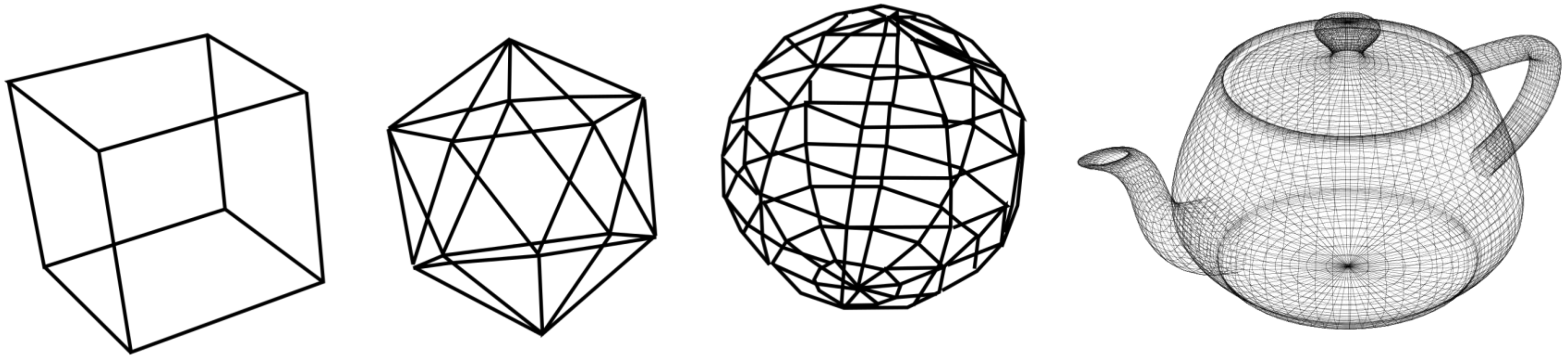
VIETNAM ACADEMY OF SCIENCE AND TECHNOLOGY
UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

COMPUTER GRAPHICS

Lecture 4: Rendering pipeline – Vertex processing

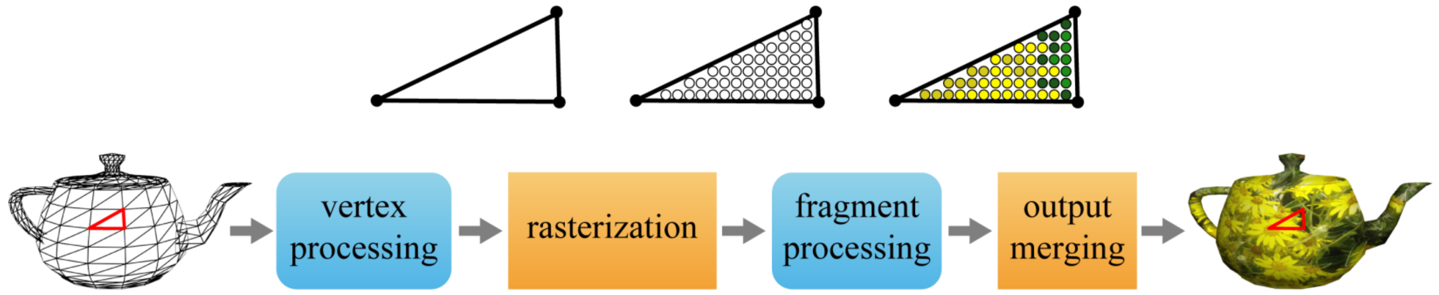
Lecturer: Dr. NGUYEN Hoang Ha



Reference: JungHyun Han. 2011. 3D Graphics for Game Programming (1st ed.), chapter 2

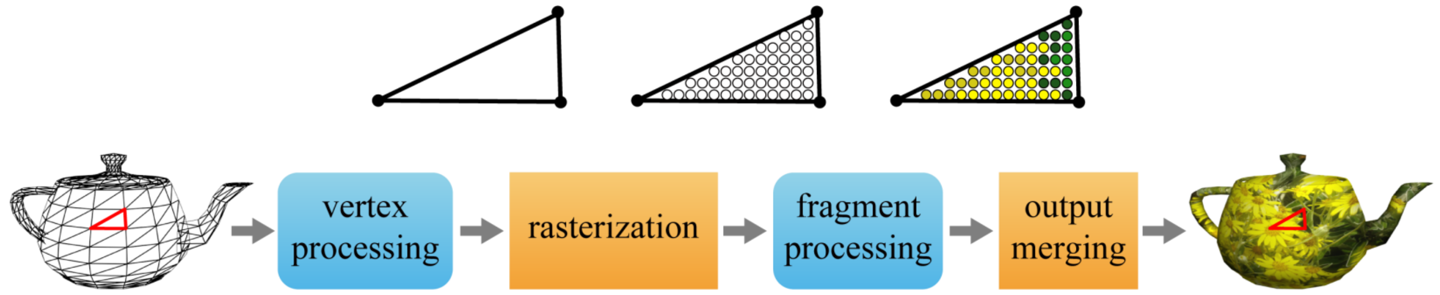
RENDERING PIPELINE OVERVIEW

Rendering Pipeline



- *Vertex processing:* operations (e.g. transformation) on every vertex.
- *Rasterization:* converts polygon into a set of *fragments* (set of data for updating a pixel in the color buffer)
- *Fragment processing:* determines color of fragments.
  - Fragment: data for updating color of a pixel.
- *Output merging:* determines pixel color

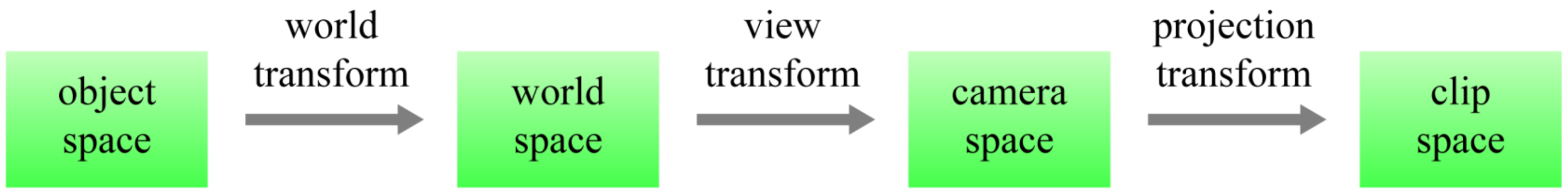
Rendering Pipeline (cont')



- Programmable: vertex and fragment processing stages. V*ertex program* and *fragment program* enable user to apply any transform to the vertex, determine the fragment color through any way you want.
- Hardwired: rasterization and output merging stages but they are configurable through user-defined parameters.

