

---

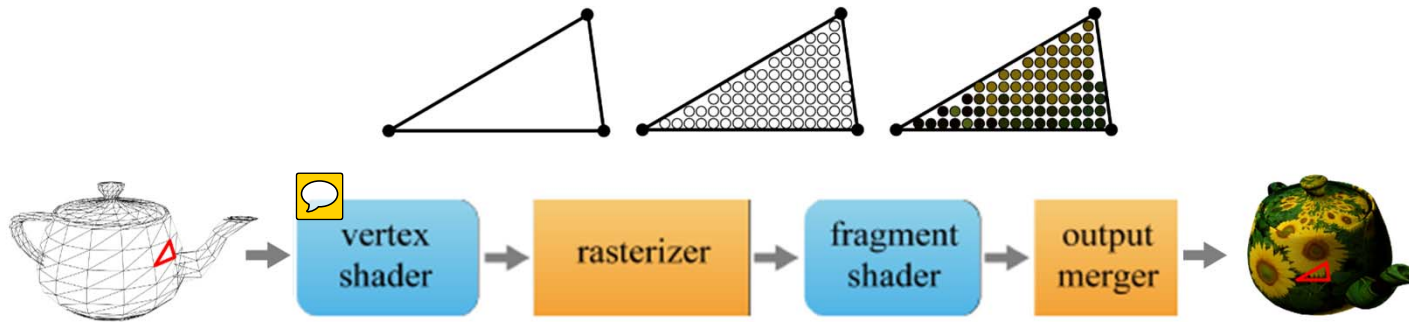
# **Chapter V**

## **Vertex Processing**

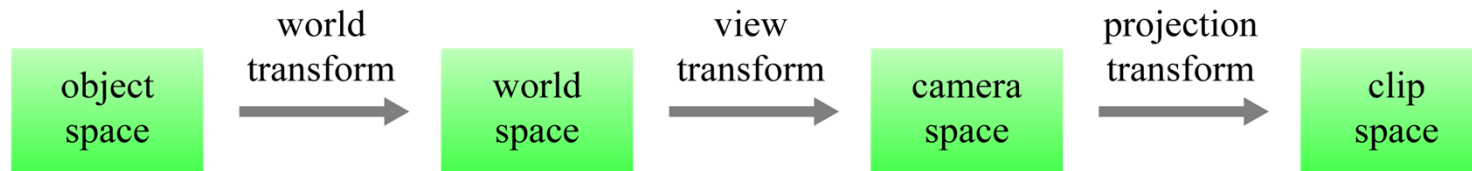
# GPU Rendering Pipeline

---

- Main stages in the *rendering pipeline*




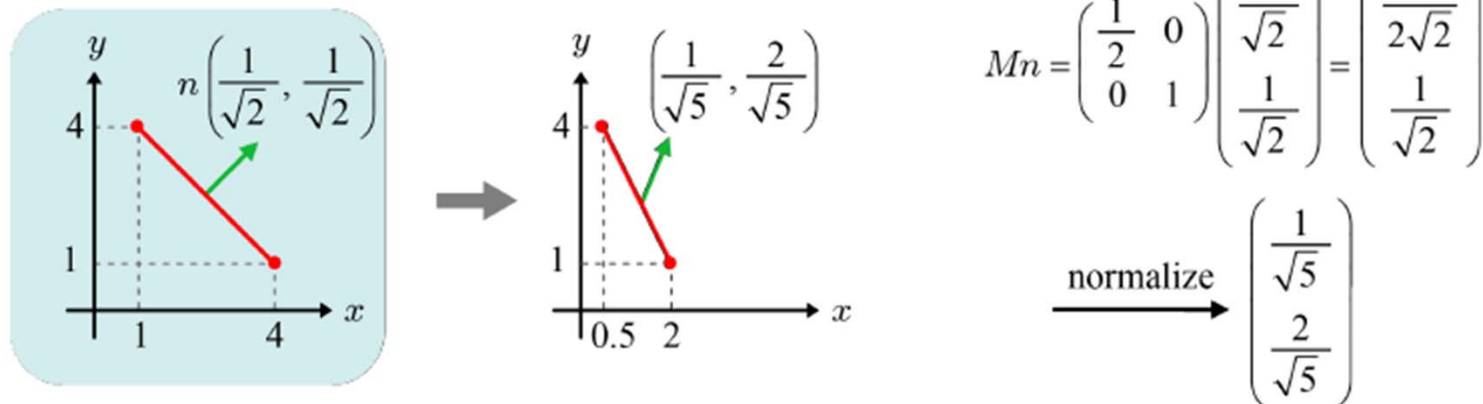
- The vertex shader (vertex program) operates on every input vertex stored in the vertex array and performs various operations.
- The essential among them is applying a series of *transforms* to the vertices.



- Let us present the view transform to the camera space and the projection transform to the clip space.

# Normal Transform

- If the world transform for vertex positions (denote by  $M$ ) includes a *non-uniform* scaling, it cannot be applied to surface normals. 
- Let's consider triangle normals.

