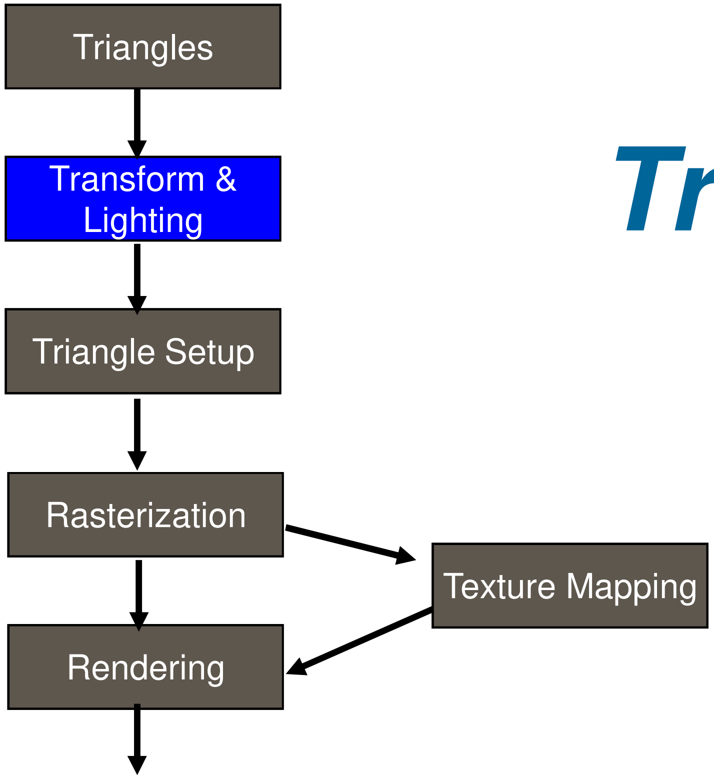
Computer Graphics

by Ruen-Rone Lee
ICL/ITRI

Wrap up from last class

- Primitives
- Rasterization – Generate Pixels that Constitute the Primitive
  - Line Rasterization
  - Triangle Rasterization
- Deriving Attributes (Colors, Depths, Texture Coordinates) of Pixels inside a Triangle

Part I:
Conventional 3D Graphics Pipeline

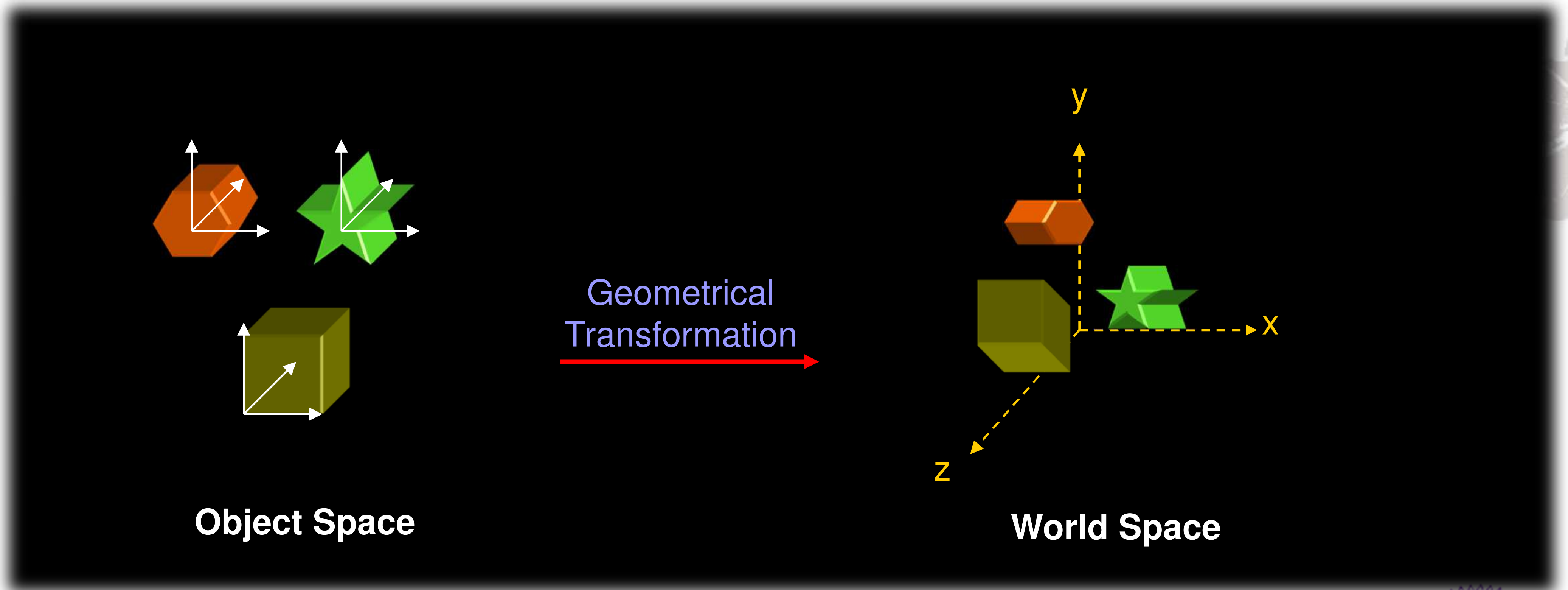


Transformation

3D Viewing Process

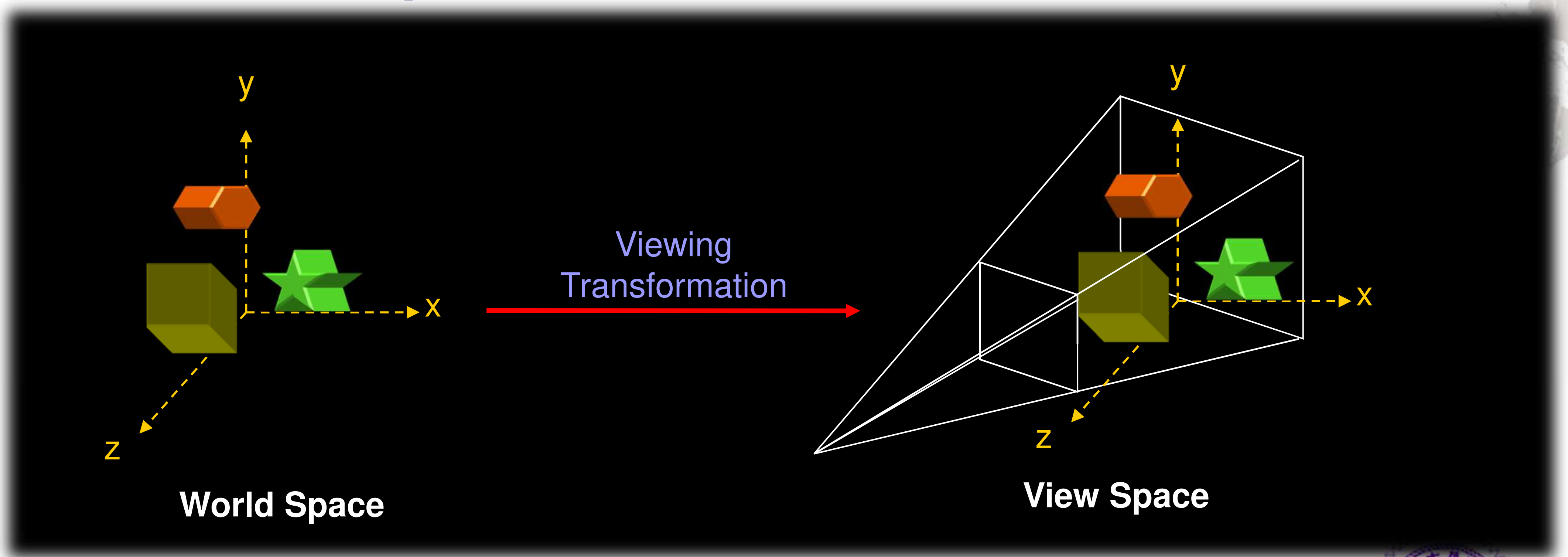
3D Viewing Process

- Geometrical Transformation
  - From Object Space to World Space



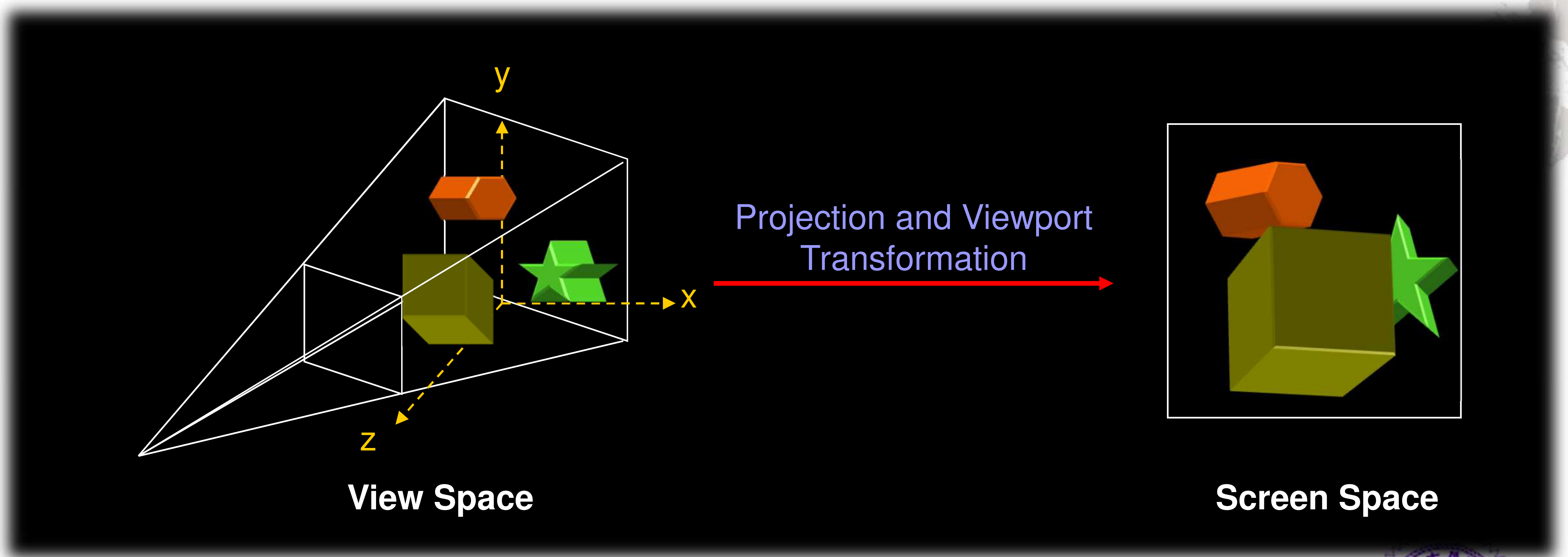
3D Viewing Process

- Viewing Transformation
  - From World Space to View Space (Eye Space or Camera Space)



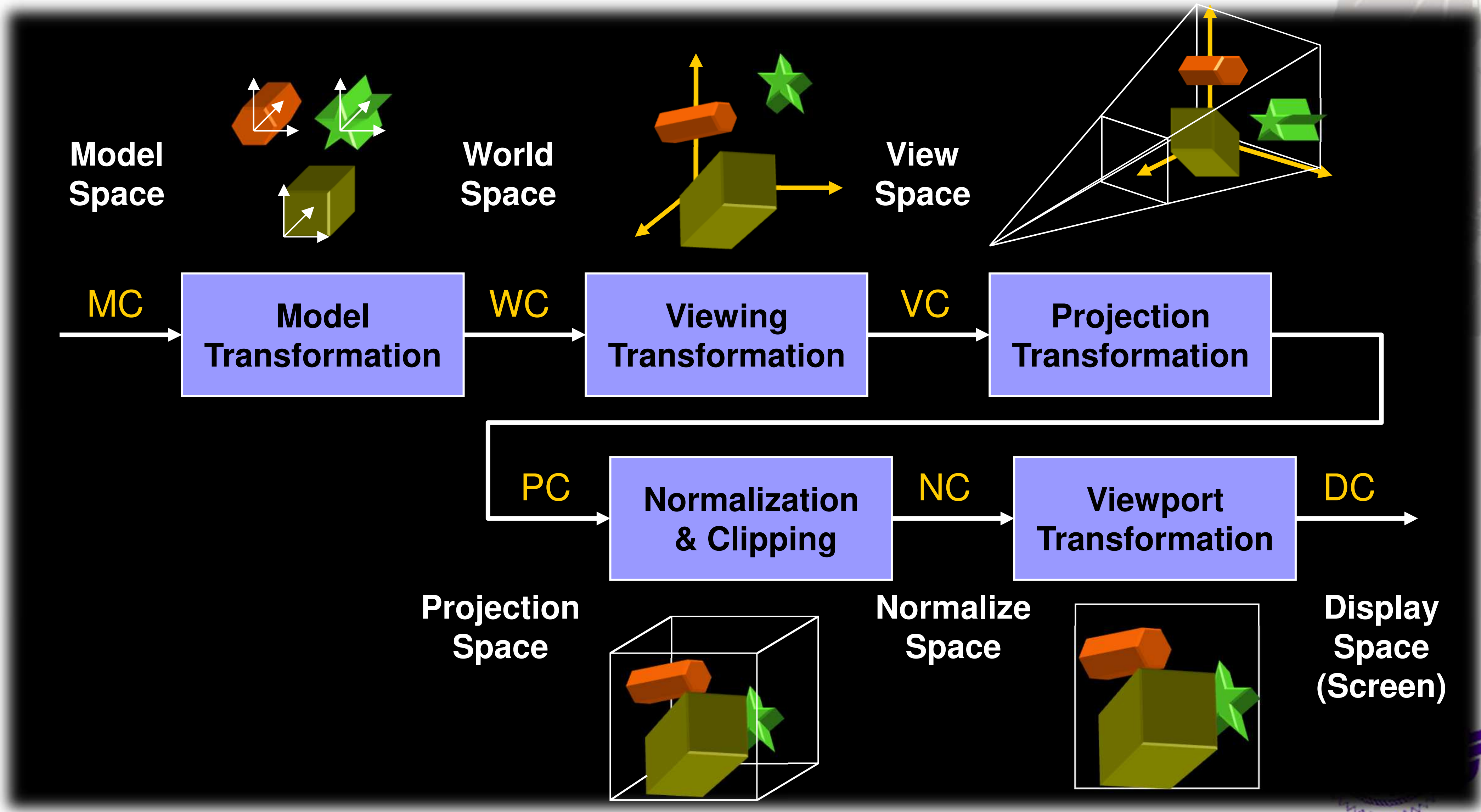
3D Viewing Process

- Projection and Viewport Transformation
  - From View Space (3D) to Screen Space (2D)



3D Viewing Process

- Coordinates Transformation

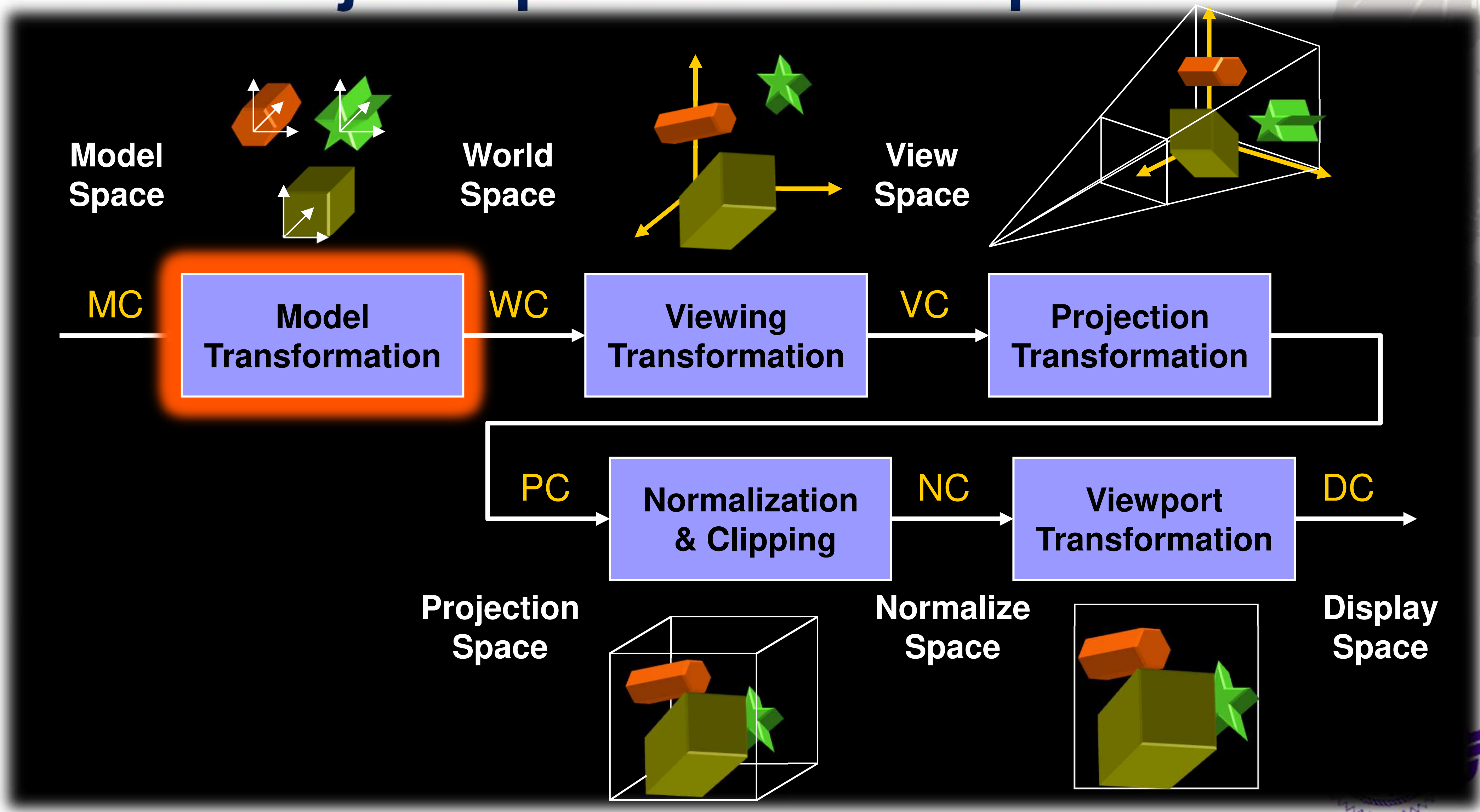


Geometrical Transformations

From Object Space to World Space

Geometrical Transformation

- From Object Space to World Space



2D Transformation

- **Translation**

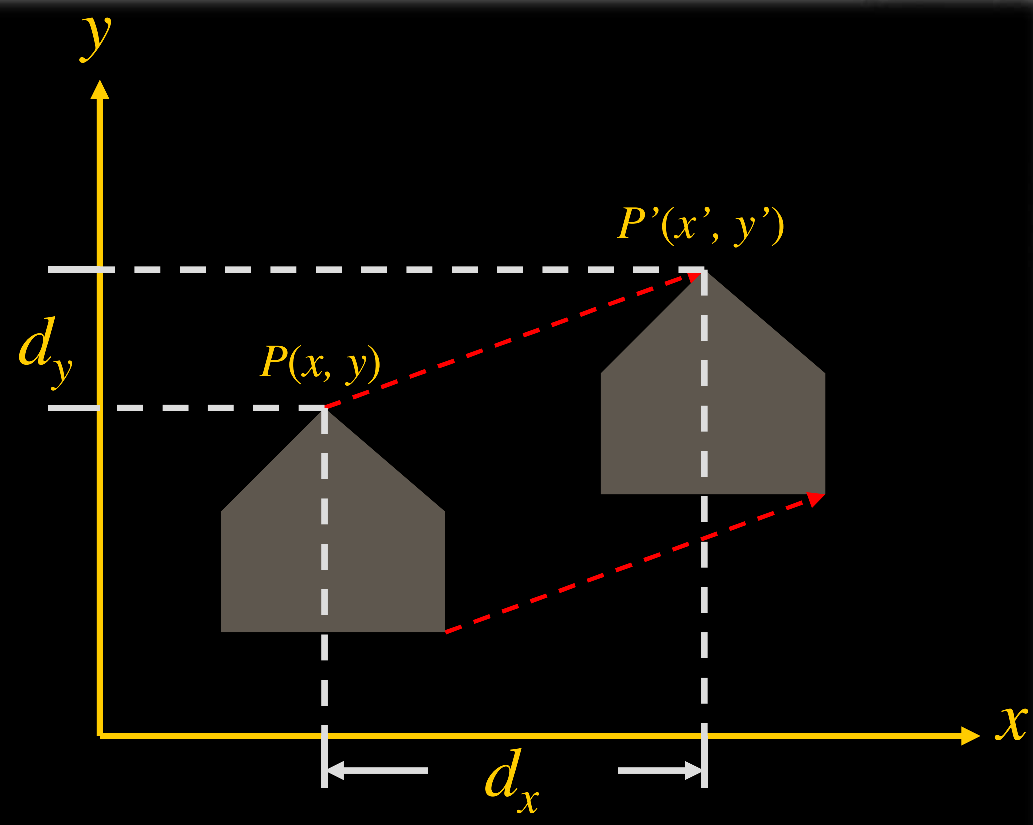
$P(x, y)$ move to $P'(x', y')$

$$x' = x + d_x$$

$$y' = y + d_y$$

$$P = \begin{bmatrix} x \\ y \end{bmatrix},\quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix},\quad T = \begin{bmatrix} d_x \\ d_y \end{bmatrix}$$

$$P' = P + T(d_x, d_y)$$



2D Transformation

- **Scaling**

Scale by s_x along the x axis

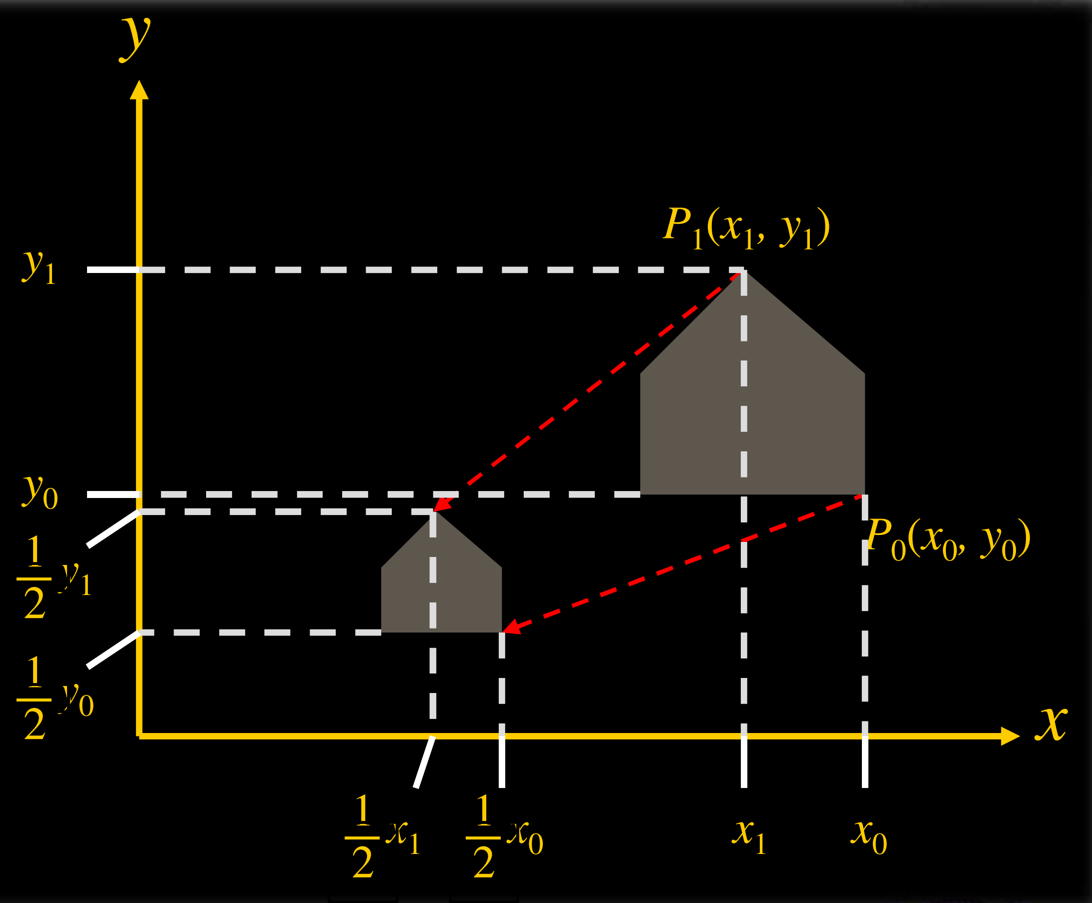
and by s_y along the y axis

$$x' = s_x \cdot x$$

$$y' = s_y \cdot y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$P' = S(s_x, s_y) \cdot P$$



2D Transformation

- **Rotation**

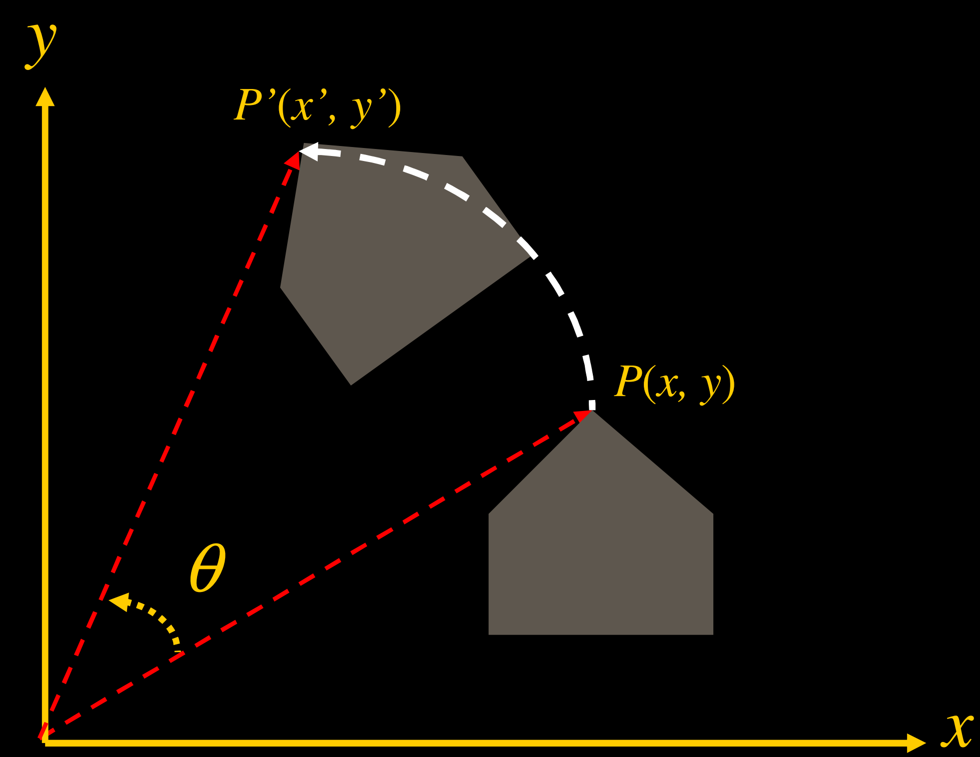
Rotate by an angle θ with respect to the origin

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

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

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$P' = R(\theta) \cdot P$$



2D Transformation

- Derivation of the rotation equation

$$x = r \cdot \cos\phi$$

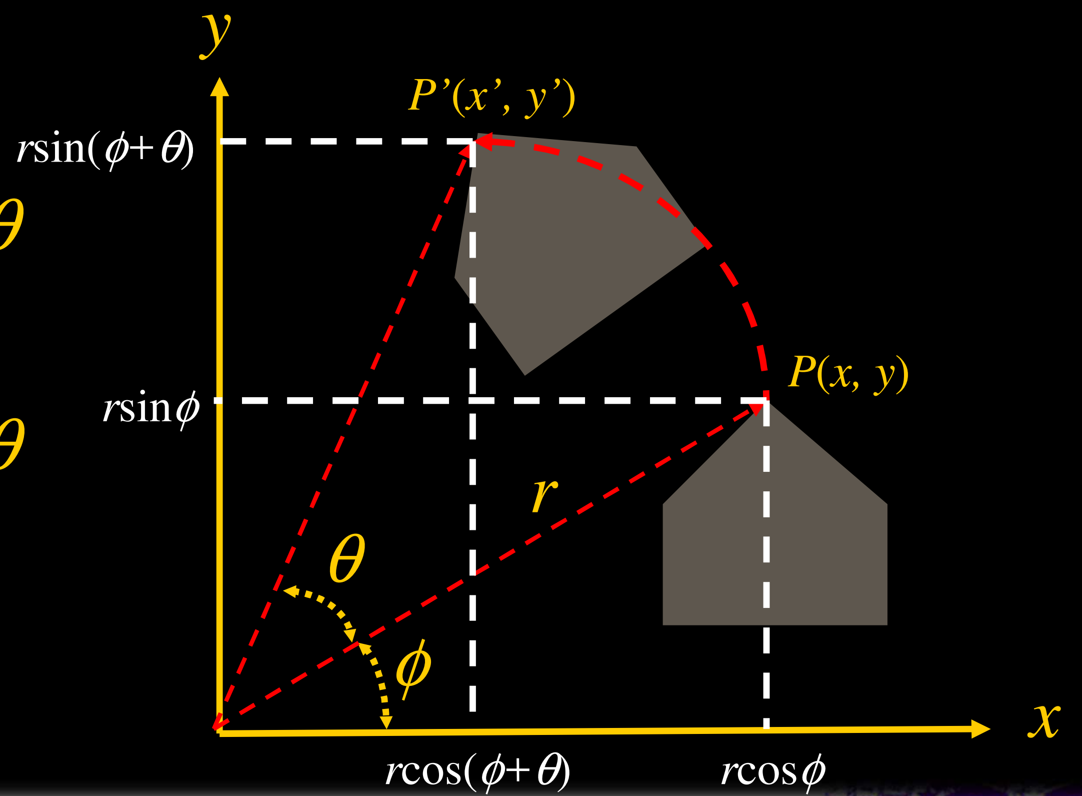
$$y = r \cdot \sin\phi$$

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

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

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

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



Homogeneous Coordinates

- **Matrix representations for translation, scaling, and rotation**

$$P' = T + P \qquad \text{Translation}$$

$$P' = S \cdot P \qquad \text{Scaling}$$

$$P' = R \cdot P \qquad \text{Rotation}$$

- **How to derive a consistent way to represent all the transformations?**

Homogeneous coordinate representation

Homogeneous Coordinates

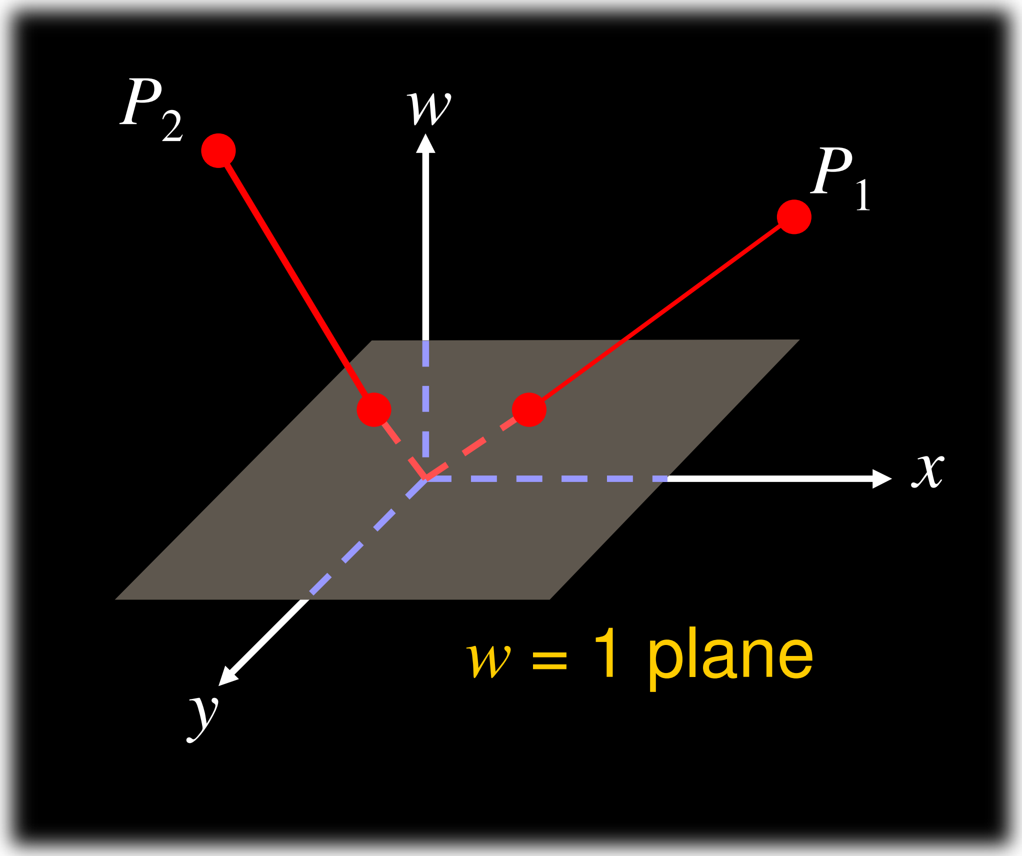
- A point (x, y) in 2D coordinates is represented by (x, y, W) in homogeneous coordinates
- Two homogeneous coordinates, (x, y, W) and (x', y', W'), represent the same point if and only if $(x', y', W') = (tx, ty, tW)$ for some $t \neq 0$
- Homogeneous coordinates to Cartesian Coordinates ($W \neq 0$)

$$(x, y, W) = (x/W, y/W, 1)$$

$$(x', y', W') = (x'/W', y'/W', 1) = (x/W, y/W, 1)$$

Homogeneous Coordinates

- All triples of the form (tx, ty, tW), with $t \neq 0$, get a line in homogeneous coordinates



Translation in Homogeneous Rep.

- 3x3 matrix form

$$x' = x + d_x \qquad y' = y + d_y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} d_x \\ d_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$P' = T(d_x, d_y) \cdot P, \qquad T(d_x, d_y) = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling in Homogeneous Rep.

- 3x3 matrix form

$$x' = s_x \cdot x$$
$$y' = s_y \cdot y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$P' = S(s_x, s_y) \cdot P, \qquad S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation in Homogeneous Rep.

