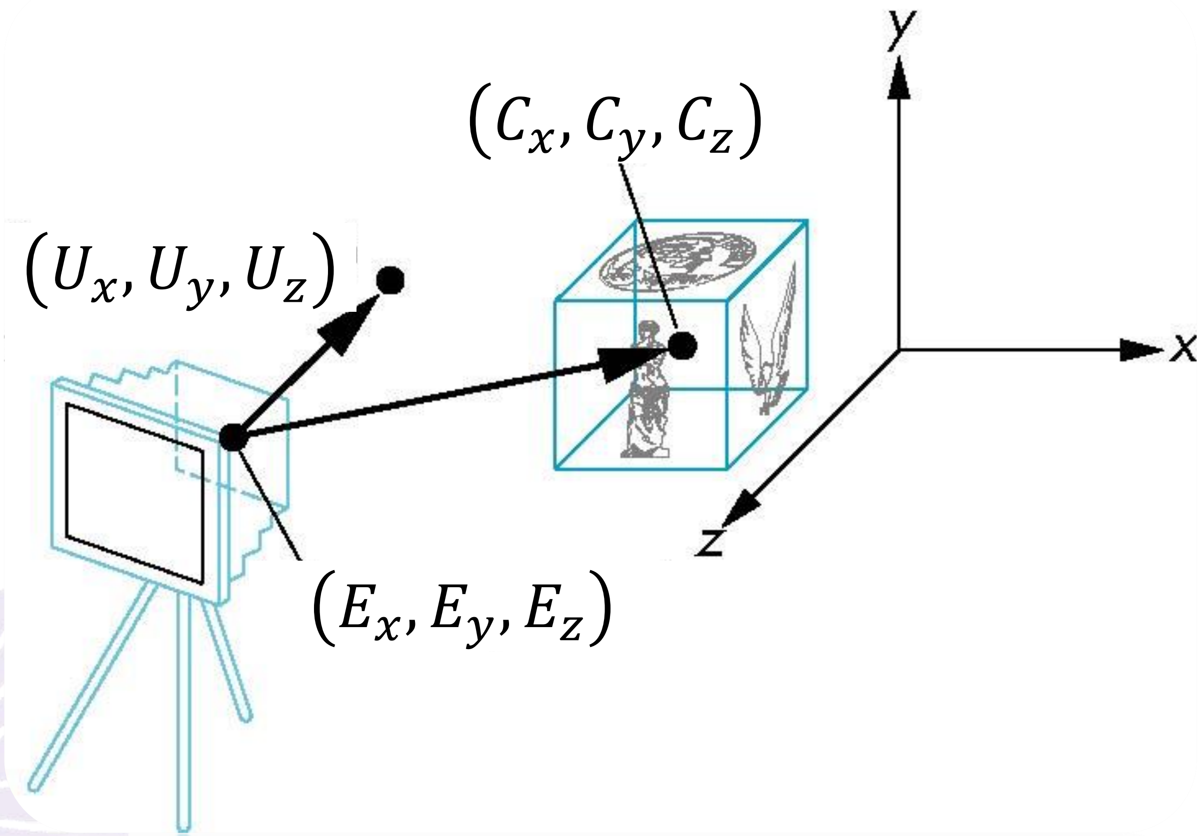
- $R = normalize(R)$
- $Q = \left(R_x \sin\frac{\theta}{2}, R_y \sin\frac{\theta}{2}, R_z \sin\frac{\theta}{2}, \cos\frac{\theta}{2}\right) = (X, Y, Z, W)$
- $C = 1 - \cos\theta$

$$M_{rot} = \begin{bmatrix} 1-(2Y^2+2Z^2) & 2XY-2ZW & 2XZ+2YW & 0 \\ 2XY+2ZW & 1-(2X^2+2Z^2) & 2YZ-2XW & 0 \\ 2XZ-2YW & 2YZ+2XW & 1-(2X^2+2Y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta + {R_x}^2 C & R_x R_y C - R_z \sin\theta & R_x R_z C + R_y \sin\theta & 0 \\ R_x R_y C + R_z \sin\theta & \cos\theta + {R_y}^2 C & R_y R_z C - R_x \sin\theta & 0 \\ R_x R_z C - R_y \sin\theta & R_y R_z C + R_x \sin\theta & \cos\theta + {R_z}^2 C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewing Matrix

- A viewing matrix is used to set camera position and direction in world space

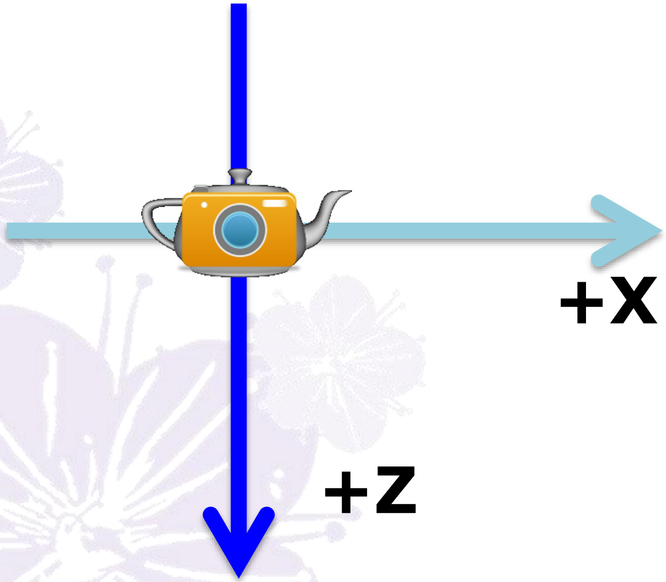


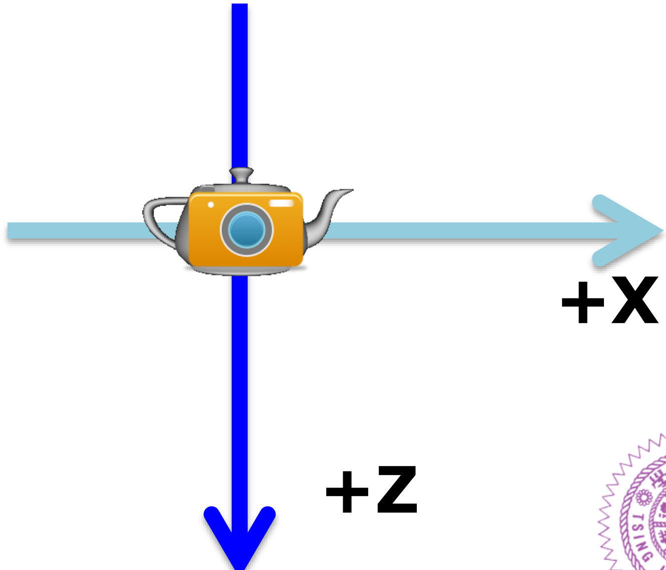
Viewing Matrix

- Position the camera/eye in the world.
  - Setup the configuration of tripod and camera.
- To “fly through” in the 3D world.
  - change viewing transformation and redraw scene

Viewing v.s. Modeling

- Move the camera in the positive z direction.
  - Translate the camera coordinate frame.
- Move the objects in the negative z direction.
  - Translate the world coordinate frame.





Viewing Matrix

- $M_{viewing} = M_{rotation} * M_{translation}$
- Given camera position E, eye look at position C and up direction U

1. Move the origin to camera position
2. Then rotate camera's ***forward direction to -z axis*** and camera up direction to +y axis

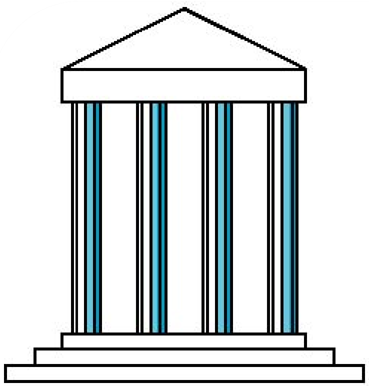
Viewing Matrix

- Forward direction $F = normalize(C - E)$
- Side direction $S = normalize\big(cross(F, U)\big)$
- $U' = normalize\big(cross(S, F)\big)$

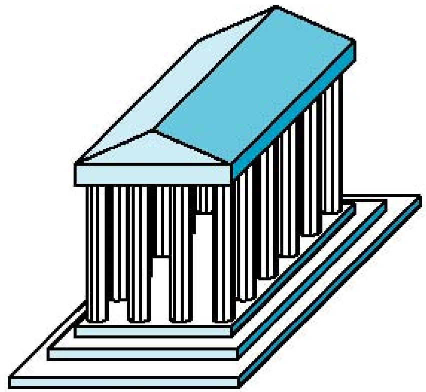
$$M_{viewing} = \begin{bmatrix} S_x & S_y & S_z & 0 \\ U'_x & U'_y & U'_z & 0 \\ \boxed{-}F_x & \boxed{-}F_y & \boxed{-}F_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(S, U, F) is, in fact, a left-handed coordinate system, so it must be converted to right-handed. This conversion also conveniently map forward to –z axis