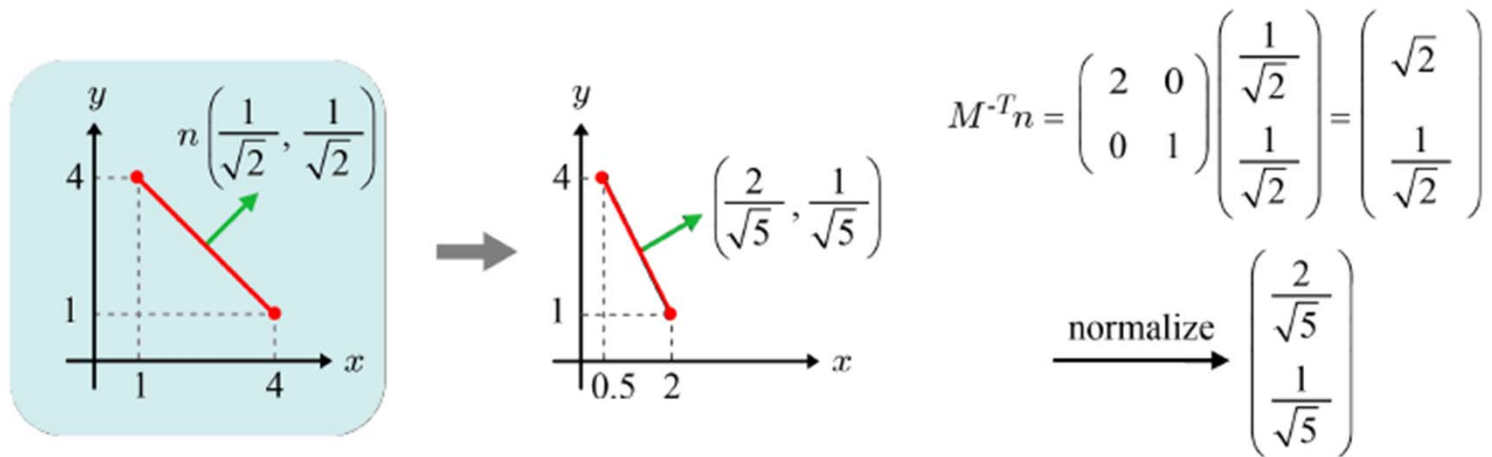


- The triangle's normal scaled by  $M$  is not any longer orthogonal to the triangle scaled by  $M$ .

## Normal Transform (cont'd)

---

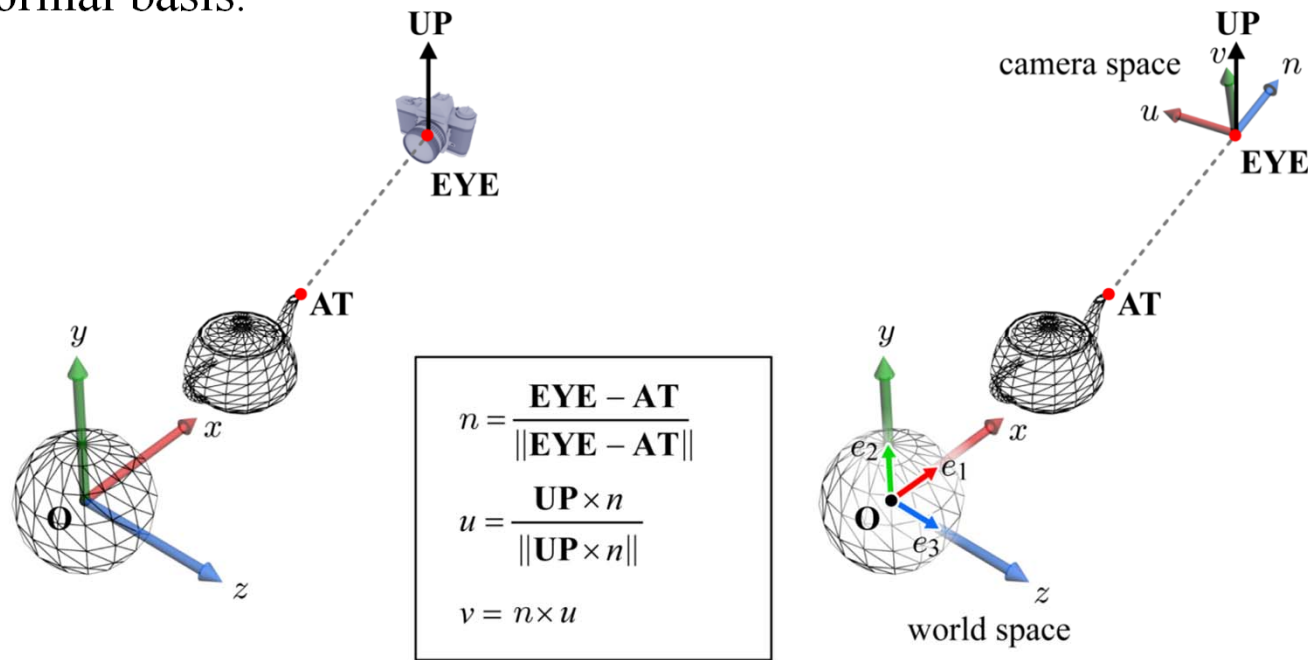
- Instead, we have to use inverse transpose of  $M$ , which is  $(M^{-1})^T$ .
- It is simply denoted by  $M^{-T}$ , which is identical to  $(M^T)^{-1}$ .



# View Transform

---

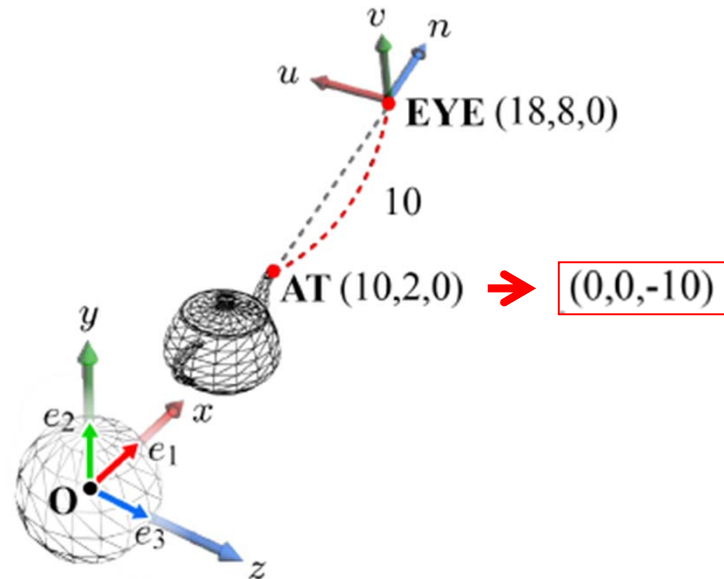
- Camera pose specification in the world space
  - **EYE**: camera position
  - **AT**: a reference point toward which the camera is aimed
  - **UP**: view up vector that describes where the top of the camera is pointing.  
(In most cases, **UP** is set to the vertical axis, y-axis, of the world space.)
- The *camera space*,  $\{u, v, n, \mathbf{EYE}\}$ , can be created. Note that  $\{u, v, n\}$  is an orthonormal basis.