- 3x3 matrix form

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

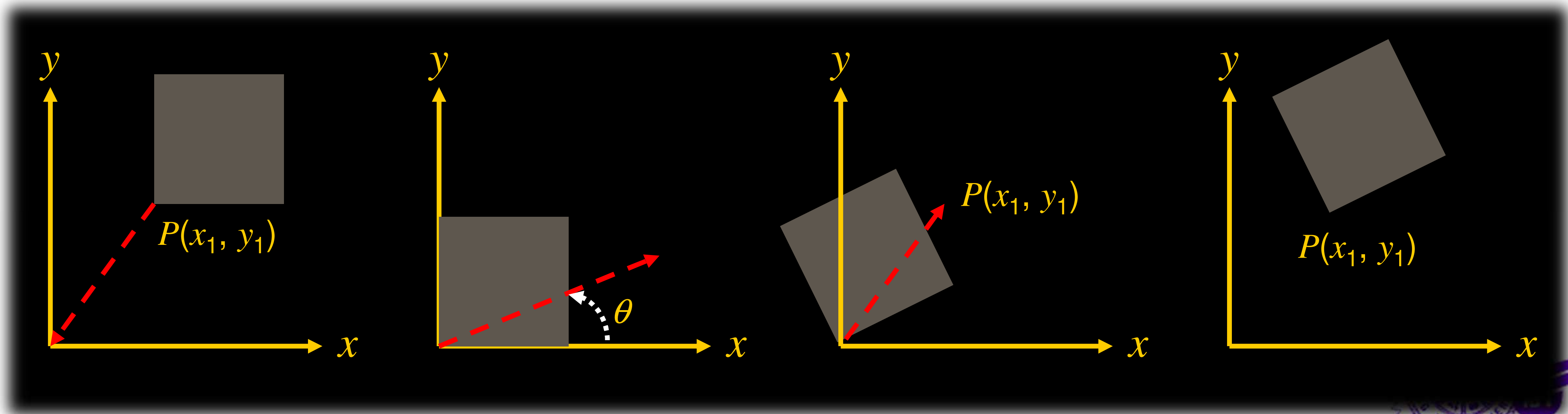
$$\dashrightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\dashrightarrow \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$P' = R(\theta) \cdot P, \qquad R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Composition of 2D Transformations

- Rotate an object about an arbitrary point $P = (x_1, y_1)$
  - Translate from P to origin
  - Rotate by θ degree
  - Translate back from origin to P



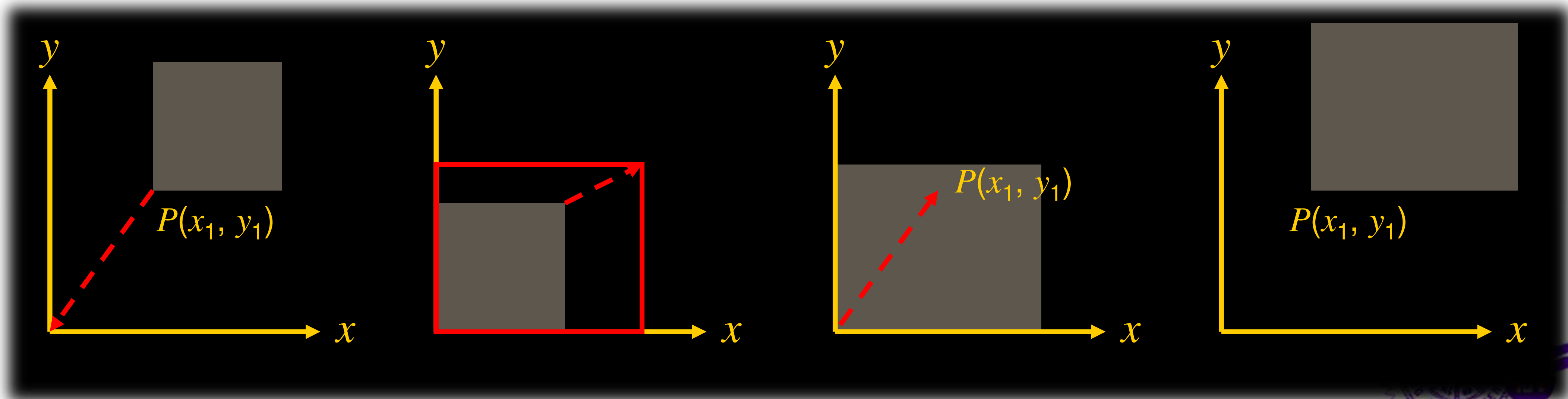
Composition of 2D Transformations

- Rotate an object about an arbitrary point $P = (x_1, y_1)$
  - Translate from P to origin, $T(-x_1, -y_1)$
  - Rotate by θ degree, $R(\theta)$
  - Translate back from origin to P, $T(x_1, y_1)$

$$T(x_1, y_1)\cdot R(\theta)\cdot T(-x_1,-y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

Composition of 2D Transformations

- Scale an object about an arbitrary point $P = (x_1, y_1)$
  - Translate from P to origin
  - Scale by (s_x, s_y)
  - Translate back from origin to P



Composition of 2D Transformations

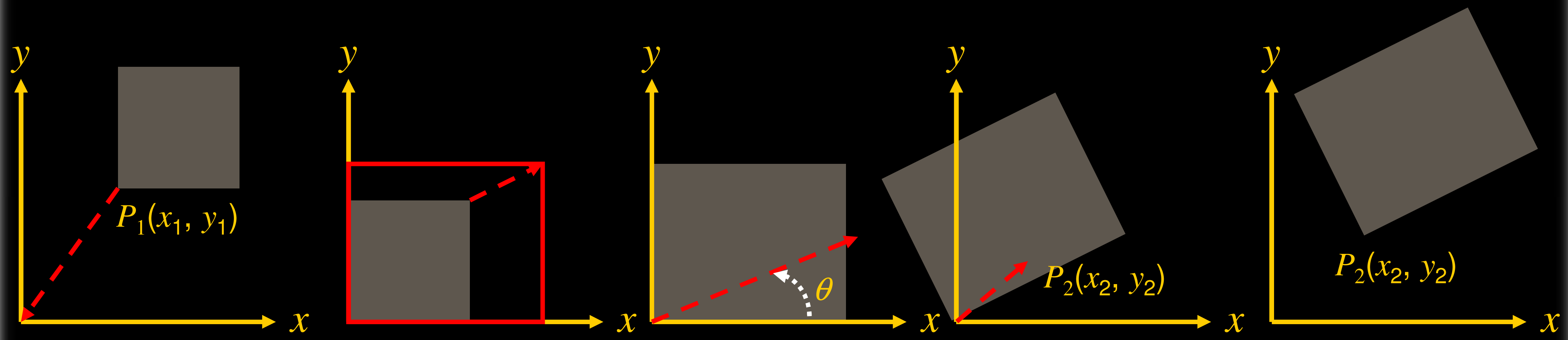
- Scale an object about an arbitrary point $P = (x_1, y_1)$
  - Translate from P to origin, $T(-x_1, -y_1)$
  - Scale by (s_x, s_y), $S(s_x, s_y)$
  - Translate back from origin to P, $T(x_1, y_1)$

$$T(x_1, y_1) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

Composition of 2D Transformations

- Scale and rotate about $P_1 = (x_1, y_1)$ and translate to point $P_2 = (x_2, y_2)$

$$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1)$$



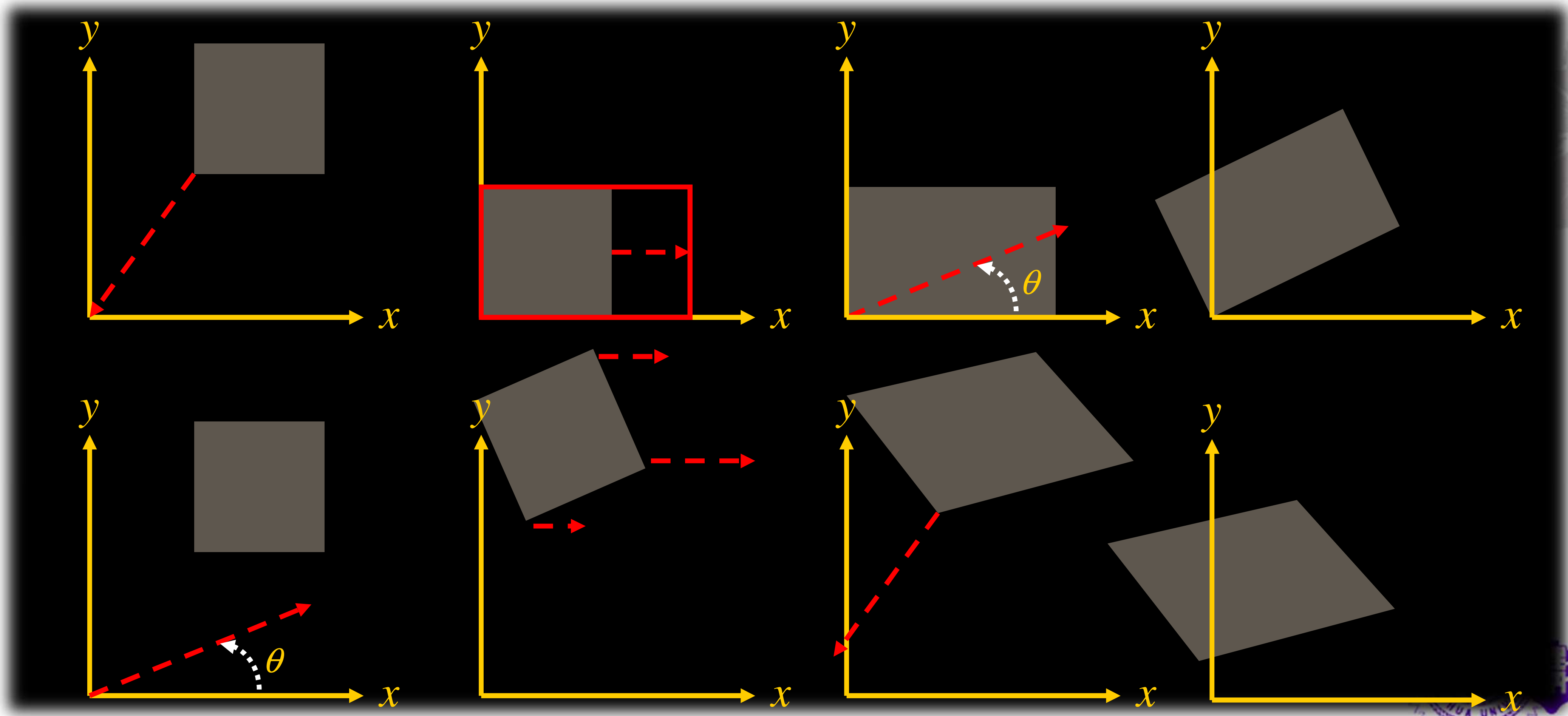
Composition of 2D Transformations

- Scale and rotate about $P_1 = (x_1, y_1)$ and translate to point $P_2 = (x_2, y_2)$
  - Translate from P_1 to origin, $T(-x_1, -y_1)$
  - Scale by (s_x, s_y), $S(s_x, s_y)$
  - Rotate by θ degree, $R(\theta)$
  - Translate from origin to P_2, $T(x_2, y_2)$

$$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1)$$

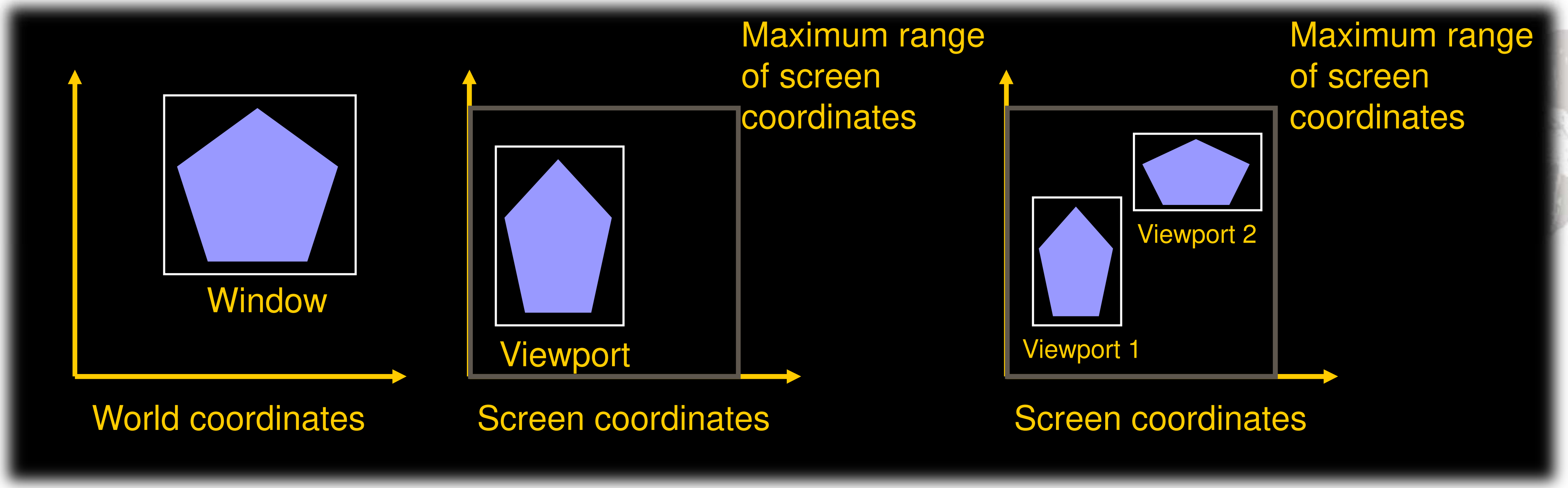
Composition of 2D Transformations

- Commutativity
  - $R(\theta)\cdot S(s_x, s_y)\cdot T(d_x, d_y) \neq T(d_x, d_y)\cdot S(s_x, s_y)\cdot R(\theta)$



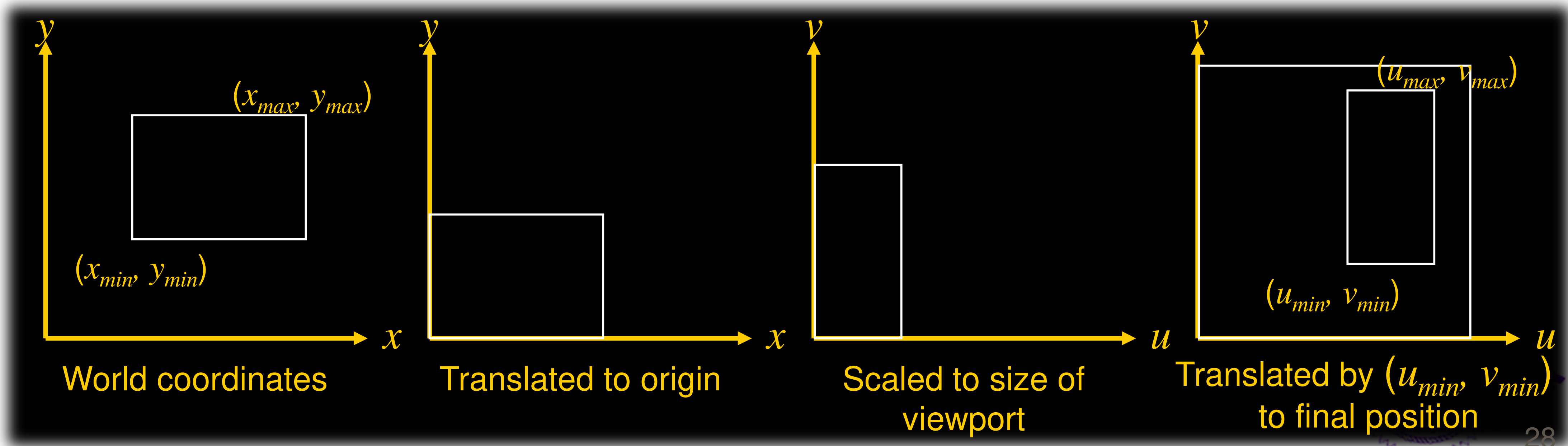
Window to Viewport Transformation

- Application of 2D Transformation



Window to Viewport Transformation

- Steps in transforming a world-coordinate window into a viewport
  - Window translated to origin
  - Window scaled to size of viewport
  - Translated by (u_{min}, v_{min}) to final position



Window to Viewport Transformation

- Composition matrix in transforming a world coordinate window into a viewport

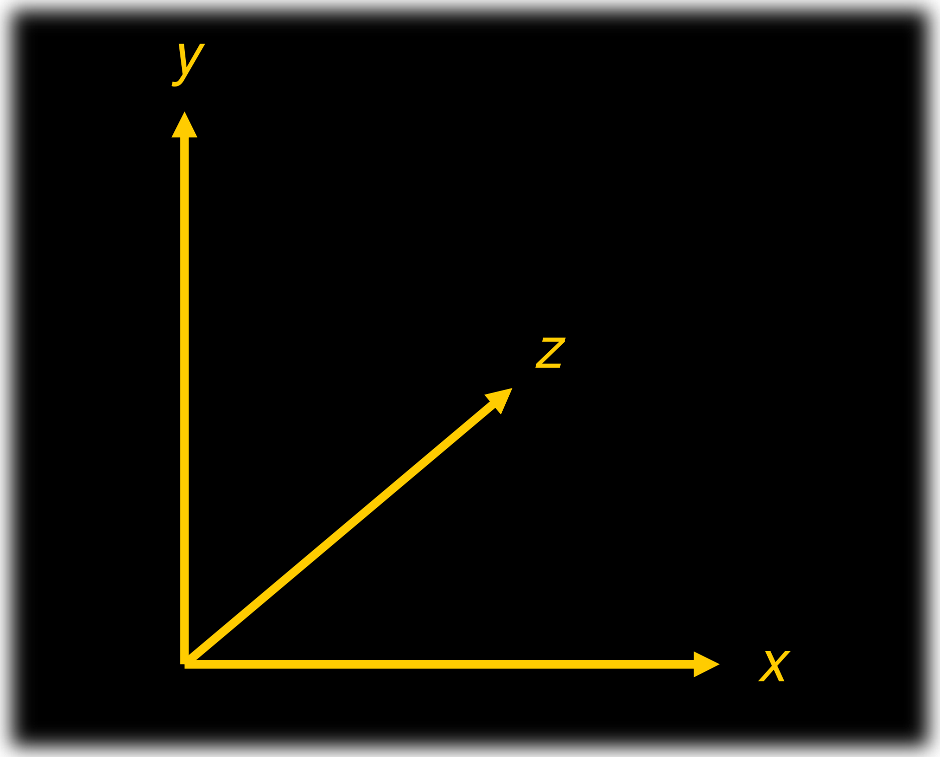
$$M_{WV} = T(u_{\min}, v_{\min}) \cdot S\left(\frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}}, \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}}\right) \cdot T(-x_{\min}, -y_{\min})$$

3D Geometrical Transformations

Similar to 2D Geometrical Transformations

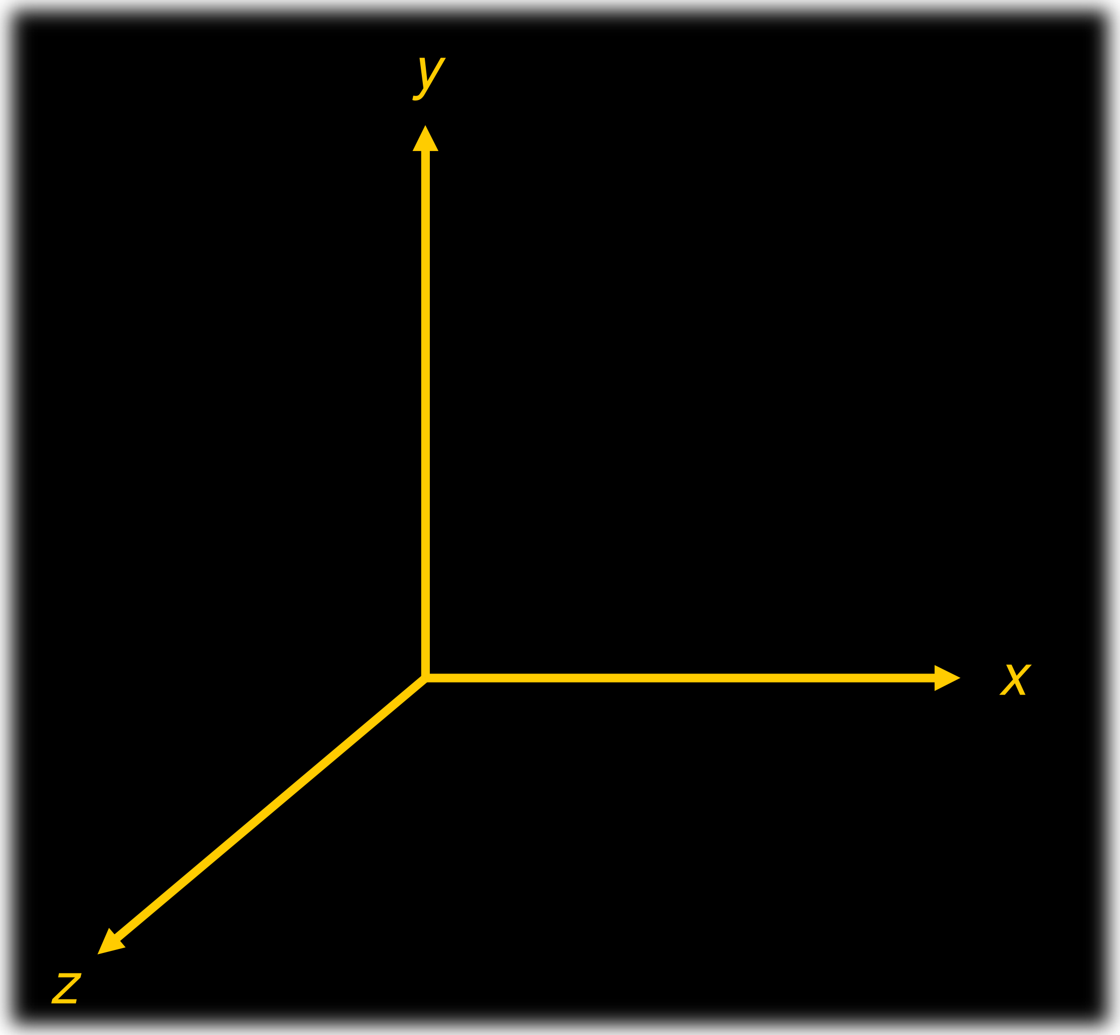
3D Coordinate Systems

- Left-hand Coordinate Systems
  - Direct3D



3D Coordinate Systems

- Right-hand Coordinate Systems
  - OpenGL



3D Transformation

- **Translation**

$$P(x, y, z) \text{ move to } P'(x', y', z')$$

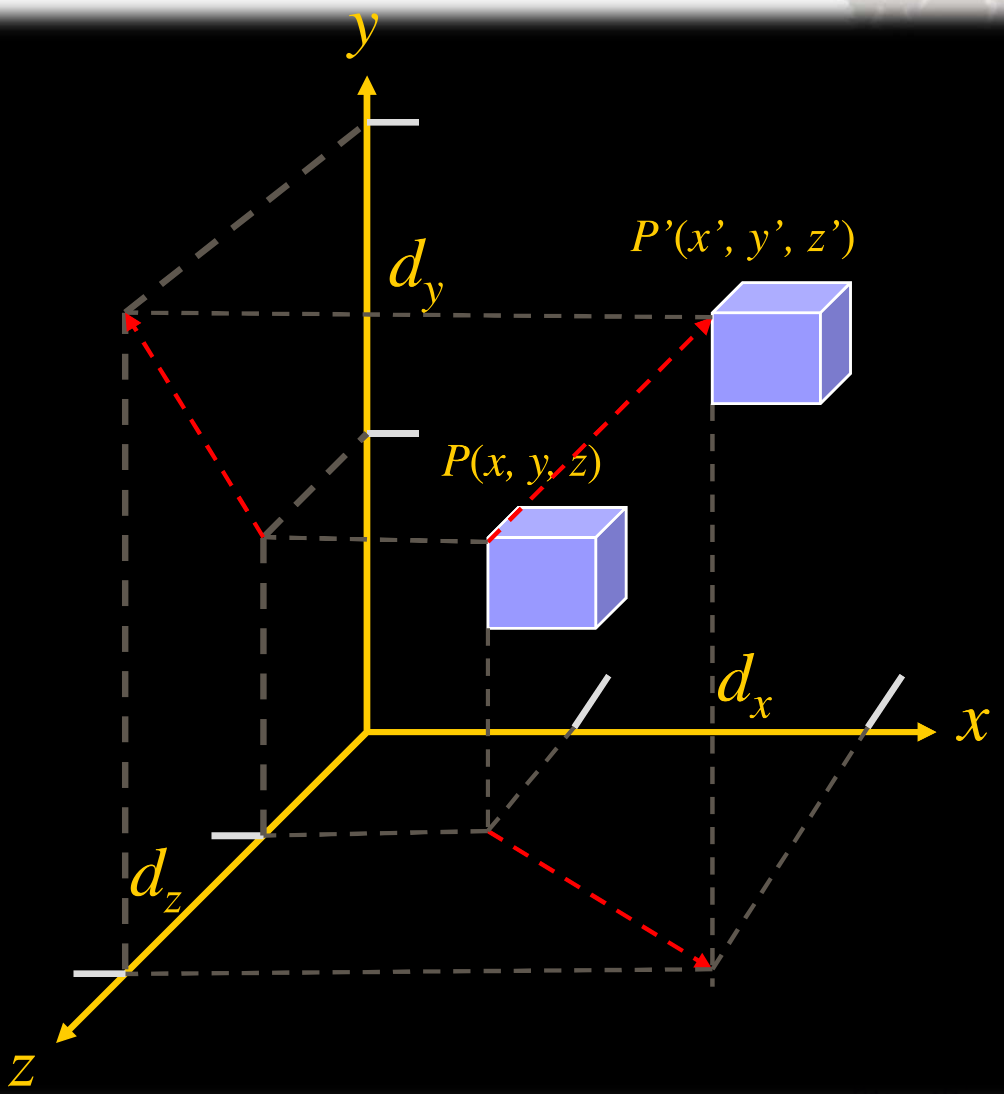
$$x' = x + d_x$$

$$y' = y + d_y$$

$$z' = z + d_z$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$P' = T(d_x, d_y, d_z) + P$$



3D Transformation

- **Scaling**

Scale by s_x, s_y, and s_z with respect to the x axis, y axis, and z axis

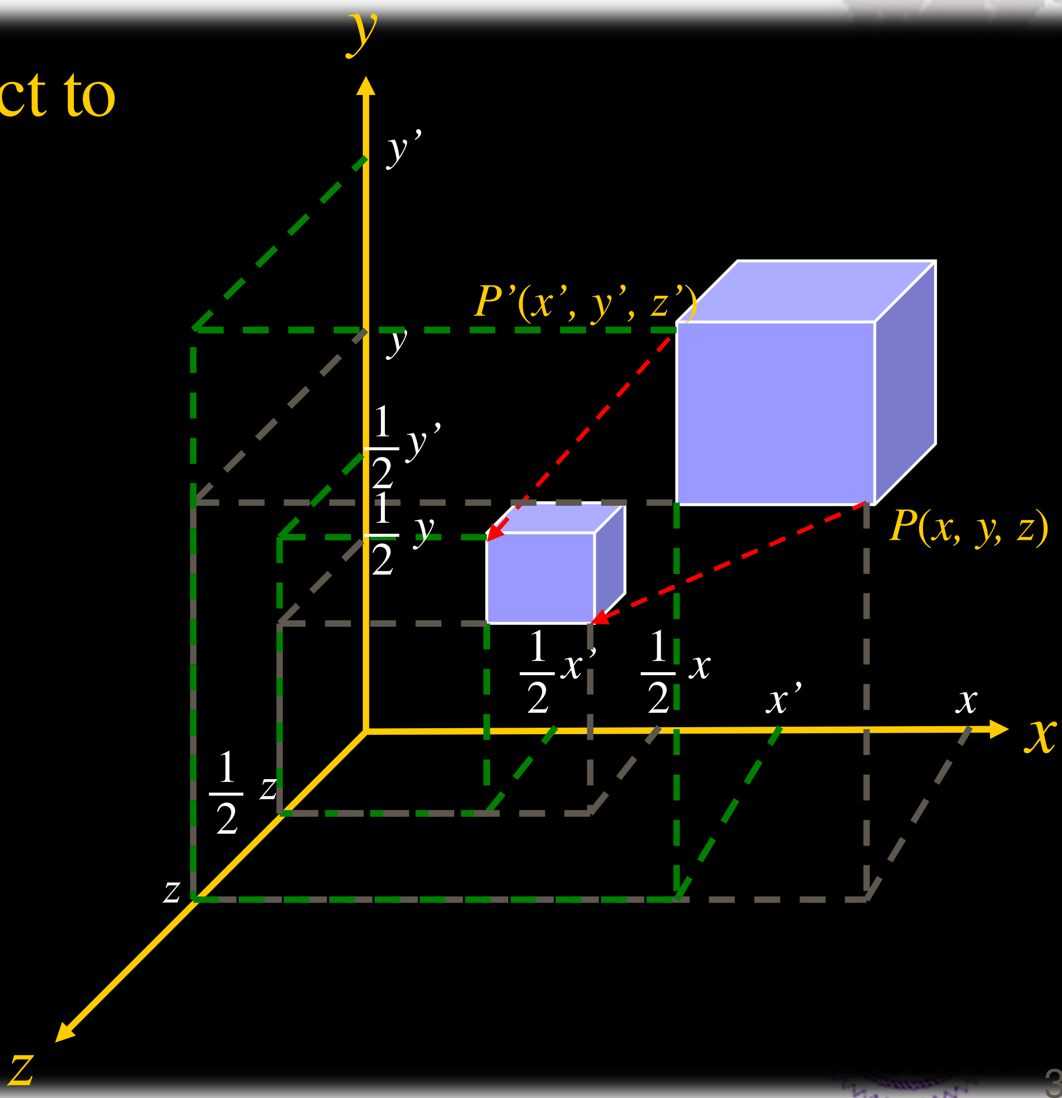
$$x' = s_x \cdot x$$

$$y' = s_y \cdot y$$

$$z' = s_z \cdot z$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$P' = S(s_x, s_y, s_z) \cdot P$$



3D Transformation

- **Rotation in x-axis**

Rotate an angle θ in x-axis

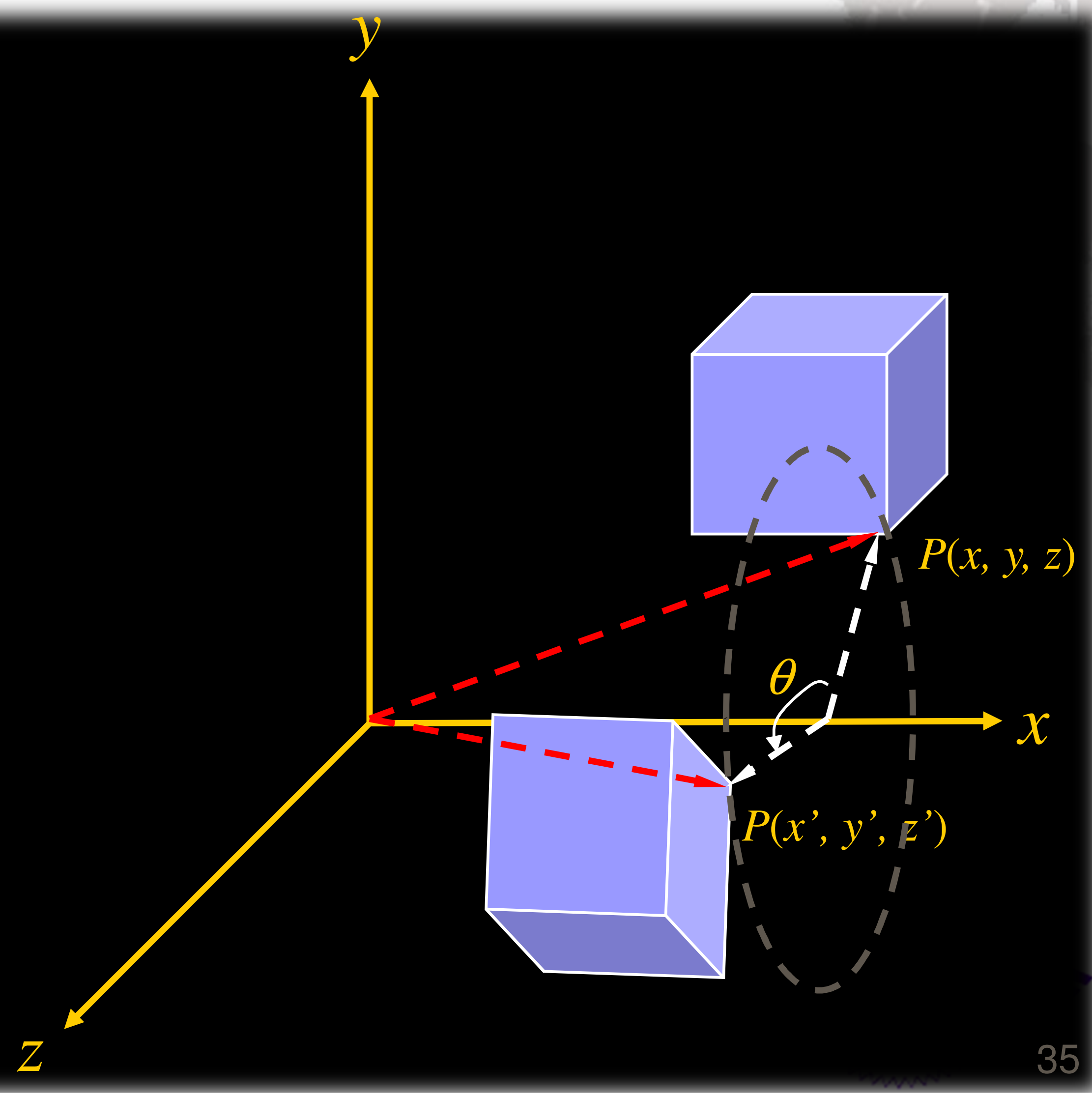
$$x' = x$$

$$y' = y \cdot \cos\theta - z \cdot \sin\theta$$

$$z' = y \cdot \sin\theta + z \cdot \cos\theta$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$P' = R_x(\theta) \cdot P$$



3D Transformation

- **Rotation in *y*-axis**

Rotate an angle θ in y-axis

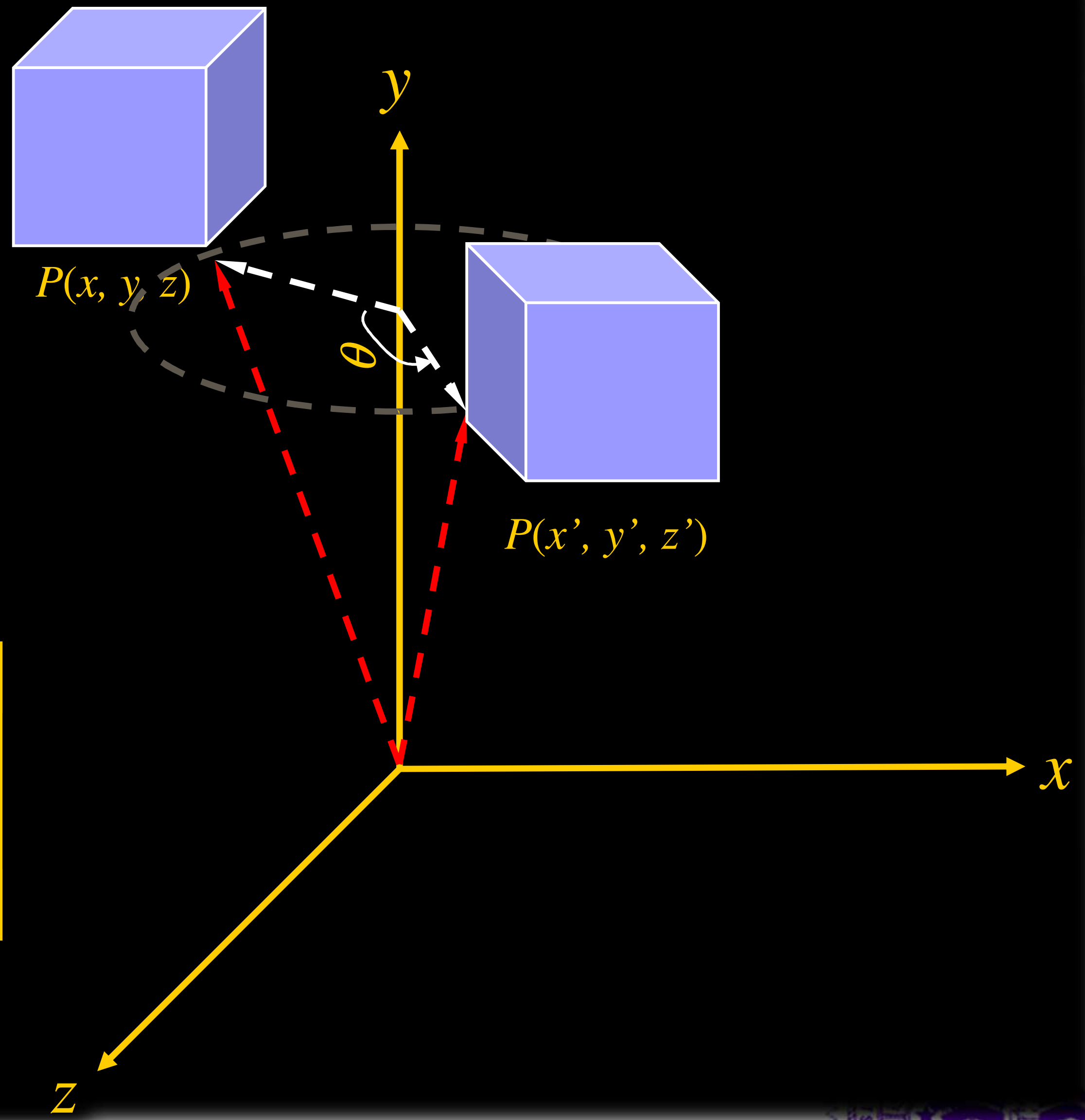
$$x' = x \cdot \cos\theta + z \cdot \sin\theta$$

$$y' = y$$

$$z' = -x \cdot \sin\theta + z \cdot \cos\theta$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$P' = R_y(\theta) \cdot P$$



3D Transformation

- **Rotation in *z*-axis**

Rotate an angle θ in z-axis

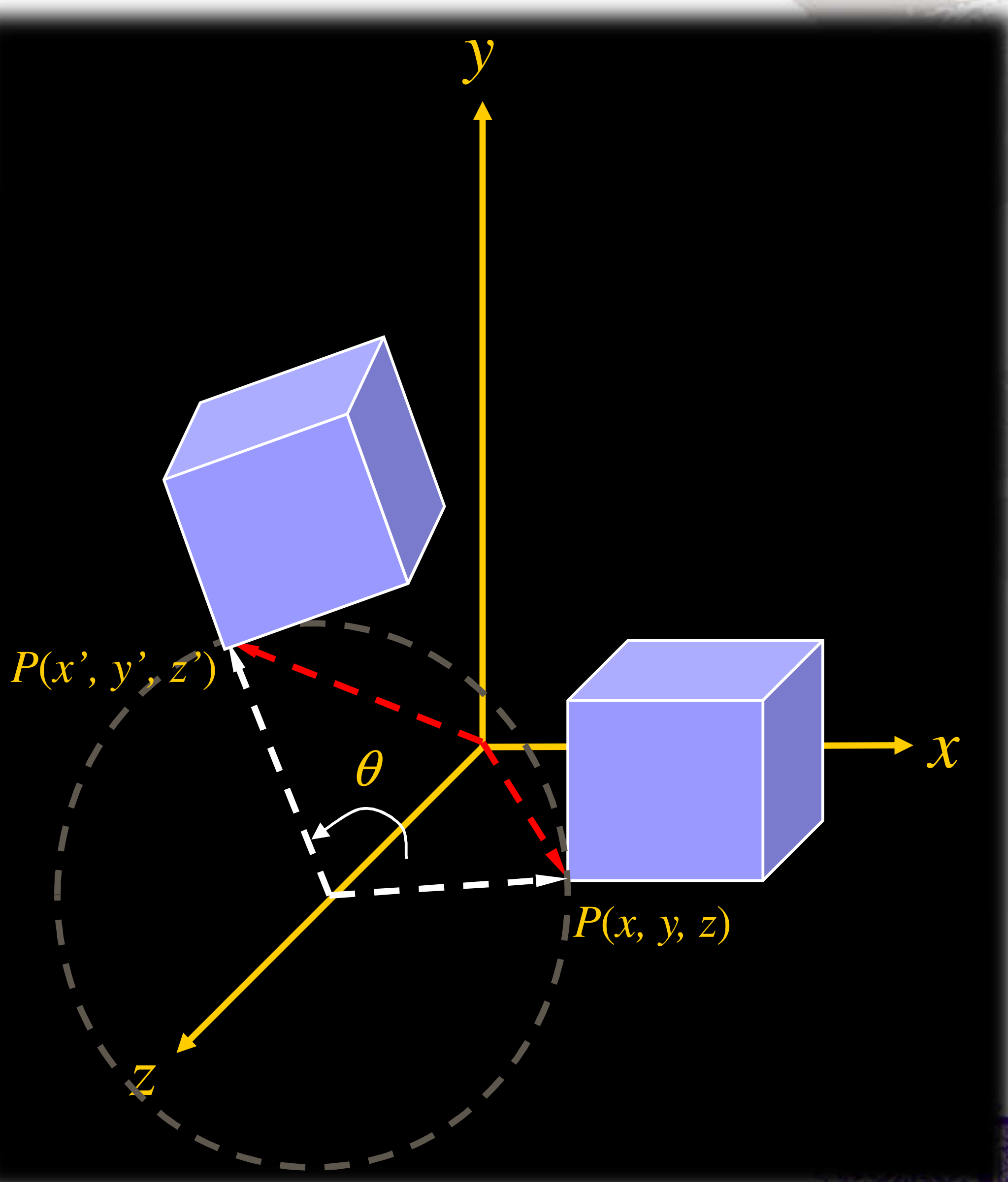
$$x' = x \cdot \cos\theta - y \cdot \sin\theta$$

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

$$z' = z$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$P' = R_z(\theta) \cdot P$$



Homogeneous Coordinates

- A point (x, y, z) in 3D coordinates is represented by (x, y, z, W) in homogeneous coordinates
- Two homogeneous coordinates (x, y, z, W) and (x', y', z', W') represent the same point if and only if $(x', y', z', W') = (tx, ty, tz, tW)$ for some $t \neq 0$
- Cartesian Coordinates

$$(x, y, z, W) = (x/W, y/W, z/W, 1)$$

$$(x', y', z', W') = (x'/W', y'/W', z'/W', 1) = (x/W, y/W, z/W, 1)$$

Translation in Homogeneous Rep.