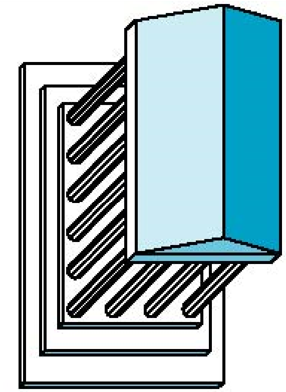
Classical Projections



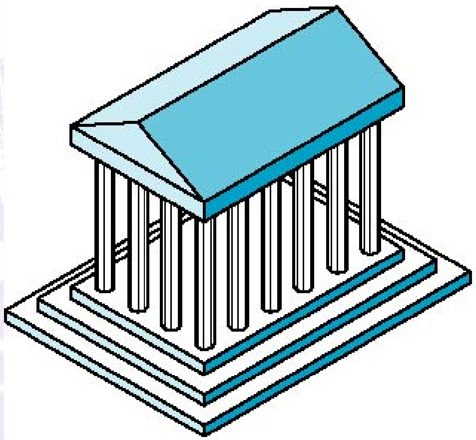
Front elevation



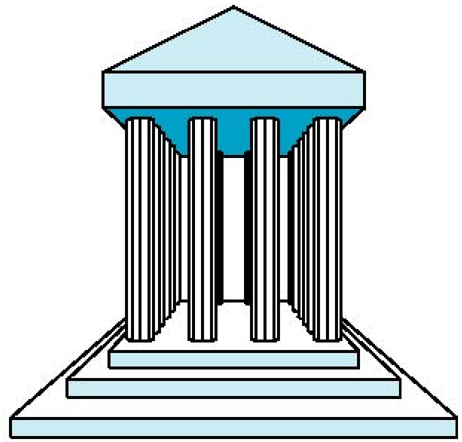
Elevation oblique



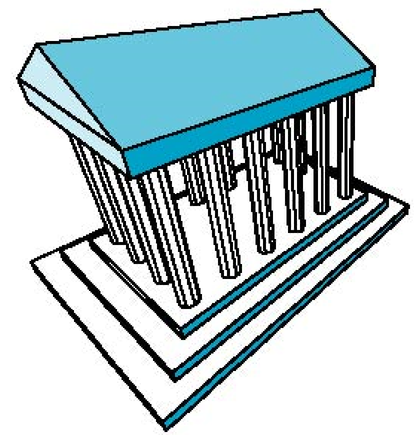
Plan oblique



Isometric



One-point perspective



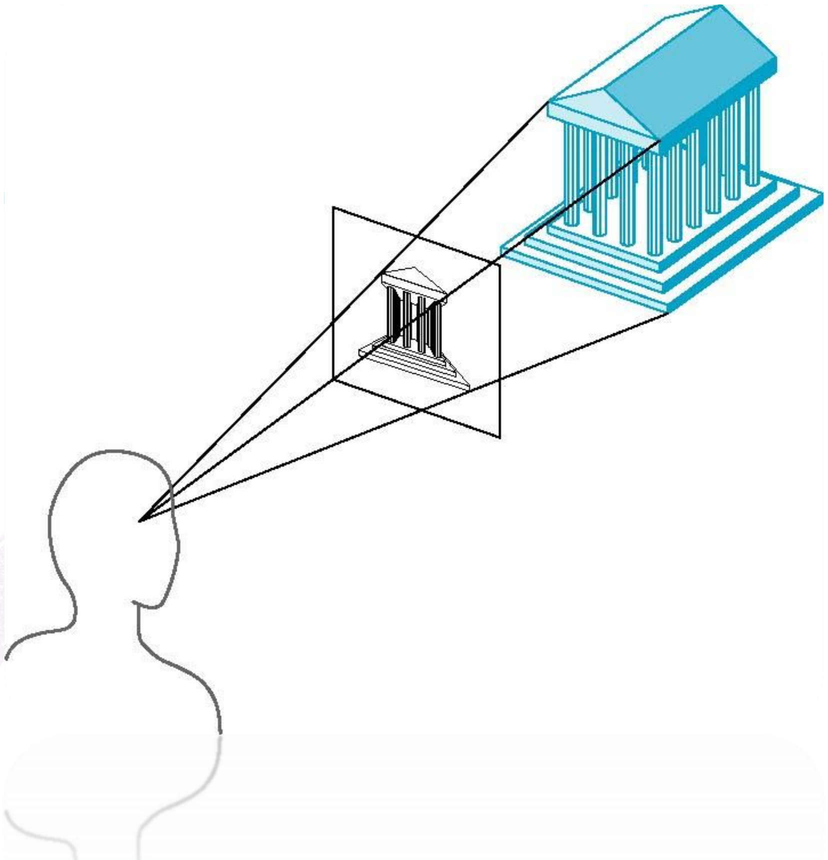
Three-point perspective

Projection Matrix

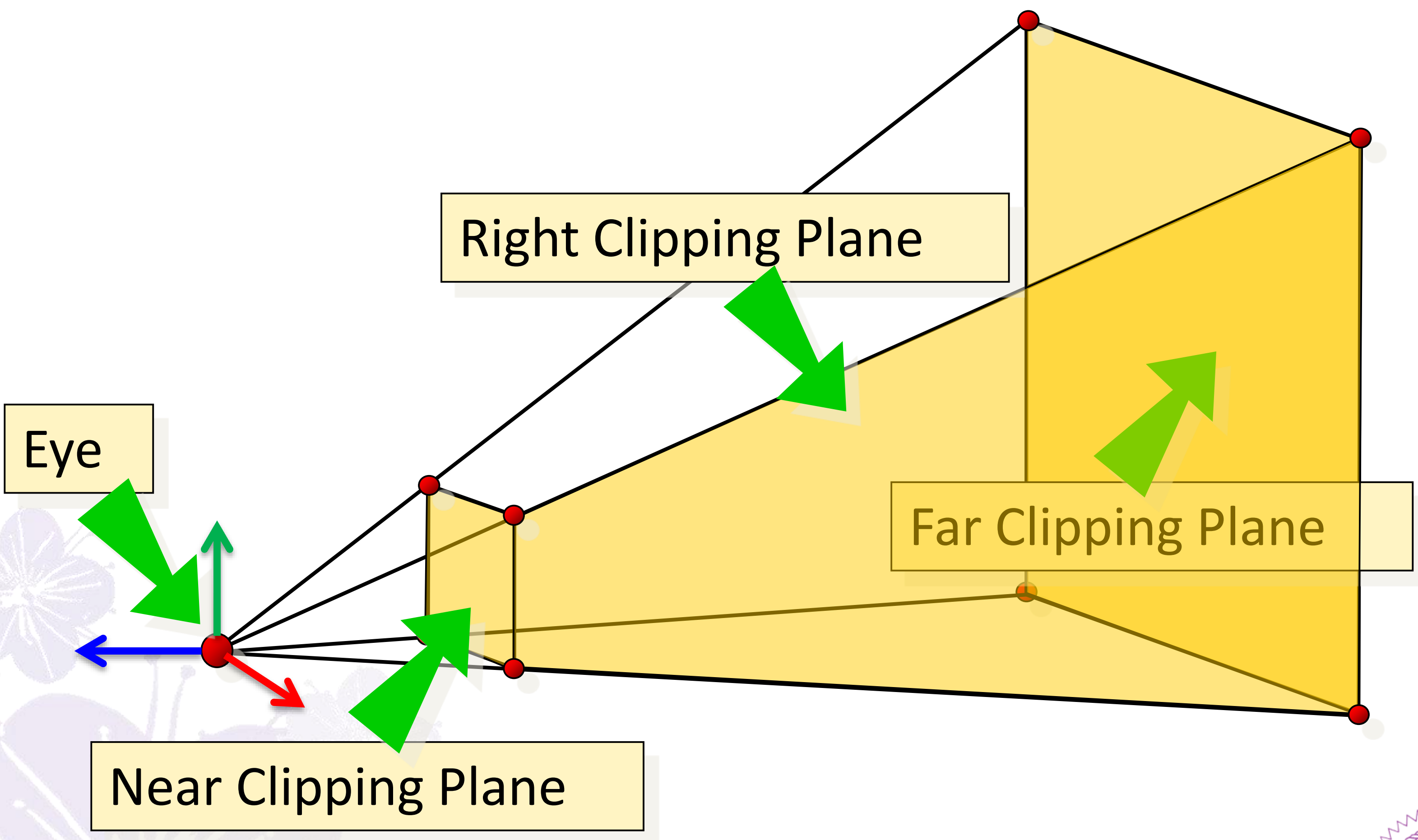
- A projection matrix is used to create a projection of the 3D scene to a 2D plane
- Like camera lenses
- It also ***converts points from right-handed coordinate (RHC) to left-handed coordinate (LHC)*** in OpenGL
- Perspective and orthographic

Perspective Projection

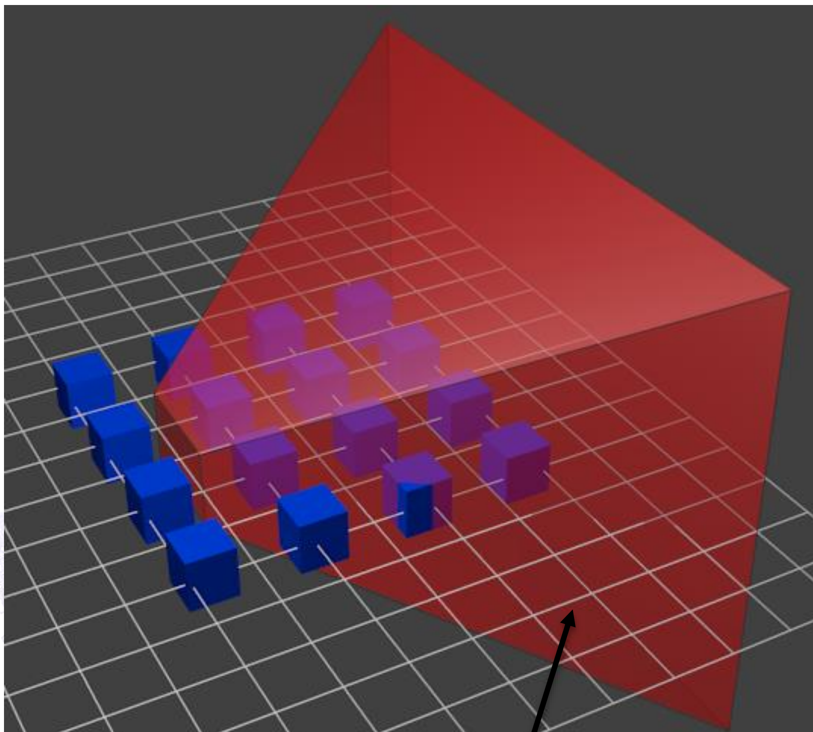
- Projectors converge at the center of projection