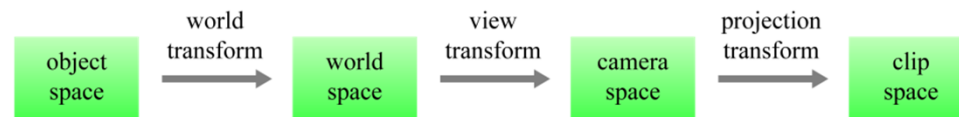
## View Transform (cont'd)

---

- A point is given different coordinates in distinct spaces.

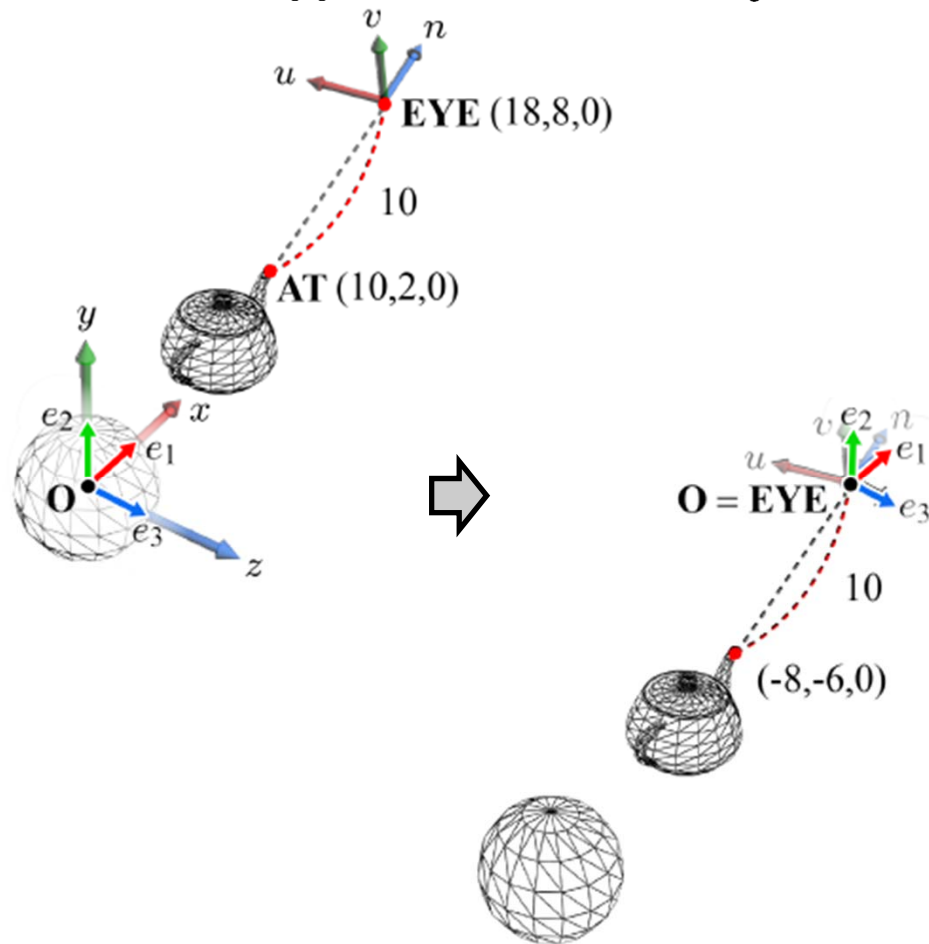


- If all the world-space objects can be newly defined in terms of the camera space in the manner of the teapot's mouth end, it becomes much easier to develop the rendering algorithms. In general, it is called *space change*.
- The space change from the world space  $\{e_1, e_2, e_3, \mathbf{O}\}$  to the camera space  $\{u, v, n, \mathbf{EYE}\}$  is the *view transform*.



## View Transform (cont'd)

- First of all, **EYE** is translated to the origin of the world space. Imagine invisible rods connecting the scene objects and the camera space. The translation is applied to the scene objects.