- **4x4 matrix form**

$$x' = x + d_x \qquad y' = y + d_y \qquad z' = z + f_z$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = T(d_x, d_y, d_z) \cdot P, \qquad T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling in Homogeneous Rep.

- **4x4 matrix form**

$$x' = s_x \cdot x$$
$$y' = s_y \cdot y$$
$$z' = s_z \cdot z$$

$$\dashrightarrow \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\dashrightarrow \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = S(s_x, s_y, s_z) \cdot P$$

Rotation in Homogeneous Rep.

- **4x4 matrix form (Rotate in x-axis)**

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

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_x(\theta) \cdot P$$

Rotation in Homogeneous Rep.

- 4x4 matrix form (Rotate in y-axis)

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

$$\dashrightarrow \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\dashrightarrow \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_y(\theta) \cdot P$$

Rotation in Homogeneous Rep.

- 4x4 matrix form (Rotate in *z*-axis)

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

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_z(\theta) \cdot P$$

Composition of 3D Transformations

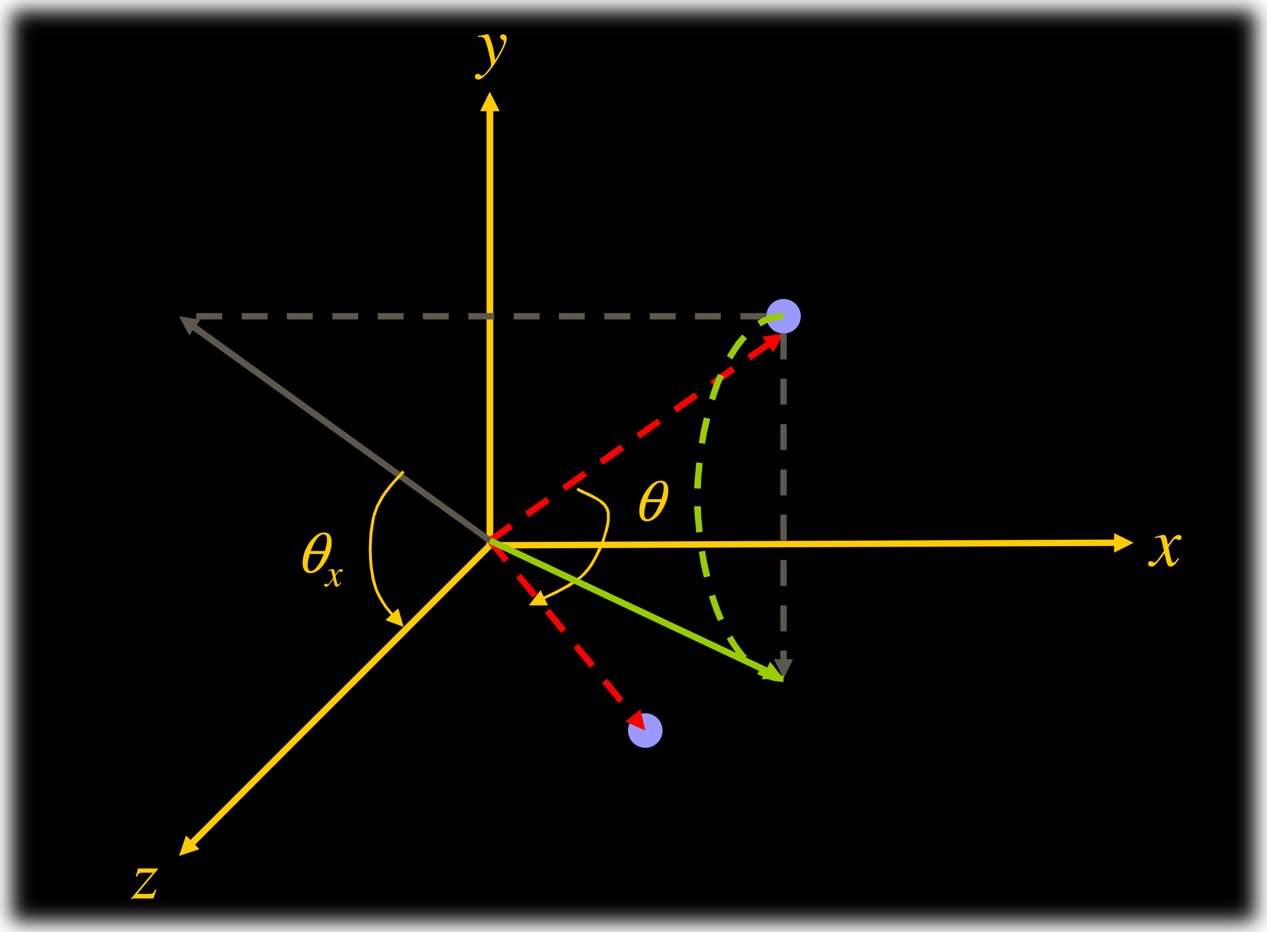
Rotation about the origin

Rotation about an arbitrary point

Rotation about an arbitrary axis

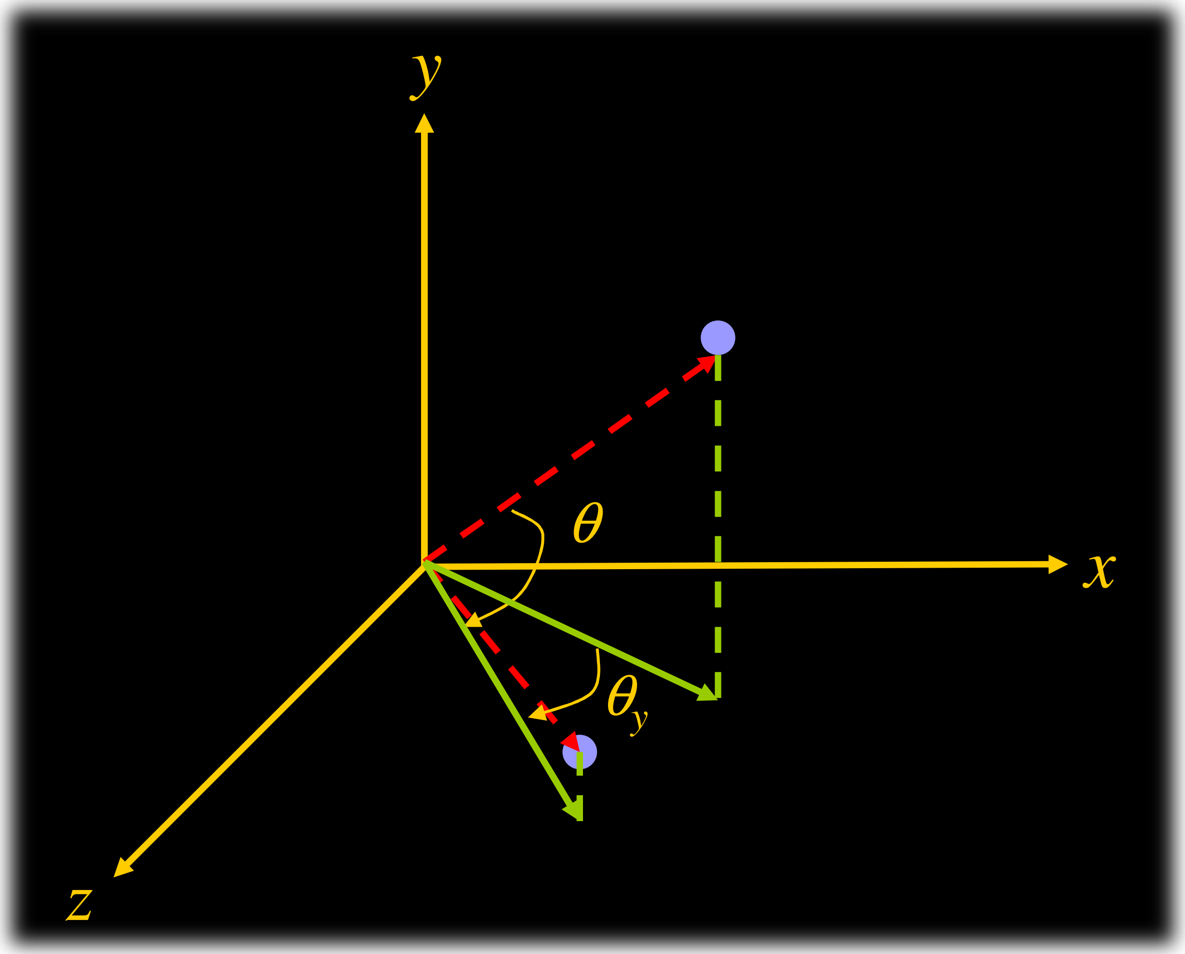
Composition of 3D Transformations

- Rotation about the origin by θ degree
  - Step 1: Rotate in x-axis by θ_x degree



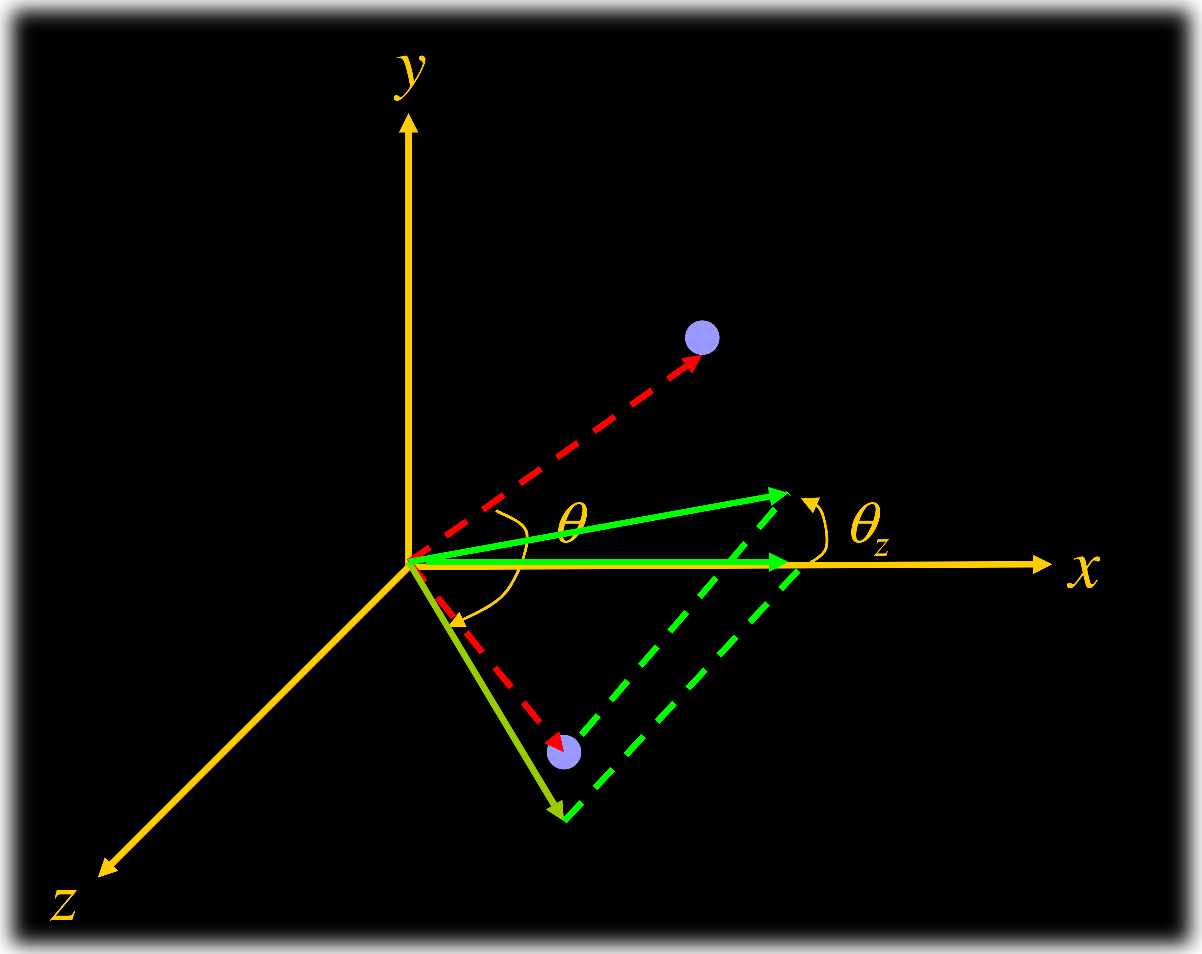
Composition of 3D Transformations

- Rotation about the origin by θ degree
  - Step 2: Rotate in y-axis by θ_y degree



Composition of 3D Transformations

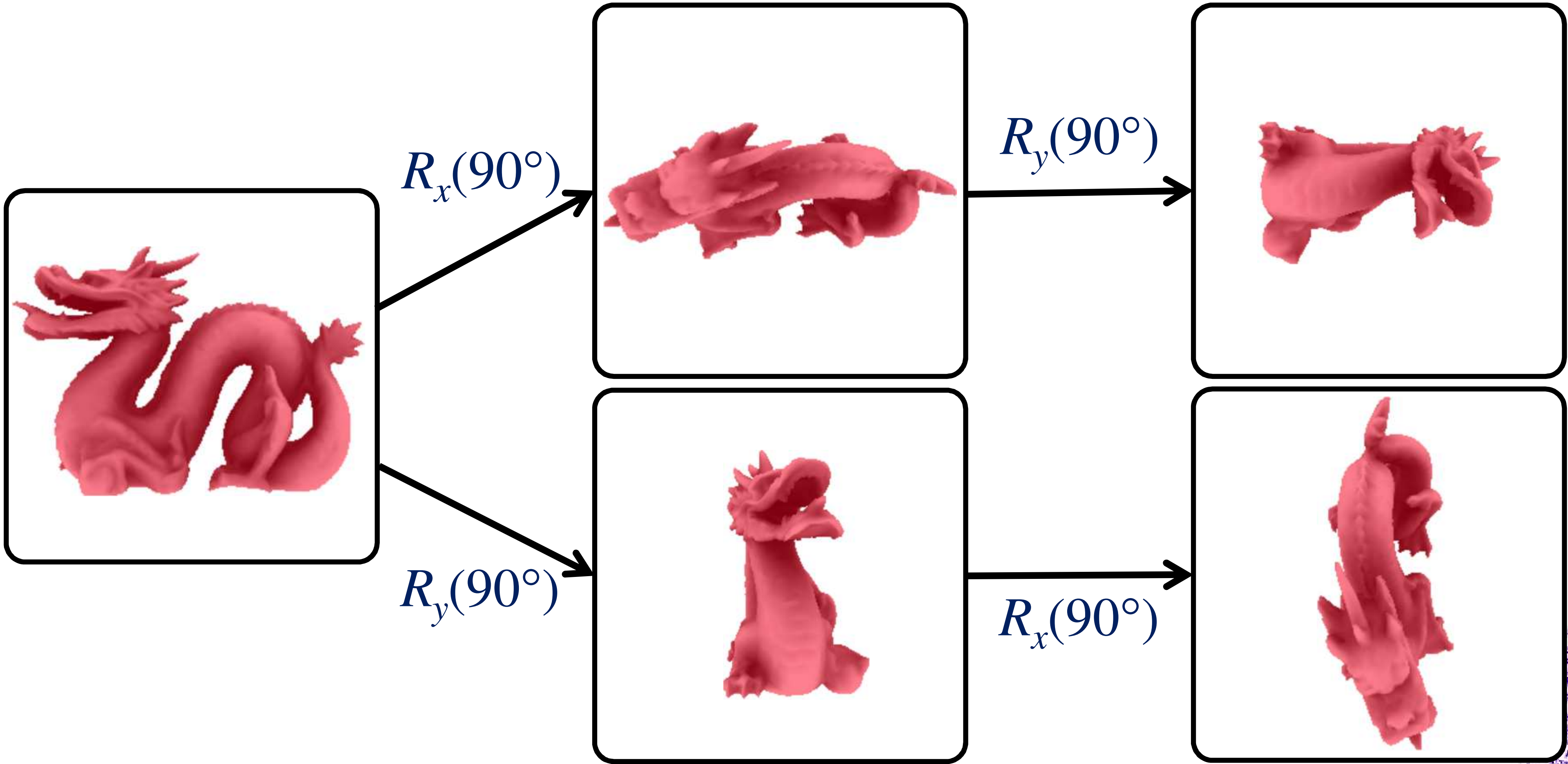
- Rotation about the origin by θ degree
  - Step 3: Rotate in z-axis by θ_z degree



Composition of 3D Transformations

- Commutative property is not held for rotation

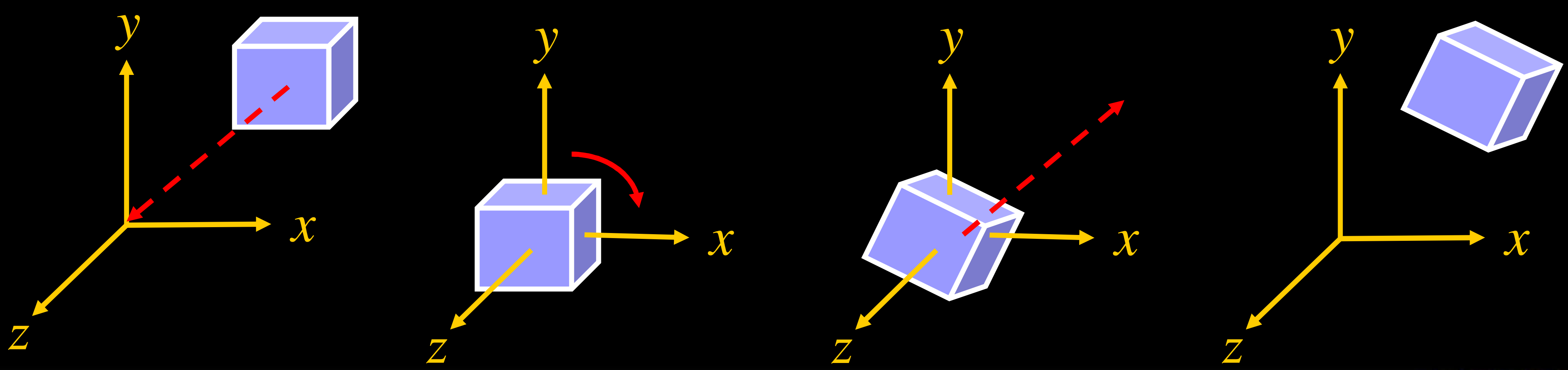
$$R_y(\theta_y)\cdot R_x(\theta_x) \neq R_x(\theta_x)\cdot R_y(\theta_y)$$



Composition of 3D Transformations

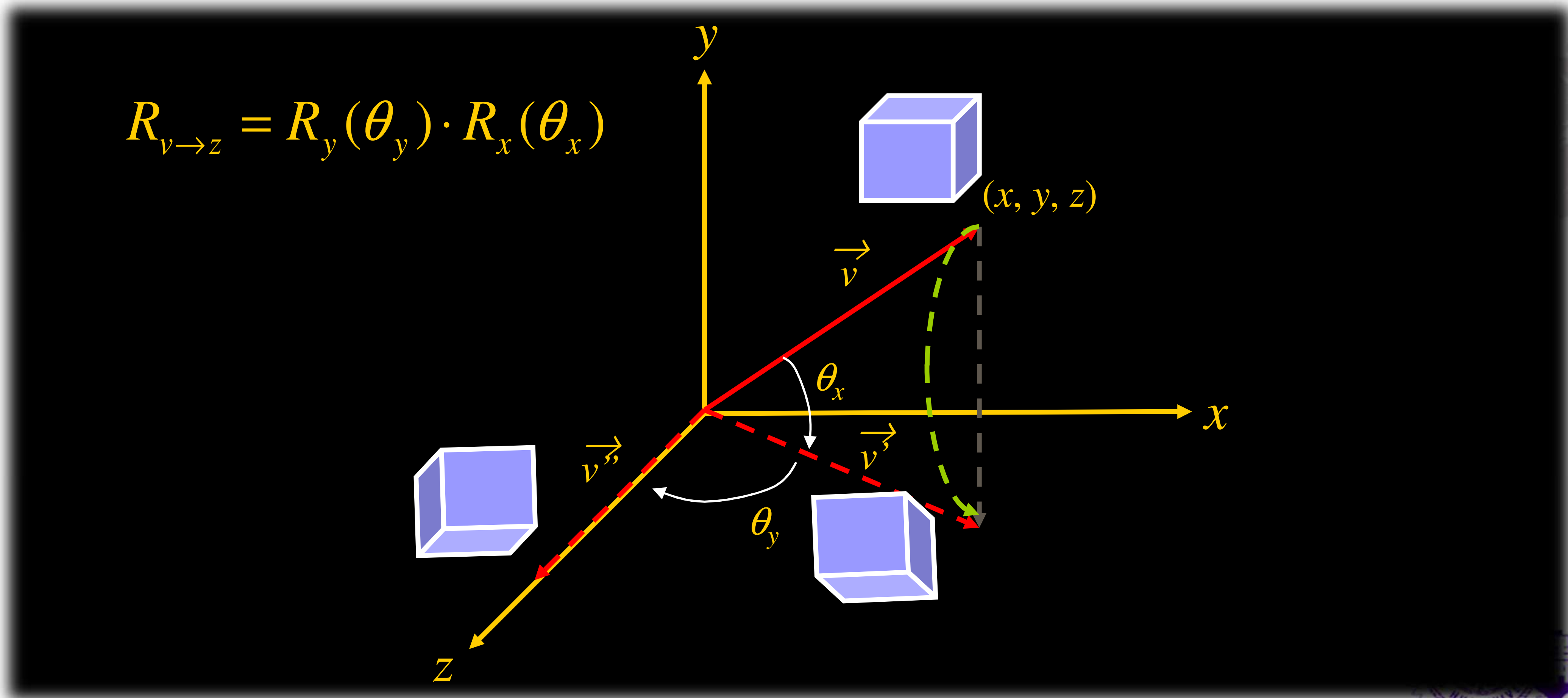
- Rotation about an arbitrary point $P_0(x, y, z)$
  - Translate from P_0 to origin
  - Rotate by θ degree // $R(\theta) = R_x(\theta_x) \cdot R_y(\theta_y) \cdot R_z(\theta_z)$
  - Translate from origin back to P_0

$$P' = (T(x, y, z) \cdot R(\theta) \cdot T(-x, -y, -z)) \cdot P$$



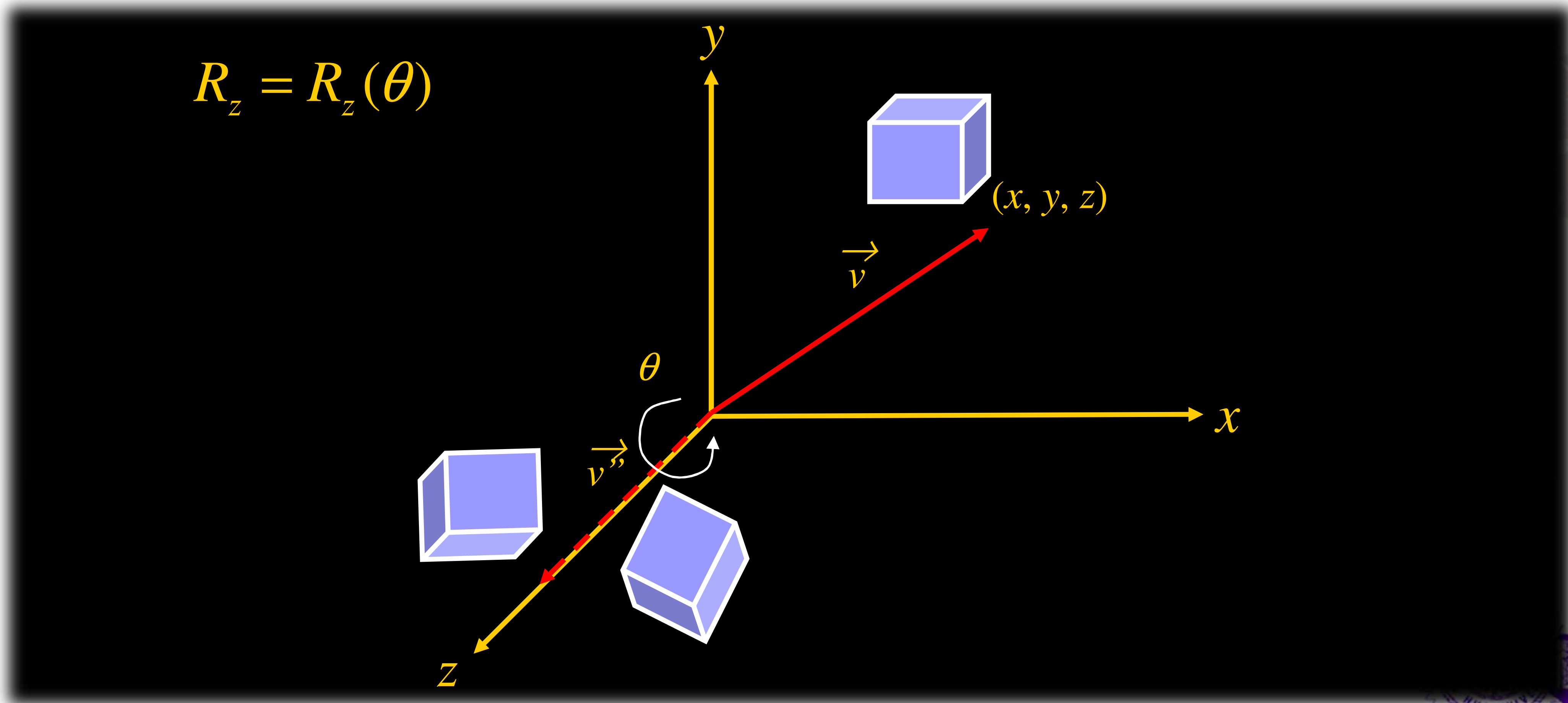
Composition of 3D Transformations

- Rotation about an arbitrary vector v
  - Step 1: Rotate the vector v to align with z-axis



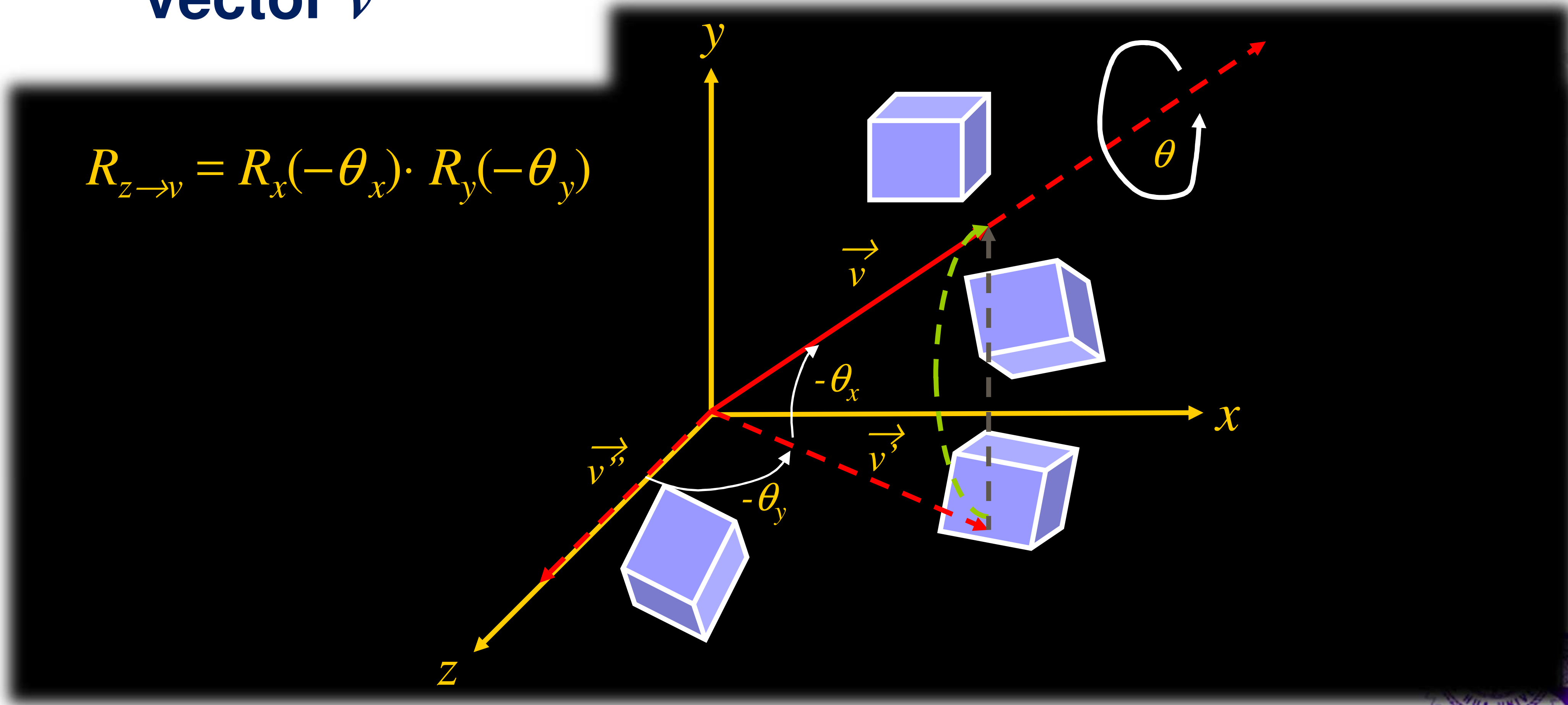
Composition of 3D Transformations

- Rotation about an arbitrary vector *v*
  - Step 2: Rotate in *z* by θ degree



Composition of 3D Transformations

- Rotation about an arbitrary vector v
  - Step 3: Rotate the z-axis back to align with vector v