Spaces and Transforms for Vertex Processing

- Typical operations on a vertex
  - Transform



  - Lighting
  - Animation

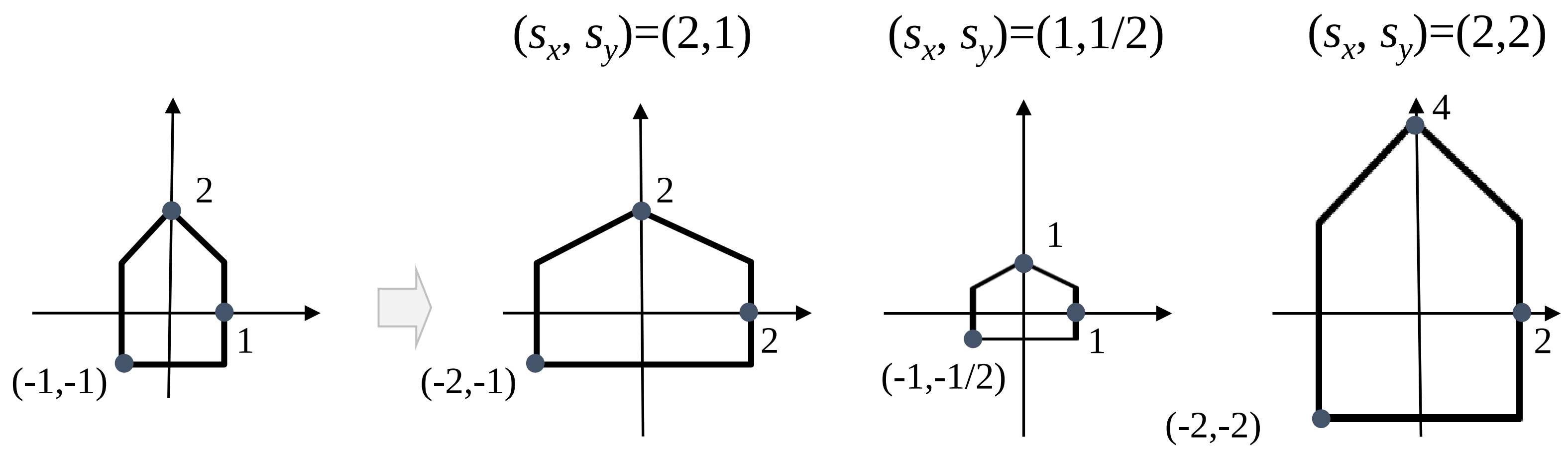
BASIC MATH FOR TRANSFORMS

Affine Transform

- The world and view transforms are built upon affine transforms: P’ = M*P + T
- Affine transform preserves:
  - Collinearity:
  - Parallelism
  - Convexity
  - Ratios of lengths of parallel line segments
  - Varycenters of weighted collections of points
- Affine transform:
  - Translation
  - Scaling
  - Rotation
  - Shear mapping

Affine Transform – Scaling

- Scaling example in 2D

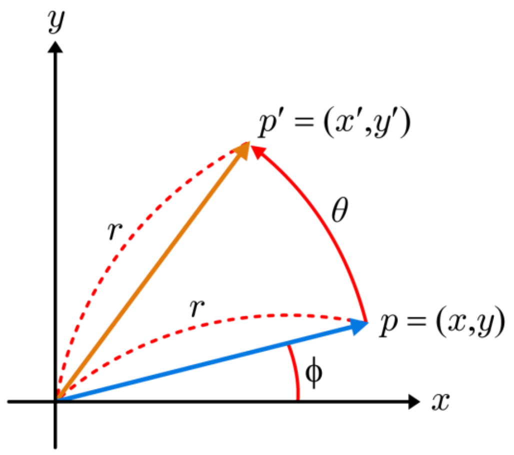


- Scaling in 3D $$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \end{pmatrix}$$

- If all of the scaling factors, sx, sy, and sz, are identical, the scaling is called **uniform**. Otherwise, it is **non-uniform**.

Affine Transform – Rotation

- 2D Rotation



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

$$\begin{aligned} x' &= r\cos(\phi + \theta) \\ &= r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ &= x\cos\theta - y\sin\theta \end{aligned}$$

$$\begin{aligned} y' &= r\sin(\phi + \theta) \\ &= r\cos\phi\sin\theta + r\sin\phi\cos\theta \\ &= x\sin\theta + y\cos\theta \end{aligned}$$

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

Affine Transform – Rotation (cont')

- 3D rotation about *z*-axis (R_z)

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

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

- 3D rotation about *x*-axis can be obtained through cyclic permutation: *x*-, *y*-, and *z*-coordinates are replaced by *y*-, *z*-, and *x*-coordinates, respectively.

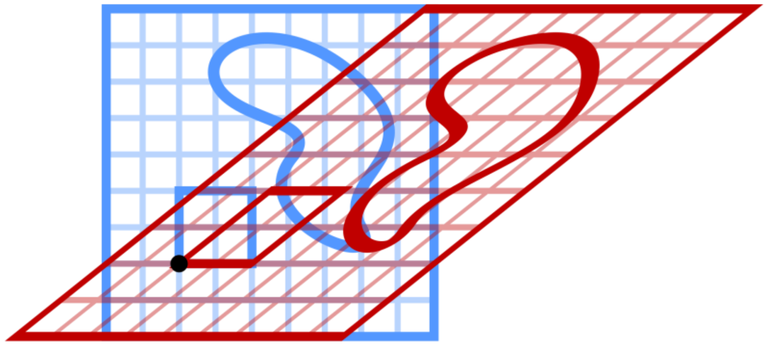
$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$$

- One more cyclic permutation leads to 3D rotation about *y*-axis.

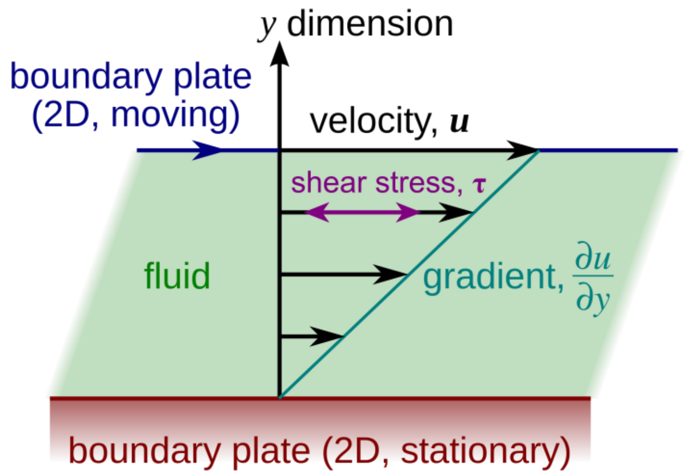
$$R_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