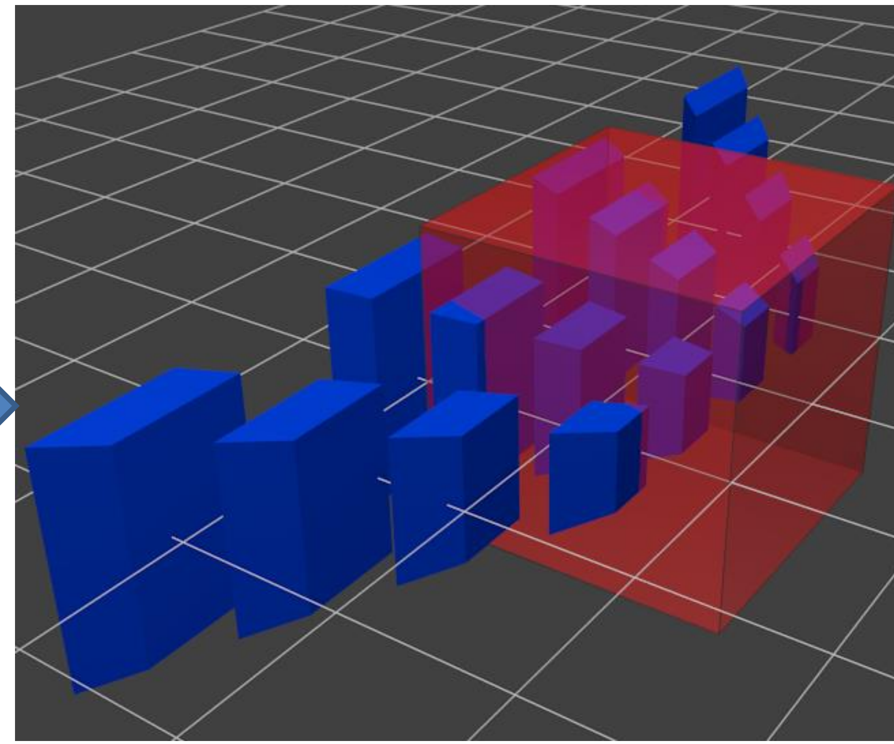


Perspective Projection



Perspective Projection Matrix



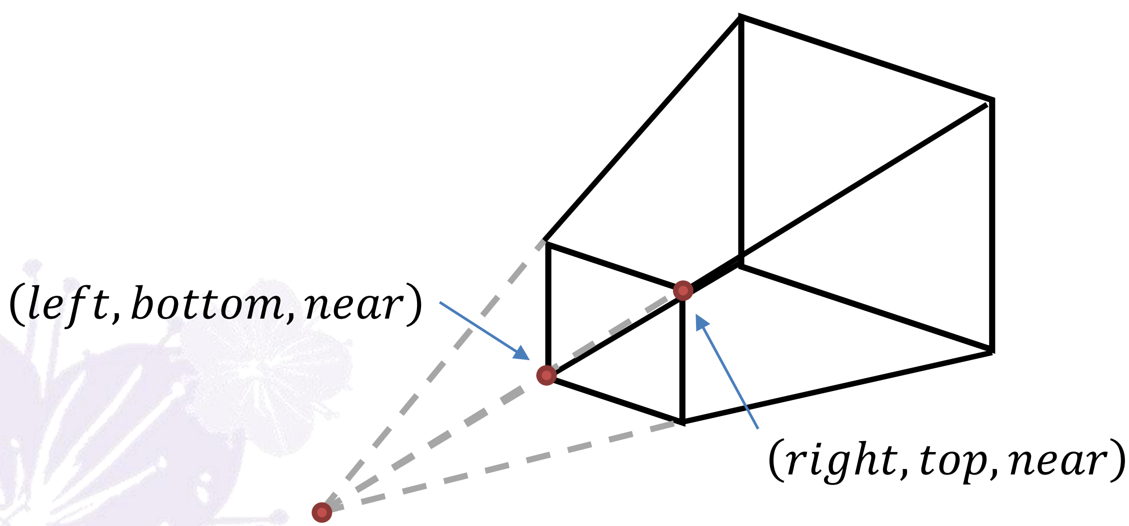


After projection

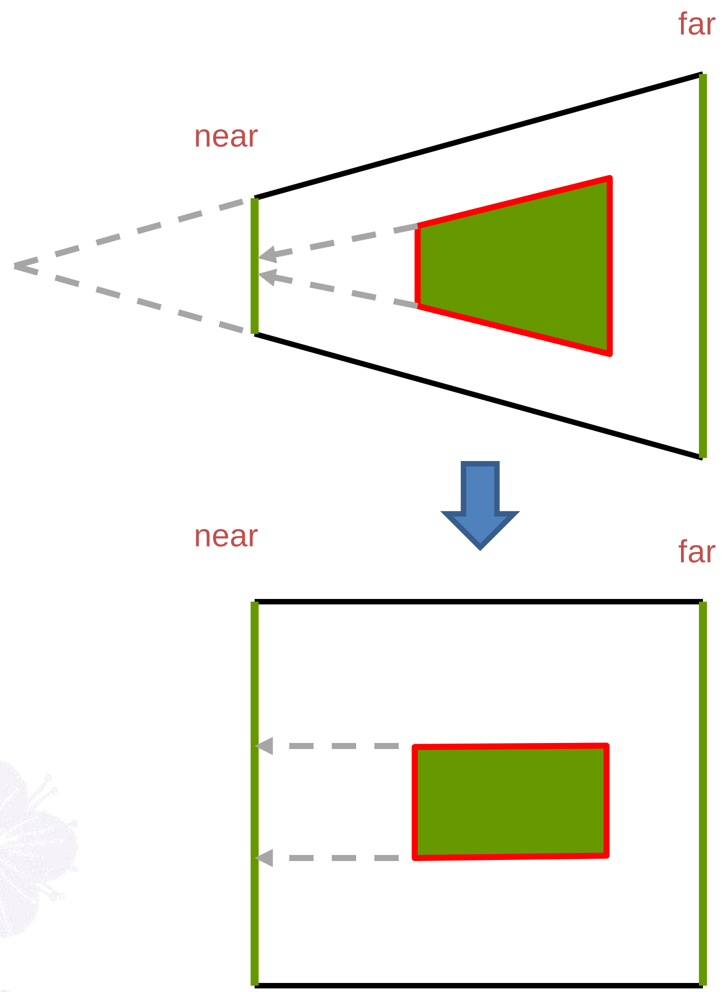
View volume

Perspective Projection Matrix

- Define a perspective projection view volume with a ***frustum***
- A frustum is defined by six parameters, $(left, right, bottom, top, near, far)$



Perspective Projection Matrix



Perspective Projection Matrix

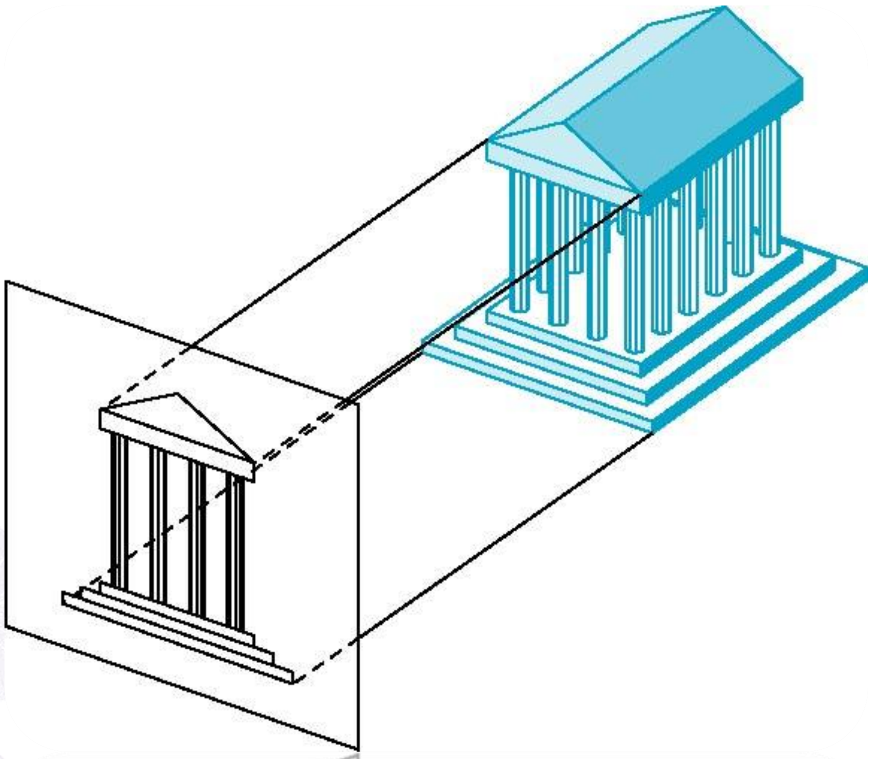
- Remember ***RHC/LHC conversion***

$$M_{perspective} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \boxed{-}\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