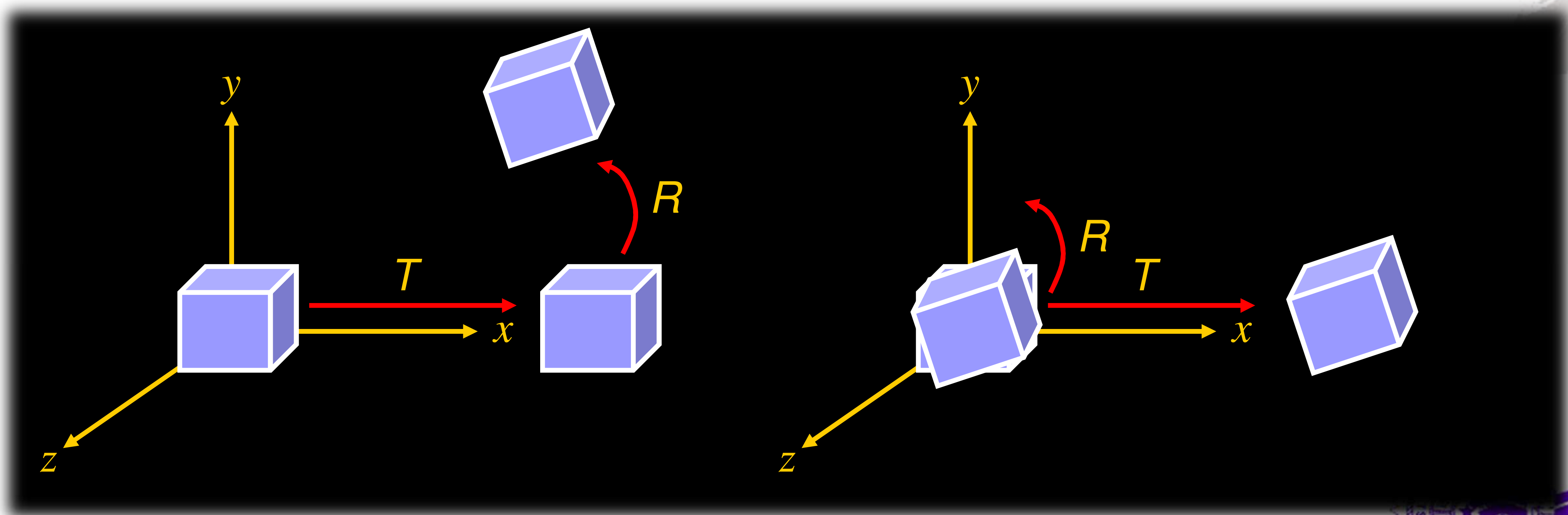
Composition of 3D Transformations

- Rotation about an arbitrary vector v
  - Step 1: Rotate the vector v to align with z-axis
  - Step 2: Rotate in z by θ degree
  - Step 3: Rotate the z-axis back to align with vector v

$$P' = \underbrace{(((R_x(-\theta_x) \cdot R_y(-\theta_y))}_{\text{Step 3}} \cdot \underbrace{R_z(\theta)}_{\text{Step 2}} \cdot \underbrace{(R_y(\theta_y) \cdot R_x(\theta_x)))}_{\text{Step 1}} \cdot P$$

Composition of 3D Transformations

- Commutative Property
  - Commutative property is not held for matrix composition
  - e.g. $T \cdot R \neq R \cdot T$



Application Order of Transformation

- **Think of the *model* transformation in reverse order to get the right sequence of transformation in OpenGL**
  - ***An example of OpenGL matrix multiplication***

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N);              /* apply transformation N */
glMultMatrixf(M);              /* apply transformation M */
glMultMatrixf(L);              /* apply transformation L */
glBegin(GL_POINTS);
  glVertex3f(v);               /* draw transformed vertex v */
glEnd();
```

The vertex transformation is $I(N(M(Lv))) = (INML)v = (NML)v$

Application Order of Transformation

- **If you are not thinking in reverse order, than all the transformation matrices derived should be transposed**
  - ***For example , the Direct3D matrix representation is applied in this order***

The previous transformation example becomes

$$(((vL^T)M^T)N^T)I = v(L^TM^TN^TI) = v(L^TM^TN^T)$$

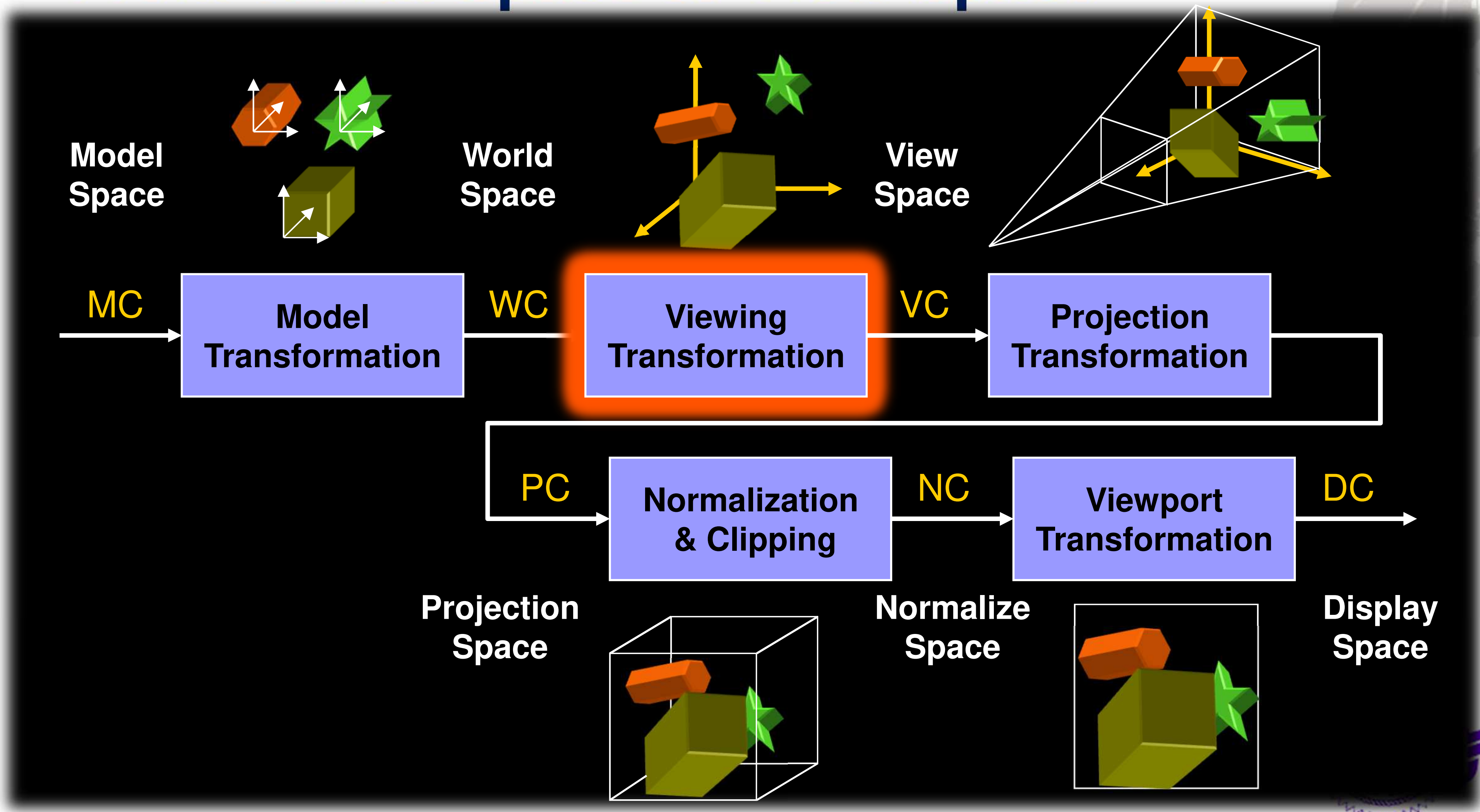
If L, M, N, I is represented as transformation matrix in OpenGL

Viewing Transformation

From World Space to View Space

3D Viewing Process

◆ From World Space to View Space

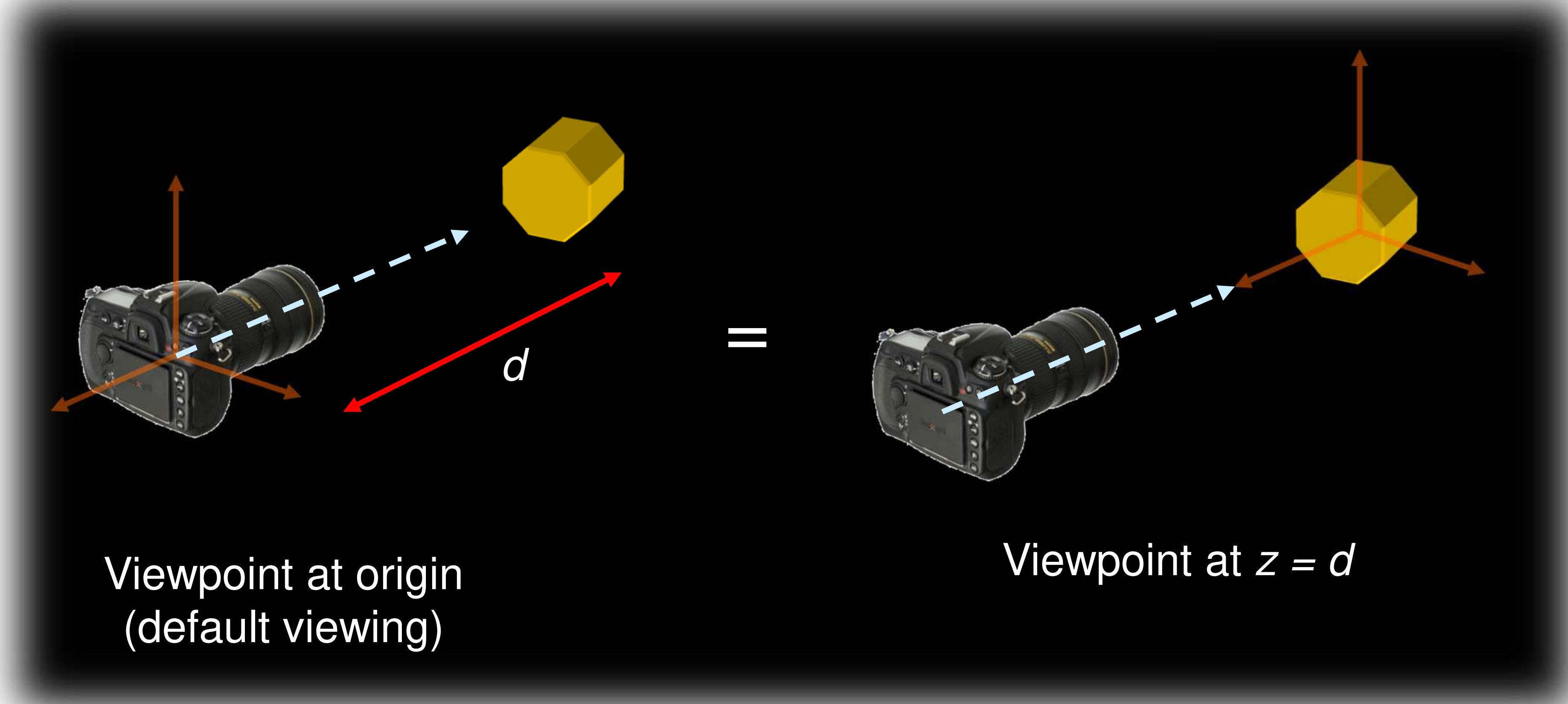


Viewing Transformation

- It defines the position and orientation of the viewpoint
- It is composed of a sequence of translation and rotation transformations
- In OpenGL, the default viewpoint is located at the origin and looking toward the negative z-axis direction

Viewing Transformation

- Example

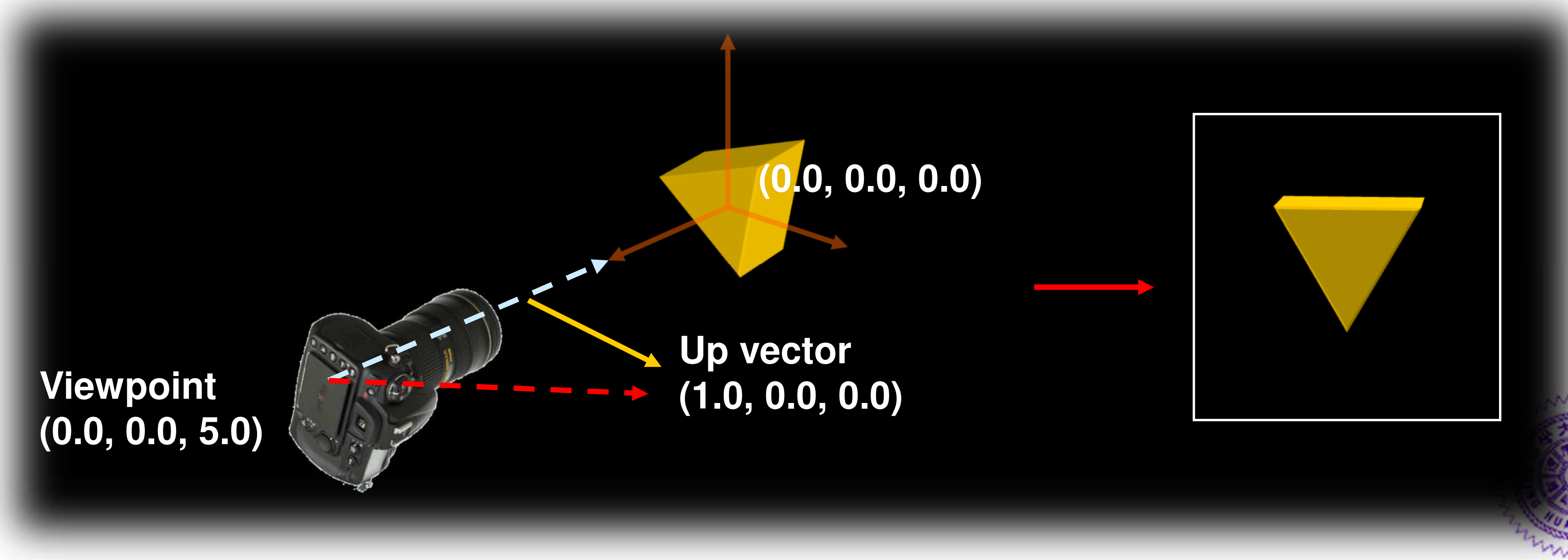


Specify the Viewing Parameters

- You need to define the viewpoint, viewing direction, and the pose when you are looking through a specific direction
- **gluLookat**(*ex*, *ey*, *ez*, // eye coordinates
  cx, *cy*, *cz*, // view center coordinates
  ux, *uy*, *uz*) // up vector
  - Viewpoint is at (*ex*, *ey*, *ez*)
  - Viewing direction is from (*ex*, *ey*, *ez*) toward (*cx*, *cy*, *cz*)
  - Up direction is from (0, 0, 0) to (*ux*, *uy*, *uz*)

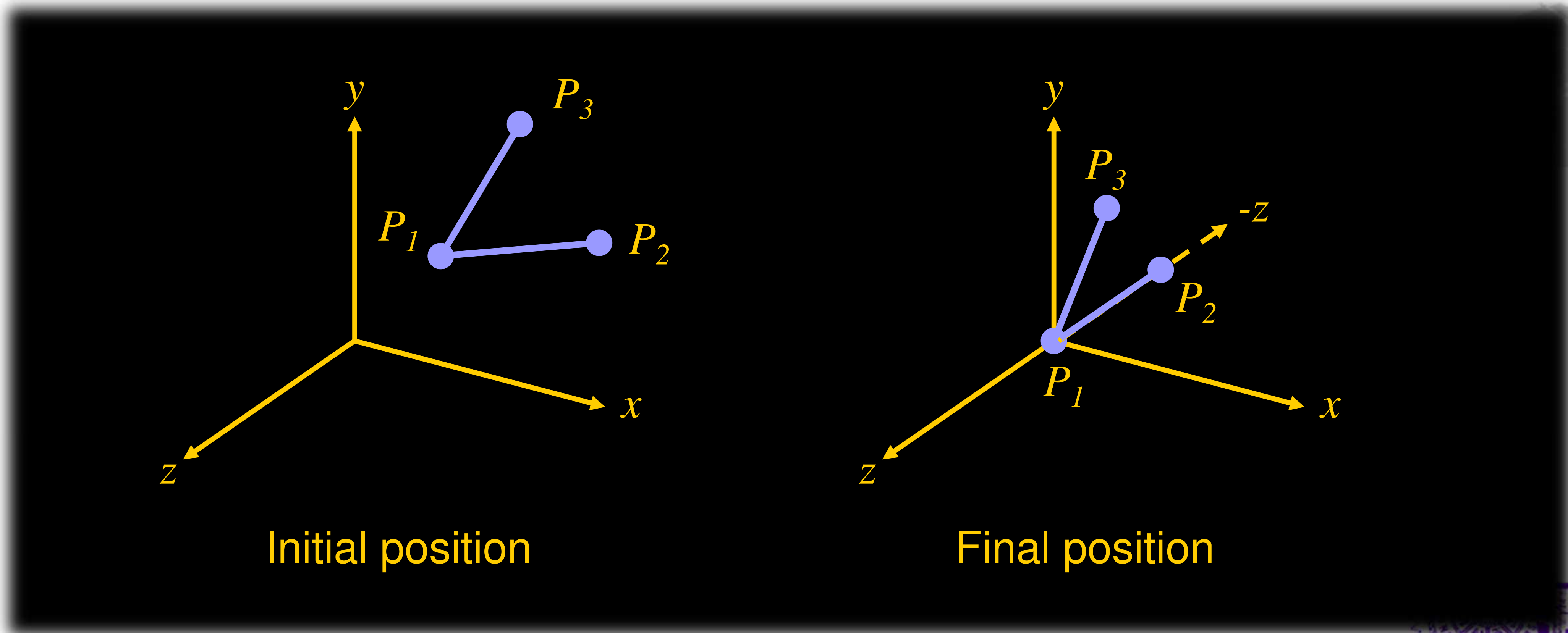
Specify the Viewing Parameters

- gluLookat(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0)
  - Viewpoint is at (0.0, 0.0, 5.0)
  - Viewing direction is from (0.0, 0.0, 5.0) toward (0.0, 0.0, 0.0)
  - Up direction is (1.0, 0.0, 0.0)



Concept of Viewing Transformation

- Transforming P_1, P_2, and P_3 such that $\overrightarrow{P_1P_2}$ aligns with the **negative** *z*-axis and $\overrightarrow{P_1P_3}$ lies on the *yz*-plane



Concept of Viewing Transformation

- Transforming P_1, P_2, and P_3 from their initial position to their final position
  - Step 1: Translate P_1 to the origin
  - Step 2: Rotate about the *y*-axis such that $\overrightarrow{P_1P_2}$ lies on the *yz*-plane
  - Step 3: Rotate about the *x*-axis such that $\overrightarrow{P_1P_2}$ align with the negative *z*-axis
  - Step 4: Rotate about the *z*-axis such that $\overrightarrow{P_1P_3}$ lies on the *yz*-plane

Concept of Viewing Transformation

- Transforming P_1, P_2, and P_3 from their initial position to their final position
  - Step 1: Translate P_1 to the origin

$$P'_1 = T(-x_1, -y_1, -z_1) \cdot P_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$P'_2 = T(-x_1, -y_1, -z_1) \cdot P_2 = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \\ 1 \end{bmatrix}$$

$$P'_3 = T(-x_1, -y_1, -z_1) \cdot P_3 = \begin{bmatrix} x_3 - x_1 \\ y_3 - y_1 \\ z_3 - z_1 \\ 1 \end{bmatrix}$$

Concept of Viewing Transformation

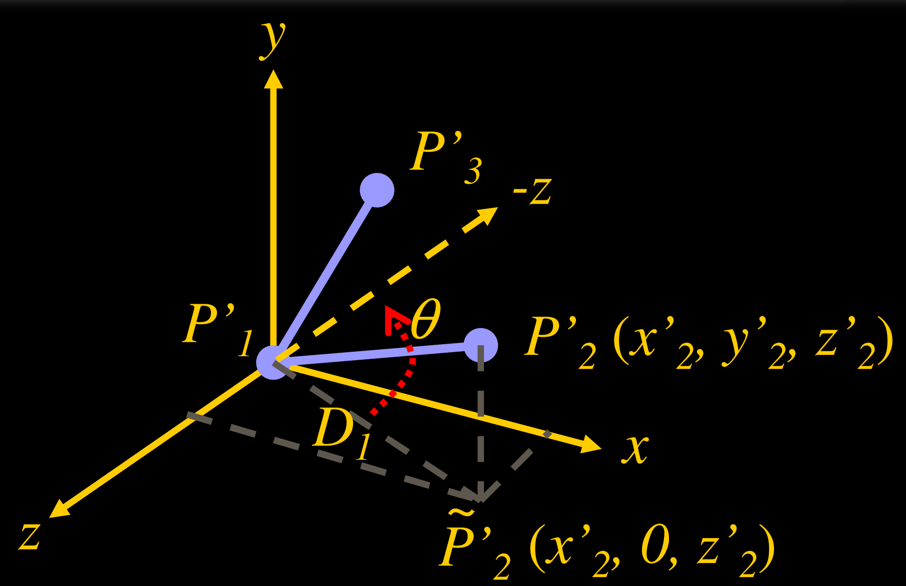
- Transforming P_1, P_2, and P_3 from their initial position to their final position
  - Step 2: Rotate about the y-axis such that $\overrightarrow{P_1P_2}$ lies in the (y, z) plane

$$P''_2 = R_y(\theta) \cdot P'_2 = \begin{bmatrix} 0 \\ y_2 - y_1 \\ D_1 \\ 1 \end{bmatrix}$$

where $D_1 = \left|\overline{P'_1 \tilde{P}'_2}\right|$

Besides, $P''_1 = R_y(\theta) \cdot P'_1$ and

$P''_3 = R_y(\theta) \cdot P'_3$



Concept of Viewing Transformation

- Transforming P_1, P_2, and P_3 from their initial position to their final position
  - Step 3: Rotate about the x-axis such that $\overrightarrow{P_1P_2}$ lies on the z-axis

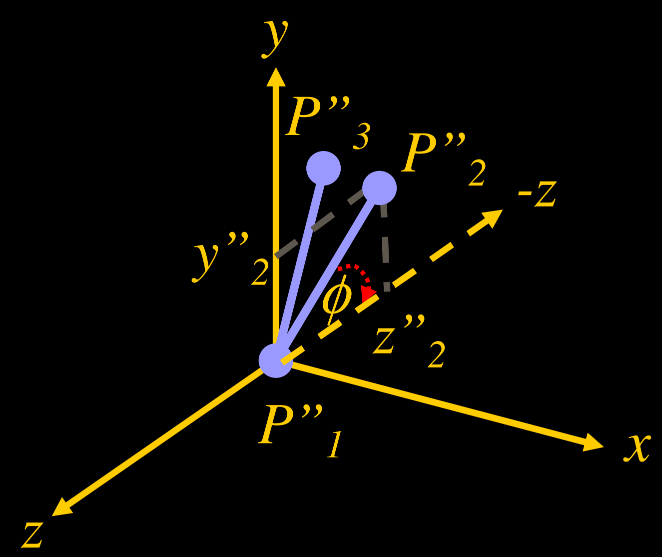
$$P'''_2 = R_x(-\emptyset) \cdot P''_2$$

$$= R_x(-\emptyset) \cdot R_y(\theta) \cdot P'_2$$

$$= R_x(-\emptyset) \cdot R_y(\theta) \cdot T \cdot P_2$$

$$P'''_1 = R_x(-\emptyset) \cdot R_y(\theta) \cdot T \cdot P_1$$

$$P'''_3 = R_x(-\emptyset) \cdot R_y(\theta) \cdot T \cdot P_3$$



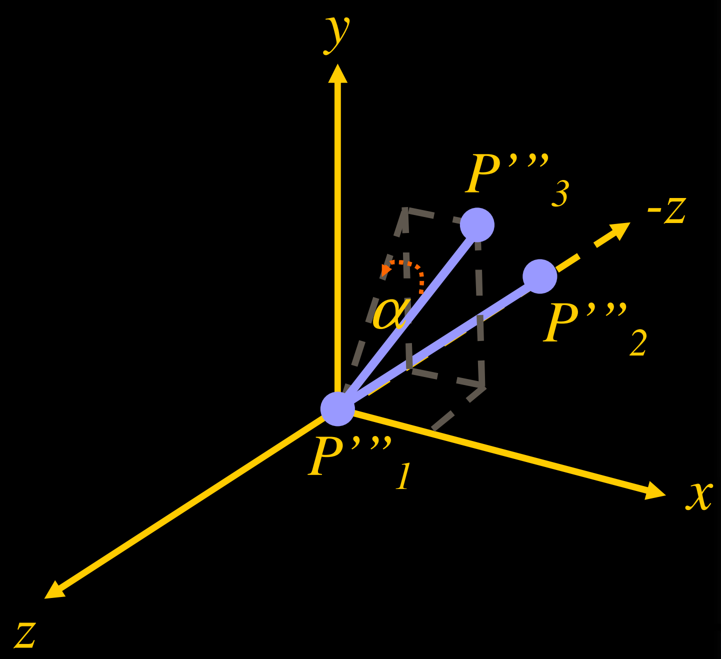
Concept of Viewing Transformation

- Transforming P_1, P_2, and P_3 from their initial position to their final position
  - Step 4: Rotate about the z-axis such that $\overrightarrow{P_1P_3}$ lies on the yz-plane

$$
\begin{aligned}
P''''_3 &= R_z(\alpha)\cdot P'''_3 \\
&= R_z(\alpha)\cdot R_x(-\emptyset)\cdot P''_3 \\
&= R_z(\alpha)\cdot R_x(-\emptyset)\cdot R_y(\theta)\cdot P'_3 \\
&= R_z(\alpha)\cdot R_x(-\emptyset)\cdot R_y(\theta)\cdot T\cdot P_3
\end{aligned}
$$

$$P''''_1 = R_z(\alpha)\cdot R_x(-\emptyset)\cdot R_y(\theta)\cdot T\cdot P_1$$

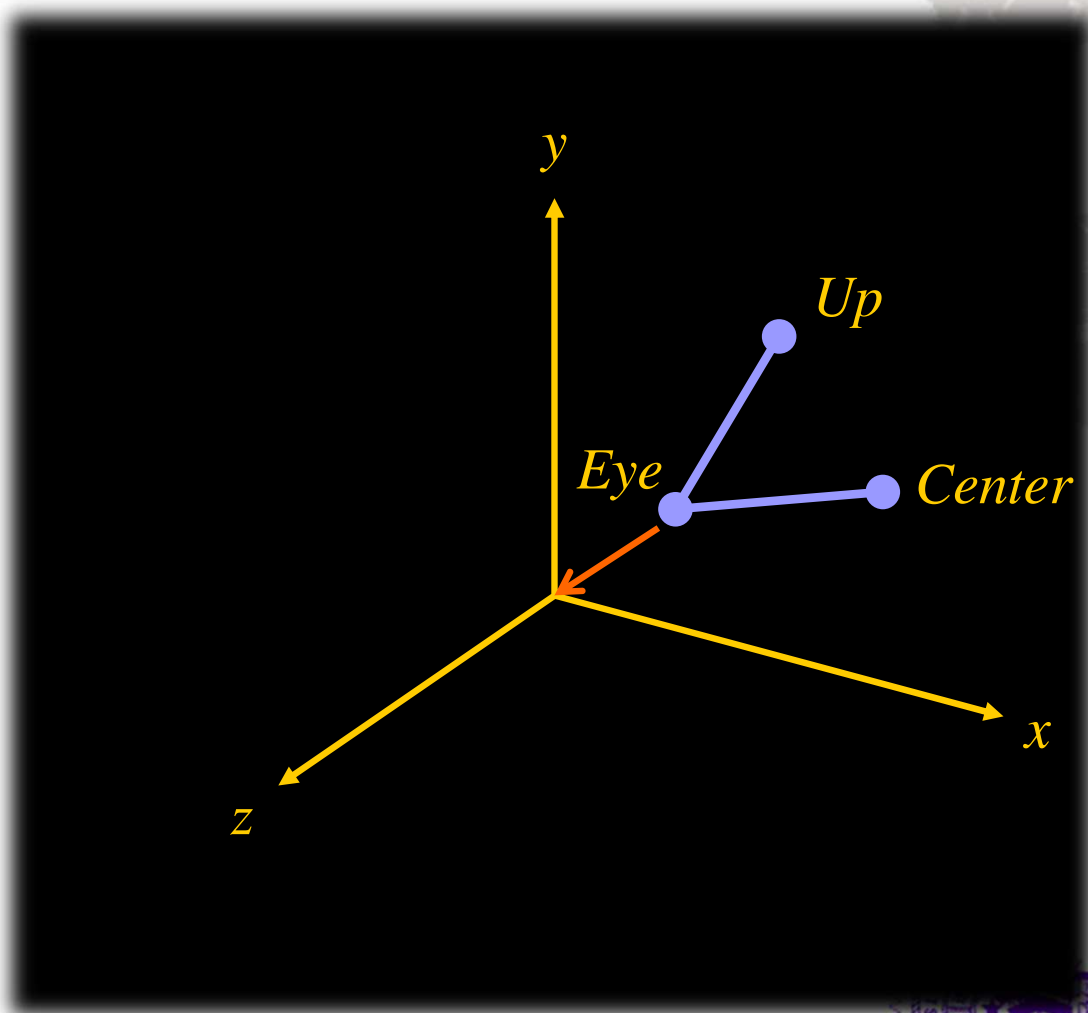
$$P''''_2 = R_z(\alpha)\cdot R_x(-\emptyset)\cdot R_y(\theta)\cdot T\cdot P_2$$



Fast Viewing Matrix Derivation

- Translate eye position to the origin

$$T = \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Fast Viewing Matrix Derivation

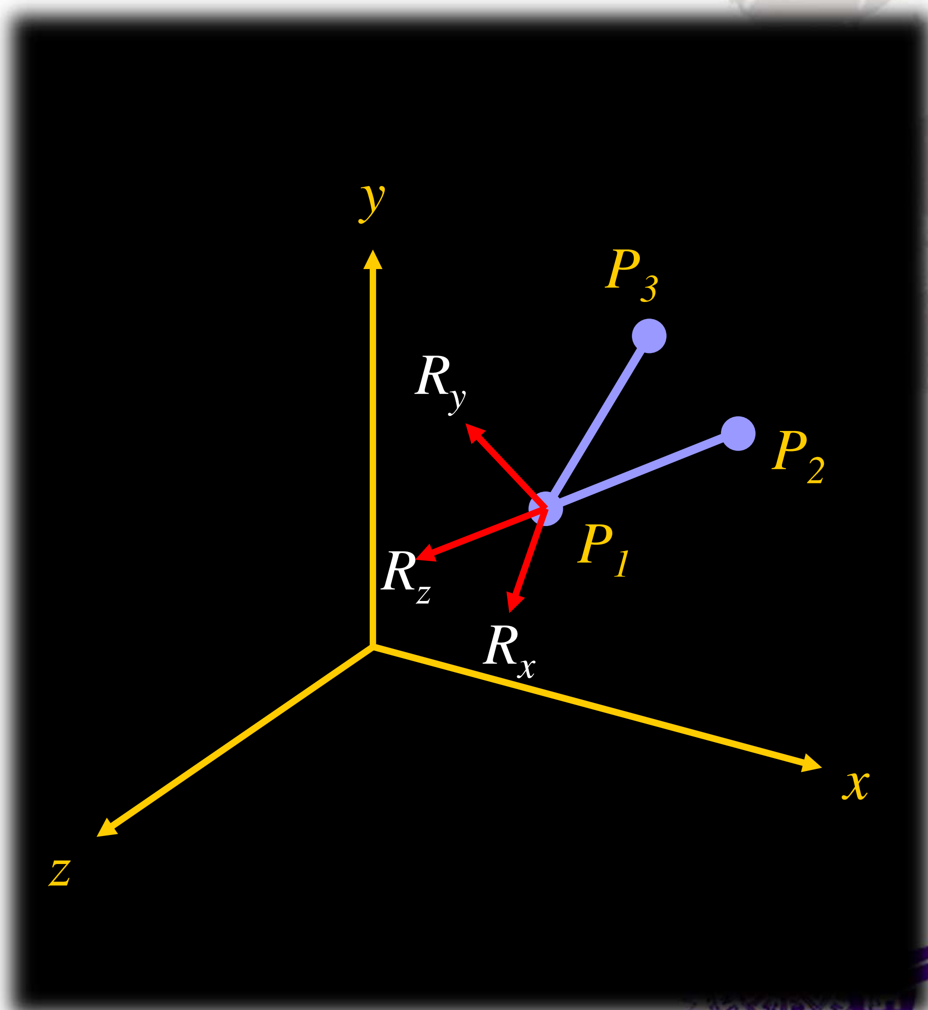
- **Find the bases with respect to the new space**

$$R = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} \\ r_{1y} & r_{2y} & r_{3y} \\ r_{1z} & r_{2z} & r_{3z} \end{bmatrix}$$

$$R_z = [r_{1z} \quad r_{2z} \quad r_{3z}]^T = \frac{-\overrightarrow{P_1P_2}}{\left|\overrightarrow{P_1P_2}\right|}$$

$$R_x = [r_{1x} \quad r_{2x} \quad r_{3x}]^T = \frac{\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}}{\left|\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}\right|}$$

$$R_y = [r_{1y} \quad r_{2y} \quad r_{3y}]^T = R_z \times R_x$$



Fast Viewing Matrix Derivation

◆ Coordinates Conversion

$$x' \times (r_{1x}, r_{2x}, r_{3x}) + y' \times (r_{1y}, r_{2y}, r_{3y}) + z' \times (r_{1z}, r_{2z}, r_{3z})$$
$$= x \times (1,0,0) + y \times (0,1,0) + z \times (0,0,1)$$

$$\begin{bmatrix} r_{1x} & r_{1y} & r_{1z} \\ r_{2x} & r_{2y} & r_{2z} \\ r_{3x} & r_{3y} & r_{3z} \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Inverse matrix

$$\boxed{\begin{bmatrix} r_{1x} & r_{2x} & r_{3x} \\ r_{1y} & r_{2y} & r_{3y} \\ r_{1z} & r_{2z} & r_{3z} \end{bmatrix}} \begin{bmatrix} r_{1x} & r_{1y} & r_{1z} \\ r_{2x} & r_{2y} & r_{2z} \\ r_{3x} & r_{3y} & r_{3z} \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} \\ r_{1y} & r_{2y} & r_{3y} \\ r_{1z} & r_{2z} & r_{3z} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} \\ r_{1y} & r_{2y} & r_{3y} \\ r_{1z} & r_{2z} & r_{3z} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Fast Viewing Matrix Derivation

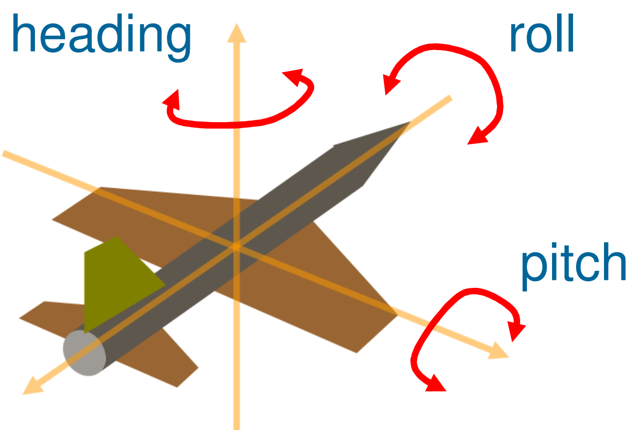
- Final Viewing Matrix is defined as

$$M_{view} = R \bullet T = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} & 0 \\ r_{1y} & r_{2y} & r_{3y} & 0 \\ r_{1z} & r_{2z} & r_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Custom Viewing

- Pilot view

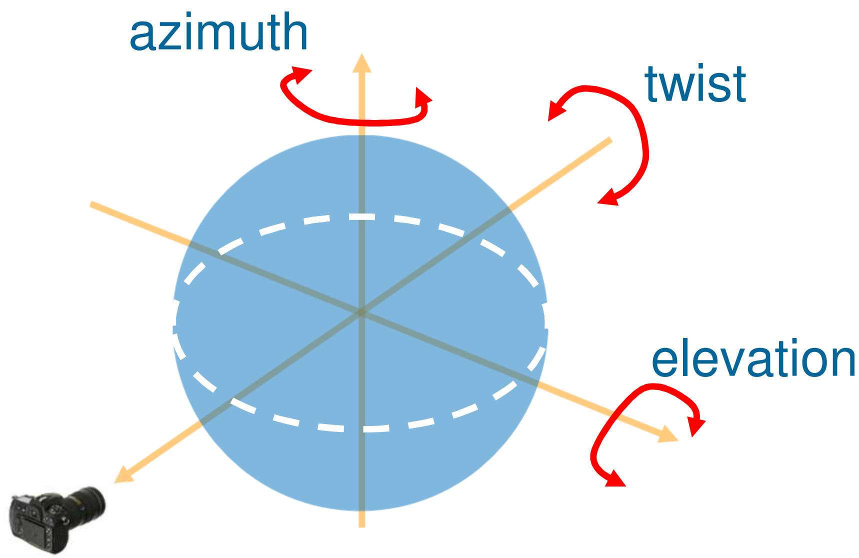
```
void pilotView(GLdouble planex, GLdouble planey,
               GLdouble planez, GLdouble roll,
               GLdouble pitch, GLdouble heading)
{
      glRotated(roll, 0.0, 0.0, 1.0);
      glRotated(pitch, 1.0, 0.0, 0.0);
      glRotated(heading, 0.0, 1.0, 0.0);  // yaw
      glTranslated(-planex, -planey, -planez);
}
```



Custom Viewing

- **Polar view**