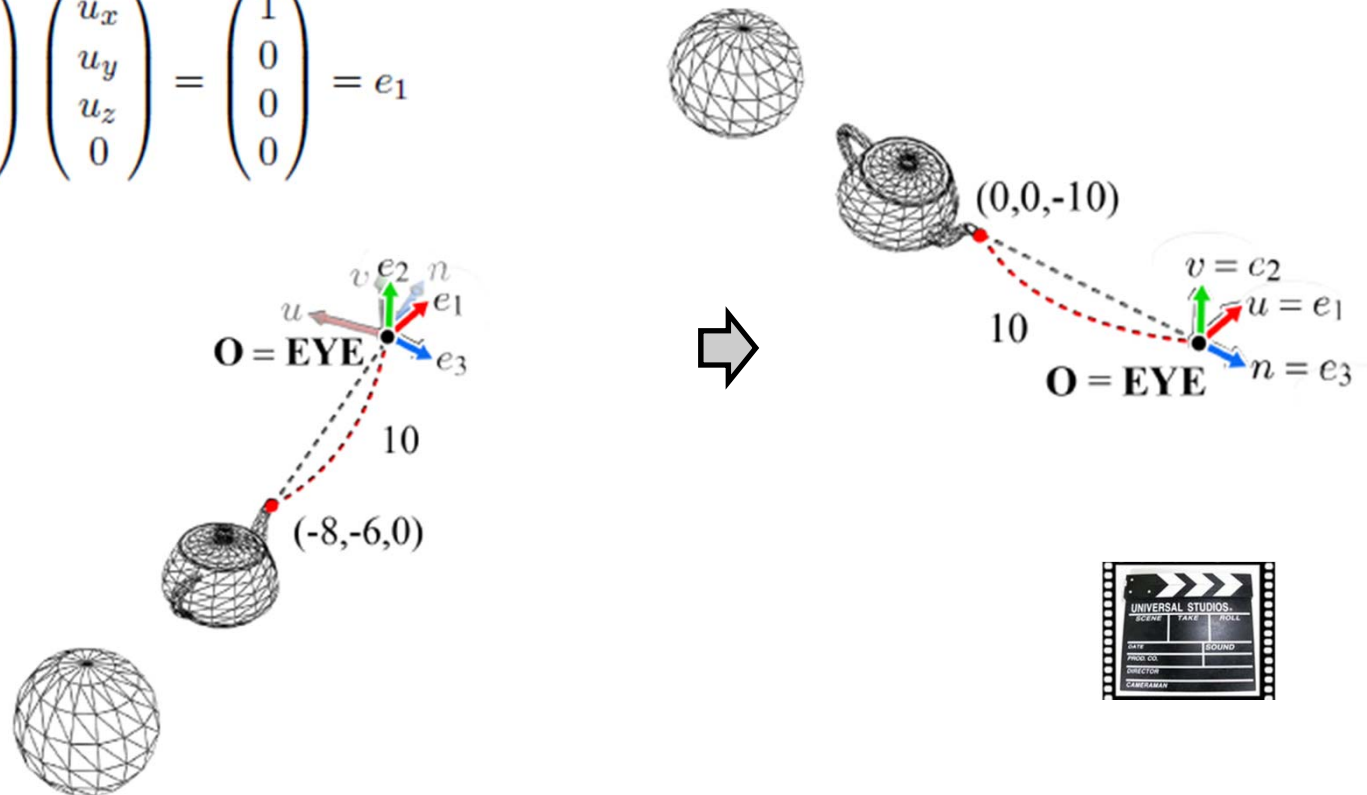
$$T_{view} = \begin{pmatrix} 1 & 0 & 0 & -\mathbf{EYE}_x \\ 0 & 1 & 0 & -\mathbf{EYE}_y \\ 0 & 0 & 1 & -\mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_{view} \begin{pmatrix} 10 \\ 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -18 \\ 0 & 1 & 0 & -8 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 10 \\ 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -8 \\ -6 \\ 0 \\ 1 \end{pmatrix}$$

## View Transform (cont'd)

- The world space and the camera space now share the origin, due to translation.
- We then need a rotation  $R_{view}$  that transforms  $\{u, v, n\}$  into  $\{e_1, e_2, e_3\}$ .

$$R_{view} u = \begin{pmatrix} \boxed{u_x} & \boxed{u_y} & \boxed{u_z} & 0 \\ \boxed{v_x} & \boxed{v_y} & \boxed{v_z} & 0 \\ \boxed{n_x} & \boxed{n_y} & \boxed{n_z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_x \\ u_y \\ u_z \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = e_1$$



- As the world and camera spaces are made identical, the world-space positions of the transformed objects can be taken as the camera-space ones.



## *View Transform (cont'd)*

---

- The view matrix

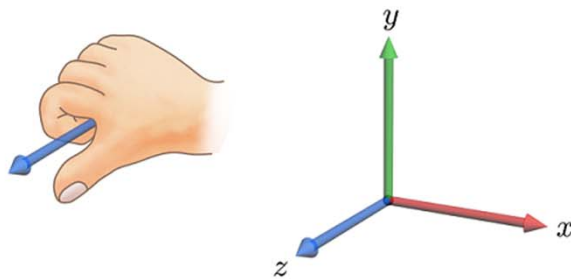
$$\begin{aligned} M_{view} &= R_{view} T_{view} \\ &= \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\mathbf{EYE}_x \\ 0 & 1 & 0 & -\mathbf{EYE}_y \\ 0 & 0 & 1 & -\mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} u_x & u_y & u_z & -\mathbf{EYE} \cdot u \\ v_x & v_y & v_z & -\mathbf{EYE} \cdot v \\ n_x & n_y & n_z & -\mathbf{EYE} \cdot n \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

# *Coordinate System's Handedness - Revisited*

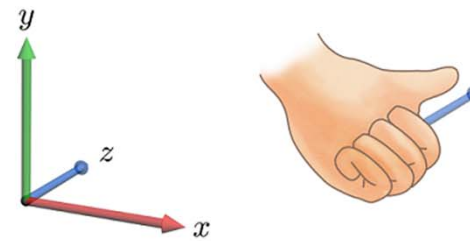
---

- Right-hand system vs. left-hand system

right-hand system (RHS)