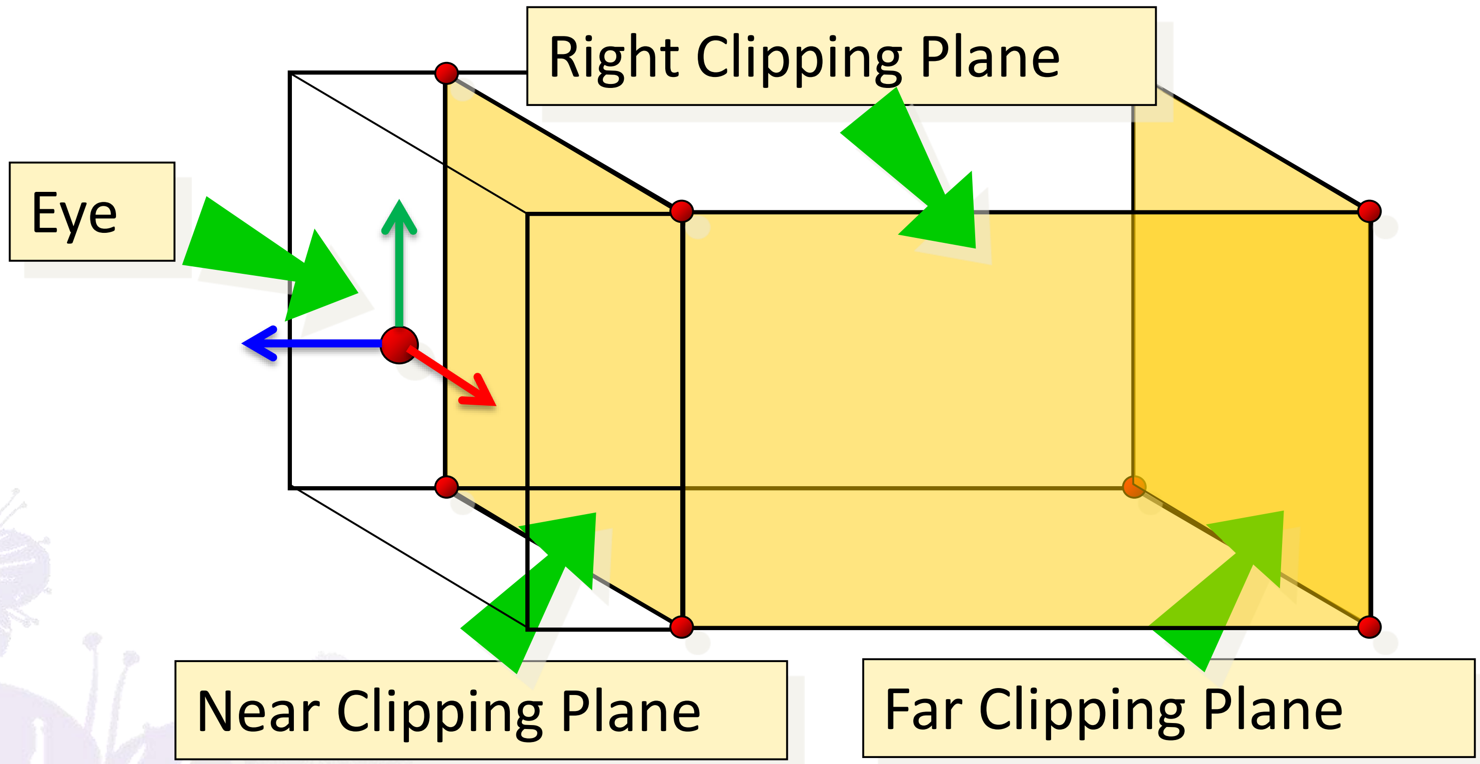
LHC/RHC conversion

Orthographic Projection Matrix

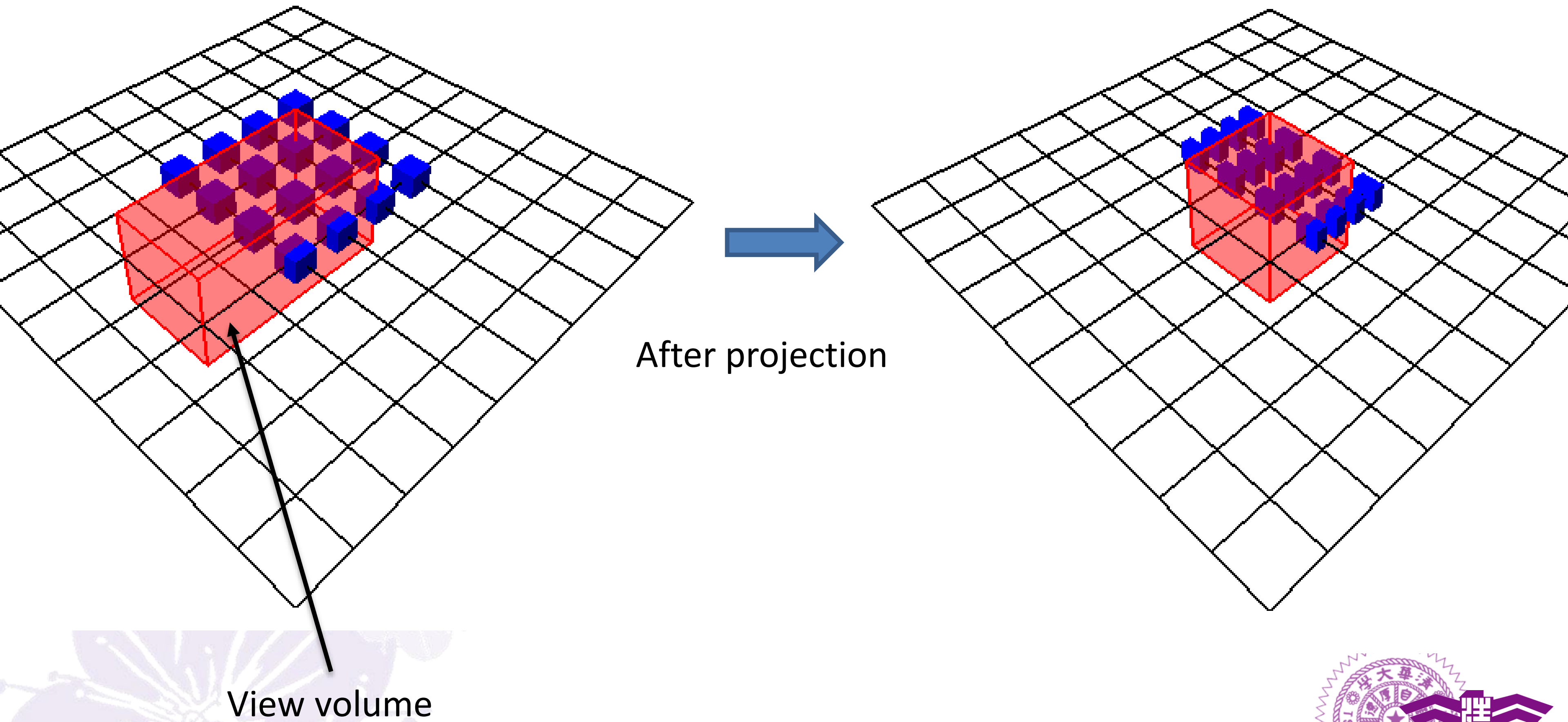
- Projectors are orthogonal to the projection surface



Orthographic Projection

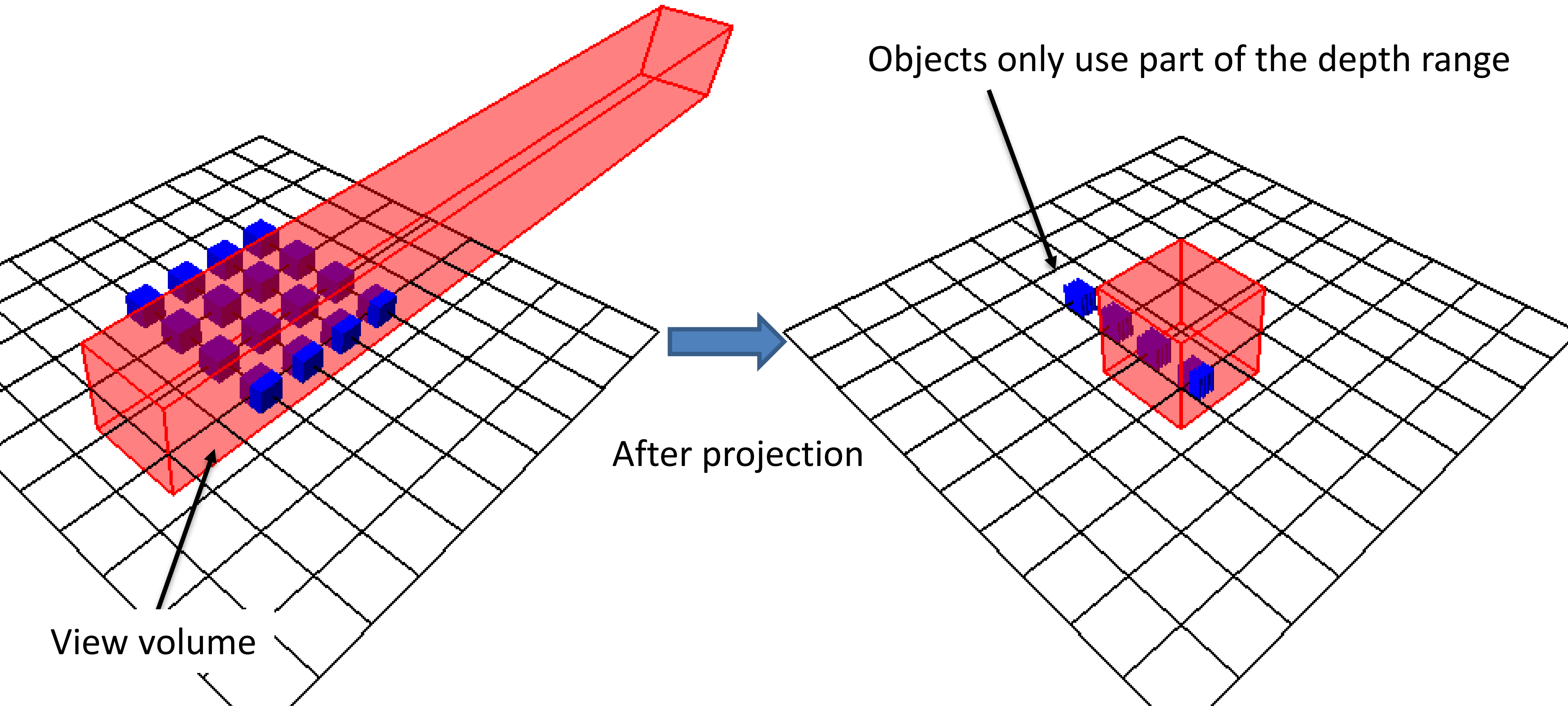


Orthographic Projection Matrix



Orthographic Projection Matrix

- Use appropriate depth range, or depth buffer may suffer from precision problems



Orthographic Projection Matrix

- $M_{orthographic} = M_{scaling} * M_{translation}$

1. Move the center of a ***view volume*** defined by left, right, bottom, top, near and far boundaries to origin $(0,0,0)$
2. Then scale the volume into ***Normalized Device Coordinate*** (**NDC**)
   - $([-1,1],[-1,1],[-1,1])$
3. Don't forget to do ***RHC/LHC conversion***!

Orthographic Projection Matrix

- $glOrtho(left, right, bottom, top, near, far)$
- $Center = [(right, top, far) + (left, bottom, near)]/2$
- $Size = (right, top, far) - Center$

$$M_{ortho} = \begin{bmatrix} \frac{1}{S_x} & 0 & 0 & 0 \\ 0 & \frac{1}{S_y} & 0 & 0 \\ 0 & 0 & \boxed{-}\frac{1}{S_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

RHC/LHC conversion

Orthographic Projection Matrix

$$M = \begin{bmatrix} \frac{1}{S_x} & 0 & 0 & -\frac{C_x}{S_x} \\ 0 & \frac{1}{S_y} & 0 & -\frac{C_y}{S_y} \\ 0 & 0 & -\frac{1}{S_z} & -\frac{C_z}{S_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Inverse

- Although the inverse of matrices can be computed using general methods:
  - Gaussian elimination, or LU decomposition
- We can exploit simple geometric observations:
  - Translation: $T^{-1}(dx, dy, dz) = T(-dx, -dy, -dz)$
  - Rotation: $R^{-1}(q) = R(-q)$
    - Holds for any rotation matrix
    - Note that $\cos(-q) = \cos(q)$ and $\sin(-q)=-\sin(q)$
    - $R^{-1}(q) = R^T(q)$
  - Scaling: $S^{-1}(sx, sy, sz) = S(1/sx, 1/sy, 1/sz)$

Matrix Concatenation

- We can form arbitrary affine transformation matrices by multiplying the rotation, translation, and scaling matrices
- Because the same transformation is applied to many vertices, the cost of computing a matrix $\mathbf{M}=\mathbf{ABCD}$ is much lower than the cost of computing $\mathbf{p}'=\mathbf{Mp}$
- The difficult part is how to form a desired transformation from the specifications in the application