Affine Transform – Shear mapping

- Shear in x direction:
    - $(x,y) \rightarrow (x + s.y, y)$

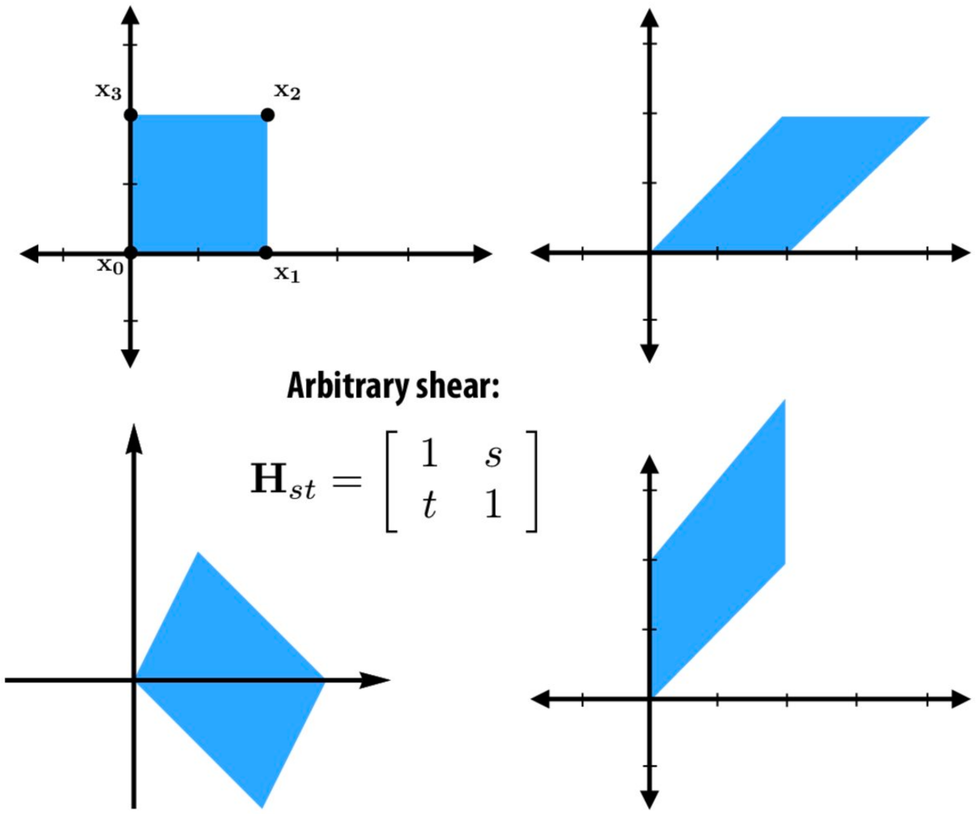


- Shear mapping in modelling fluid dynamics



y dimension
boundary plate (2D, moving)
velocity, $\boldsymbol{u}$
shear stress, τ
fluid
gradient, $\frac{\partial u}{\partial y}$
boundary plate (2D, stationary)

Affine Transform – Shear mapping



Shear in x:

$$\mathbf{H}_{xs} = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$$

Arbitrary shear:

$$\mathbf{H}_{st} = \begin{bmatrix} 1 & s \\ t & 1 \end{bmatrix}$$

Shear in y:

$$\mathbf{H}_{ys} = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

Homogeneous Coordinates

- How to express the combination of some Affine transform
  - → Tough in Cartesian coordinates
  - → Simple in Homogeneous coordinates
- Homogeneous coordinates
  - A point in a n-dimention space is express by n+1 components
  - The homogeneous coordinates (x, y, z, w) correspond to the 3D Cartesian coordinates (x/w, y/w, z/w).
  - A point presented Homogeneous coordinates by corresponds to finite coordinates. E.g: (1,2,3,1), (2,4,6,2) and (3,6,9,3) are different homogeneous coordinates for the same Cartesian coordinates (1,2,3).
  - In CG, the w-component of the homogeneous coordinates is used to distinguish between vectors and points.
    - If w is 0, (x, y, z, w) represent a vector.
    - Otherwise, a point.

Affine Transform in Homogeneous Coordinates

- Translation is not linear transforms, and is represented as *vector addition*. $\begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ z + d_z \end{pmatrix}$
- We can describe translation as *matrix multiplication* if we use the *homogeneous coordinates*.

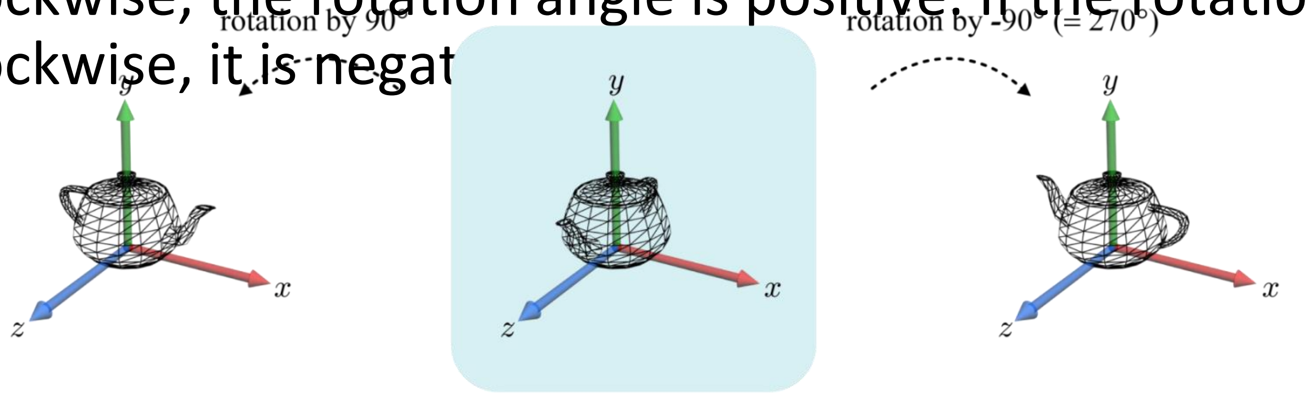
$$\begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{pmatrix}$$

- The matrices for scaling and rotation should be extended into 4x4 matrices. E.g. scaling:

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \\ 1 \end{pmatrix}$$

Rotation

- Look at the origin of the coordinate system such that the axis of rotation points toward you. If the rotation is counter-clockwise, the rotation angle is positive. If the rotation is clockwise, it is negat