```
void polarView(GLdouble distance, GLdouble twist,
               GLdouble elevation, GLdouble azimuth)
{
      glTranslate(0.0, 0.0, -distance);
      glRotated(twist, 0.0, 0.0, 1.0);
      glRotated(elevation, 1.0, 0.0, 0.0);
      glRotated(azimuth, 0.0, 1.0, 0.0);
}
```



Transformation in Reverse Order

- Each vertex is doing modeling transformation first and then viewing transformation

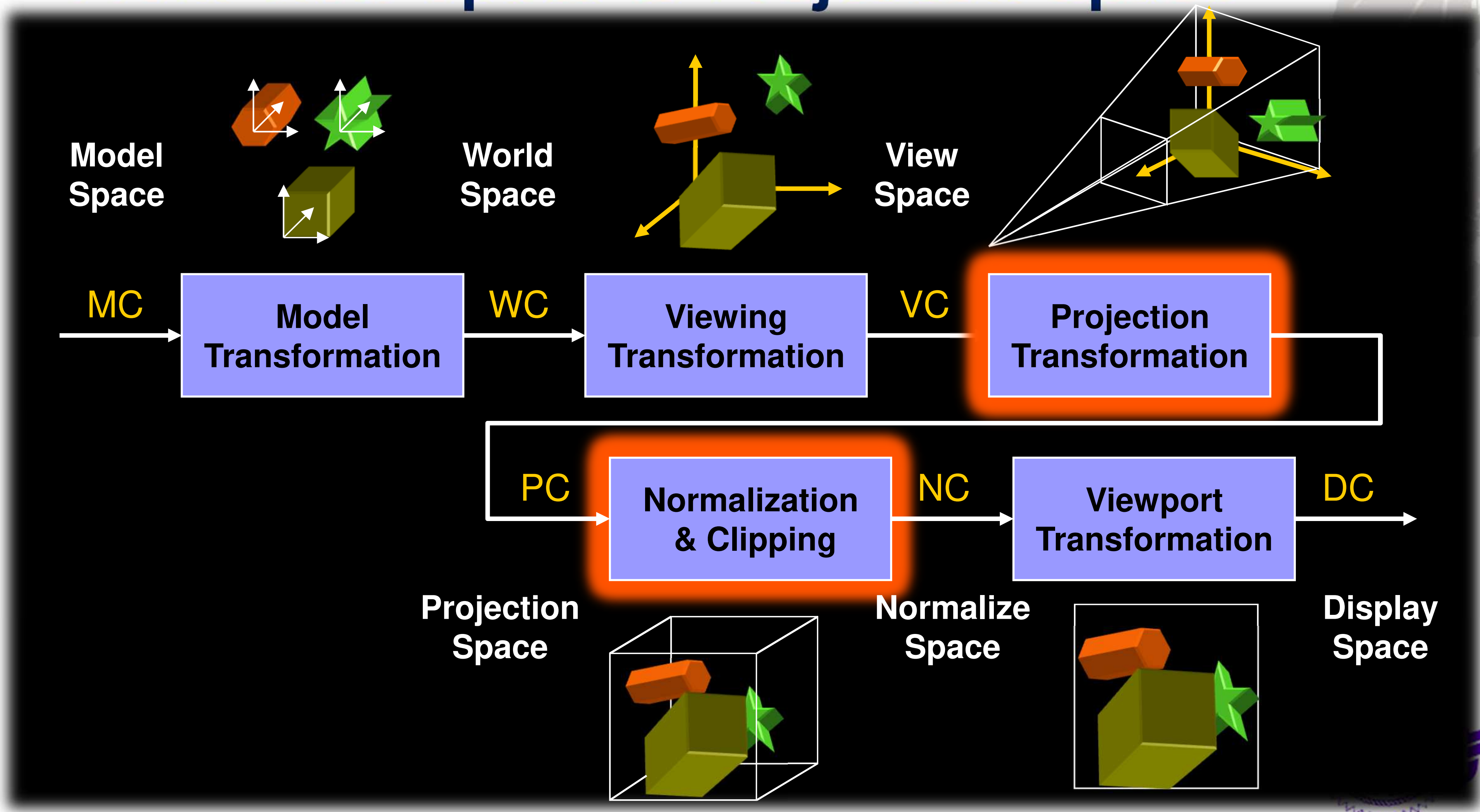
$$
\begin{aligned}
VMv = V(Mv) &= V(M_n M_{n-1} \ldots M_2 M_1 M_0 v) \\
&= V(M_n(M_{n-1}(\ldots(M_2(M_1(M_0 v)))\ldots)))
\end{aligned}
$$

Projection

From View Space to Projection Space

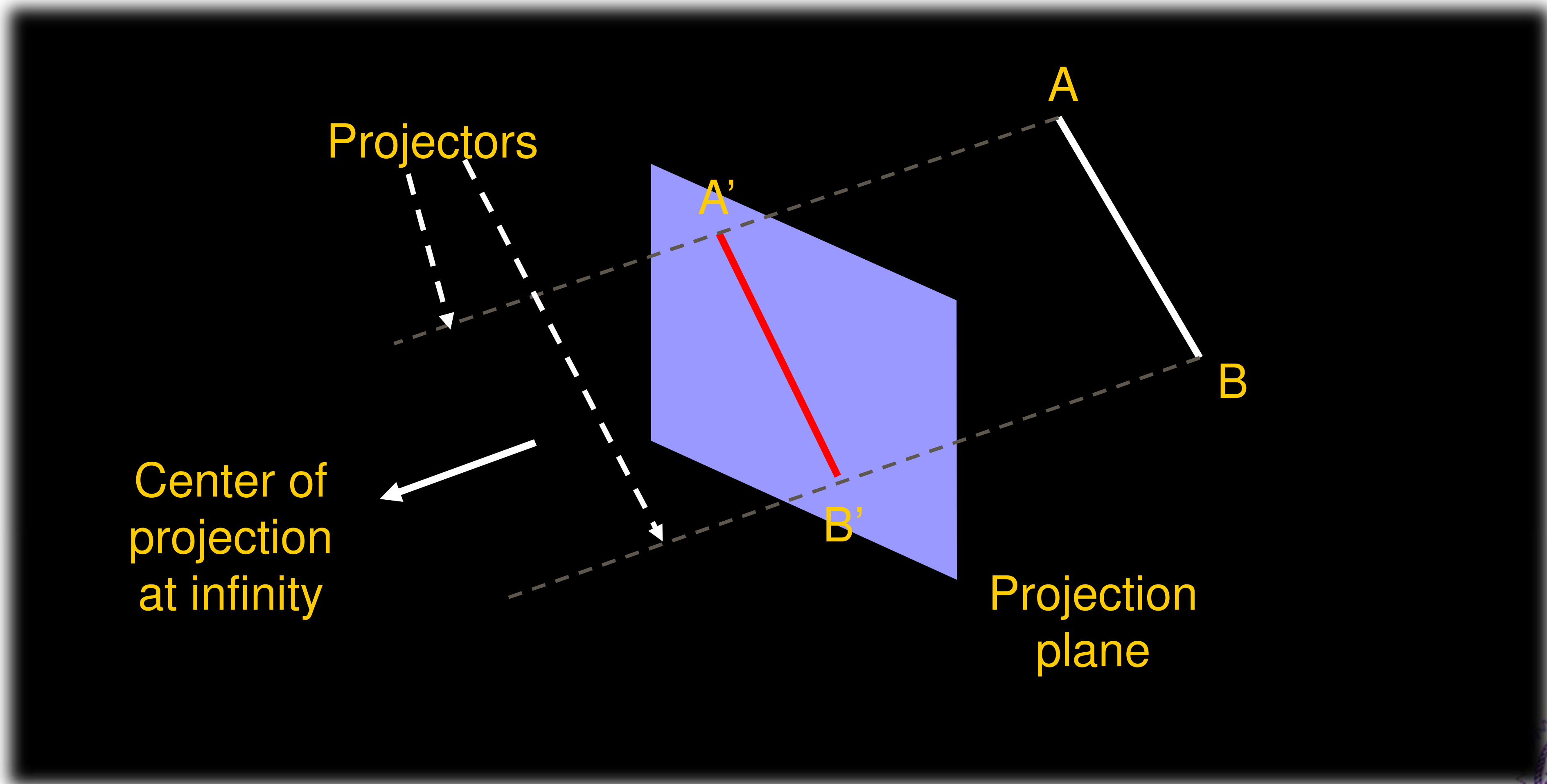
3D Viewing Process

- From View Space to Projection Space



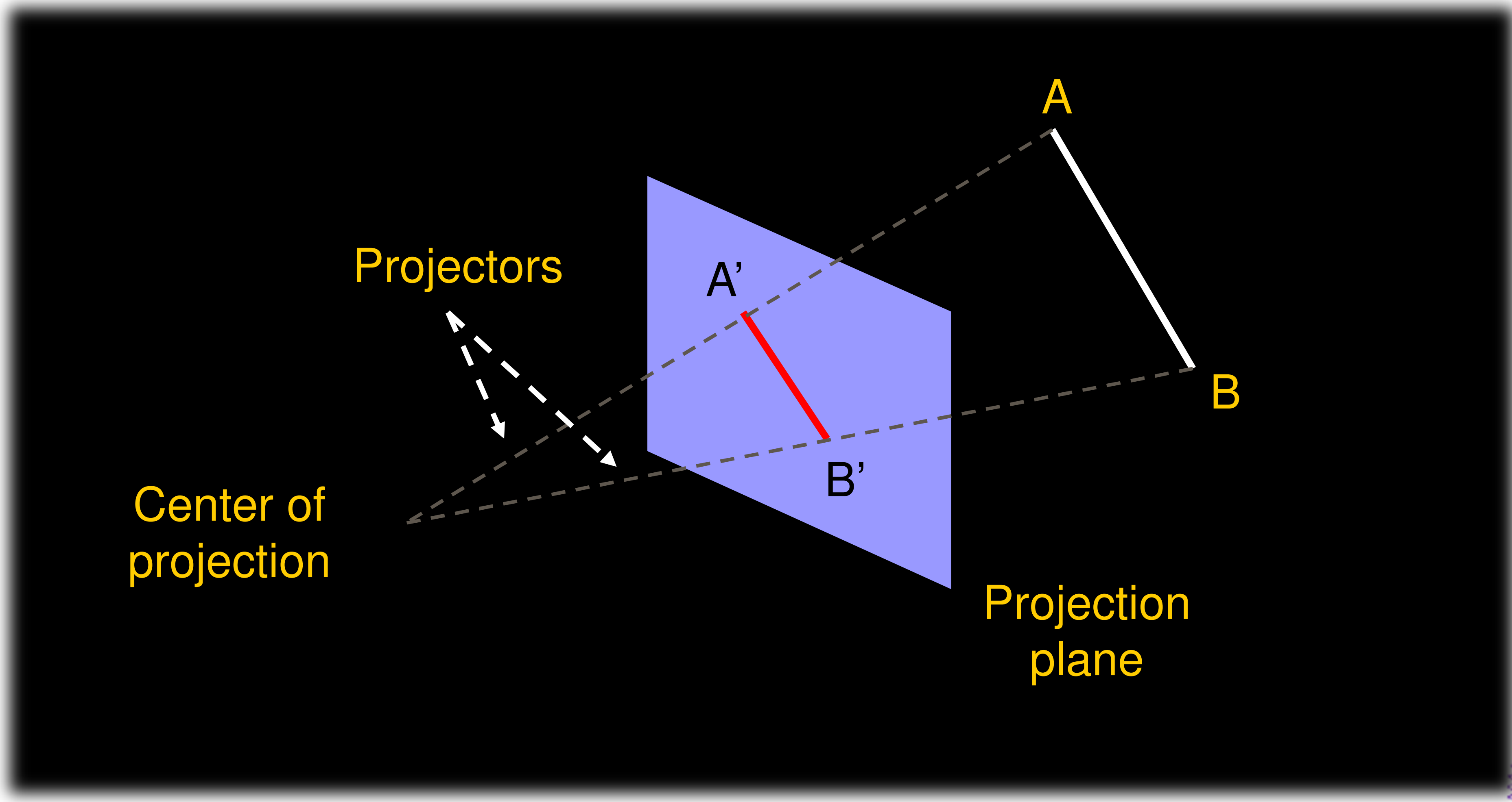
Projections

- **Parallel projection**



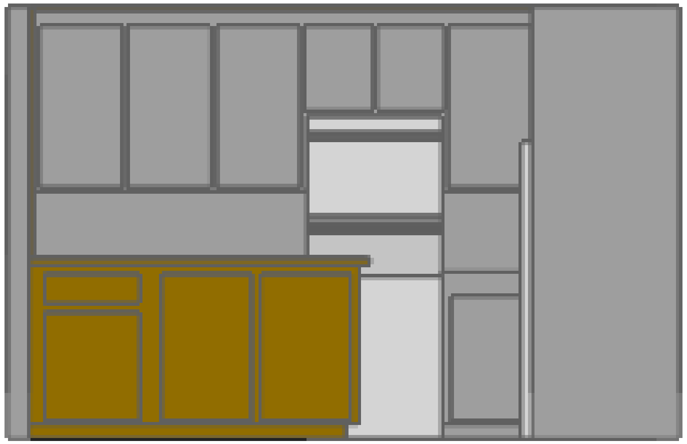
Projections

- Perspective projection

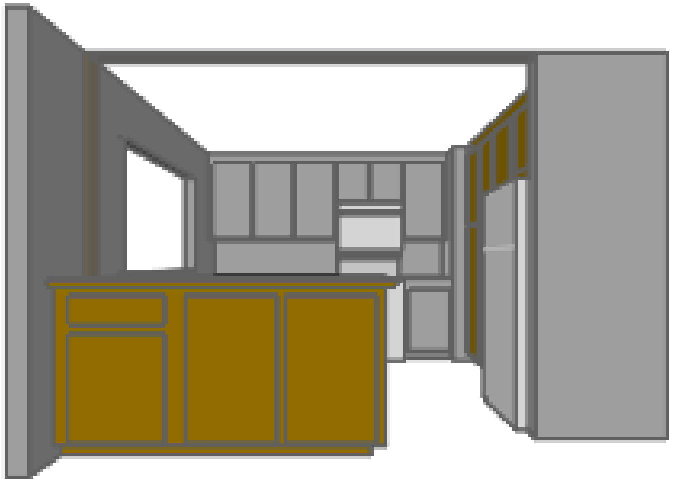


Projections

- **Parallel projection vs. perspective projection**



Parallel projection



Perspective projection

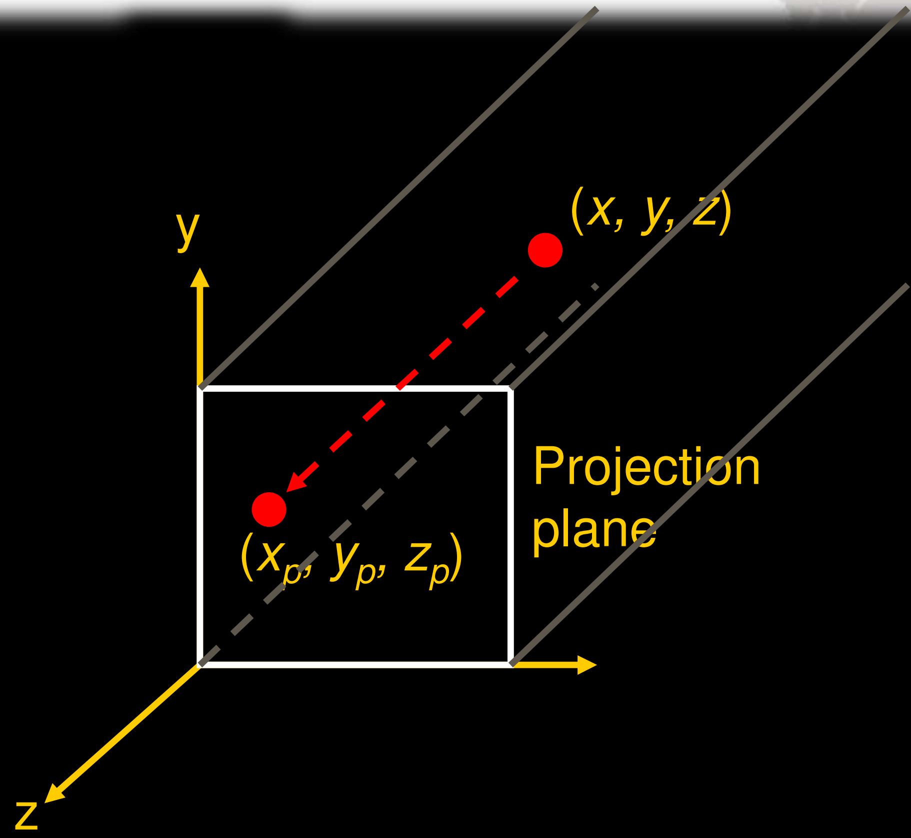
Parallel Projection

- **Projection onto a projection plane at z = 0**

$$x_p = x,\ y_p = y,\ z_p = 0$$

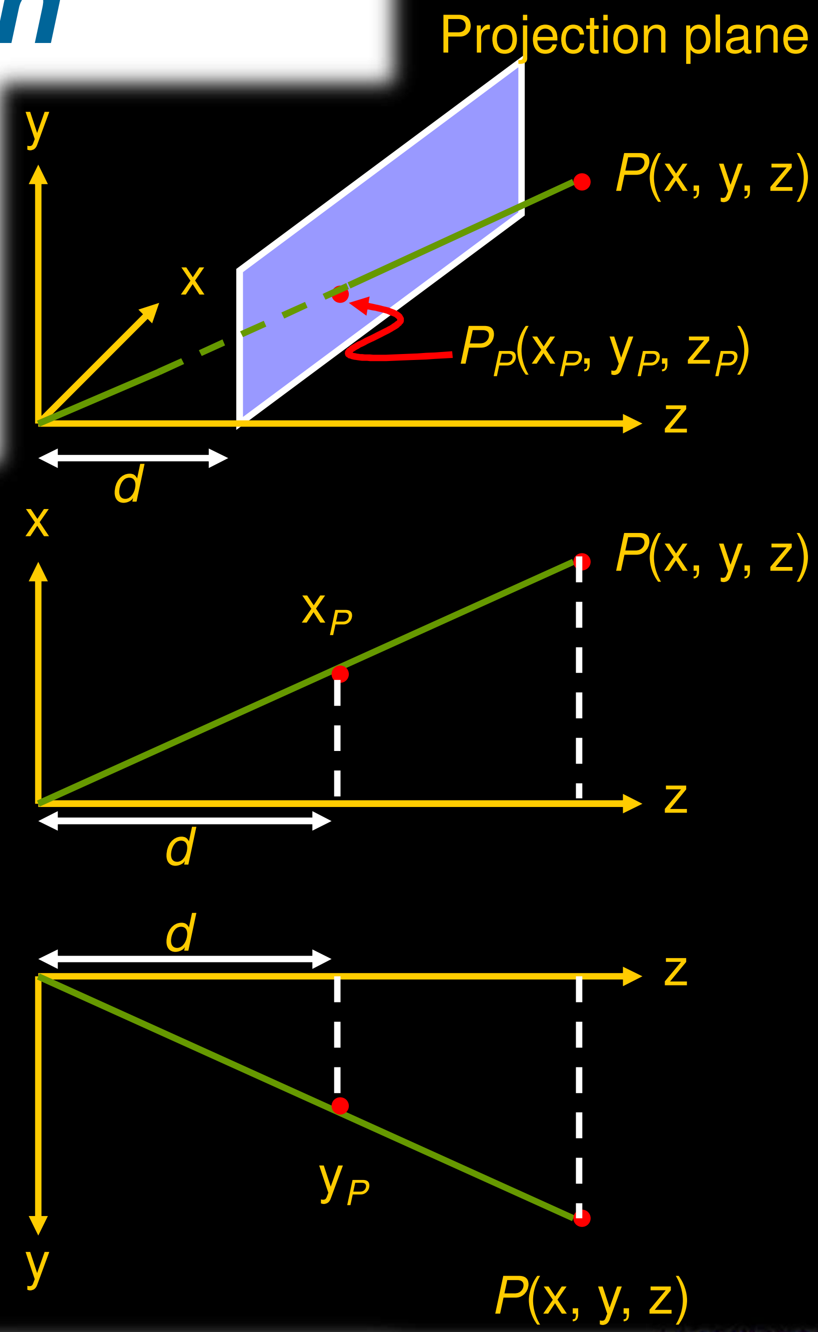
$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix}$$

$$= M_{ort} \cdot P$$



Perspective Projection

- **Projection onto a plane at z = *d***



$$\frac{x_p}{d} = \frac{x}{z}, \frac{y_p}{d} = \frac{y}{z}$$

$$x_p = \frac{x}{z/d}, y_p = \frac{y}{z/d}$$

$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Perspective Projection

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = M_{per} \cdot P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$[X \quad Y \quad Z \quad W]^T = \begin{bmatrix} x & y & z & \dfrac{z}{d} \end{bmatrix}^T$$

From homogeneous coordinates to Cartesian coordinate

$$\left(\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W}\right) = (x_p, y_p, z_p) = \left(\frac{x}{z/d}, \frac{y}{z/d}, d\right)$$

Perspective Projection

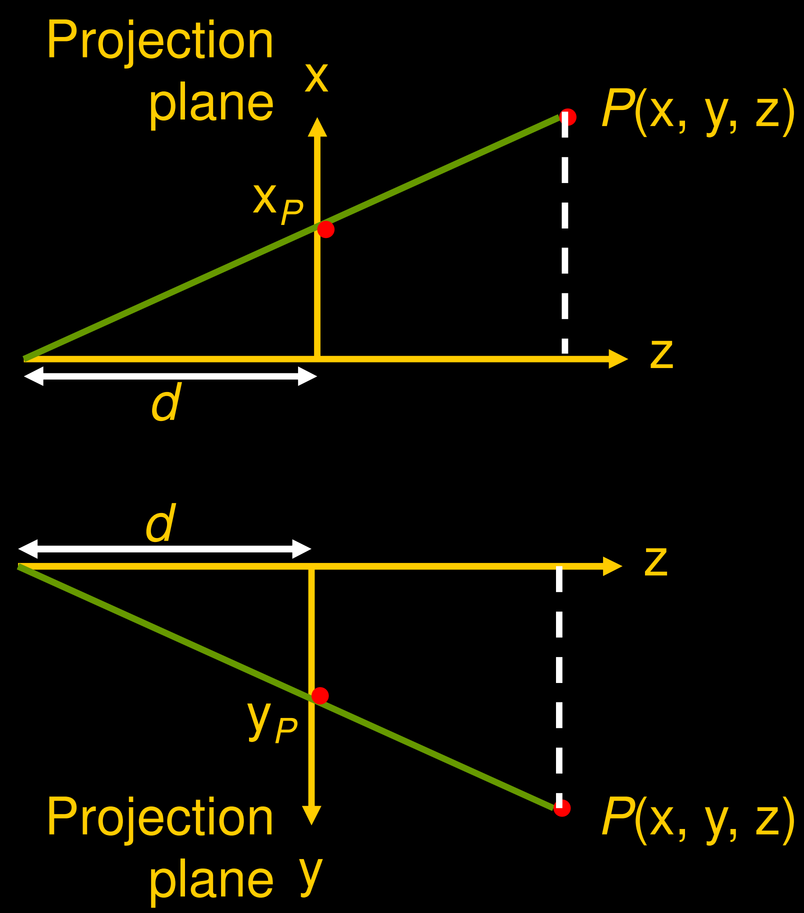
- Alternative Perspective Projection

$$\frac{x_p}{d} = \frac{x}{z+d}, \frac{y_p}{d} = \frac{y}{z+d}$$

$$x_p = \frac{d \cdot x}{z+d} = \frac{x}{(z/d)+1}$$

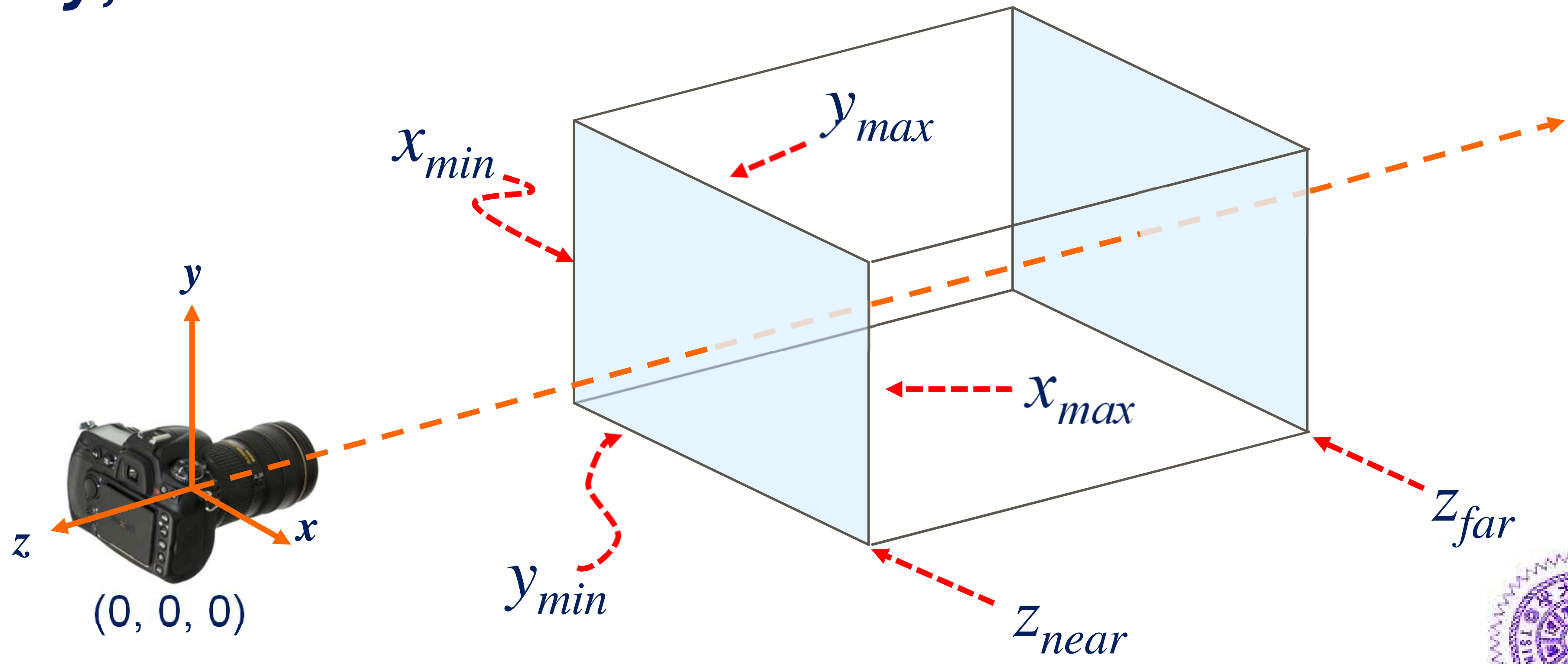
$$y_p = \frac{d \cdot y}{z+d} = \frac{y}{(z/d)+1}$$

$$M'_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}$$



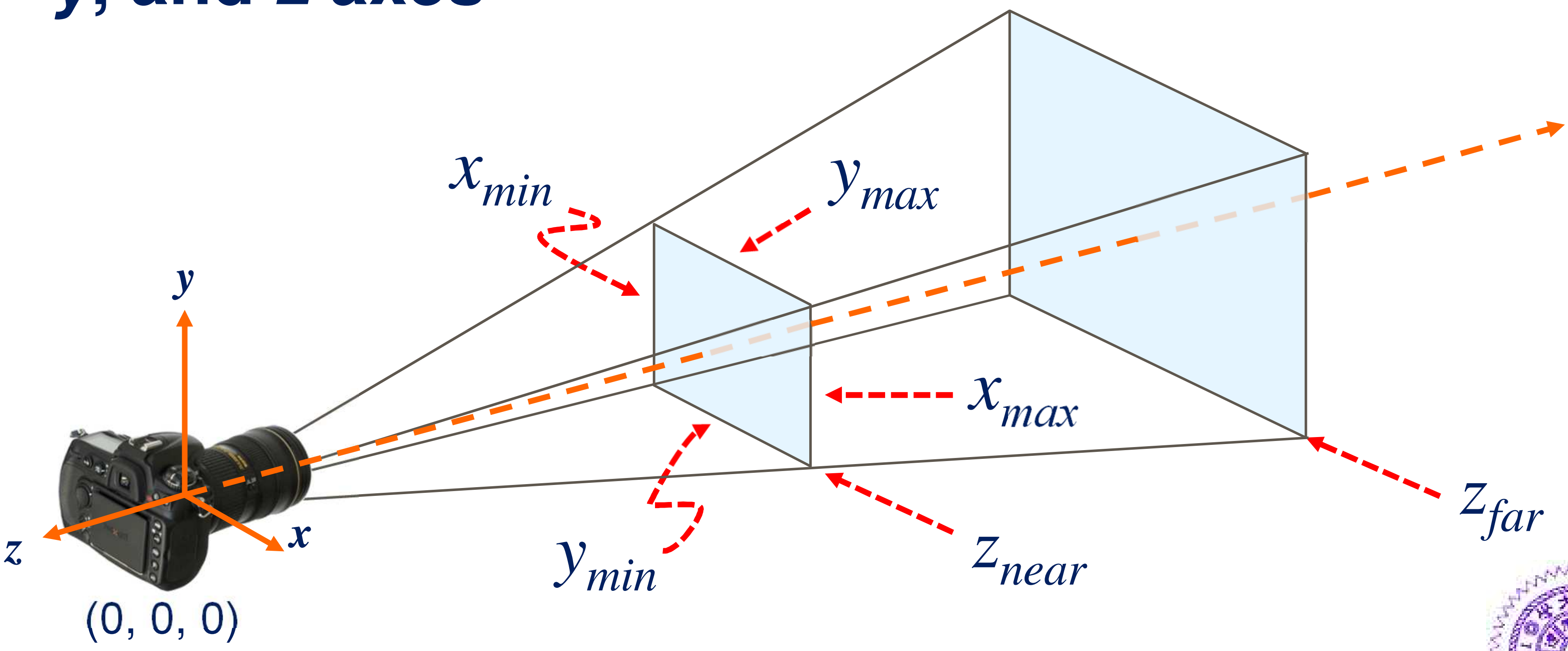
View Volume

- View Volume in Orthographic Projection
  - Assume the eye is at the origin, then specify x_{max}, x_{min}, y_{max}, y_{min}, z_{near}, z_{far} as the boundaries of *x*, *y*, and *z* axes



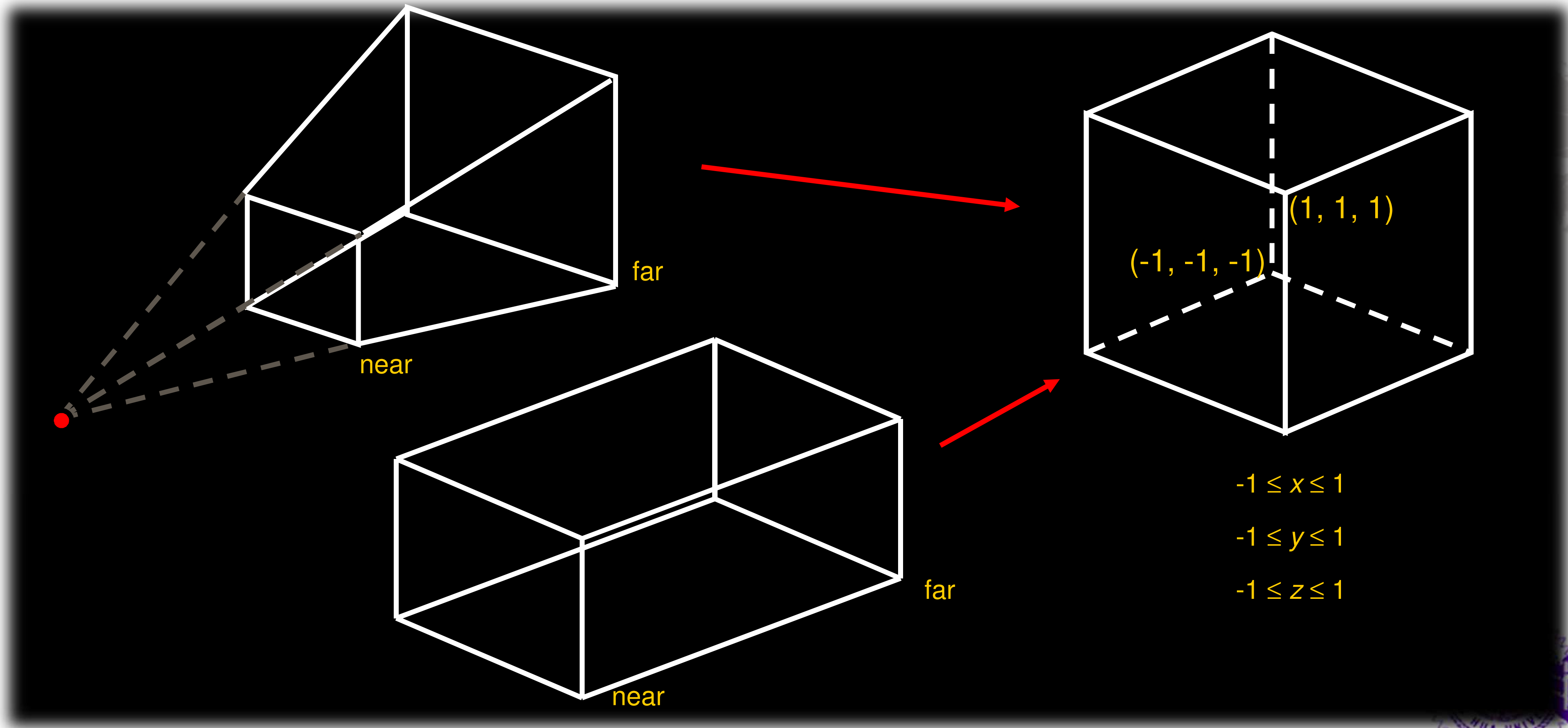
View Volume

- View Volume in Perspective Projection
  - Assume the eye is at the origin, then specify x_{max}, x_{min}, y_{max}, y_{min}, z_{near}, z_{far} as the boundaries of *x*, *y*, and *z* axes



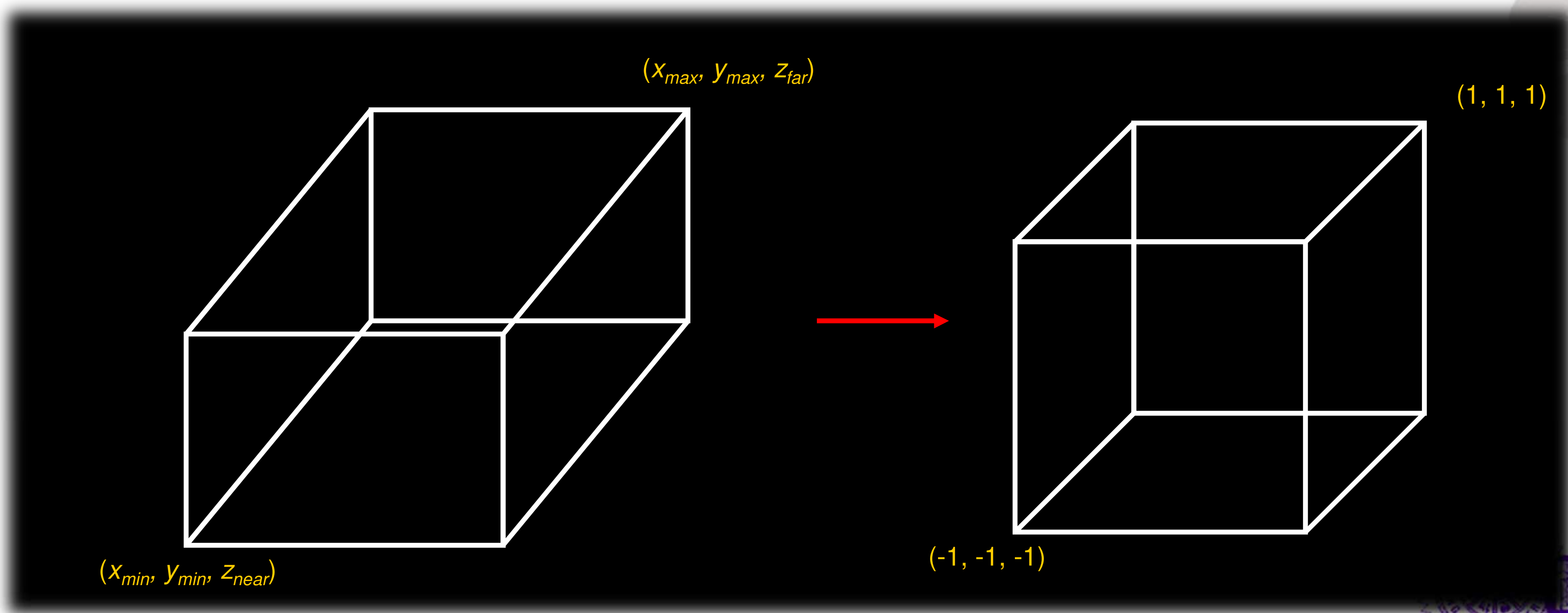
View Volume Normalization

- **Make the view volume clipping easy under normalized clipping volume**



Orthogonal Normalization

- Composition of orthogonal normalization
  - Translate the view volume center to origin
  - Scale to have each side of length 2



Orthogonal Normalization

- Composition of orthogonal normalization

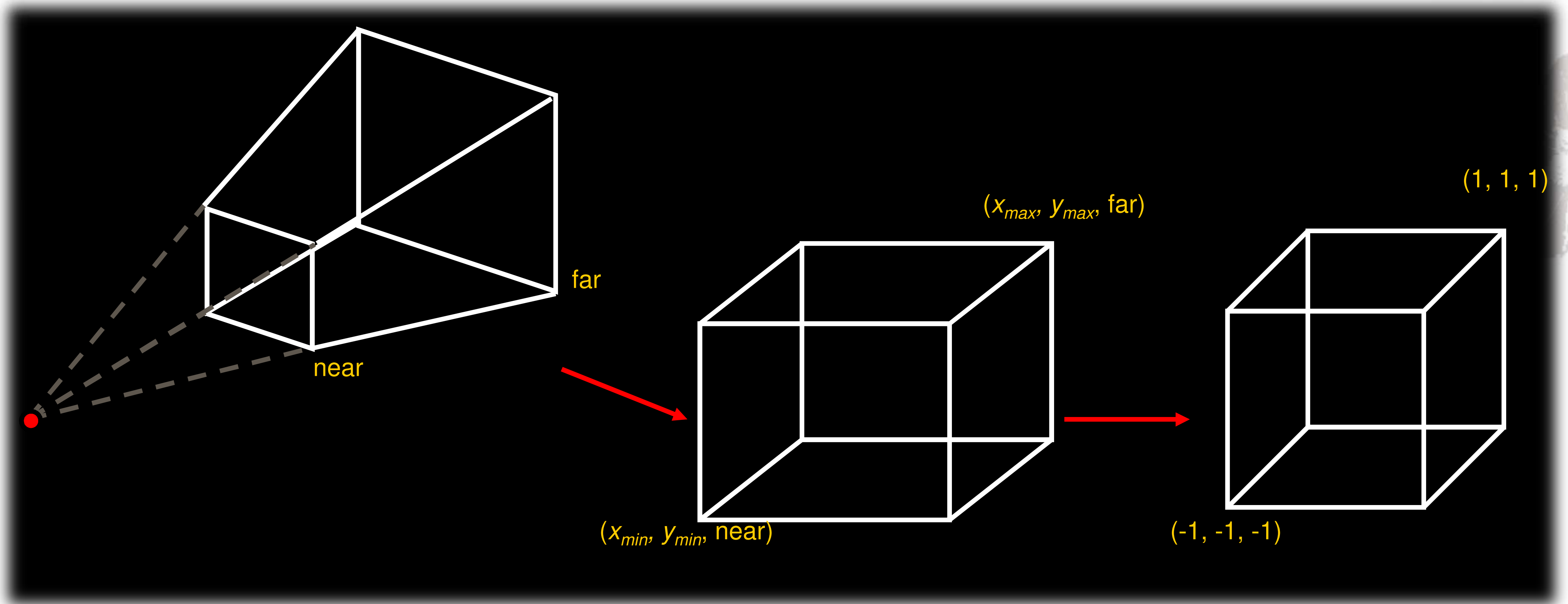
$$M_{orthonorm} = S\left(\frac{2}{x_{\max} - x_{\min}}, \frac{2}{y_{\max} - y_{\min}}, \frac{2}{z_{far} - z_{near}}\right)$$

$$\cdot T\left(-\frac{x_{\max} + x_{\min}}{2}, -\frac{y_{\max} + y_{\min}}{2}, -\frac{z_{far} + z_{near}}{2}\right)$$

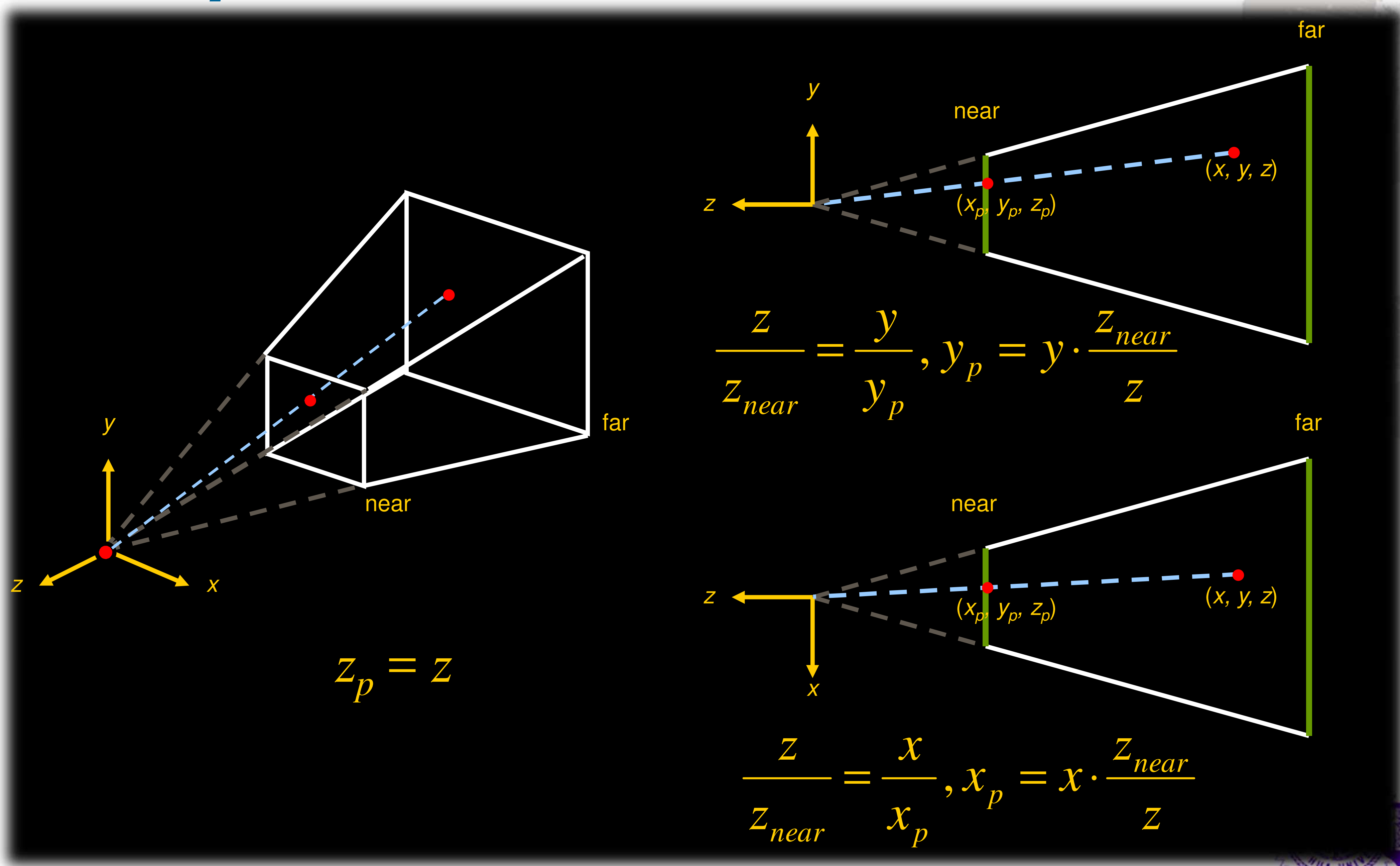
$$= \begin{bmatrix} \frac{2}{x_{\max} - x_{\min}} & 0 & 0 & -\frac{x_{\max} + x_{\min}}{x_{\max} - x_{\min}} \\ 0 & \frac{2}{y_{\max} - y_{\min}} & 0 & -\frac{y_{\max} + y_{\min}}{y_{\max} - y_{\min}} \\ 0 & 0 & \frac{2}{z_{far} - z_{near}} & -\frac{z_{far} + z_{near}}{z_{far} - z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Normalization

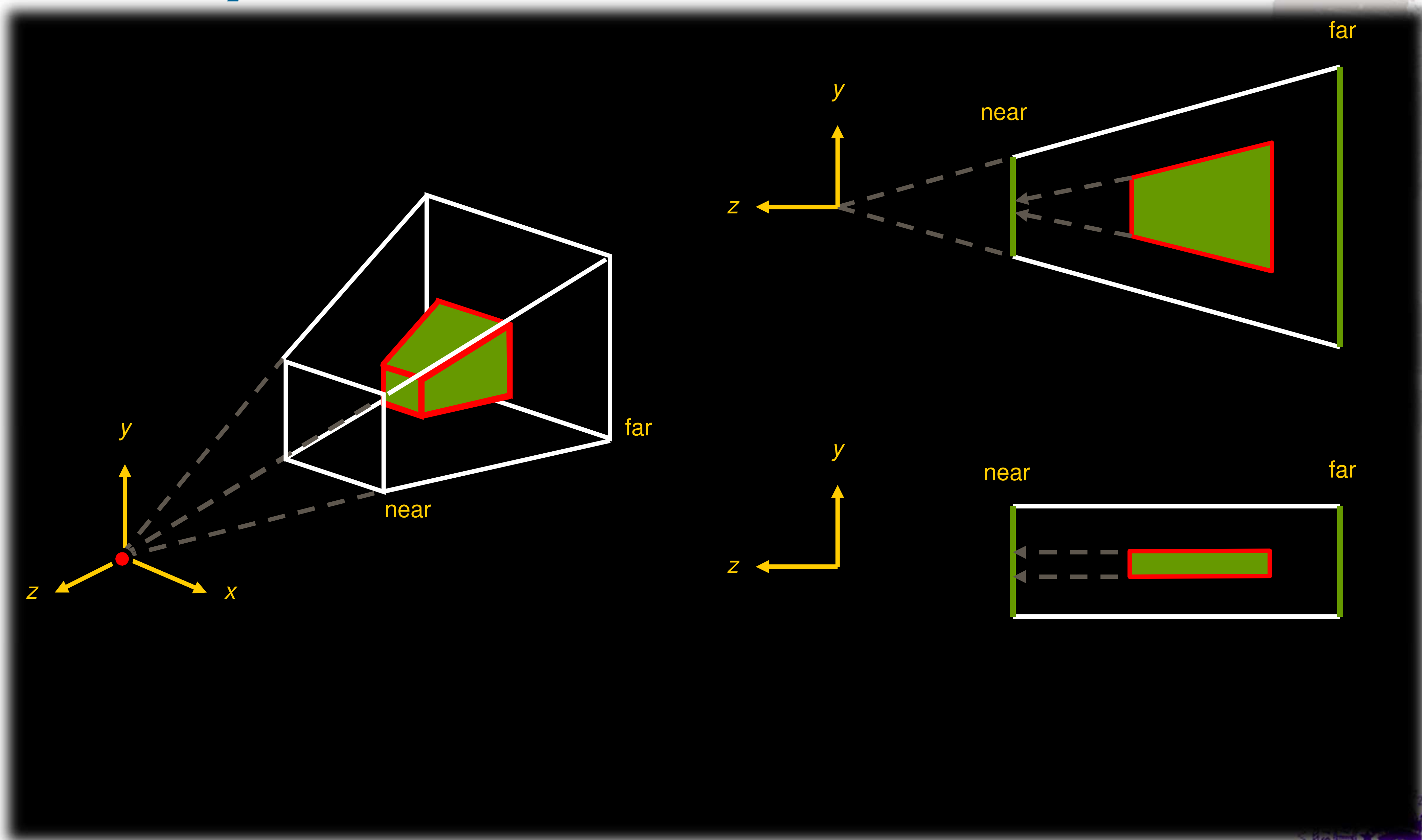
- Composition of perspective normalization



Perspective Normalization



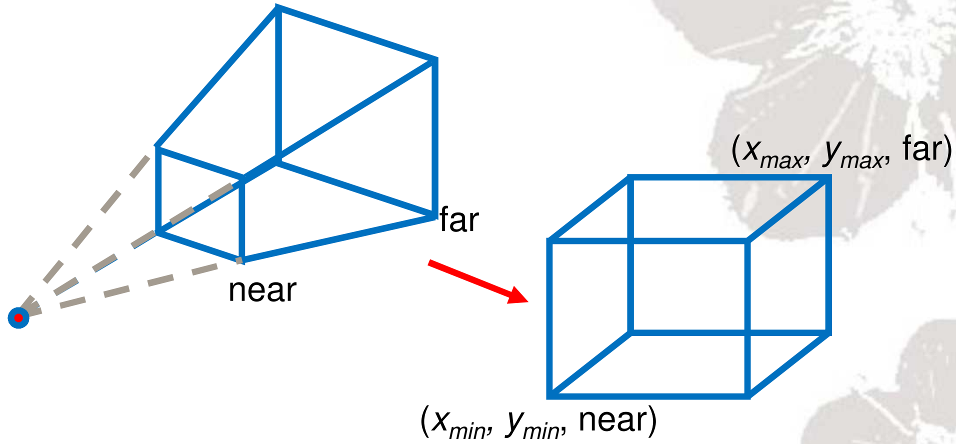
Perspective Normalization



Perspective Normalization

- **From Frustum to Cuboid**

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} z_{near} & 0 & 0 & 0 \\ 0 & z_{near} & 0 & 0 \\ 0 & 0 & S_z & T_z \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

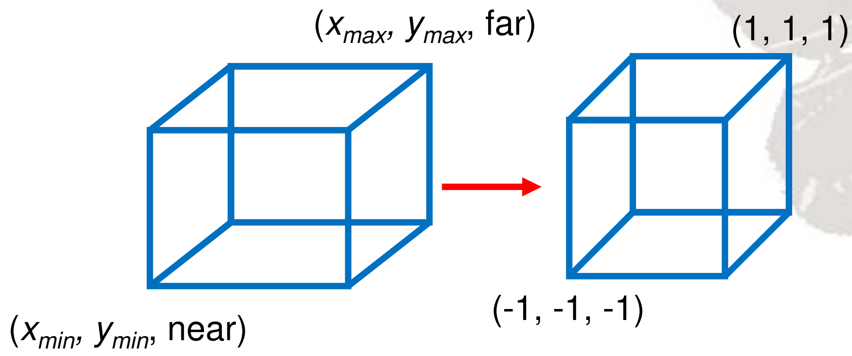


$$x_p = \frac{x'}{w'} = x \cdot \frac{z_{near}}{z} \qquad y_p = \frac{y'}{w'} = y \cdot \frac{z_{near}}{z} \qquad z_p = \frac{z'}{w'} = \frac{S_z \cdot z + T_z}{z}$$

$$\begin{cases} z_p = z_{near}, \text{if } z = z_{near} \\ z_p = z_{far}, \quad \text{if } z = z_{far} \end{cases} \qquad \begin{cases} \dfrac{S_z \cdot z_{near} + T_z}{z_{near}} = z_{near} \\ \dfrac{S_z \cdot z_{far} + T_z}{z_{far}} = z_{far} \end{cases} \qquad \begin{cases} S_z = z_{far} + z_{near} \\ T_z = -z_{far} \cdot z_{near} \end{cases}$$

Perspective Normalization

- **From Cuboid to Cube**
  - **Similar to orthogonal normalization**



$$M_{orthonorm} = S\left(\frac{2}{x_{\max} - x_{\min}}, \frac{2}{y_{\max} - y_{\min}}, \frac{2}{z_{far} - z_{near}}\right)$$

$$\cdot T\left(-\frac{x_{\max} + x_{\min}}{2}, -\frac{y_{\max} + y_{\min}}{2}, -\frac{z_{far} + z_{near}}{2}\right)$$

Perspective Normalization

$$M_{persnorm} = \underbrace{\begin{bmatrix} \frac{2}{x_{\max}-x_{\min}} & 0 & 0 & -\frac{x_{\max}+x_{\min}}{x_{\max}-x_{\min}} \\ 0 & \frac{2}{y_{\max}-y_{\min}} & 0 & -\frac{y_{\max}+y_{\min}}{y_{\max}-y_{\min}} \\ 0 & 0 & \frac{2}{z_{far}-z_{near}} & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{From cuboid to cube}} \cdot \underbrace{\begin{bmatrix} z_{near} & 0 & 0 & 0 \\ 0 & z_{near} & 0 & 0 \\ 0 & 0 & z_{far}+z_{near} & -z_{far}z_{near} \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{From frustum to cuboid}}$$

$$= \boxed{\begin{bmatrix} \frac{2\cdot z_{near}}{x_{\max}-x_{\min}} & 0 & -\frac{x_{\max}+x_{\min}}{x_{\max}-x_{\min}} & 0 \\ 0 & \frac{2\cdot z_{near}}{y_{\max}-y_{\min}} & -\frac{y_{\max}+y_{\min}}{y_{\max}-y_{\min}} & 0 \\ 0 & 0 & \frac{z_{far}+z_{near}}{z_{far}-z_{near}} & \frac{-2z_{far}z_{near}}{z_{far}-z_{near}} \\ 0 & 0 & 1 & 0 \end{bmatrix}}$$

Final perspective normalization matrix

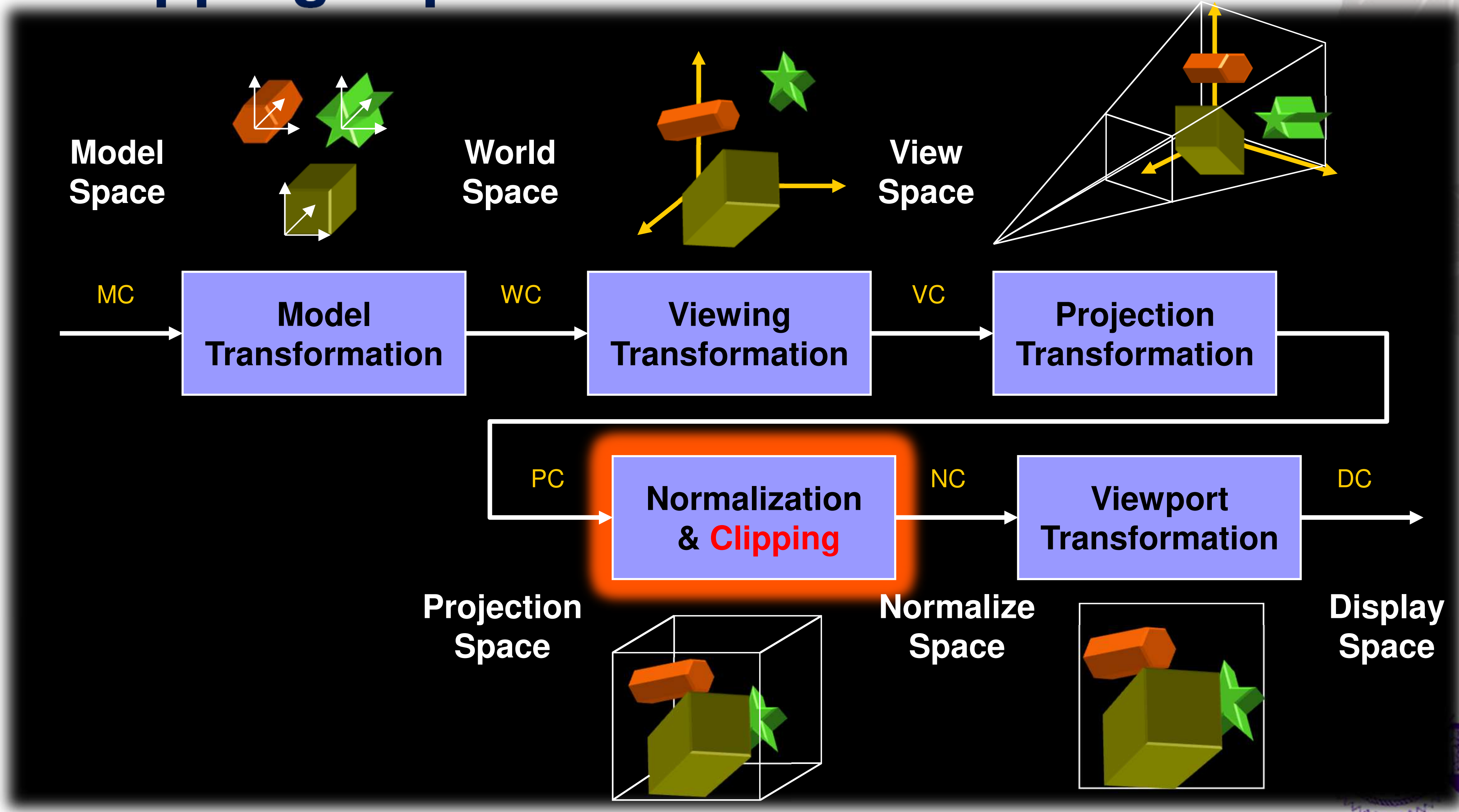
Clipping

2D Clipping

View Volume Clipping

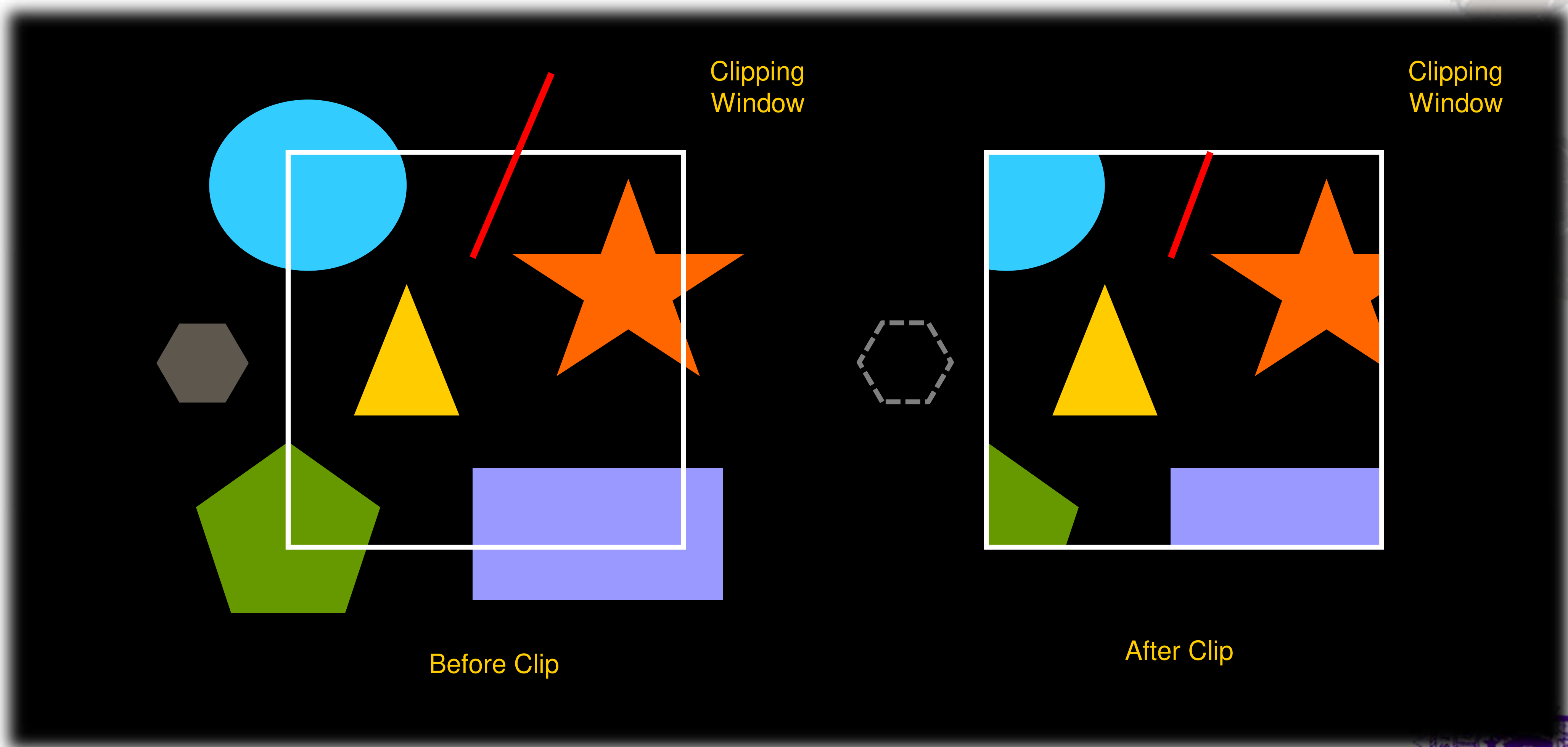
Review of 3D Viewing Process

- Clipping is performed after normalization



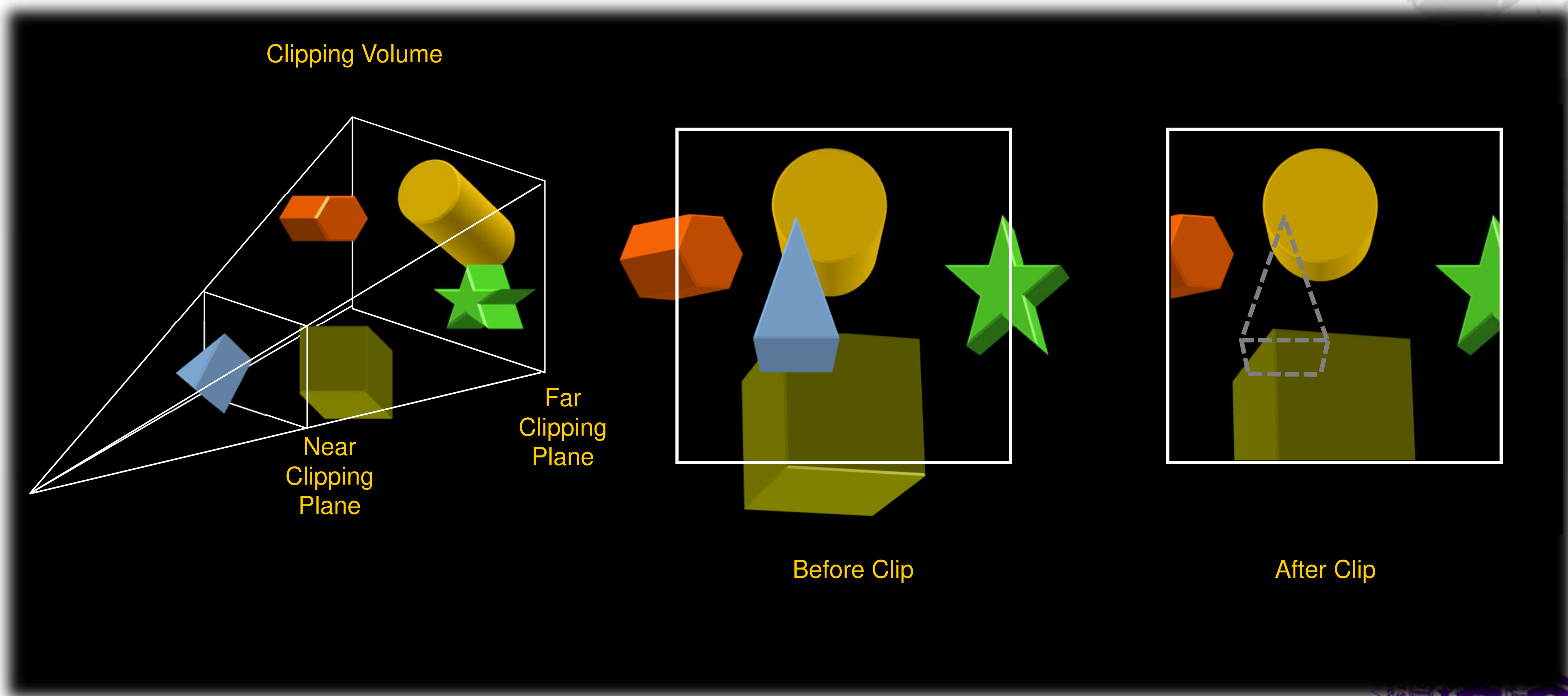
2D Clipping

- Clipping against 2D clipping window



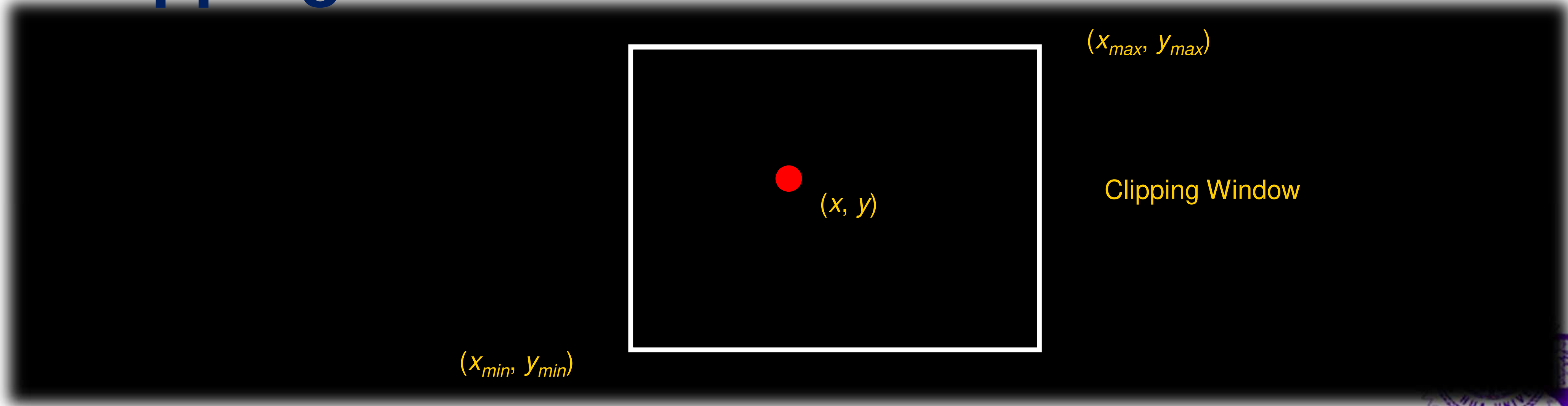
3D Clipping

- Clipping against 3D clipping volume



2D Clipping

- Define a clipping window with (x_{min}, y_{min}) and (x_{max}, y_{max}) as the lower-left and top-right coordinates, respectively.
- $x_{min} \le x \le x_{max}$ and $y_{min} \le y \le y_{max}$ must be satisfied for a point (x, y) to be inside the clipping window

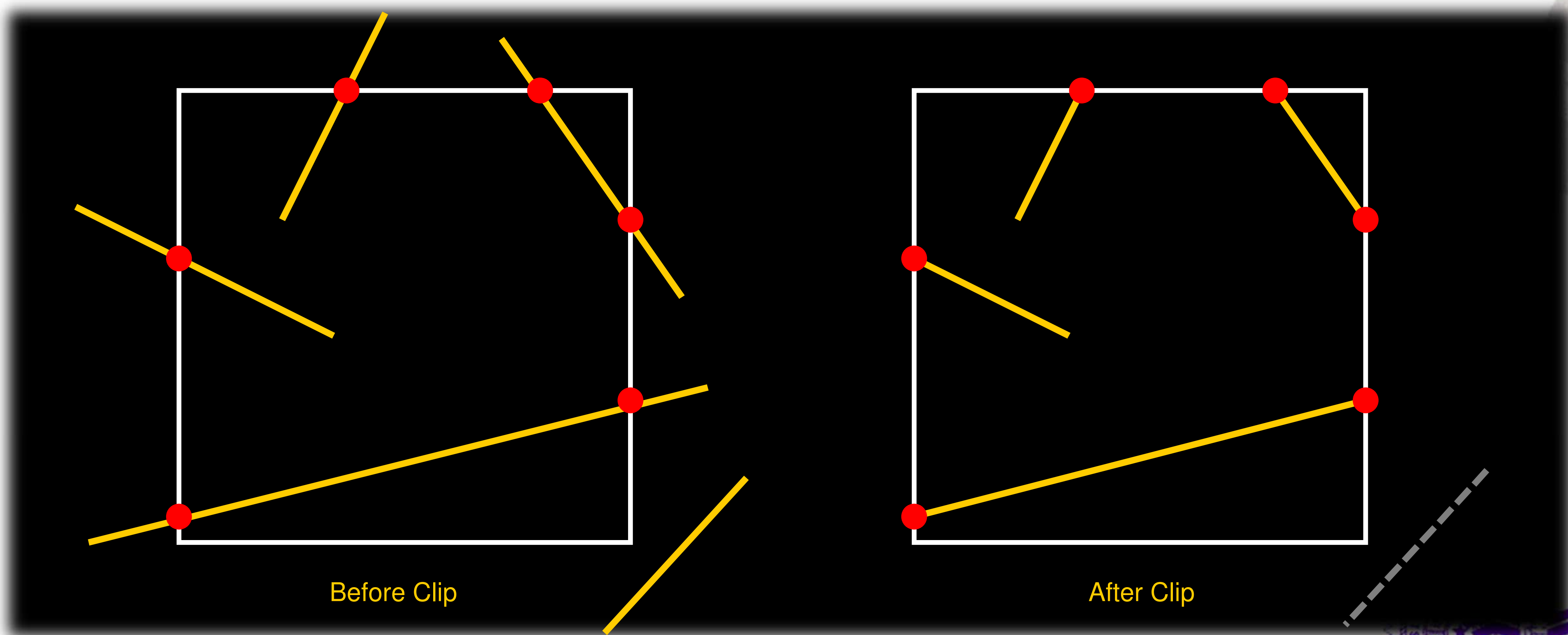


2D Clipping

- Clipping line segments
- Clipping polygons
- Other primitives can be converted into line segments or polygons before clipping

Clipping 2D Line Segments

- Compute intersections with each side of clipping window



Clipping 2D Line Segments

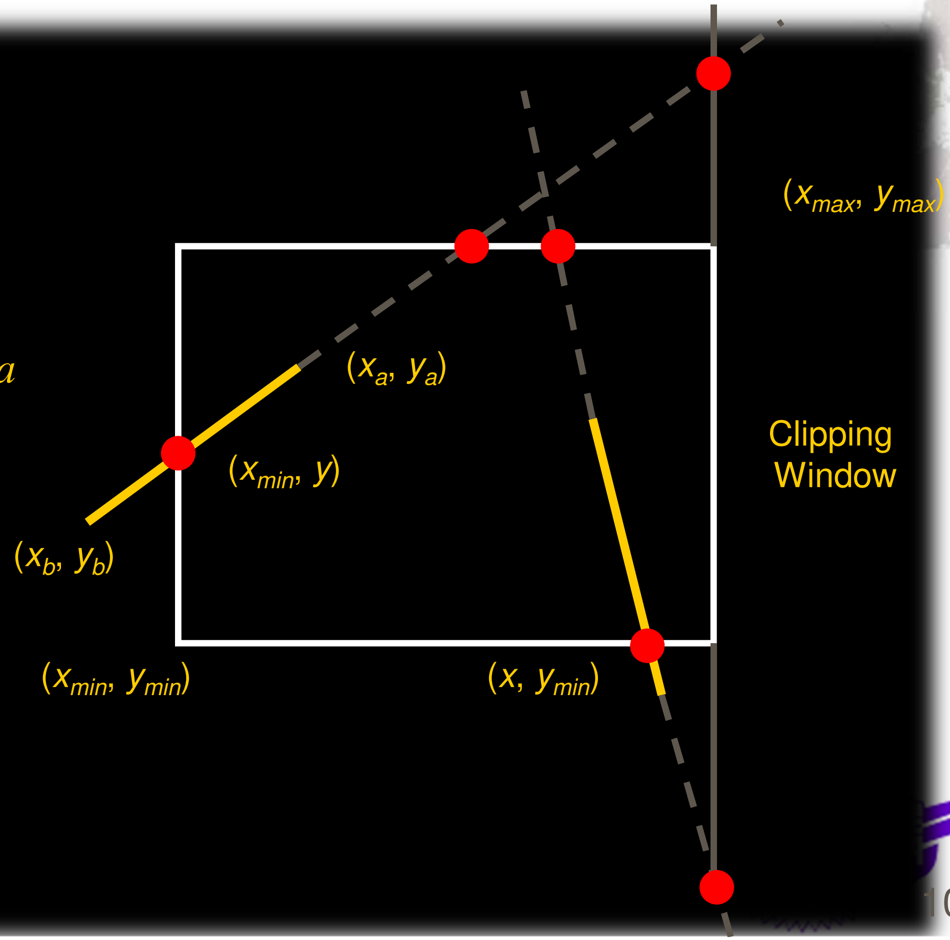
- For each line segment, $y = mx+b$, compute the intersection with each side of clipping window

$$\begin{cases} y = mx+b \\ x = x_{\min} \end{cases} \qquad y_{\min} \leq y \leq y_{\max}$$

$$\begin{cases} y = mx+b \\ x = x_{\max} \end{cases} \qquad y_b \leq y \leq y_a \text{ if } y_b \leq y_a$$

$$\begin{cases} y = mx+b \\ y = y_{\min} \end{cases} \qquad x_{\min} \leq x \leq x_{\max}$$

$$\begin{cases} y = mx+b \\ y = y_{\max} \end{cases} \qquad x_b \leq x \leq x_a \text{ if } x_b \leq x_a$$



Clipping 2D Line Segments

- **Compute intersection point using parametric equations**
  - **Example for intersection with $y = y_{min}$**