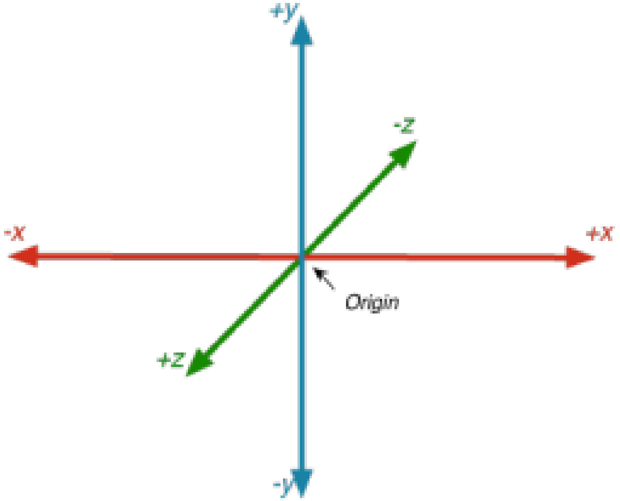
Order of Transformations

- The matrix on the right-hand-side is applied to vertex first
- Mathematically, the following are equivalent:

$$\mathbf{p}' = \mathbf{ABCp} = \mathbf{A}(\mathbf{B}(\mathbf{Cp}))$$

- Many references use column matrices to represent points. In terms of column matrices

$$\mathbf{p}'^{T} = \mathbf{p}^{T}\mathbf{C}^{T}\mathbf{B}^{T}\mathbf{A}^{T}$$



COORDINATE SPACES

Coordinate Spaces

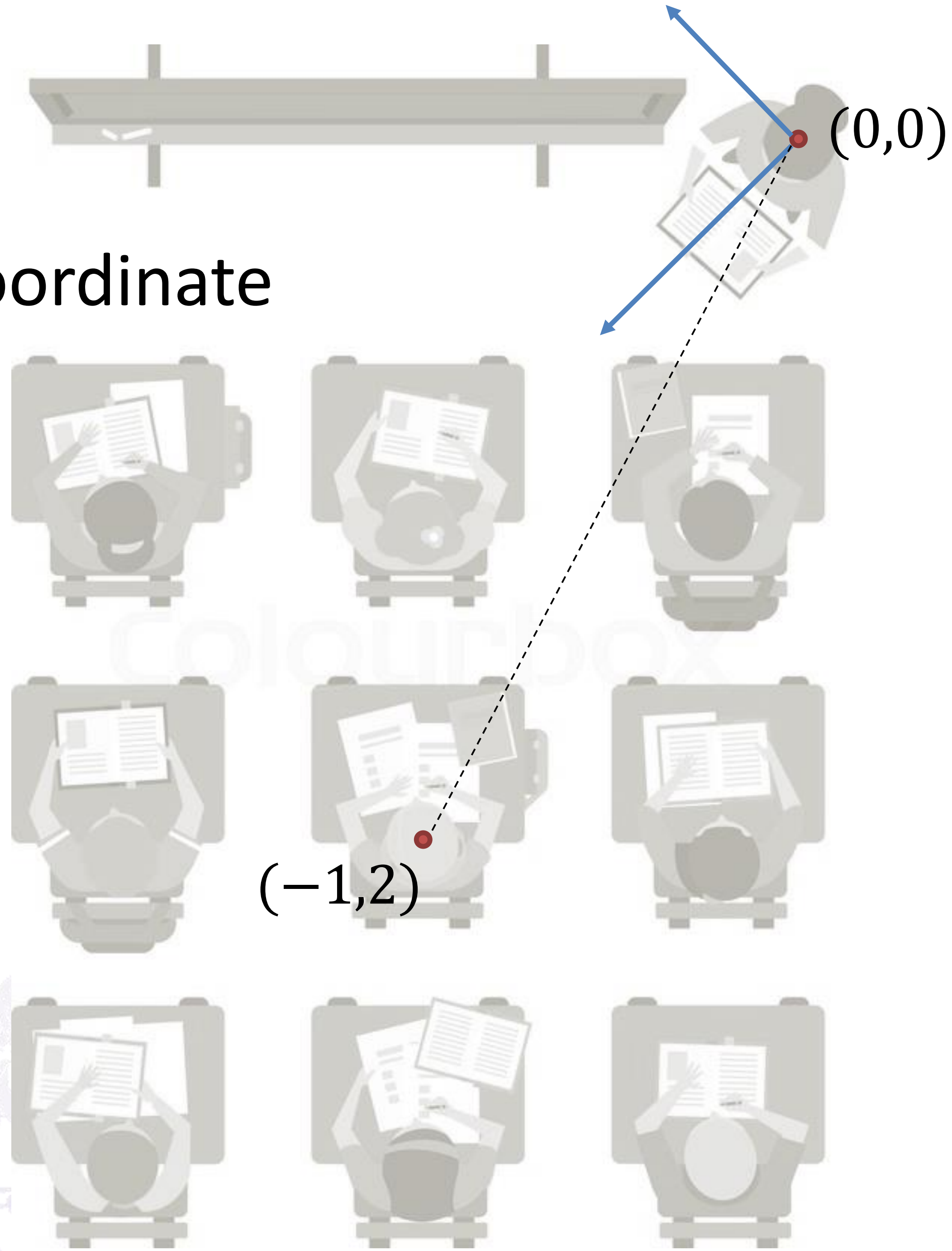
“I understand how the engines work now. It came to me in a dream. The engines don't move the ship at all. The ship stays where it is and the engines move the universe around it.”