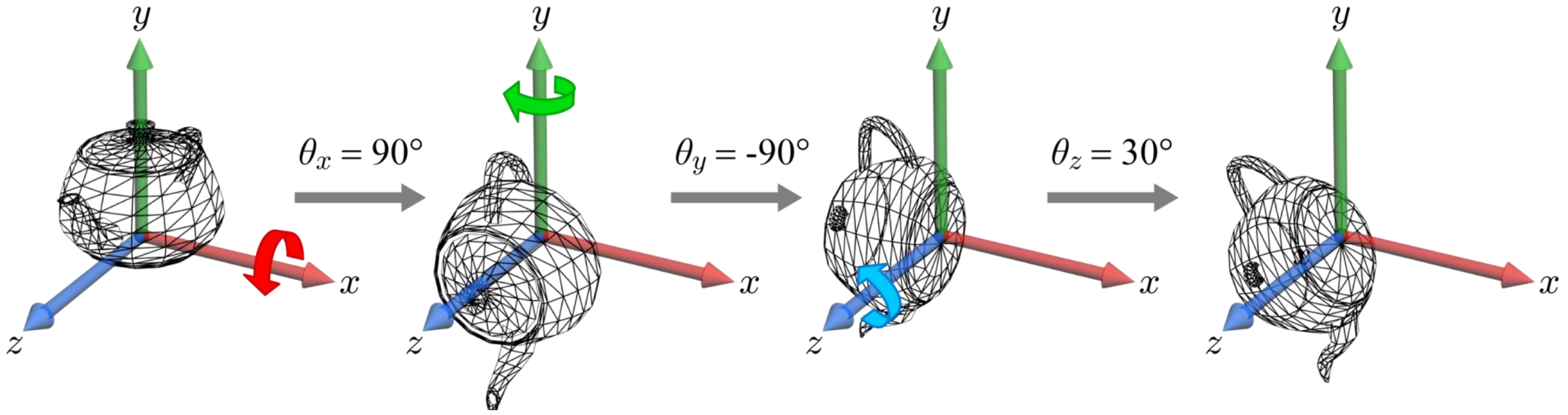
$$R_y(-90^\circ) = \begin{pmatrix} cos(-90^\circ) & 0 & sin(-90^\circ) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(-90^\circ) & 0 & cos(-90^\circ) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} cos\ 270^\circ & 0 & sin\ 270^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -sin\ 270^\circ & 0 & cos\ 270^\circ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Euler Transform

- When we successively rotate an object about the x-, y-, and z-axes, the object acquires a specific orientation.
- The rotations angles (θx,θy,θz) are called the Euler angles. When three rotations are combined into one, it is called Euler transform.



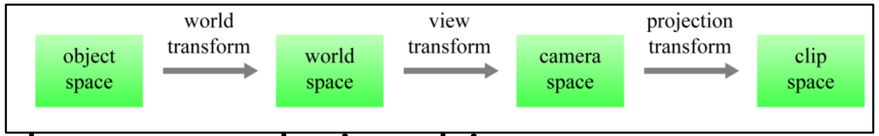
$$R_z(30°)R_y(-90°)R_x(90°) = \begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & -\frac{\sqrt{3}}{2} & \frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & -\frac{\sqrt{3}}{2} & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