$$\begin{cases} x = x_a + t_0(x_b - x_a) \\ y = y_a + t_0(y_b - y_a) \end{cases} \qquad \begin{cases} x = x_{\min} + t_1(x_{\max} - x_{\min}) \\ y = y_{\min} \end{cases}$$

$$\begin{cases} x = x_a + t_0(x_b - x_a) = x_{\min} + t_1(x_{\max} - x_{\min}) \\ y = y_a + t_0(y_b - y_a) = y_{\min} \end{cases}$$

Solving t_0 and t_1, assume $t_0 = a$ and $t_1 = b$,

if $0 \le a \le 1$ and $0 \le b \le 1$, then

$x = x_{\min} + b(x_{\max} - x_{\min})$, $y = y_{\min}$ is the intersection point

Clipping 2D Line Segments

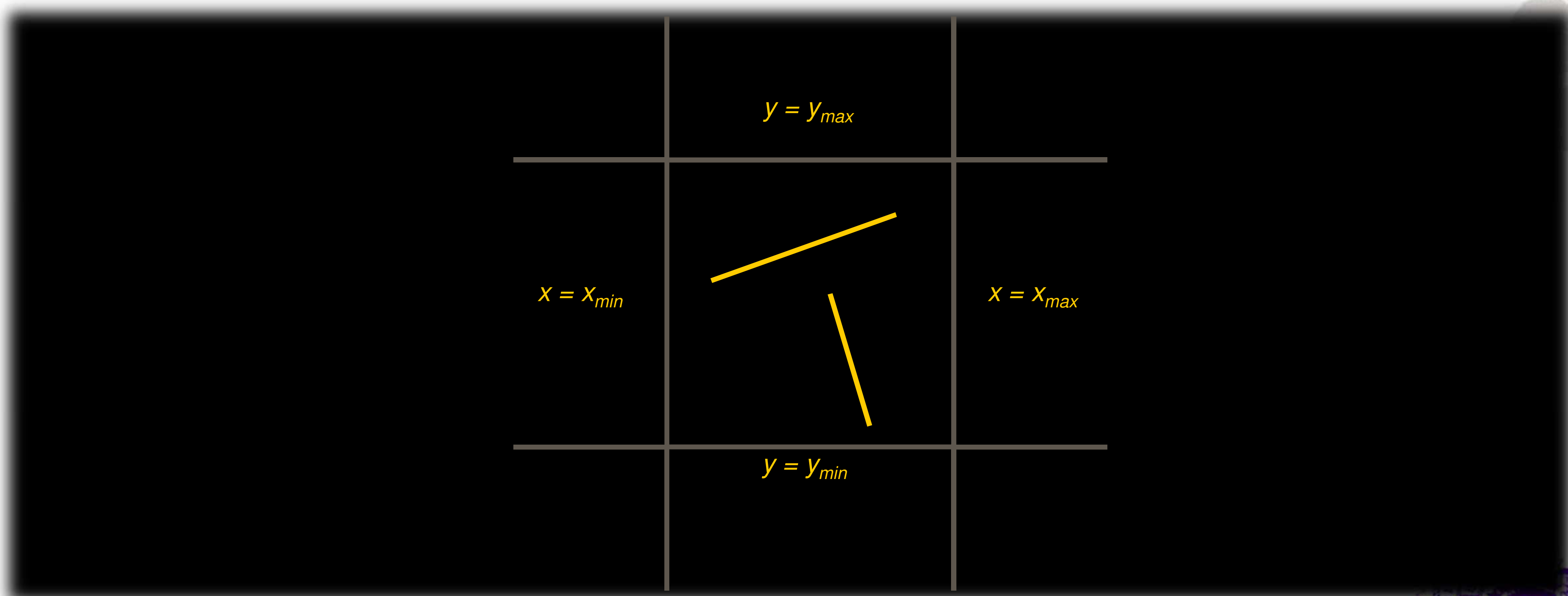
- Intersection derivation costs a lot of computations, such as division and multiplication.
- Reduce the number of intersection calculation can greatly speed up the clipping process.
- Do the intersection calculation only when it is necessary

Clipping 2D Line Segments

- Cohen-Sutherland line clipping algorithm
  - Step 1: check for trivially accepted
  - Step 2: Region check for trivially rejected
  - Step 3: Subdivide the line segment, repeat Step 1 and Step 2 until what remains is complete inside the clip rectangle or can be trivially rejected

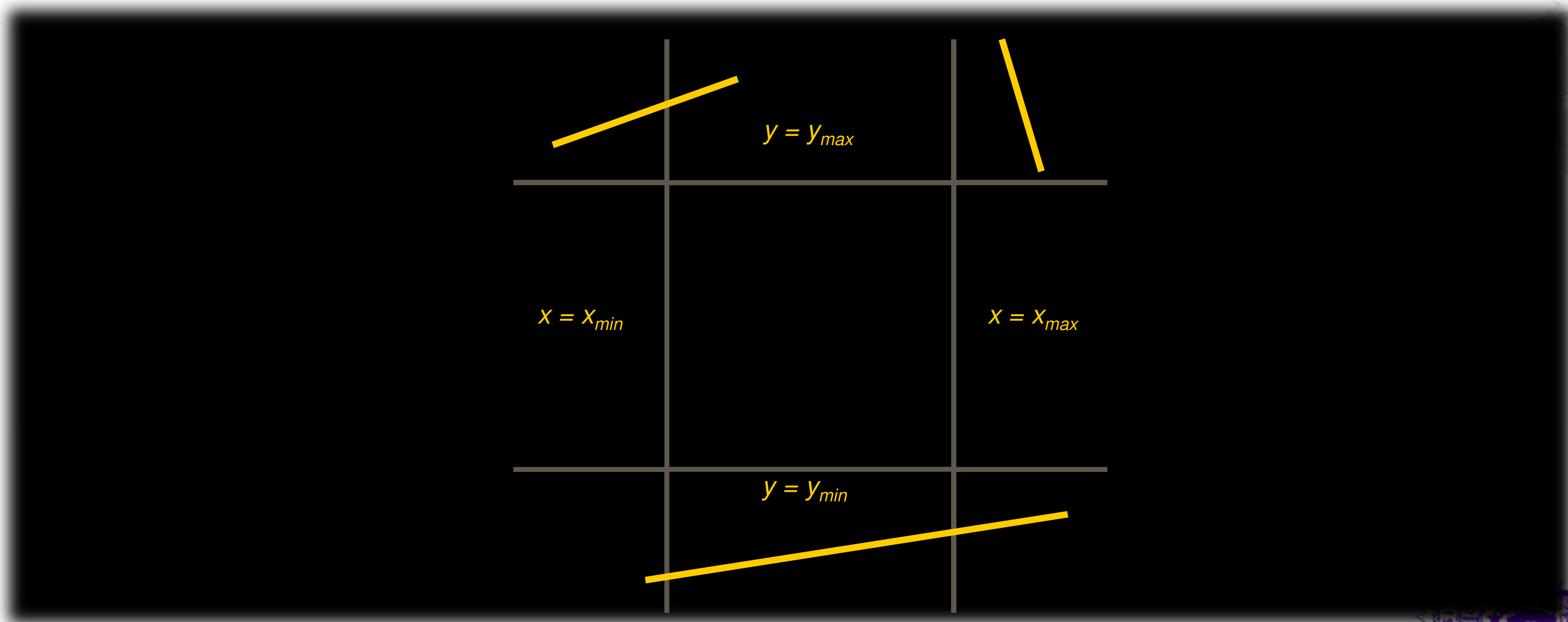
Cohen-Sutherland Algorithm

- Case for trivially accepted
  - Both endpoints of a line segments lie inside the clipping rectangle



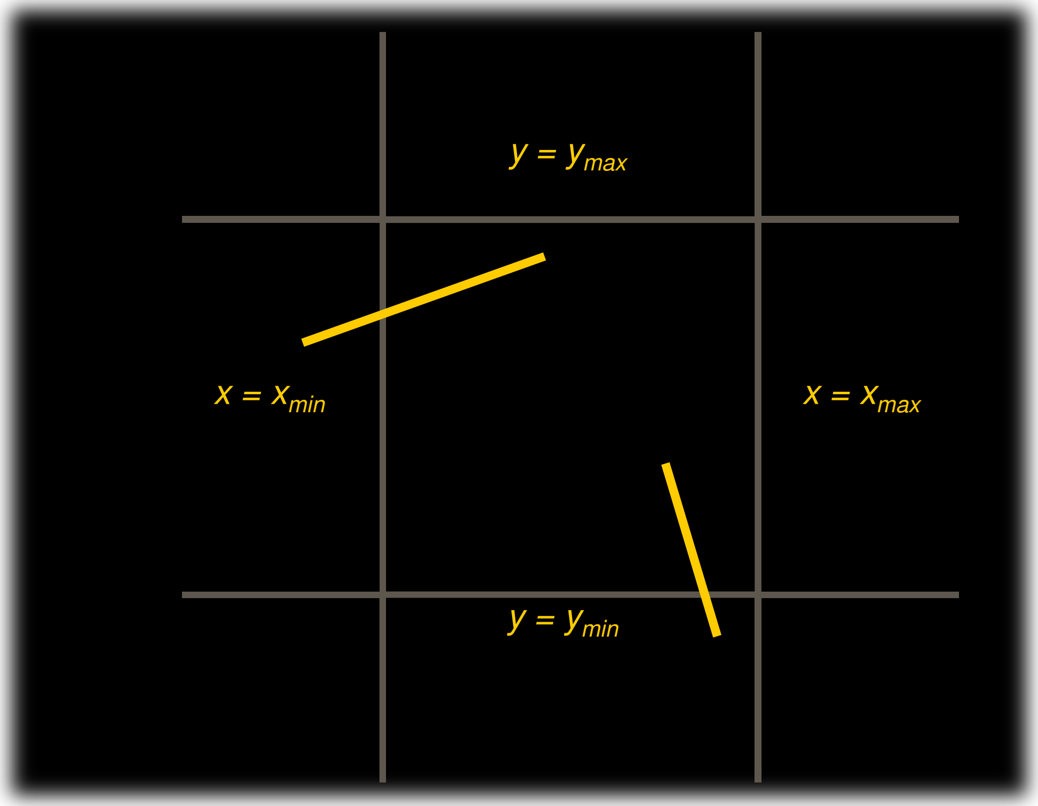
Cohen-Sutherland Algorithm

- Case for trivially rejected
  - Both endpoints of a line segments lie outside one of the clipping boundary



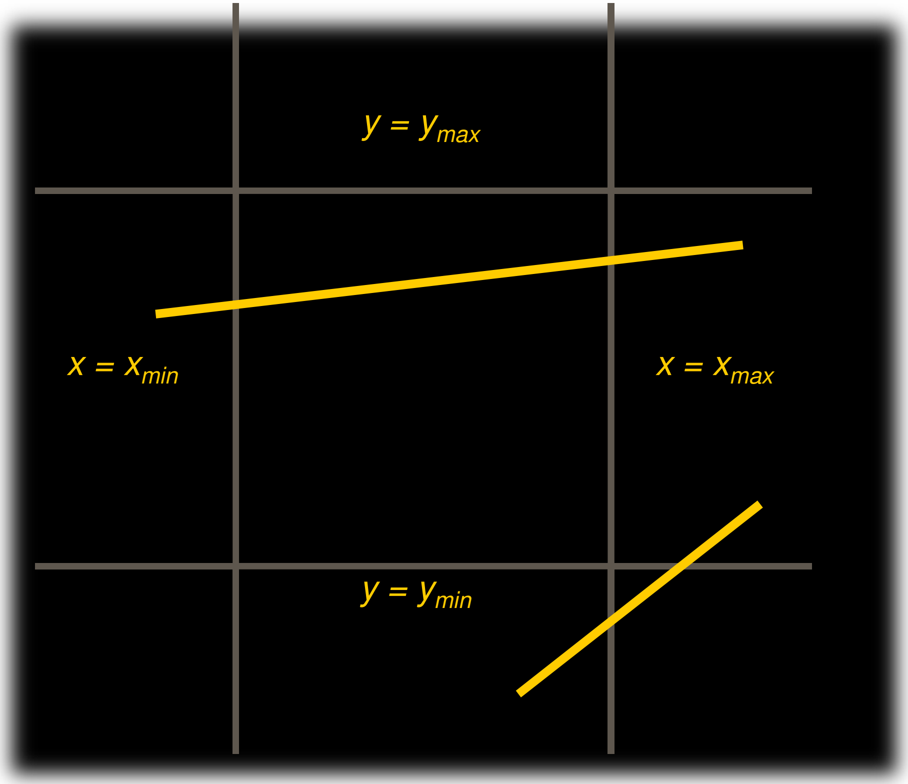
Cohen-Sutherland Algorithm

- Cases that require intersection calculation
  - One endpoint of a line segments lies inside the clipping rectangle and the other endpoint is outside



Cohen-Sutherland Algorithm

- Cases that require intersection calculation
  - Both endpoints of a line segments lie outside the clipping rectangle and the endpoints are not on the same side of a clipping boundary



Cohen-Sutherland Algorithm

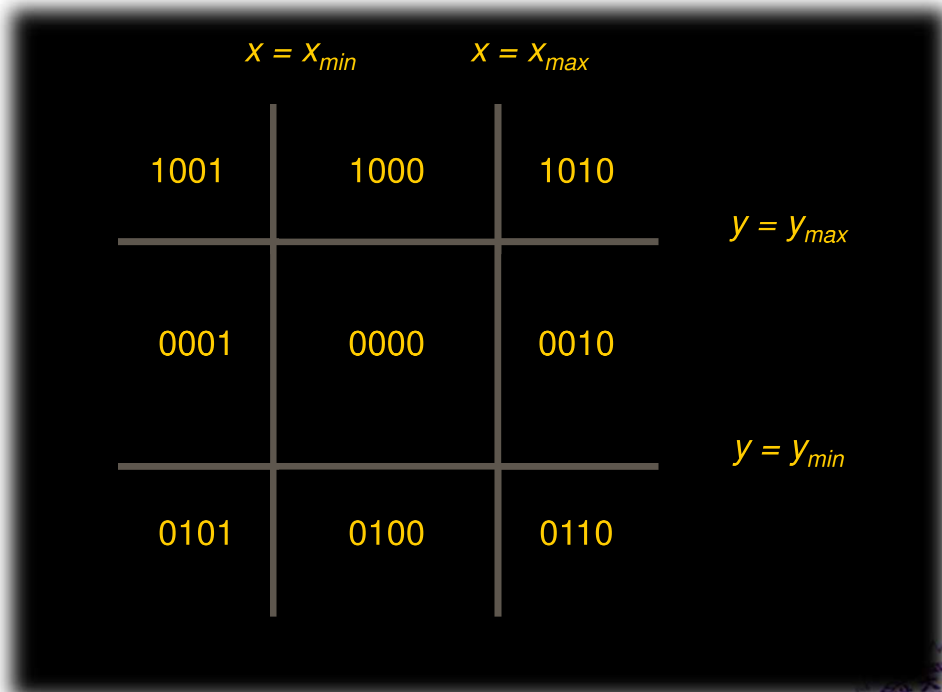
- Outcodes
  - For each endpoint (x, y) of a line segment, an outcode (b_0, b_1, b_2, b_3) is defined as

$b_0 = 1$, if $y > y_{max}$

$b_1 = 1$, if $y < y_{min}$

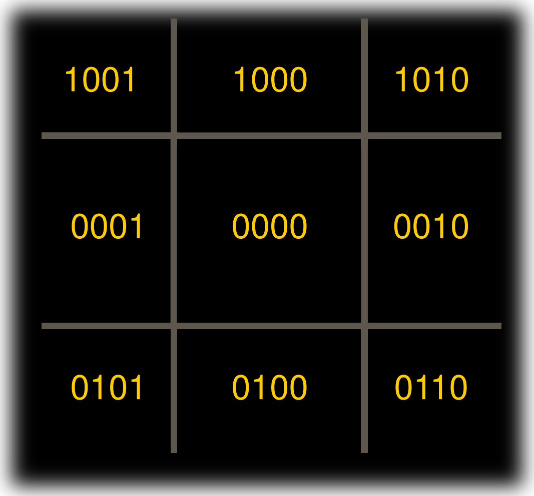
$b_2 = 1$, if $x > x_{max}$

$b_3 = 1$, if $x < x_{min}$



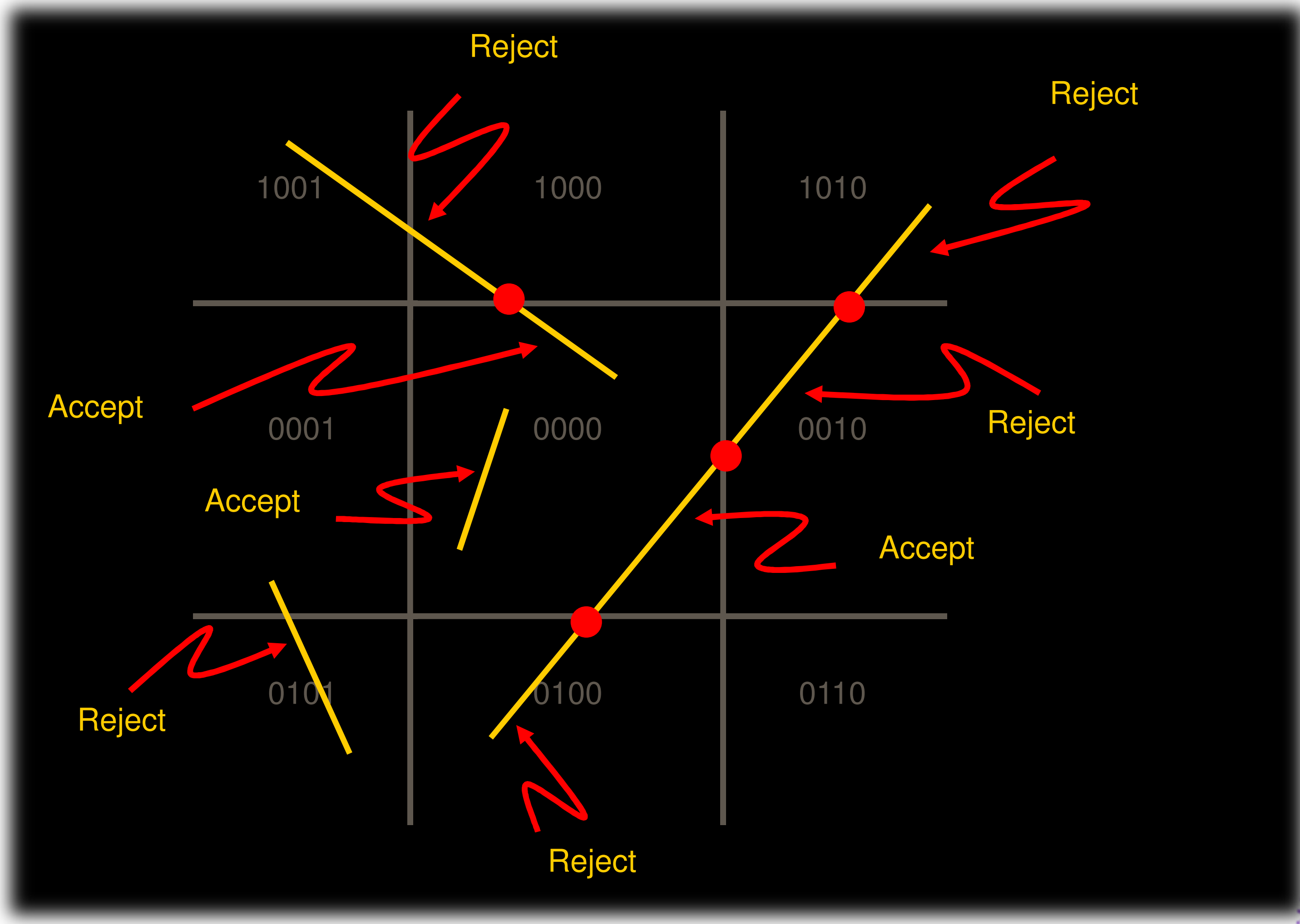
Cohen-Sutherland Algorithm

- **Assume $(b_0, b_1, b_2, b_3)_A$ and $(b_0, b_1, b_2, b_3)_B$ are the outcodes of endpoints (A, B) of a line segment, respectively**
  - **If $((b_0, b_1, b_2, b_3)_A)$ & $((b_0, b_1, b_2, b_3)_B) \neq 0000$ implies trivially rejected**
  - **if $(b_0, b_1, b_2, b_3)_A = (b_0, b_1, b_2, b_3)_B = 0000$ implies trivially accepted**



Cohen-Sutherland Algorithm

- Examples

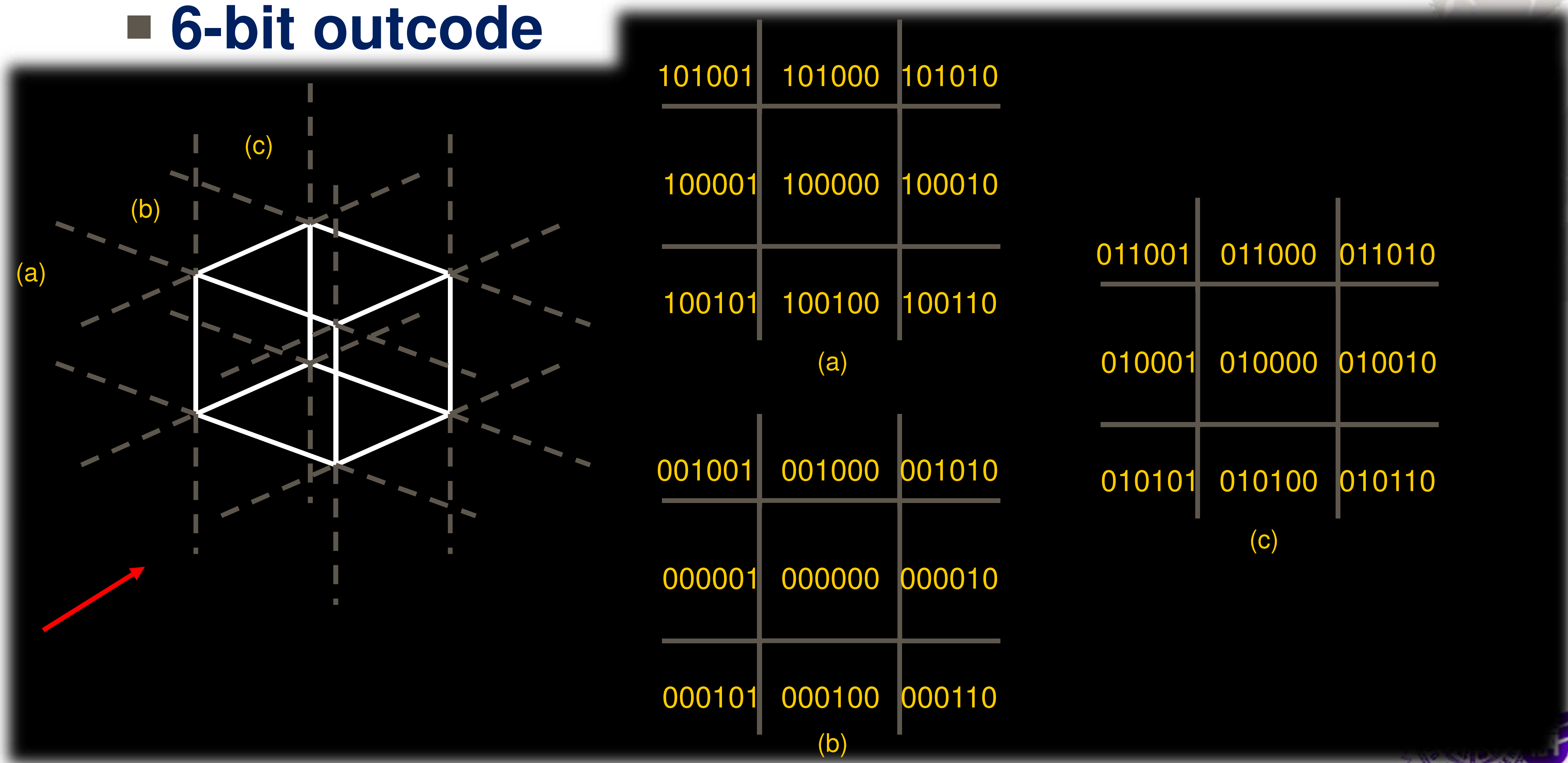


Cohen-Sutherland Algorithm

- Advantages
  - Most line segments can be easily accepted or rejected without any intersection calculation
  - Less intersection calculations are required in compare to brute force approach

Clipping in 3D

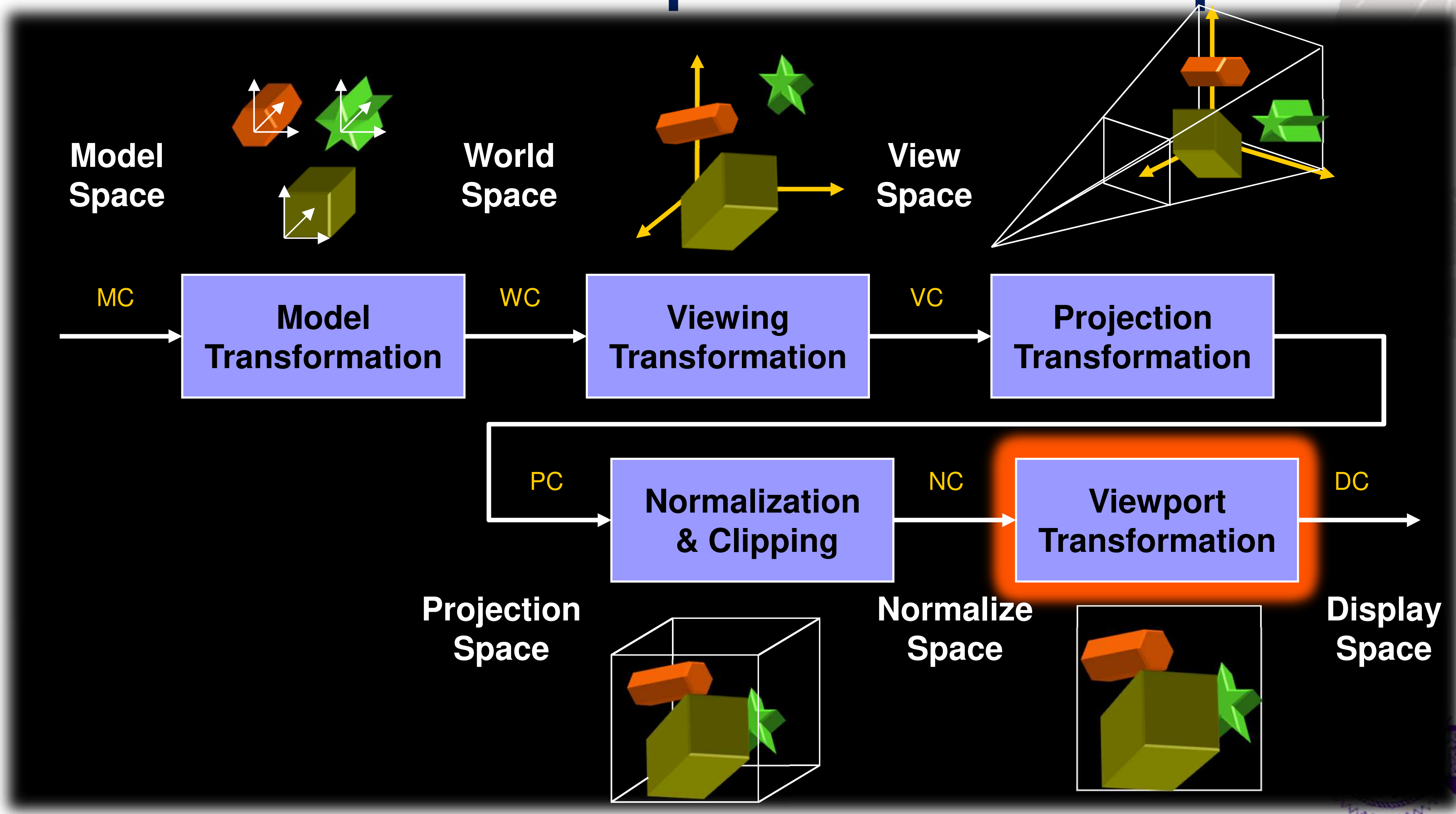
- Extension of Cohen-Sutherland algorithm
  - 6-bit outcode



Viewport Transformation

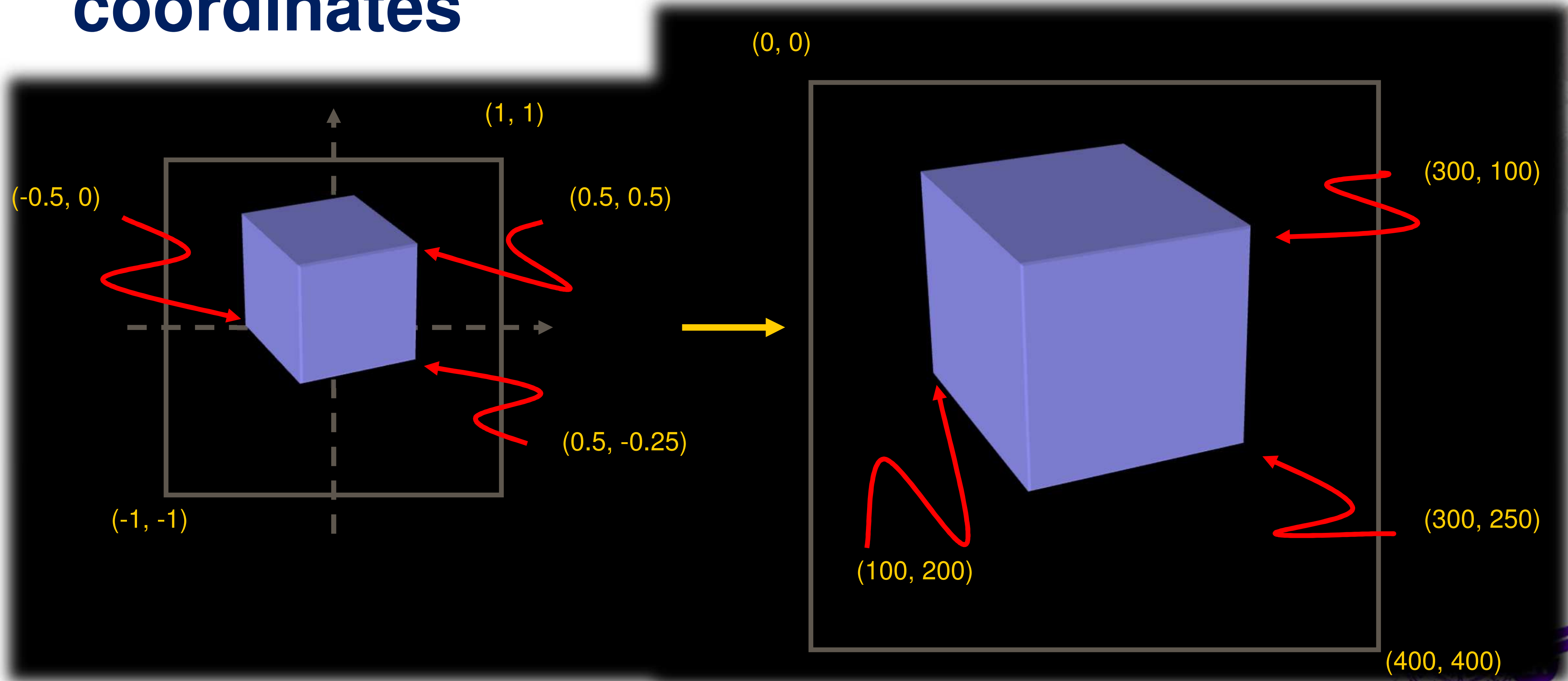
Review of 3D Viewing Process

◆ From Normalized Space to Screen Space



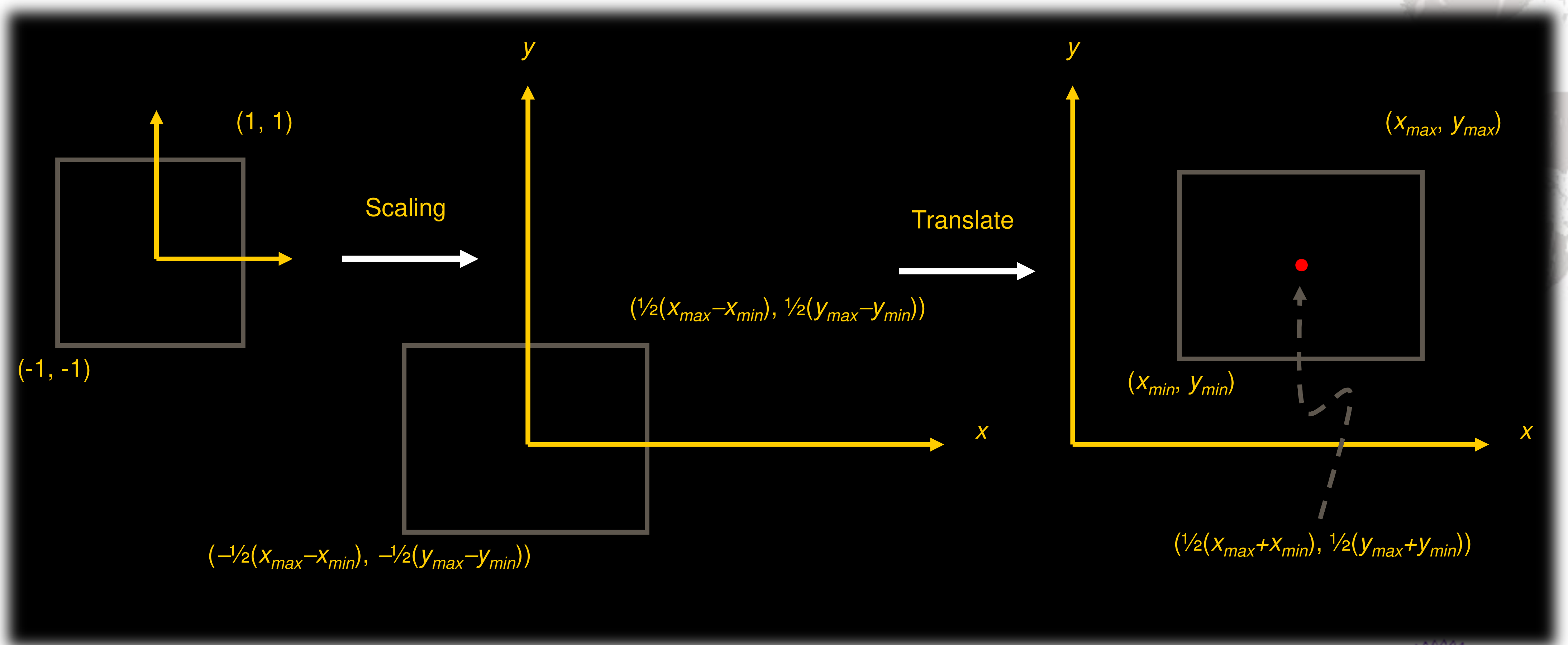
Viewport Transformation

- Transform from normalized clipping coordinates to 2D display device coordinates



Viewport Transformation

- Composition of viewport transformation



Viewport Transformation

- **Matrix representation of viewport transformation**

$$M_{viewport} = T\left(\frac{x_{\max}+x_{\min}}{2}, \frac{y_{\max}+y_{\min}}{2}, 0\right)\cdot S\left(\frac{x_{\max}-x_{\min}}{2}, \frac{y_{\max}-y_{\min}}{2}, 1\right)$$

$$= \begin{bmatrix} \frac{x_{\max}-x_{\min}}{2} & 0 & 0 & \frac{x_{\max}+x_{\min}}{2} \\ 0 & \frac{y_{\max}-y_{\min}}{2} & 0 & \frac{y_{\max}+y_{\min}}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewport Transformation

- **Depth Range Adjustment**
  - **E.g., set the depth range within [0, 1]**

$$M_{viewport} = T\left(\frac{x_{\max}+x_{\min}}{2}, \frac{y_{\max}+y_{\min}}{2}, \boxed{0.5}\right)\cdot S\left(\frac{x_{\max}-x_{\min}}{2}, \frac{y_{\max}-y_{\min}}{2}, \boxed{\frac{1}{2}}\right)$$

Translate the range from [-0.5, 0.5] to [0, 1]

Scale the range from [-1, 1] to [-0.5, 0.5]

OpenGL Transformations

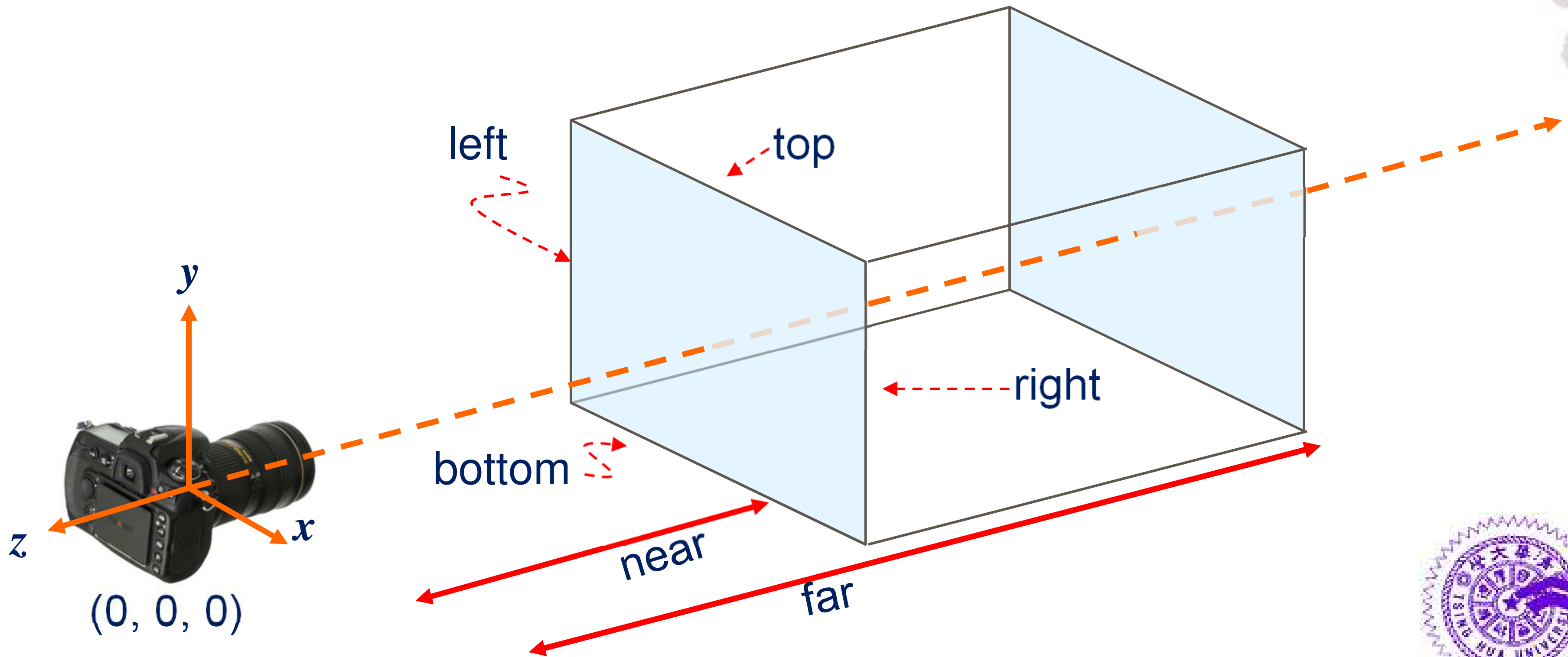
glOrtho
glFrustum
gluPerspective
gluLookAt

OpenGL Conventions

- In order to derive the similar equations for OpenGL glOrtho, glFrustum, and gluPerspective, the following terms should be aware of
  - $x_{max} = Right;\ x_{min} = Left$
  - $y_{max} = Top;\ y_{min} = Bottom$
  - $z_{near} = -Near;\ z_{far} = -Far$
- It is because the definition of Near and Far in the APIs are the distances to the viewpoint instead of the Z coordinates

glOrtho

- **OpenGL Orthographic Projection**
  - glOrtho(*left*, *right*, *bottom*, *top*, *near*, *far*)
    - *left*, *right*, *bottom*, *top*: coordinates
    - *near*, *far*: distances to viewpoint.



glOrtho

- **OpenGL Orthographic Transformation Matrix**
  - **Orthographic (parallel) projection and orthographic normalization**

$$\begin{pmatrix} \frac{2}{Right-Left} & 0 & 0 & t_x \\ 0 & \frac{2}{Top-Bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{Far-Near} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

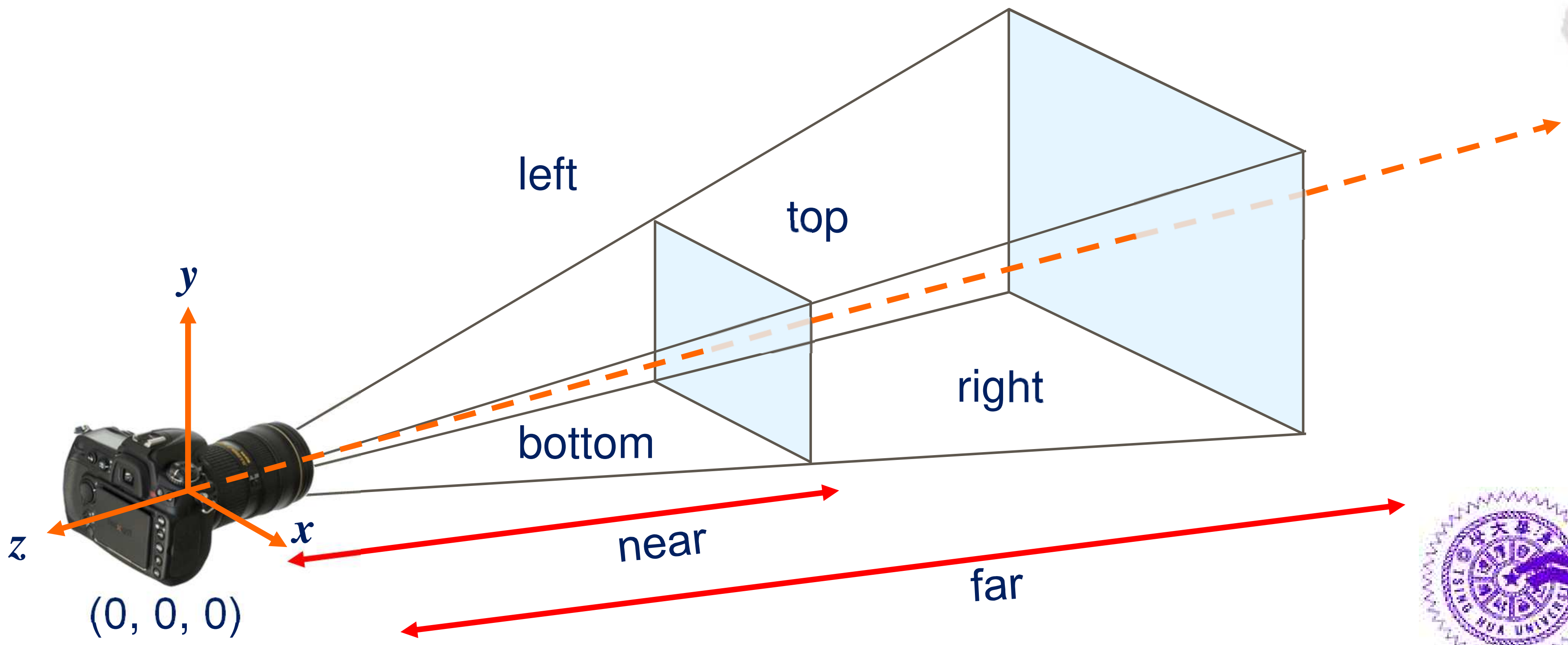
$$t_x = -\frac{Right+Left}{Right-Left}$$

$$t_y = -\frac{Top+Bottom}{Top-Bottom}$$

$$t_z = -\frac{Far+Near}{Far-Near}$$

glFrustum

- **OpenGL Perspective Projection**
  - **glFrustum(*left*, *right*, *bottom*, *top*, *near*, *far*)**
    - ***left*, *right*, *bottom*, *top*: coordinates**
    - ***near*, *far*: distances to viewpoint**



glFrustum

- **OpenGL Perspective Transformation Matrix**
  - **Perspective projection and perspective normalization**

$$\begin{pmatrix} \frac{2\,Near}{Right-Left} & 0 & A & 0 \\ 0 & \frac{2\,Near}{Top-Bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$A = \frac{Right + Left}{Right - Left}$$

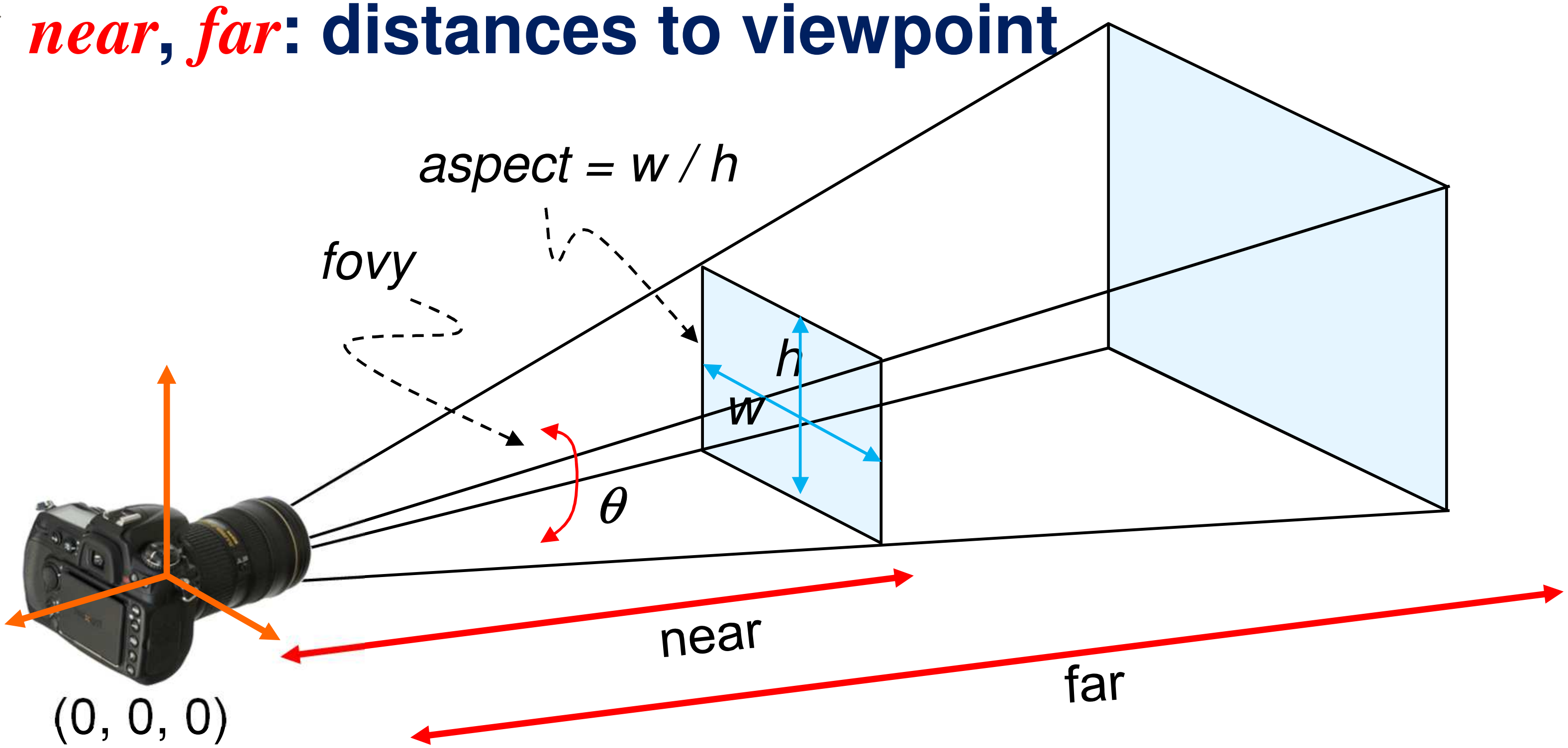
$$B = \frac{Top + Bottom}{Top - Bottom}$$

$$C = -\frac{Far + Near}{Far - Near}$$

$$D = -\frac{2Far\,Near}{Far - Near}$$

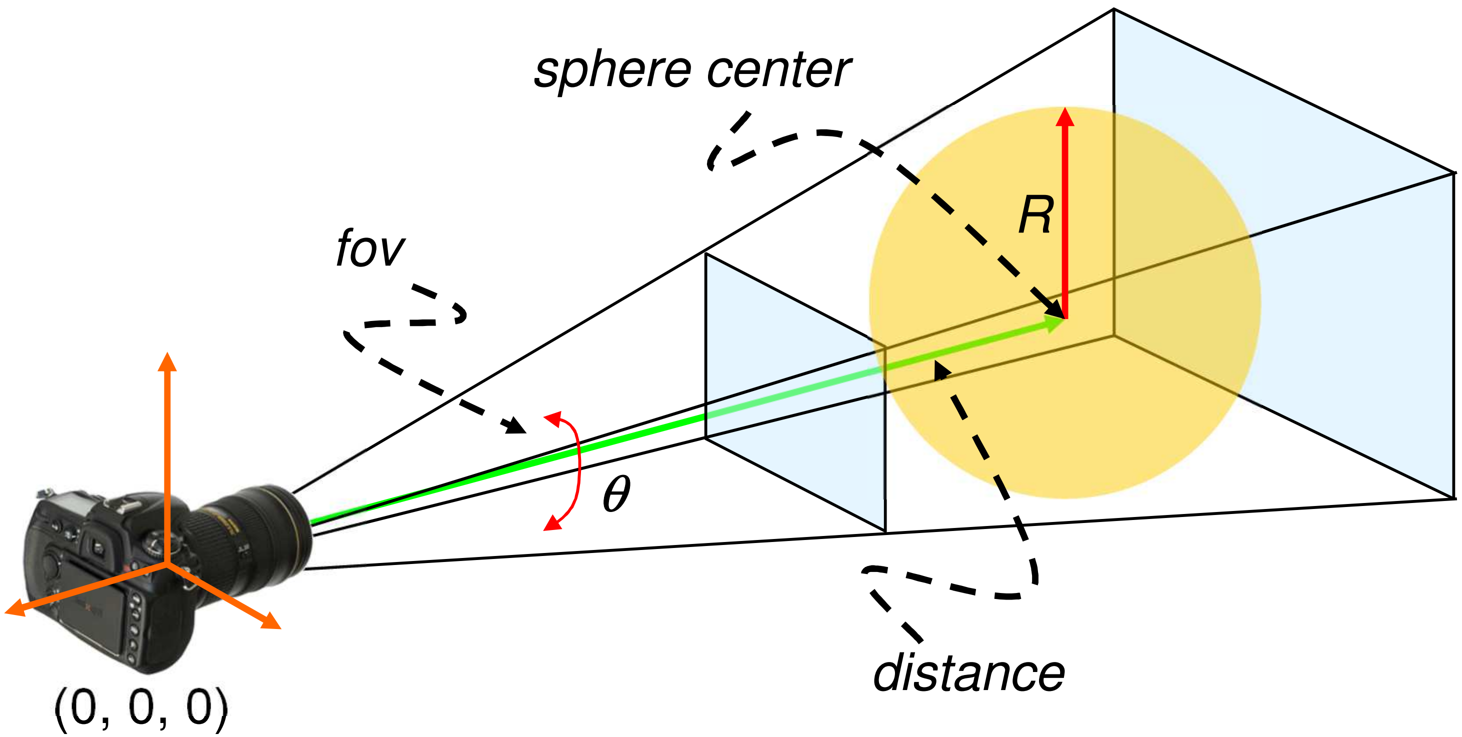
gluPerspective

- OpenGL Perspective Projection
  - gluPerspective(*fovy*, *aspect*, *near*, *far*)
    - *fovy*: field of view angle
    - *aspect*: aspect ratio of width to height
    - *near*, *far*: distances to viewpoint



gluPerspective

- How to derive a good field of view angle
  - Determine the bounding sphere of the scene
  - $fov = 2 \times \tan^{-1}(R/distance)$
    - *R*: sphere radius; *distance*: camera to sphere center



gluPerspective

- **OpenGL Perspective Transformation Matrix**
  - **Perspective projection and perspective normalization**

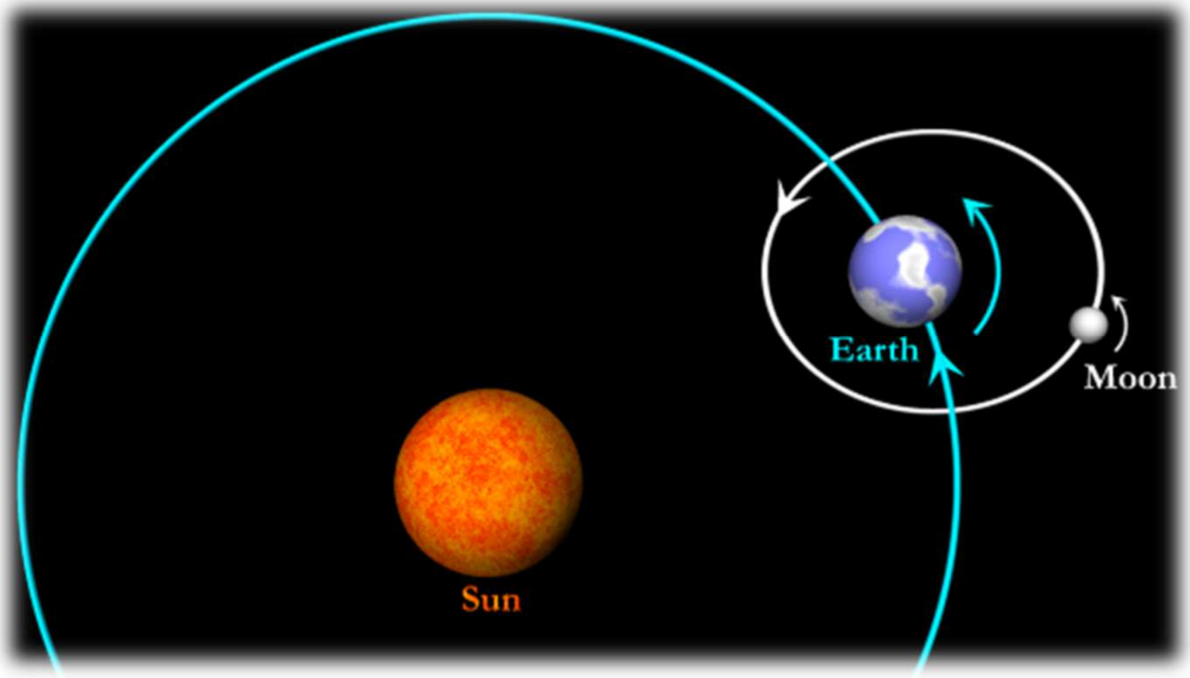
$$\begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{far + near}{near - far} & \frac{2 \cdot far \cdot near}{near - far} \\ 0 & 0 & -1 & 0 \end{pmatrix} \qquad f = \cot\left(\frac{fovy}{2}\right)$$

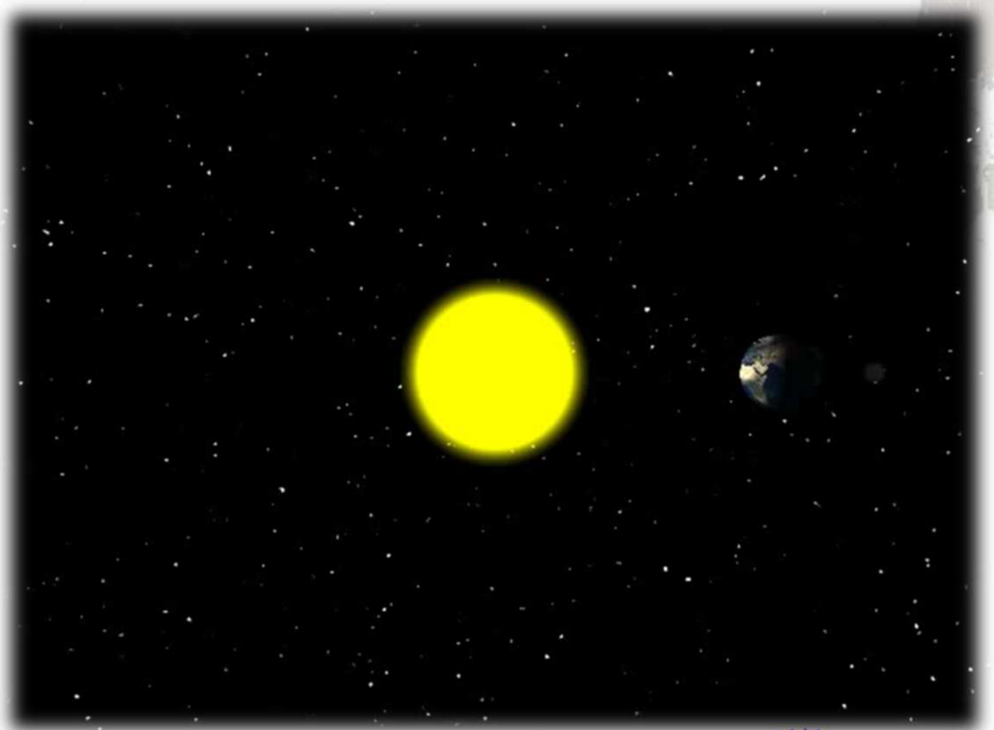
Managing Transformation Matrices

Matrix Stack

Why need to manage the matrices?

- Consider a simplified solar system with sun, earth, and moon



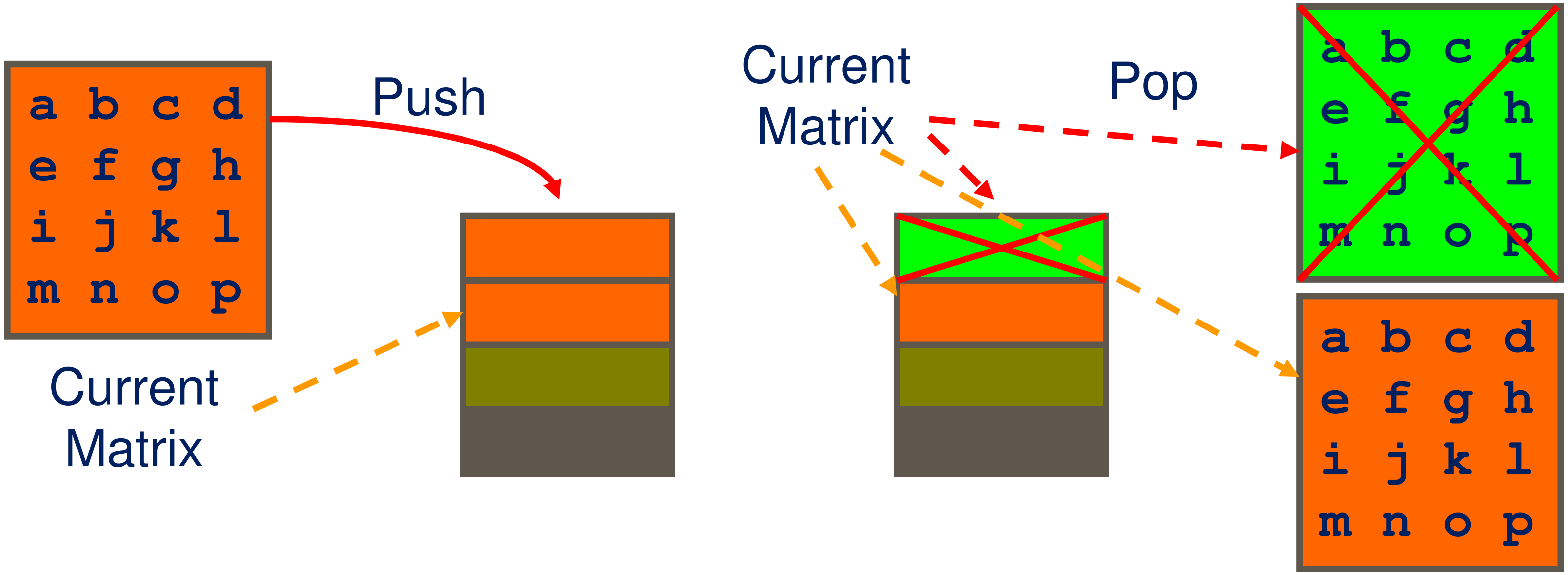


Matrix Stack

- Matrix stack is useful to remember some transformation matrix at certain point and restore it back when necessary
- OpenGL used to manage the matrix stack for user but is now deprecated
- You can implemented a similar mechanism by your own or use other way to manage your matrices

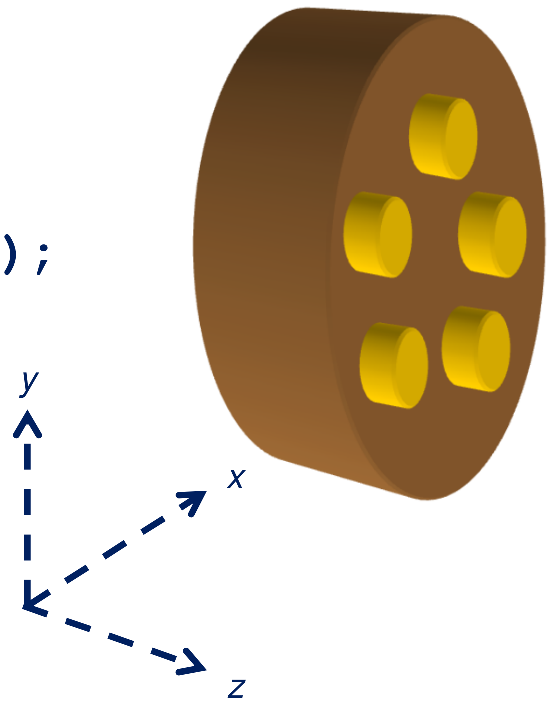
Matrix Stack in OpenGL

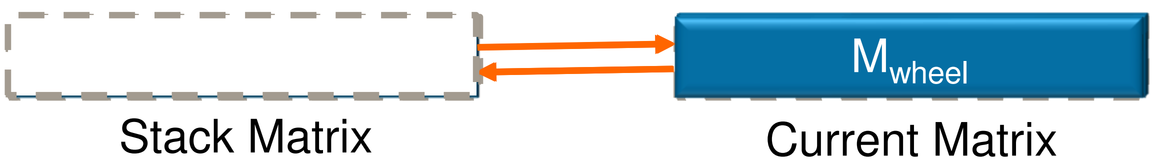
- Top of the matrix stack is the current matrix
- Any transformation is deal with the current matrix, i.e. the top of matrix stack
- Use *PushMatrix* or *PopMatrix* to push into or pop out of the matrix stack



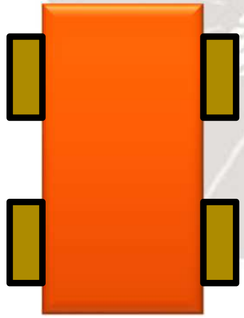
Example of Using Matrix Stack