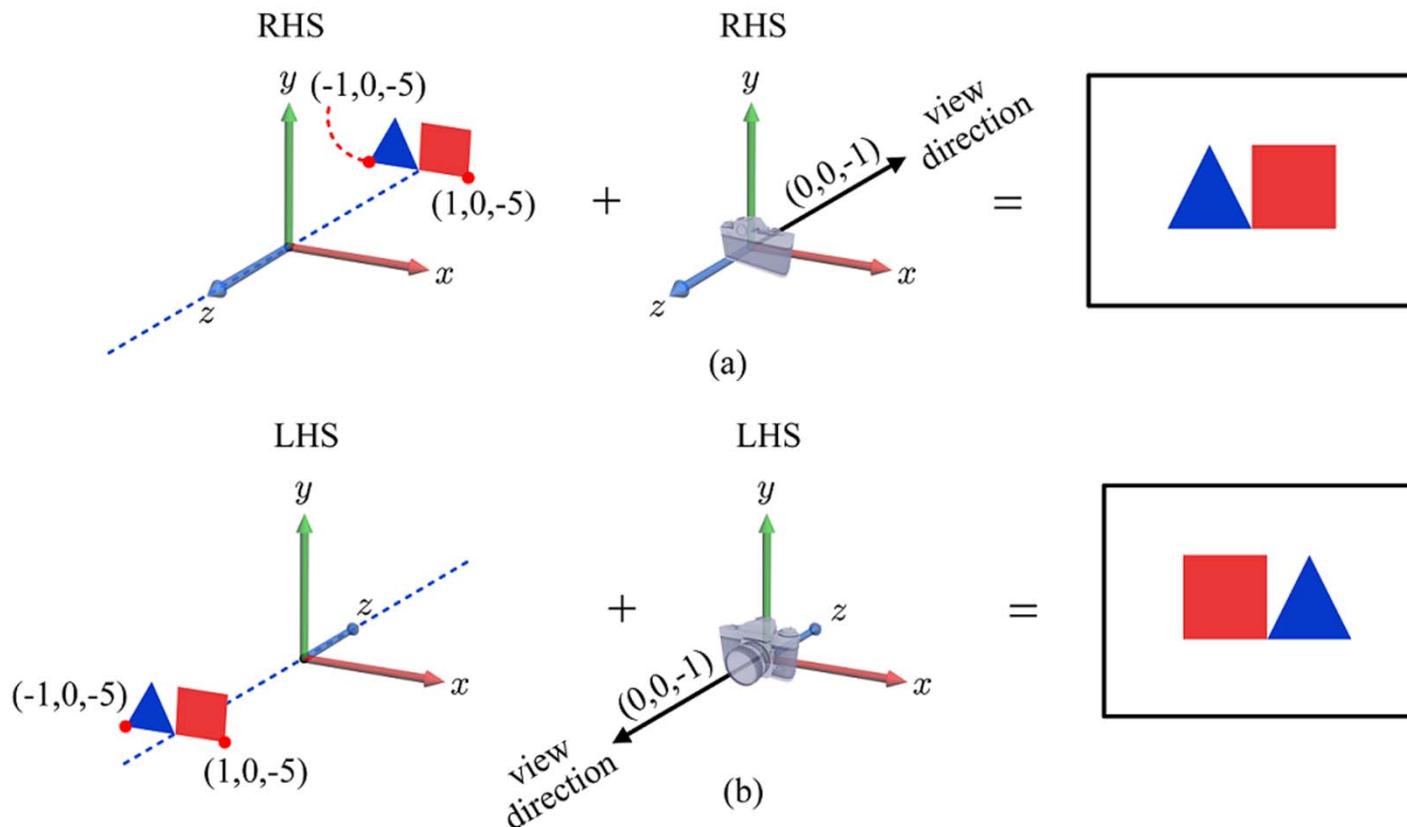


left-hand system (LHS)



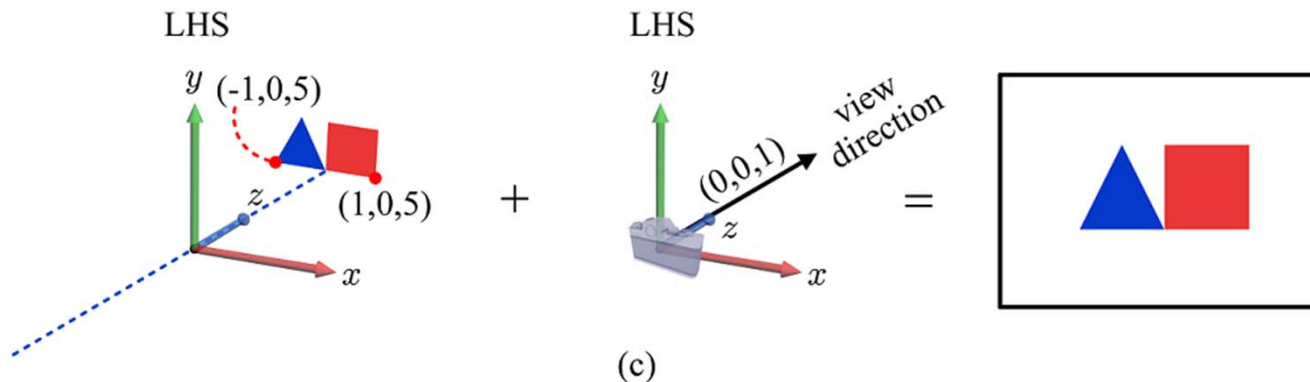
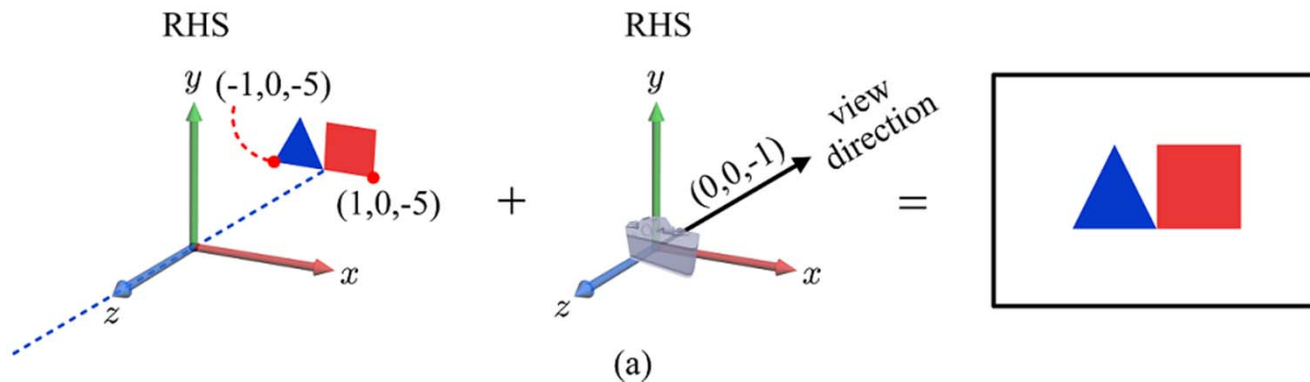
# Conversion between RHS and LHS

- Porting an application between RHS and LHS requires the order of triangle vertices to be changed:  $\langle p_1, p_2, p_3 \rangle \rightarrow \langle p_1, p_3, p_2 \rangle$ . Will it be all?