WORD TRANSFORM

To assemble models together

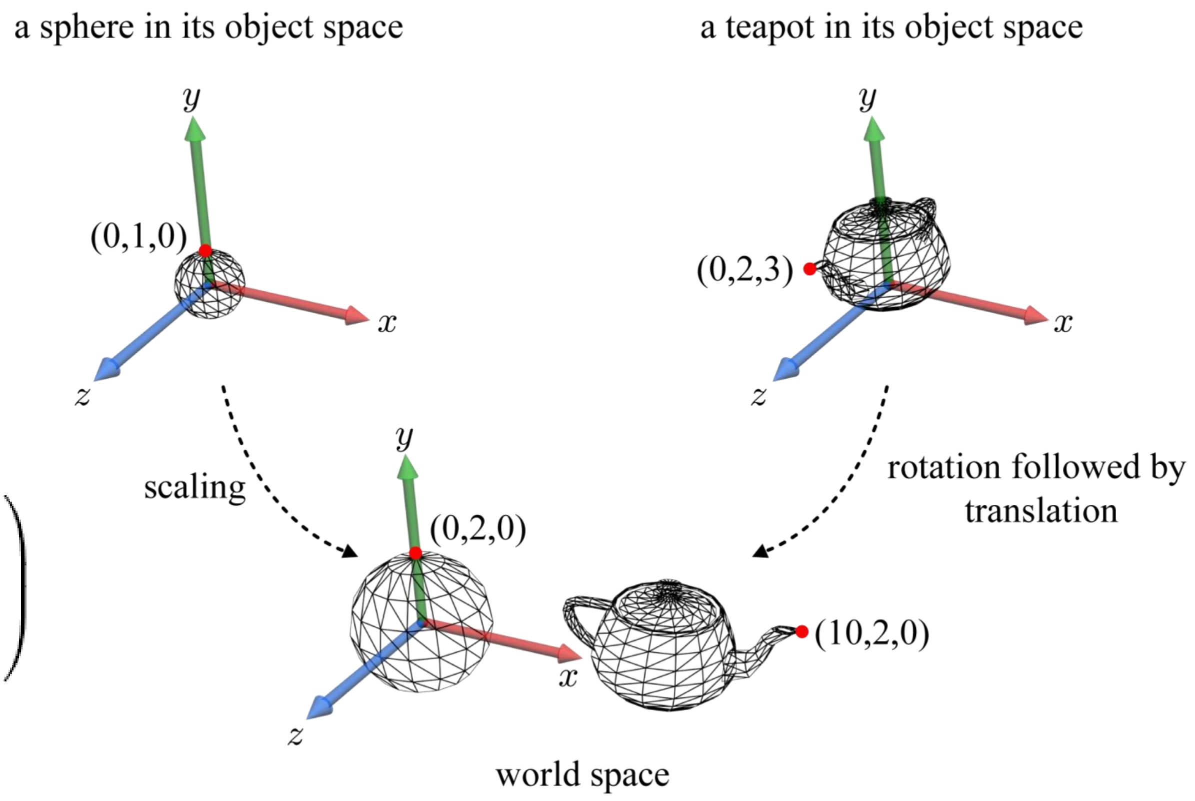
World Transform



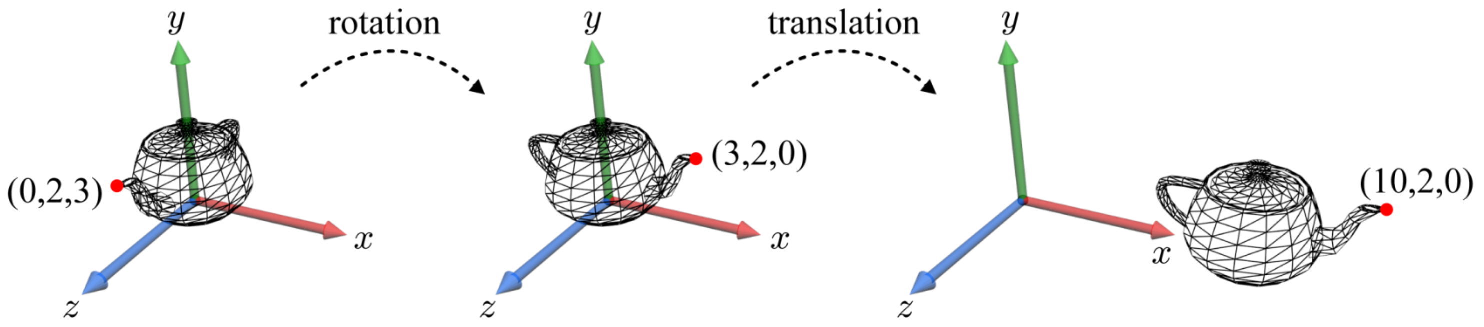
- Objects originally have no relationship
- The *world transform* ‘assembles’ all models into a single coordinate system called *world space*.

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \\ 0 \\ 1 \end{pmatrix}$$


a sphere in its object space
a teapot in its object space
y
x
z
(0,1,0)
(0,2,3)
scaling
rotation followed by translation
(0,2,0)
(10,2,0)
world space

World Transform Example



$$\boxed{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta_x & -sin\theta_x & 0 \\ 0 & sin\theta_x & cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} cos\theta_y & 0 & sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta_y & 0 & cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} cos\theta_z & -sin\theta_z & 0 & 0 \\ sin\theta_z & cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}$$

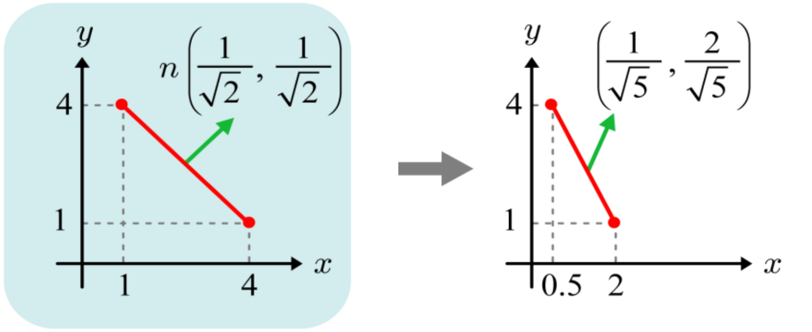
$$R = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 0 \\ 1 \end{pmatrix}$$

$$\boxed{\begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}} \quad T = \begin{pmatrix} 1 & 0 & 0 & 7 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 & 7 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 10 \\ 2 \\ 0 \\ 1 \end{pmatrix}$$

$$TR = \begin{pmatrix} 1 & 0 & 0 & 7 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 7 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 1 & 7 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 10 \\ 2 \\ 0 \\ 1 \end{pmatrix}$$

Normal Transform

- Transformed normal is not orthogonal to the transformed triangle



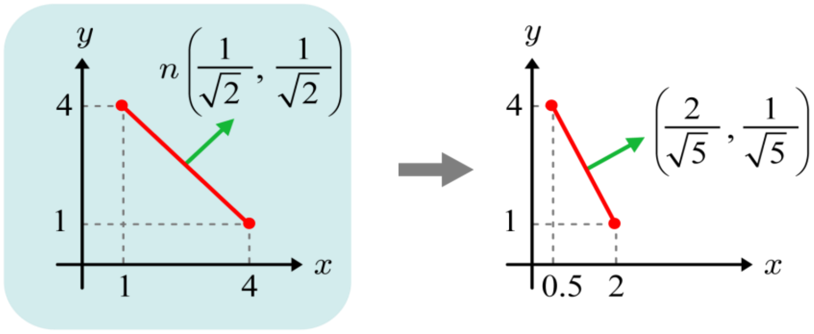
y
x
4
1
1
4
$n\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$
0.5
2
$\left(\frac{1}{\sqrt{5}}, \frac{2}{\sqrt{5}}\right)$

$$Mn = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \frac{1}{2\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

$$\xrightarrow{\text{normalize}} \begin{pmatrix} \frac{1}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} \end{pmatrix}$$

(a)

- Transforms with $(M^{-1})^T$, the normal remains orthogonal to the triangle.



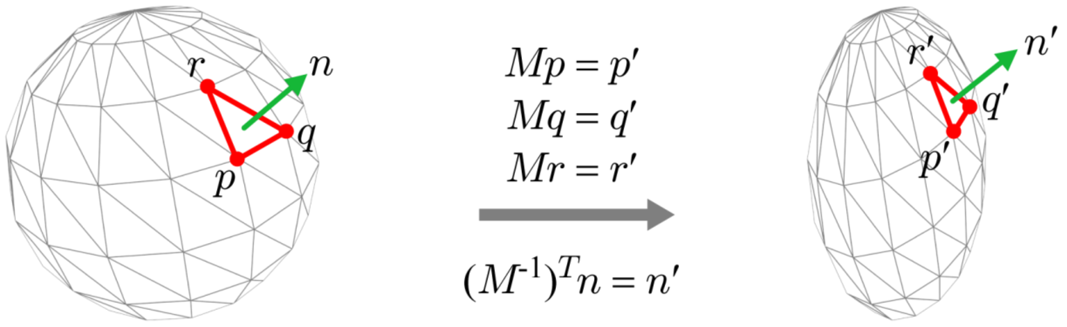
y
x
4
1
1
4
$n\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$
0.5
2
$\left(\frac{2}{\sqrt{5}}, \frac{1}{\sqrt{5}}\right)$

$$(M^{-1})^T n = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \sqrt{2} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

$$\xrightarrow{\text{normalize}} \begin{pmatrix} \frac{2}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} \end{pmatrix}$$

(b)