「現在我明白那引擎怎麼運作了。這個想法在我的夢中浮現。引擎完全沒有移動太空船。太空船待在同一個地方，而引擎移動了環繞它的宇宙。」

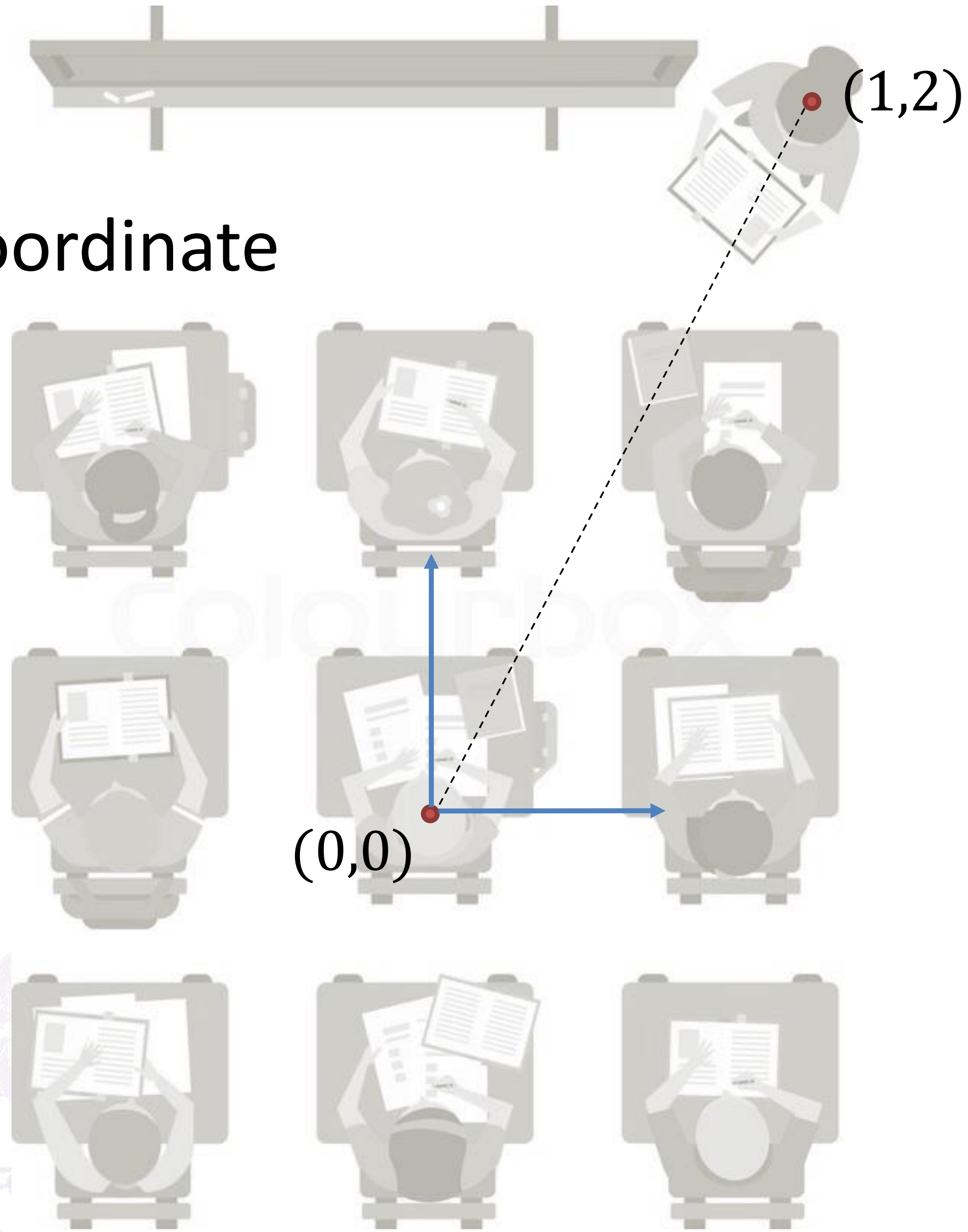
— Futurama《飛出個未來》

The Classroom Analogy

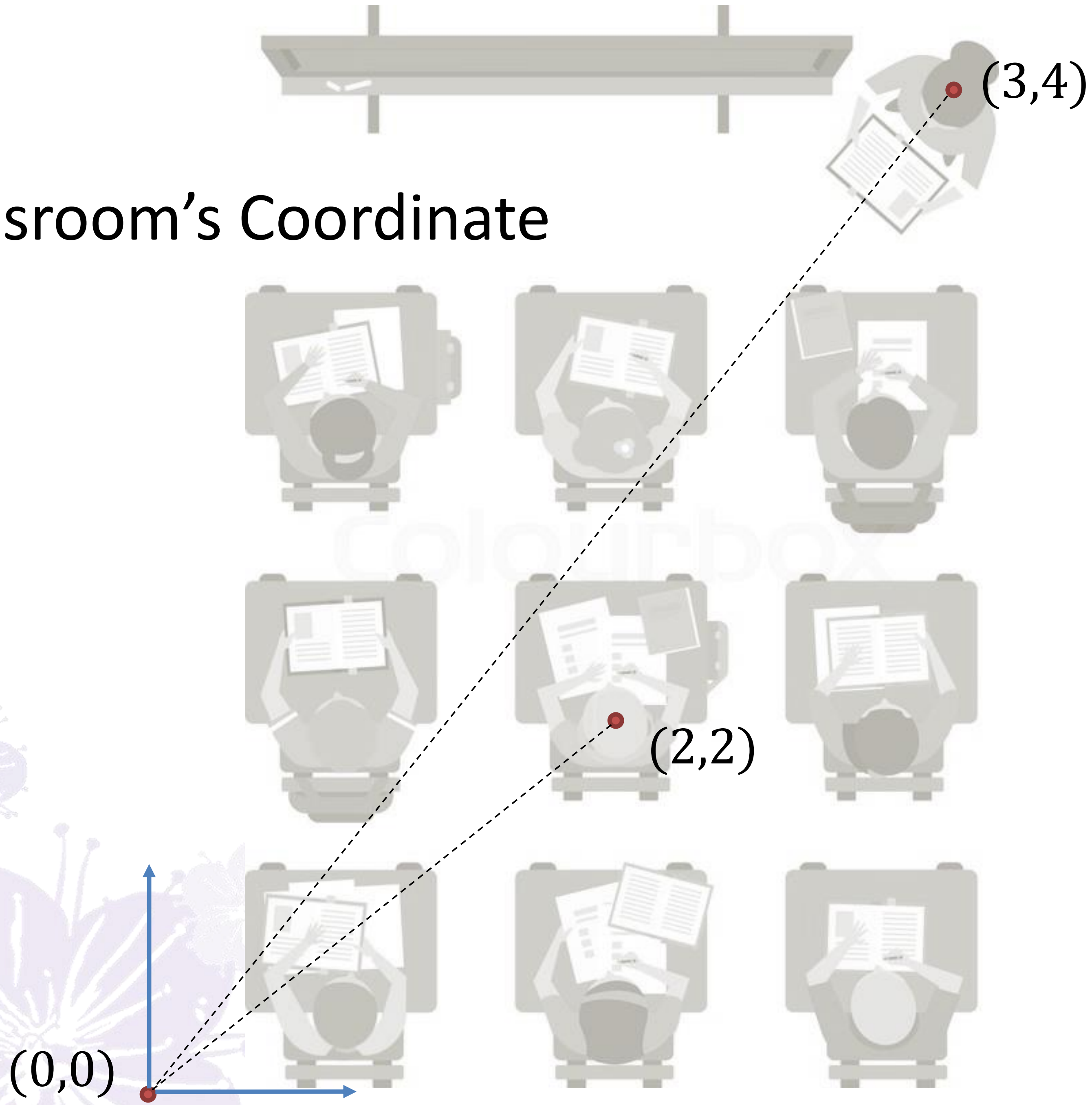
Teacher's Coordinate

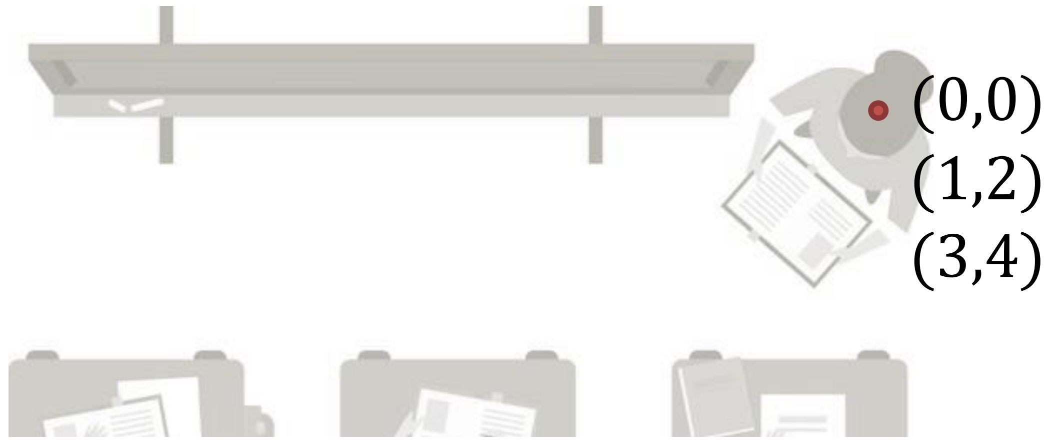


Student's Coordinate

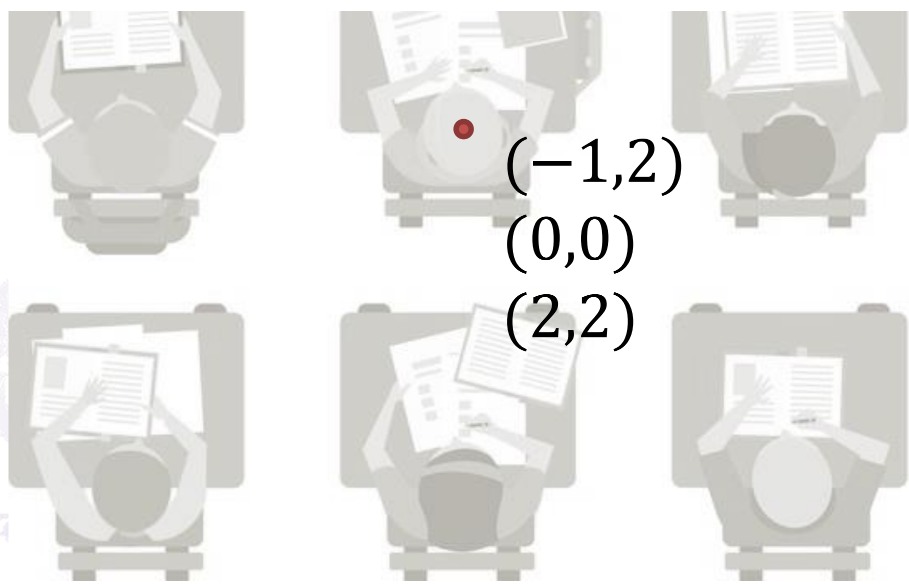


Classroom's Coordinate

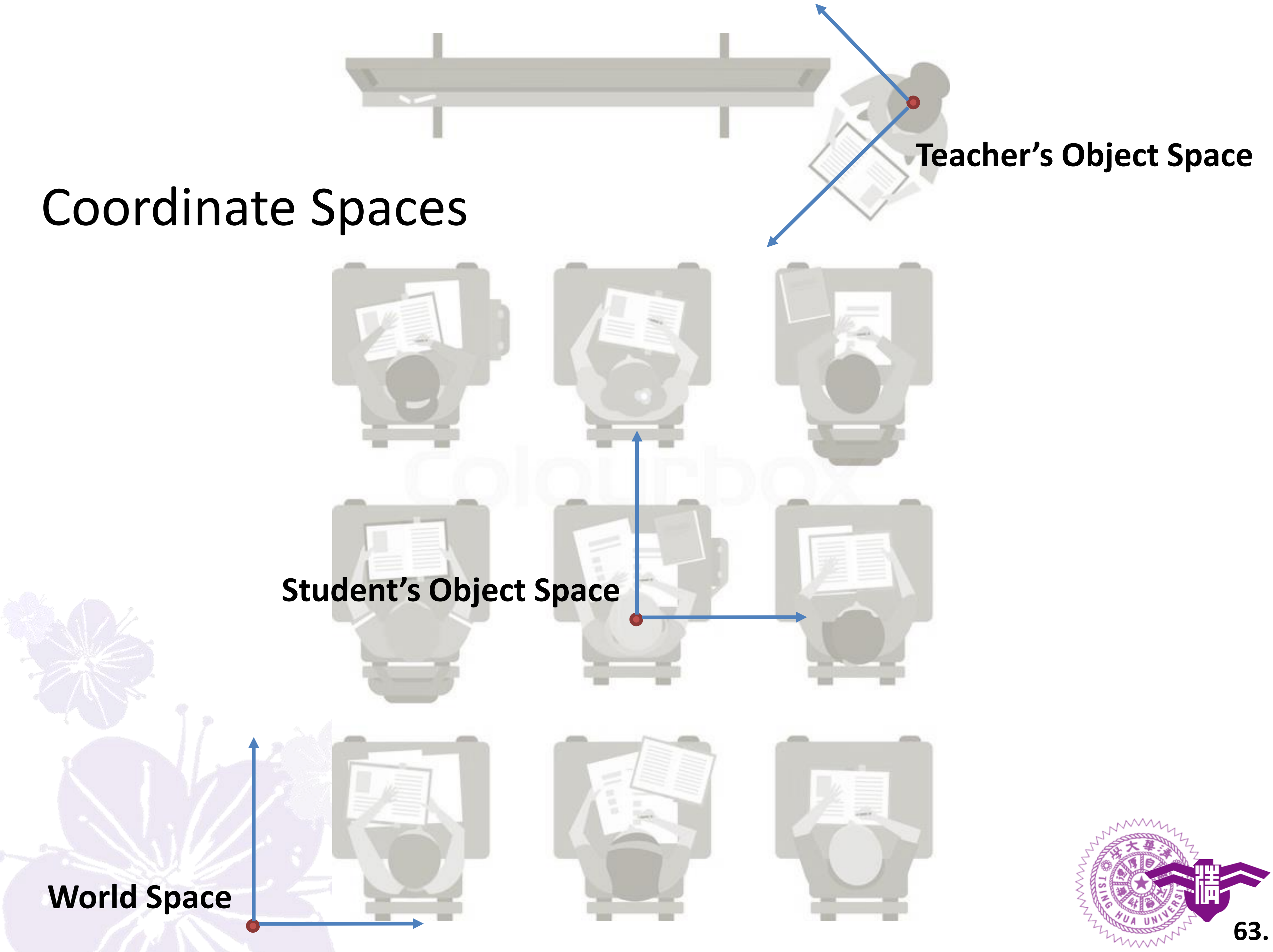




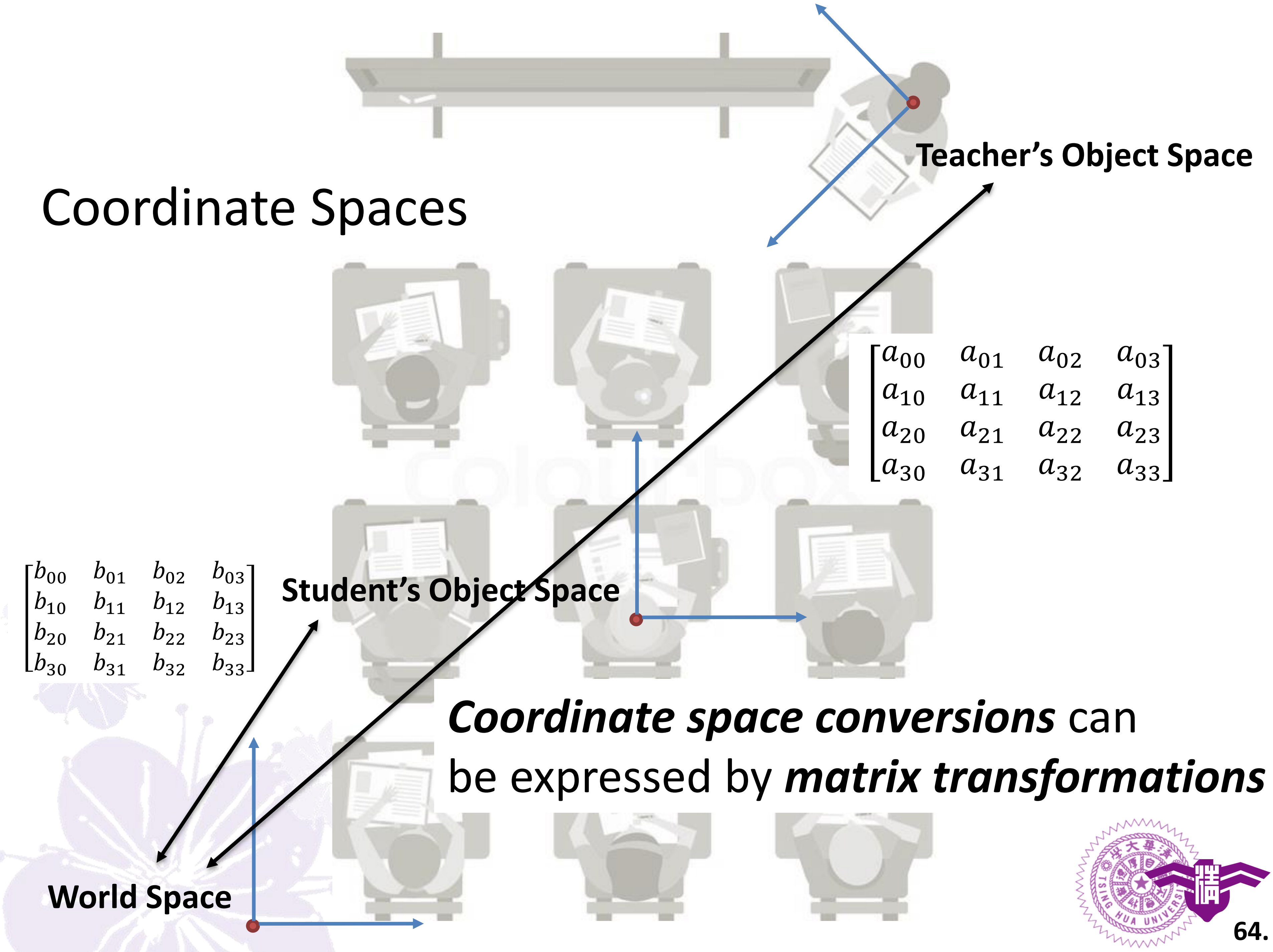
Both teacher and student can be ***represented*** by many ***different coordinates***, But it is ***always the same classroom***