## Conversion between RHS and LHS (cont'd)

- In order to avoid the reflected image shown in the previous page, the  $z$ -coordinates of both the objects and view parameters should be negated.

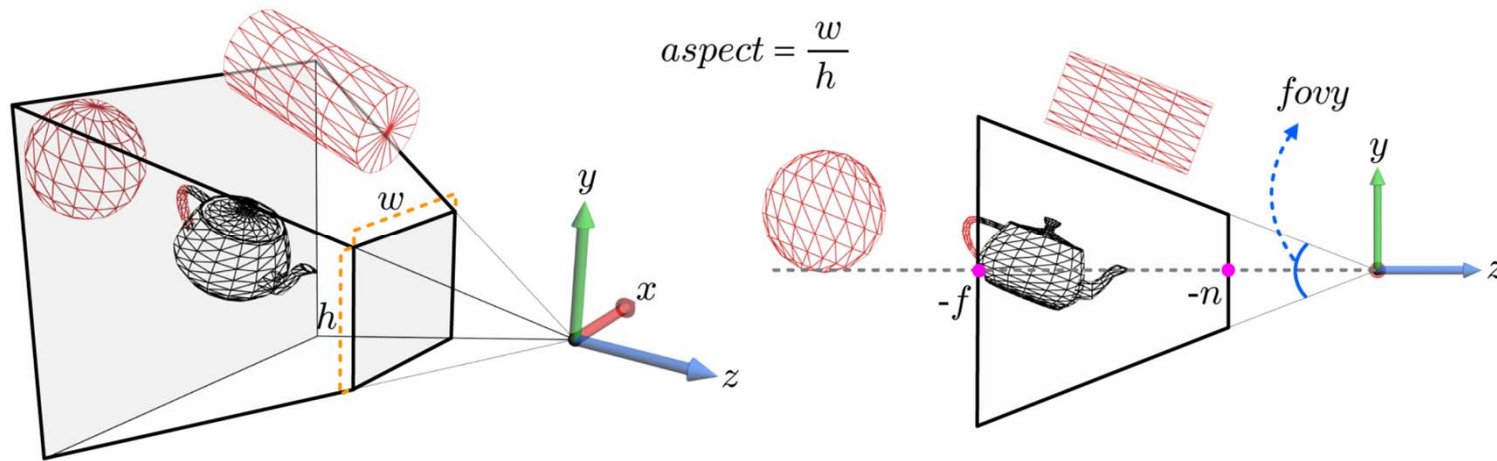


- Note that  $z$ -negation is equivalent to the  $z$ -axis inversion.

## View Frustum

---



- Let us denote the basis of the camera space by  $\{x, y, z\}$  instead of  $\{u, v, n\}$ .
- Recall that, for constructing the view transform, we defined the external parameters of the camera, i.e., **EYE**, **AT**, and **UP**. Now let us control the camera's internals. It is analogous to choosing a lens for the camera and controlling zoom-in/zoom-out.
- The *view frustum parameters*, *fovy*, *aspect*, *n*, and *f*, define a truncated pyramid.

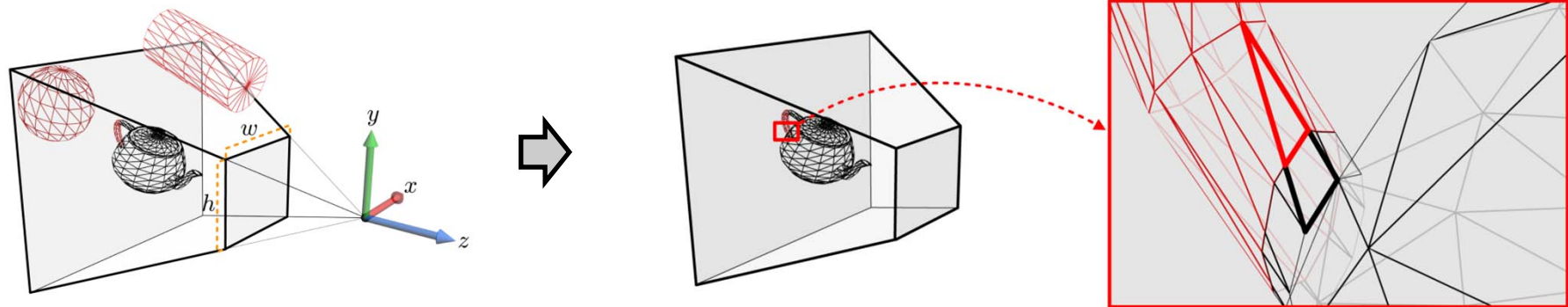


- The near and far planes run counter to the real-world camera or human vision system, but have been introduced for the sake of computational efficiency.