```
draw_wheel_and_bolts()
{
   int i;
   draw_wheel();
   for(i=0;i<5;i++){
      glPushMatrix();
         glRotatef(72.0*i, 0.0, 0.0, 1.0);
         glTranslatef(3.0, 0.0, 0.0);
         draw_bolt();
      glPopMatrix();
   }
}
```

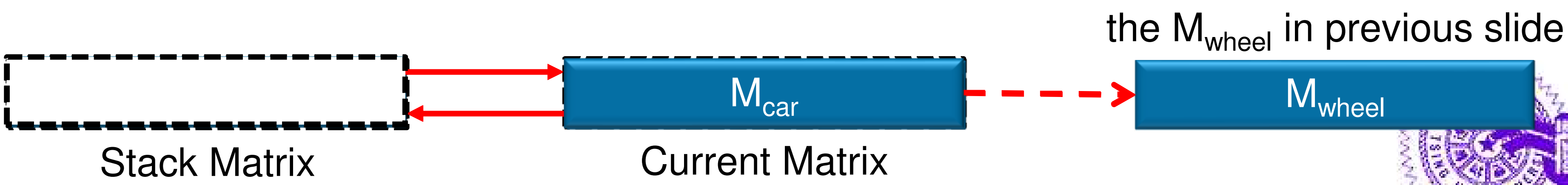




Example of Using Matrix Stack