Coordinate Spaces



Coordinate Spaces



Coordinate space conversions can be expressed by ***matrix transformations***

Coordinate Spaces

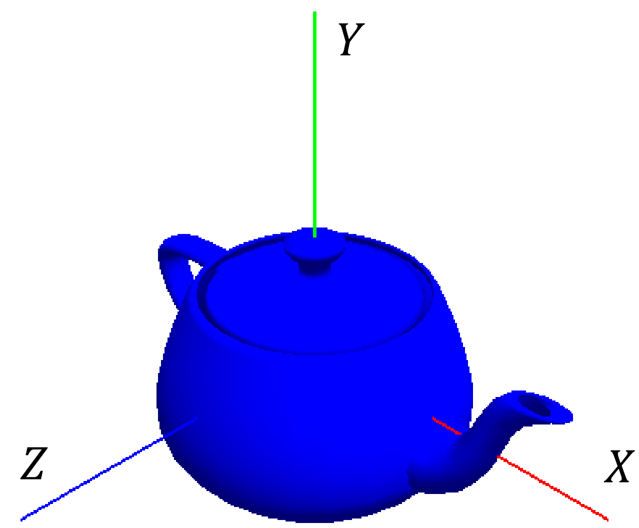
- Why do we care about coordinate spaces?
- OpenGL is, ultimately, an API that ***draws pixels*** on the screen
- How to express *"I want to draw a triangle?"*
- To make this problem scalable and easy to human, we ***break down*** (divide-and-conquer) the process and model it as a ***series of space transformation***

Coordinate Spaces

Coordinate Space	Type
Object space	Right-handed coordinate
↓ Model matrix	
World space	Right-handed coordinate
↓ View matrix	
Eye space	Right-handed coordinate
↓ Projection matrix	
Clip space	Left-handed coordinate
↓ Perspective division	
Normalized device space	Left-handed coordinate
↓ Viewport transformation	
Screen space	Left-handed coordinate

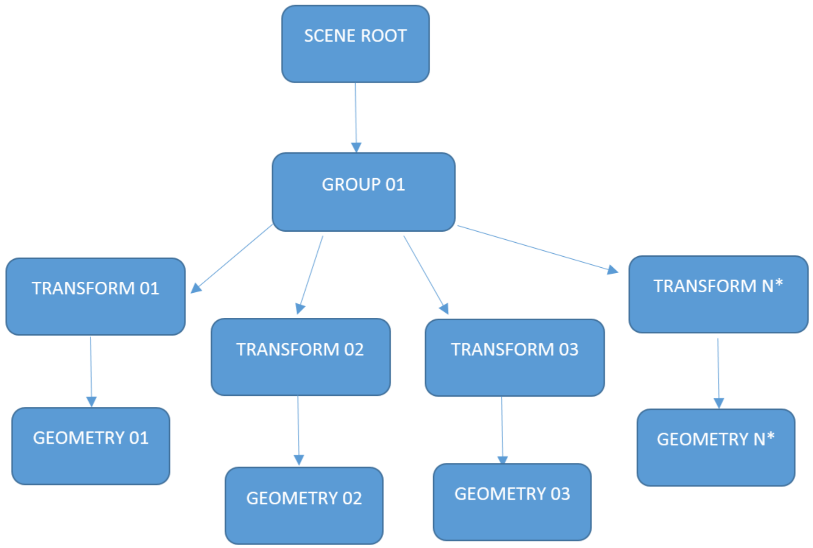
Object Space

- Also called ***model space***
- Raw input points. The position is relative to a local origin. Positions are supposed to be in right-handed coordinate
- Your input model ***may be*** in left-handed coordinate. If so, you need to ***convert*** it



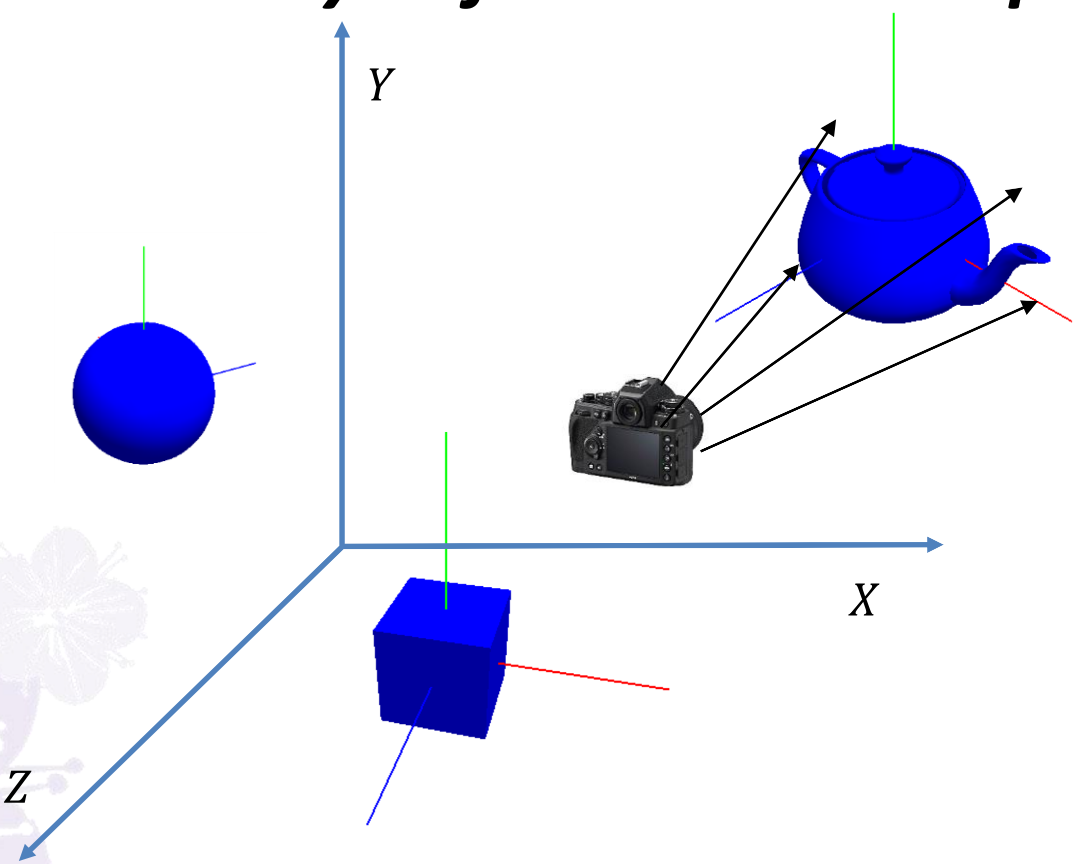
Object Space -> World Space

- Apply transformations (translation, scaling and rotation) to each different object space
- This is often modeled as a graph or a tree, and referred to as a ***scene graph***



World Space

- Points from different object spaces
- ***Camera is usually defined in world space***



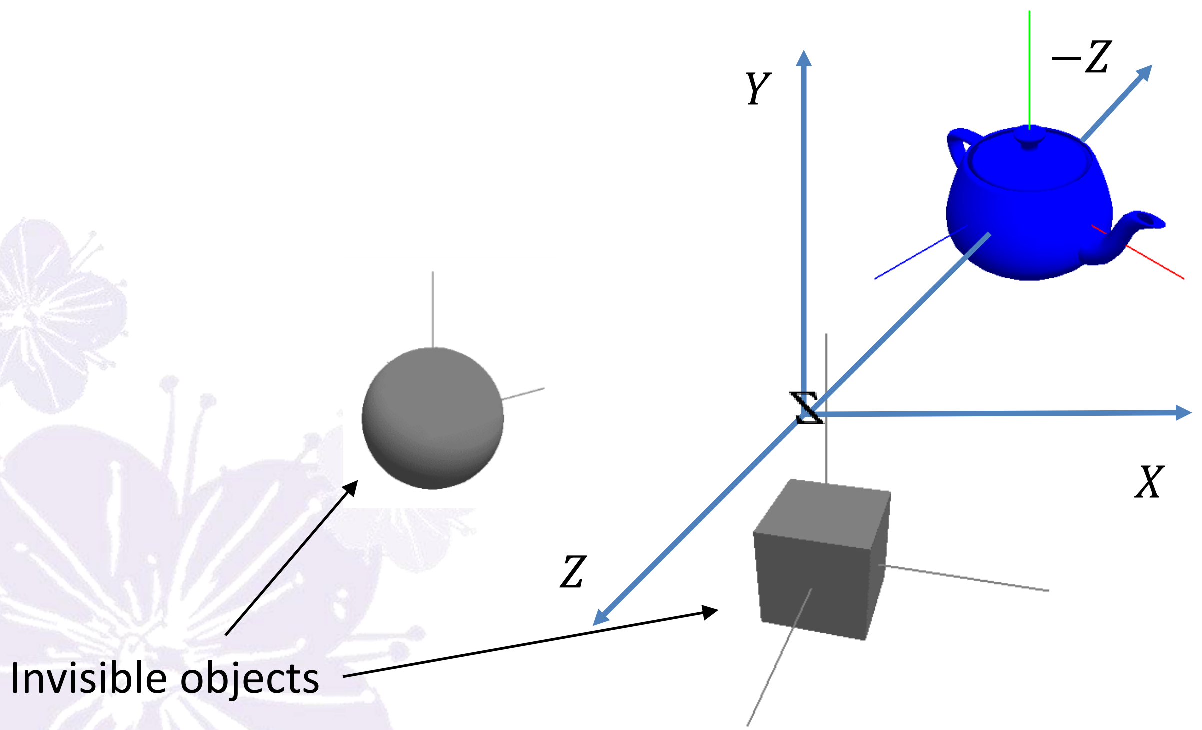
World Space -> Eye Space

- Use viewing matrix

$$M_{viewing} = \begin{bmatrix} S_x & U'_x & -F_x & 0 \\ S_y & U'_y & -F_y & 0 \\ S_z & U'_z & -F_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Eye Space