## View Frustum (cont'd)

---

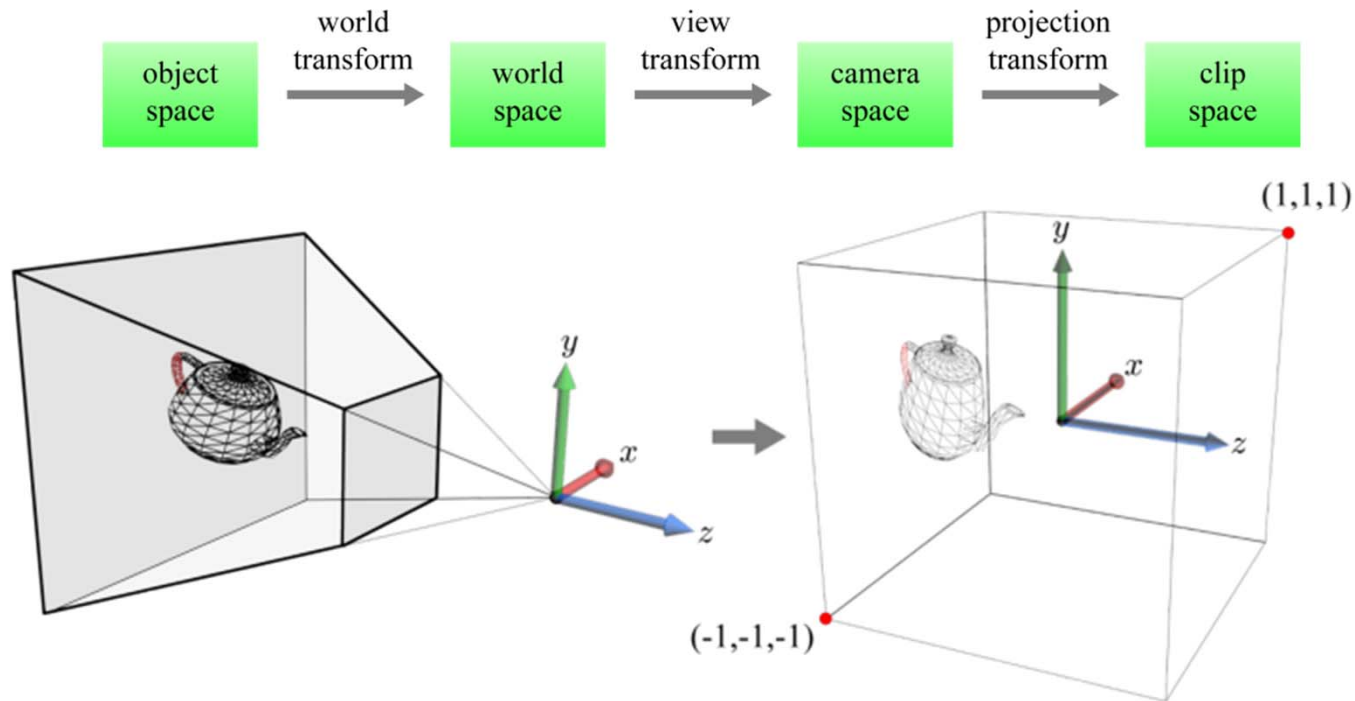
- The cylinder and sphere would be discarded by so-called view-frustum culling whereas the teapot would survive. 
- If a polygon intersects the boundary of the view frustum, it is *clipped* with respect to the boundary, and only the portion inside the view frustum is processed for display. 



# Projection Transform

---

- The pyramidal view frustum is not used for clipping. Instead, a transform is determined that deforms the view frustum into the axis-aligned 2x2x2-sized cube centered at the origin. It is called *projection transform*. The camera-space objects are projection-transformed and then clipped against the cube.

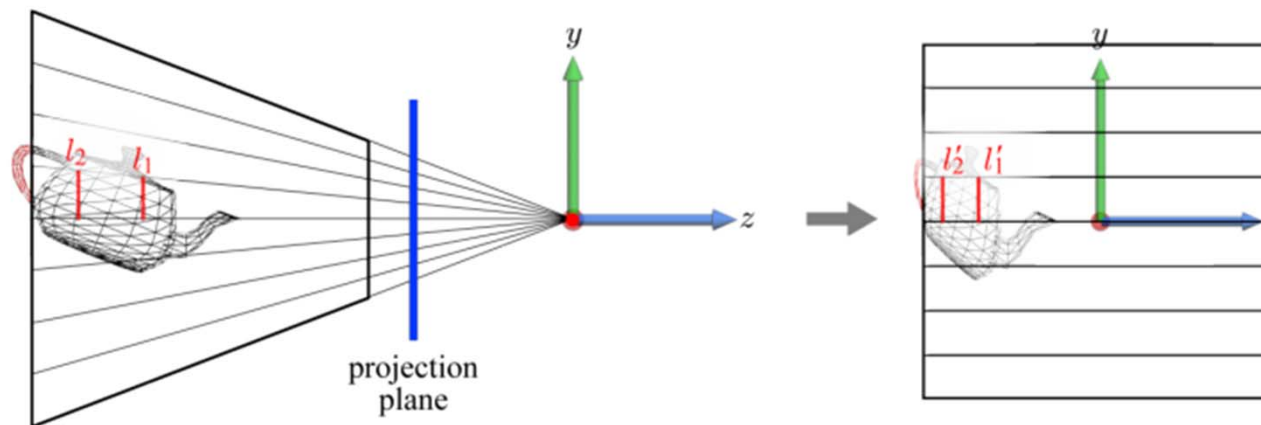


- The projection-transformed objects are said to be in the *clip space*, which is just the renamed camera space.

## Projection Transform (cont'd)

---

- The view frustum can be taken as a convergent pencil of projection lines. The lines converge on the origin, where the camera (**EYE**) is located.
- All 3D points on a projection line are mapped onto a single 2D point in the projected image. It brings the effect of *perspective projection*, where objects farther away look smaller.



- The projection transform ensures that the projection lines become parallel to the  $z$ -axis. Now viewing is done along the universal projection line. The projection transform brings the effect of perspective projection “within a 3D space.”