Normal Transform (cont')



$$n^T(q - p) = 0$$

$$Mp = p'$$

$$Mq = q'$$

$$n^T(M^{-1}q' - M^{-1}p') = 0$$

$$n^T M^{-1}(q' - p') = 0$$

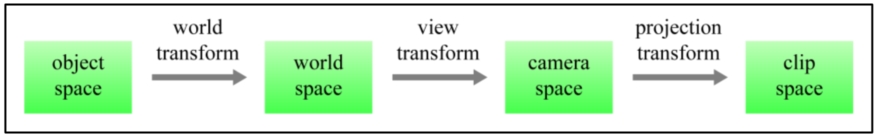
$$(q' - p')^T (M^{-1})^T n = 0$$

$$(r' - p')^T (M^{-1})^T n = 0$$

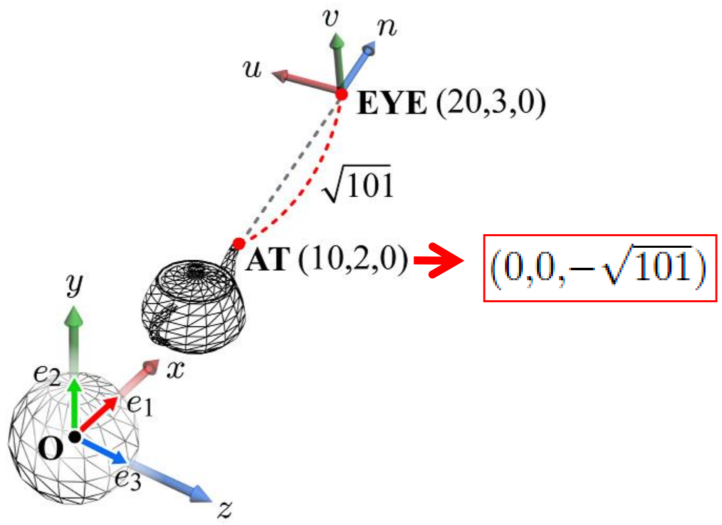
VIEW TRANSFORM

To convert points from the world space to the camera space

View Transform



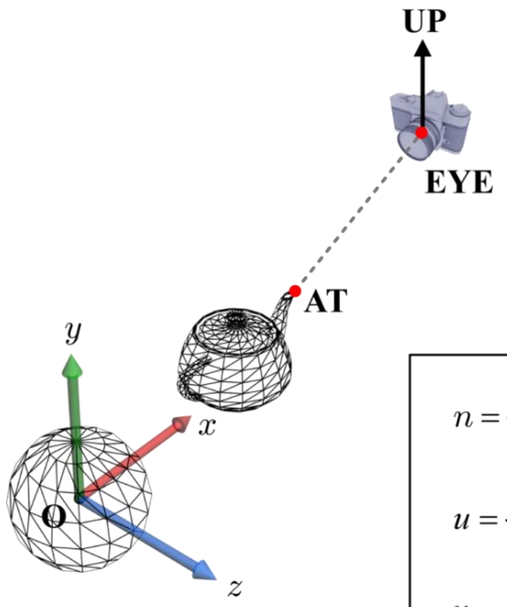
- A point is given different coordinates in distinct spaces.



- If all the world-space objects can be newly defined in terms of the camera space, it becomes much easier to develop the rendering algorithms.
- The view transform converts each vertex from the world space to the camera space.

View Transform (cont')

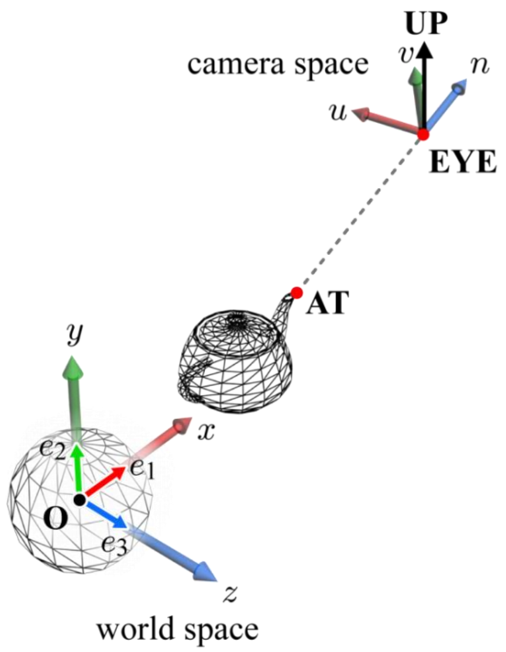
- Goal: camera pose specification
  - EYE: camera position
  - AT: a reference point toward which the camera is aimed
  - UP: view up vector that roughly describes where the top of the camera is pointing. (In most cases, UP is set to the y-axis of the world space.)
- Then, the camera space, {EYE, u, v, n}, can be created.



$$n = \frac{\mathbf{EYE} - \mathbf{AT}}{\|\mathbf{EYE} - \mathbf{AT}\|}$$

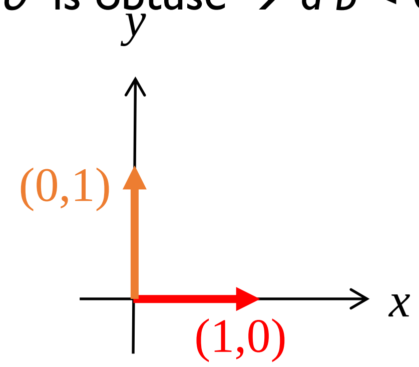
$$u = \frac{\mathbf{UP} \times n}{\|\mathbf{UP} \times n\|}$$

$$v = n \times u$$

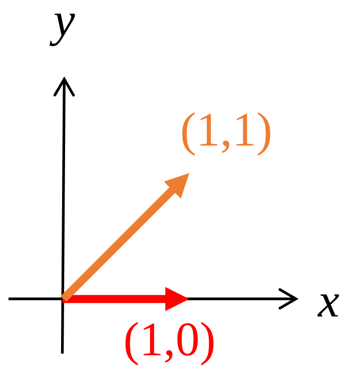


Dot Product

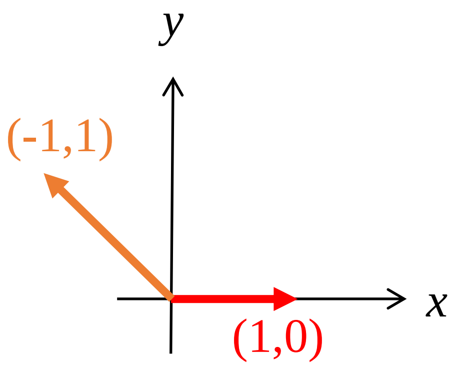
- Given vectors, *a* and *b*, whose coordinates are $(a_1, a_2, .. , a_n)$ and $(b_1, b_2, .. , b_n)$, respectively, the dot product $a \cdot b$ is defined to be $a_1b_1+a_2b_2+ .. +a_nb_n$.
- $a \cdot b = \|a\|\|b\|\cos\theta$, where $\|a\|$ and $\|b\|$ denote the lengths of *a* and *b*, respectively, and ϑ is the angle between *a* and *b*.
  - *a* and *b* are perpendicular → $a \cdot b = 0$.
  - θ is acute angle → $a \cdot b > 0$.
  - θ is obtuse → $a \cdot b < 0$.