- Also called ***camera space*** or ***view space***
- Everything in front of camera (or all visible objects) is moved to ***-z axis*** in eye space



Eye Space -> Clip Space

- Use projection matrices
  - Perspective or orthographic

Orthographic

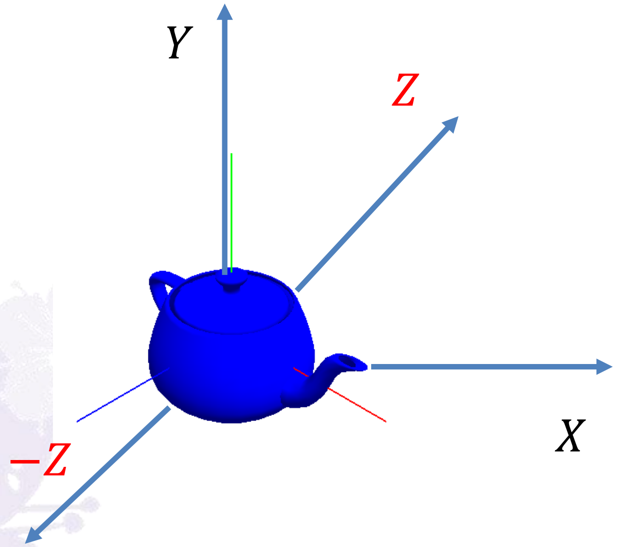
$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Clip Space

- After ***projection transformation***, it is left-handed coordinate and the ***center of the view volume is moved to origin***
- In vertex shaders, you assign a ***clip space point*** to the predefined output variable ***gl_Position***



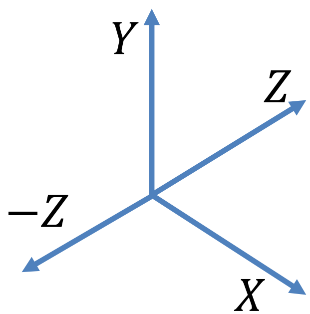
The shape (spaceship) is not changed;
But the coordinate system (universe) around it is changed

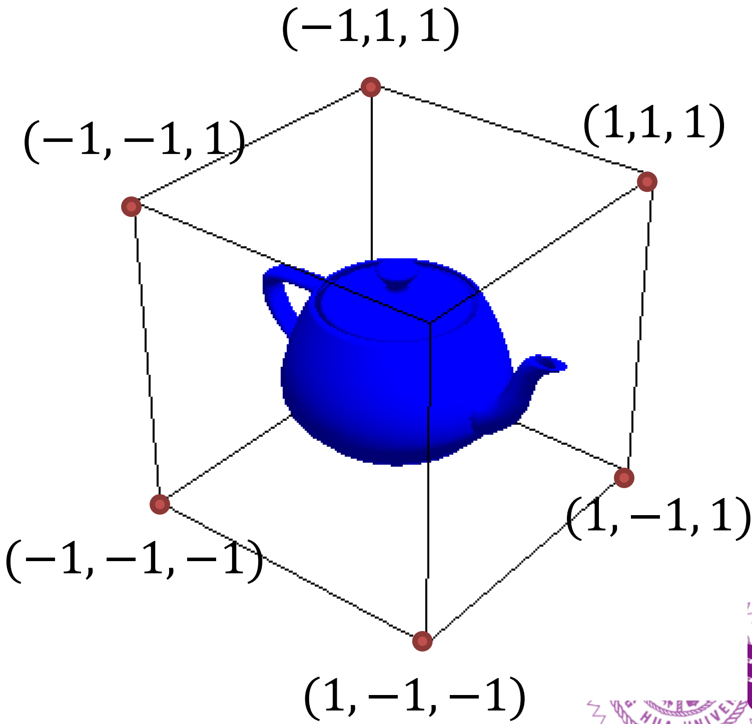
Clip Space -> NDC

- OpenGL will automatically and forcibly perform ***perspective division*** to transform points to normalized device space
- Perspective division is, actually, very simple:
- $(x, y, z, w) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, \frac{w}{w}\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right)$
- What happens if $w = 0$? OpenGL treats it as $(\infty, \infty, \infty, 0)$, which is out of visible range

Normalized Device Space

- Everything visible in the final output is inside the cube $\{(x, y, z)|x, y, z = [-1,1]\}$
- The near plane in the view volume is now $z = -1$, and the far plane is $z = 1$

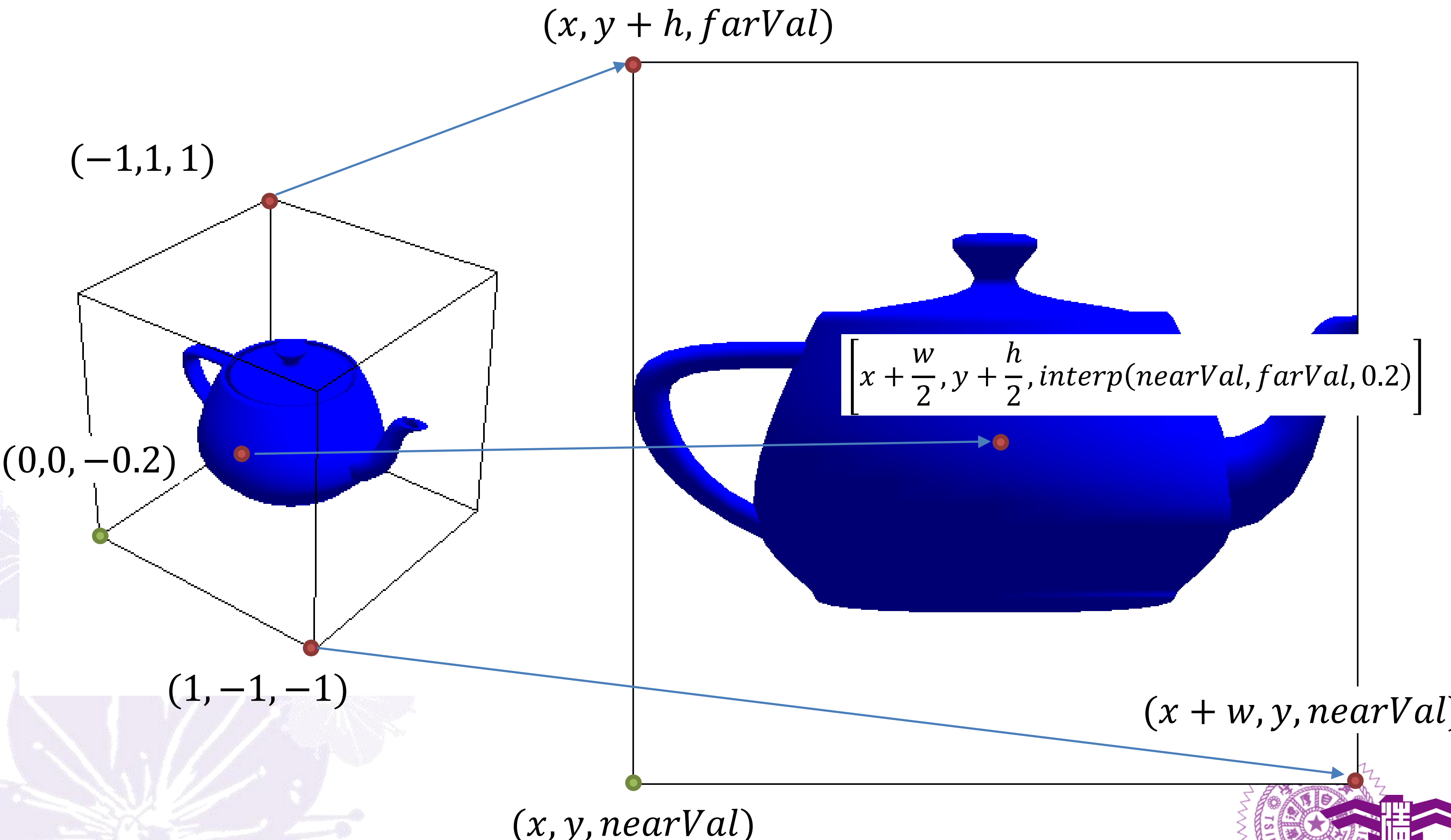




NDC -> Screen Space

- ***Viewport transformation***
- The viewport transformation is determined by the parameters set by $glViewport(x, y, w, h)$ and $glDepthRange(nearVal, farVal)$
- $glViewport(x, y, w, h)$ is used to map x, y value of vertices
- $glDepthRange(nearVal, farVal)$ is used to map z value of vertices

NDC -> Screen Space



Screen Space

- Each point is transformed into its ***actual*** pixel position in the window with a depth value
- The final pixel position (x, y) is in the rectangle area defined by the $glViewport(x, y, w, h)$ call
- The final pixel depth is in the range $[0,1]$

Coordinate Spaces

Coordinate Space	Type
Object space	Right-handed coordinate
↓ Model matrix	
World space	Right-handed coordinate
↓ View matrix	
Eye space	Right-handed coordinate
↓ Projection matrix	
Clip space	Left-handed coordinate
↓ Perspective division	
Normalized device space	Left-handed coordinate
↓ Viewport transformation	
Screen space	Left-handed coordinate

← Performed in main program or programmable shader by the user

← Performed automatically (and forcibly) by OpenGL

Matrix Transformation

- So, after all, how do I program these coordinate space blah blah blah…?
- **V’ = Projection * Viewing * Modeling * V**

main.cpp

```
mat4 model;
mat4 view;
mat4 proj;
…
mat4 mvp = proj * view * model;
glUniformMatrix4fv(0, 1, GL_FALSE, &mvp[0][0]);
```

vert.glsl

```
mat4 um4mvp;
in vec4 vertex;
void main()
{
    gl_Position = um4mvp * vertex;
}
```

Questions You Might Ask…

- **Q1**: Why do we put objects to ***-z axis*** in eye space? Isn't ***+z axis*** good?
- **Answer:** In fact, -z axis of RHC is +z axis in LHC. Visible objects are always on +z axis in LHC. It is just a ***convention*** to use RHC for model, world and eye space, and LHC for clip, NDC and screen space. You don't need to follow this convention

Questions You Might Ask…

- **Q2**: Some code I found is not the same as this slide! The ***viewing matrix*** and the ***projection matrix*** are different! Who is right?
- **Answer:** As mentioned in Q1, some people decided to use LHC for all the coordinates (In fact, Direct X does so). They put visible objects on ***+z axis*** and don't do ***RHC/LHC conversion*** in projection matrix. As long as your coordinate system ends up in ***LHC***, the result will be the same

Questions You Might Ask…

- **Q3:** Why do OpenGL use $z = [-1,1]$ in NDC? The final visible z value is in range $[0,1]$ anyway? Why not just use $z = [0,1]$ in NDC?
- **Answer:** The NDC is a normalized representation such that each component is in range $[-1,1]$. A $z = [0,1]$ range means that we will have a ***rectangle*** NDC, which is more complicated