## *Projection Transform (cont'd)*

---

- Projection transform matrix

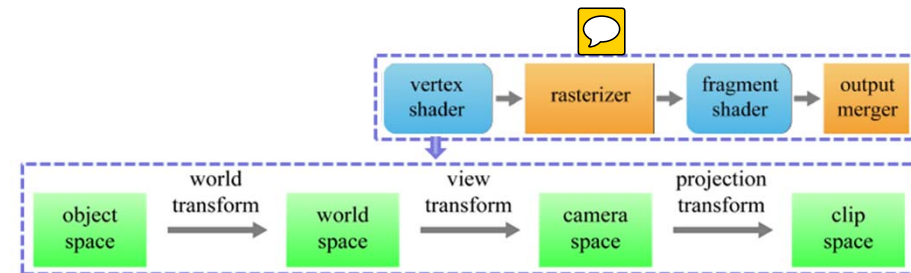
$$\begin{pmatrix} \frac{\cot(\frac{fovy}{2})}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- Derivation of the matrix is presented in the textbook.

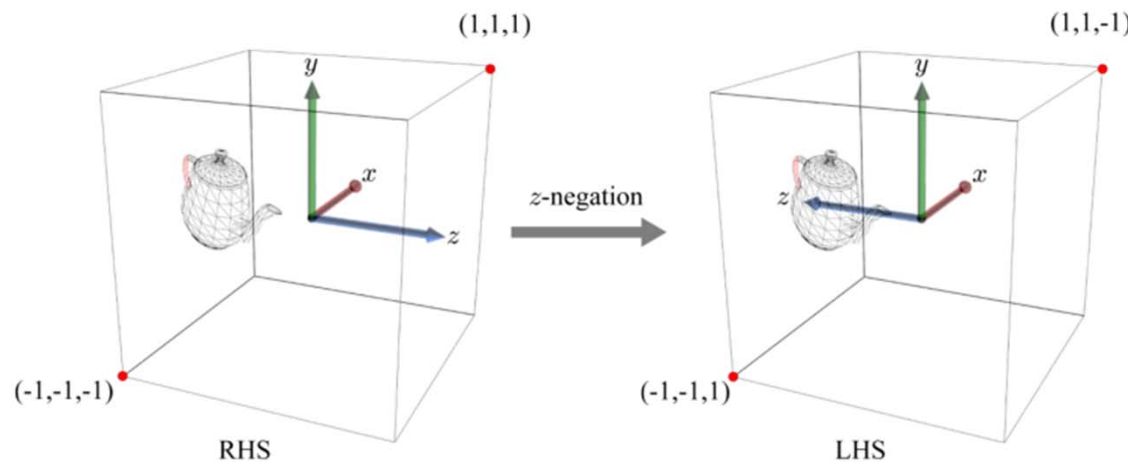
# Projection Transform (cont'd)

- Projection transform matrix

$$\begin{pmatrix} \frac{\cot(\frac{fovy}{2})}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



- The projection-transformed objects will enter the rasterizer.
- Unlike the vertex shader, the rasterizer is implemented in hardware, and assumes that the clip space is left-handed. In order for the vertex shader to be compatible with the hard-wired rasterizer, the objects should be  $z$ -negated.

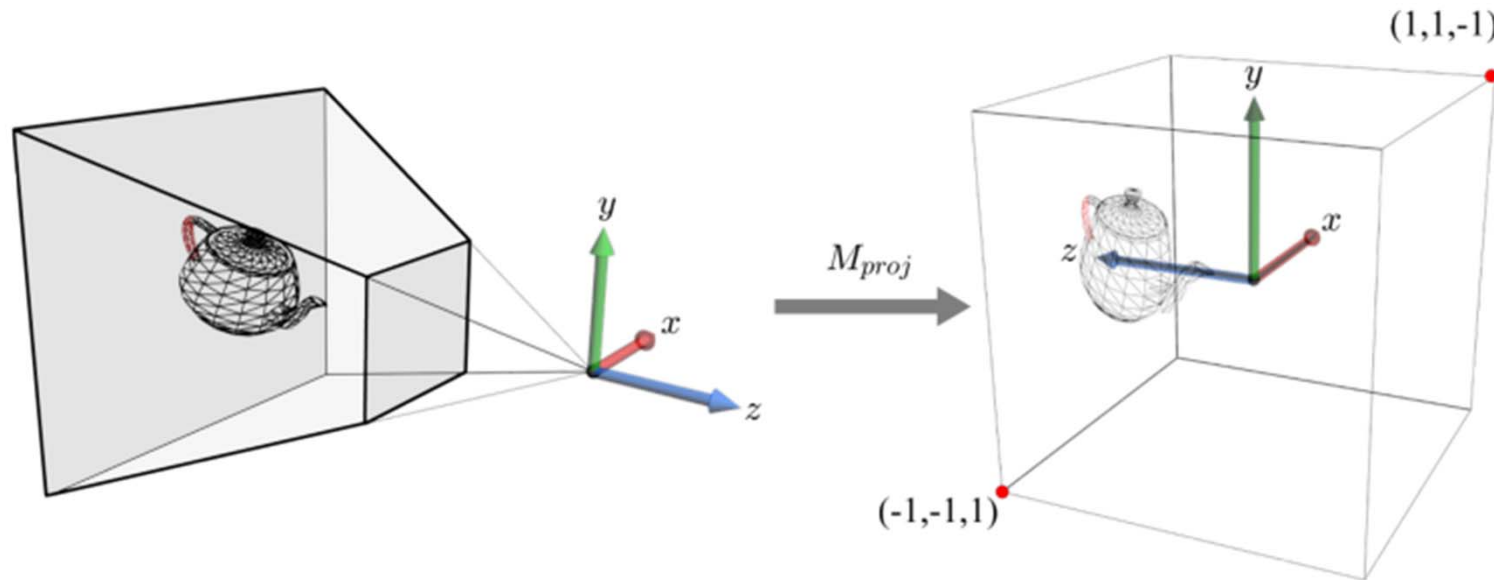


$$\begin{pmatrix} \frac{\cot(\frac{fovy}{2})}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

## *Projection Transform (cont'd)*

---

- Projection transform from the camera space (RHS) to the clip space (LHS)



- We do not have to do vertex order change. It is because the vertex order problem can be resolved by configuring the hard-wired rasterizer. This will be discussed later.