```
draw_body_and_wheel_and_bolts()
{
    draw_car_body();
    glPushMatrix();
        glTranslatef(40, 0, 30);   /*move to first wheel position*/
        draw_wheel_and_bolts();
    glPopMatrix();
    glPushMatrix();
        glTranslatef(40, 0, -30);    /*move to 2nd wheel position*/
        draw_wheel_and_bolts();
    glPopMatrix();
    ...                          /*draw last two wheels similarly*/
}
```



Composition of 3D Transformations

◆ Example I: Building a Solar System

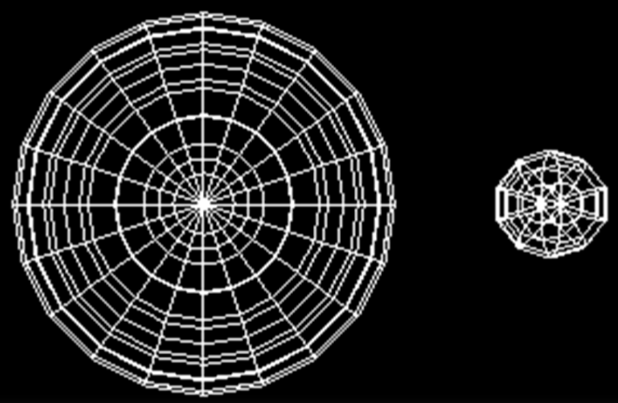
```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix();
    glutWireSphere(1.0, 20, 16);    /* draw sun */
    glRotatef((GLfloat) year, 0.0, 1.0, 0.0);
    glTranslatef(2.0, 0.0, 0.0);
    glRotatef((GLfloat) day, 0.0, 1.0, 0.0);
    glutWireSphere(0.2, 10, 8);     /* draw smaller planet */
    glPopMatrix();
    glutSwapBuffers();
}
```

Store the sun position

Eg. 360°×(date/365)

Eg. 360°×(hour/24)

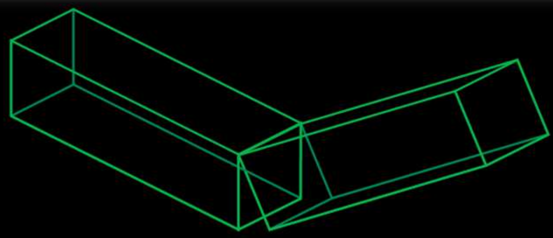
Restore the sun position



Composition of 3D Transformations

◆ Example II: Articulated Robot Arm

	`void display(void)`
	`{`
	`glClear(GL_COLOR_BUFFER_BIT);`
M_0	`glPushMatrix();`
M_1	`glTranslatef(-1.0, 0.0, 0.0);` `glRotatef((GLfloat) shoulder, 0.0, 0.0, 1.0);` `glTranslatef(1.0, 0.0, 0.0);`
$M_0 \cdot M_1$	`glPushMatrix();`
$M_0 \cdot M_1 \cdot S_1$	`glScalef(2.0, 0.4, 0.4);`
	`glutWireCube(1.0);`
$M_0 \cdot M_1$	`glPopMatrix();`
M_2	`glTranslatef(1.0, 0.0, 0.0);` `glRotatef((GLfloat) elbow, 0.0, 0.0, 1.0);` `glTranslatef(1.0, 0.0, 0.0);`
$M_0 \cdot M_1 \cdot M_2$	`glPushMatrix();`
$M_0 \cdot M_1 \cdot M_2 \cdot S_2$	`glScalef(2.0, 0.4, 0.4);`
	`glutWireCube(1.0);`
$M_0 \cdot M_1 \cdot M_2$	`glPopMatrix();`
M_0	`glPopMatrix();`
	`glutSwapBuffers();`
	`}`



Stack

Current

Stack				$M_0 \cdot M_1$	$M_0 \cdot M_1$	
Stack		M_0	M_0	M_0	M_0	M_0
Current	M_0	M_0	$M_0 \cdot M_1$	$M_0 \cdot M_1$	$M_0 \cdot M_1 \cdot S_1$	$M_0 \cdot M_1$

Stack		$M_0 \cdot M_1 \cdot M_2$	$M_0 \cdot M_1 \cdot M_2$		
Stack	M_0	M_0	M_0	M_0	
Current	$M_0 \cdot M_1 \cdot M_2$	$M_0 \cdot M_1 \cdot M_2$	$M_0 \cdot M_1 \cdot M_2 \cdot S_2$	$M_0 \cdot M_1 \cdot M_2$	M_0

Q&A