$(1,0)\cdot(0,1) = 0$



$(1,0)\cdot(1,1) = 1$

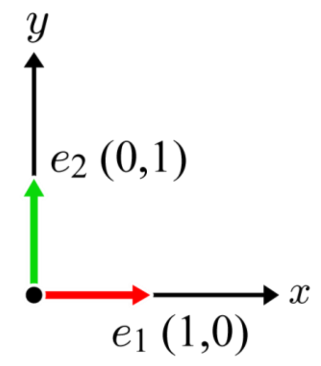


$(1,0)\cdot(-1,1) = -1$

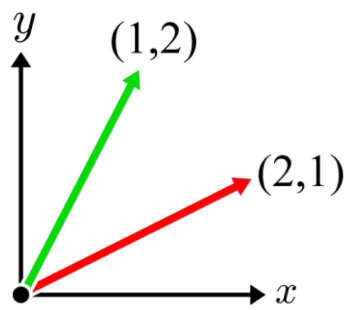
- If *a* is a unit vector, $a \cdot a = 1$.

Orthonormal Basis

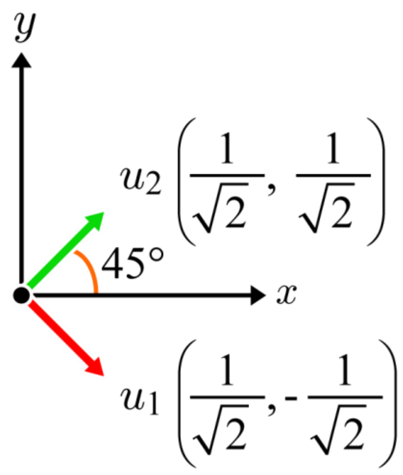
- Orthonormal basis = an orthogonal set of unit vectors



orthonormal
standard



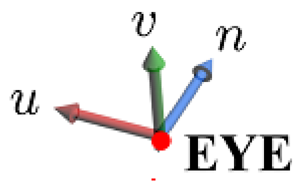
non-orthonormal
non-standard



y
$u_2\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$
45°
x
$u_1\left(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right)$

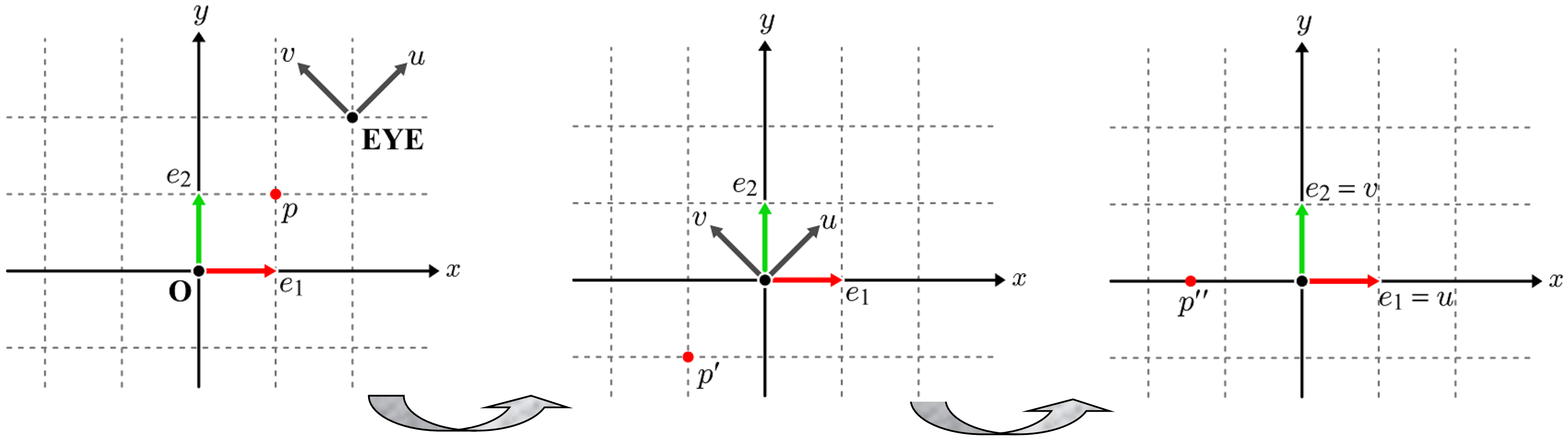
orthonormal
non-standard

- The camera space has an orthonormal basis {*u*, *v*, *n*}.
- Note that *u*·*u*=*v*·*v*=*n*·*n*=1 and *u*·*v*=*v*·*n*=*n*·*u*=0.



2D Analogy for View Transform

- The coordinates of *p* are (1,1) in the world space but (-√2,0) in the camera space.
- Let's superimpose the camera space to the world space while imagining invisible rods connecting *p* and the camera space such that the transform is applied to *p*.



translation by (-2,-2)

$$T = \begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}$$

rotation by -45º

$$R = \begin{pmatrix} cos(\text{-}45°) & \text{-}sin(\text{-}45°) & 0 \\ sin(\text{-}45°) & cos(\text{-}45°) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- As the camera space becomes identical to the world space, the world-space coordinates of *p''* can be taken as the camera-space coordinates.

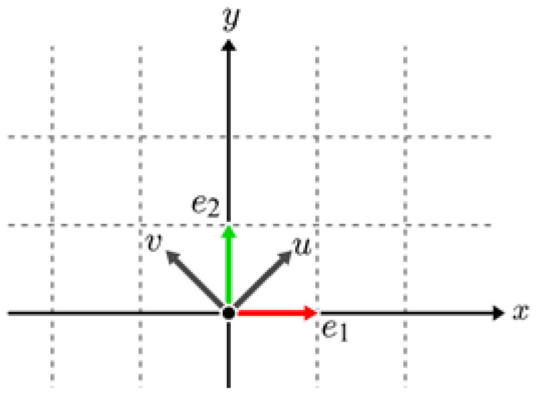
2D Analogy for View Transform (cont')

- Let's see if the combination of *T* and *R* correctly transforms *p*.

$$RT = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & -2\sqrt{2} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- How to compute *R*? It is obtained using the camera-space basis vectors.

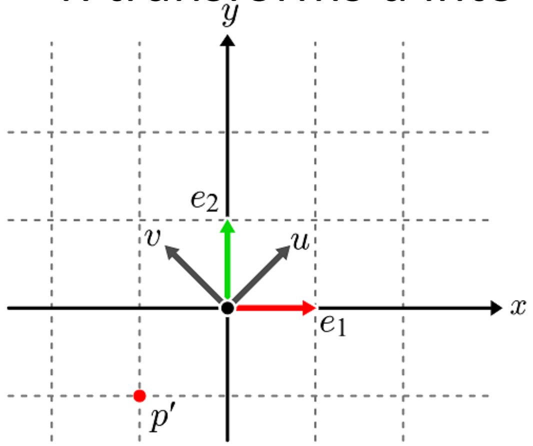
$$p'' = RTp = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & -2\sqrt{2} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -\sqrt{2} \\ 0 \\ 1 \end{pmatrix}$$

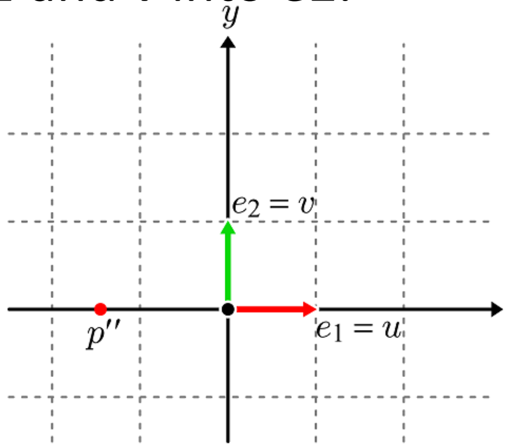


$$R = \begin{array}{c} u \\ v \\ \\ \end{array} \begin{pmatrix} \boxed{\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}}} & 0 \\ \boxed{-\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}}} & 0 \\ 0 \quad\quad 0 & 1 \end{pmatrix}$$

2D Analogy for View Transform (cont')

- R transforms u into e1 and v into e2.





$$\begin{array}{c}u\\v\\ \end{array}\begin{pmatrix}\boxed{\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}} \quad 0}\\ \boxed{-\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}} \quad 0}\\ 0 \quad\quad 0 \quad\quad 1\end{pmatrix}\overset{u}{\boxed{\begin{pmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\\ 0\end{pmatrix}}}=\overset{e_1}{\boxed{\begin{pmatrix}1\\0\\0\end{pmatrix}}}$$

$$\begin{array}{c}u\\v\\ \end{array}\begin{pmatrix}\boxed{\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}} \quad 0}\\ \boxed{-\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}} \quad 0}\\ 0 \quad\quad 0 \quad\quad 1\end{pmatrix}\overset{v}{\boxed{\begin{pmatrix}-\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\\ 0\end{pmatrix}}}=\overset{e_2}{\boxed{\begin{pmatrix}0\\1\\0\end{pmatrix}}}$$

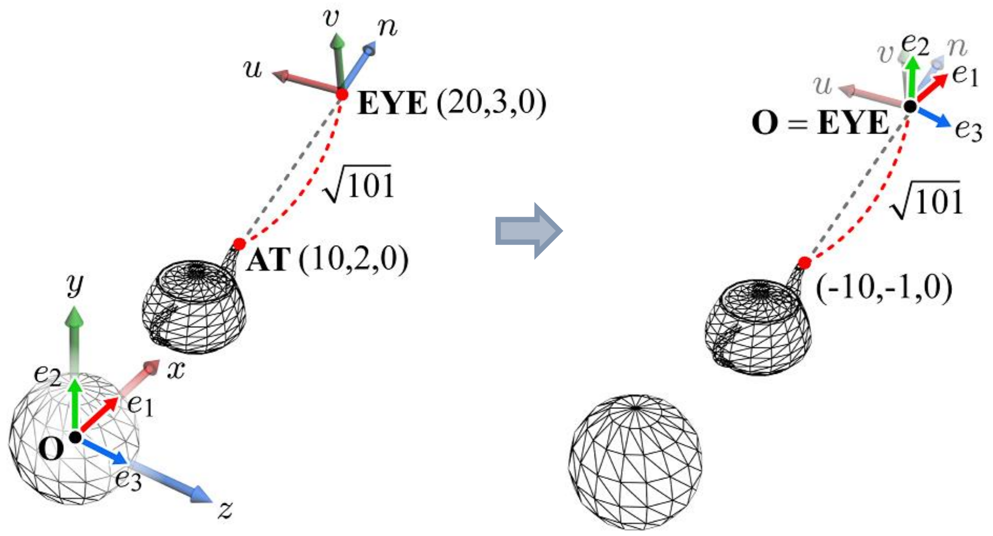
- R converts the coordinates defined in terms of the basis {e1, e2}, e.g., (-1,-1), into those defined in terms of the basis {u, v}, e.g., (-√2,0). In other words, R performs the basis change from {e1, e2} to {u, v}.

$$\begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0\\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}-1\\-1\\0\end{pmatrix}=\begin{pmatrix}-\sqrt{2}\\0\\0\end{pmatrix}$$

- The problem of space change is decomposed into translation and basis change.

View Transform (cont')

- Let us do the same thing in 3D. 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.



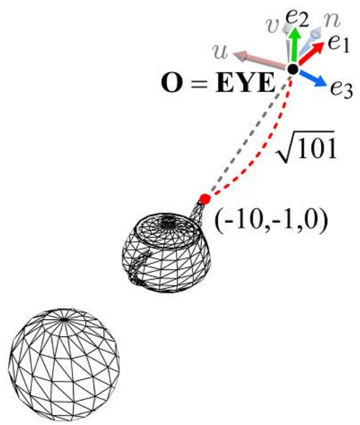
$$\begin{pmatrix} 1 & 0 & 0 & -\mathbf{EYE}_x \\ 0 & 1 & 0 & -\mathbf{EYE}_y \\ 0 & 0 & 1 & -\mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

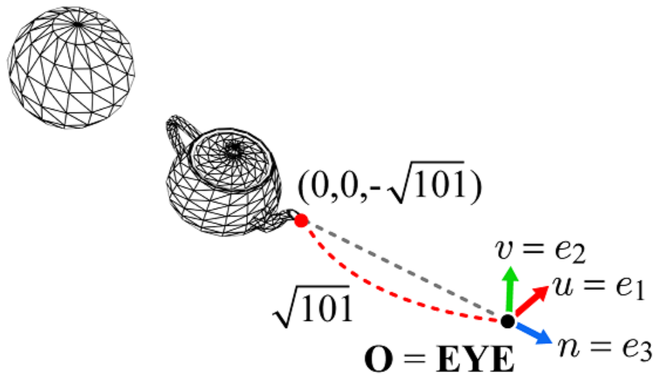
$$\begin{pmatrix} 1 & 0 & 0 & -20 \\ 0 & 1 & 0 & -3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 10 \\ 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -10 \\ -1 \\ 0 \\ 1 \end{pmatrix}$$

View Transform (cont')

- The world space and the camera space now share the origin, due to translation.
- We then need a rotation that transforms *u*, *v*, and *n* into e_1, e_2, and e_3, respectively, i.e., $Ru=e_1$, $Rv=e_2$, and $Rn=e_3$. *R* performs the *basis change* from $\{e_1, e_2, e_3\}$ to $\{u, v, n\}$.

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





View Transform (cont')

- The view matrix

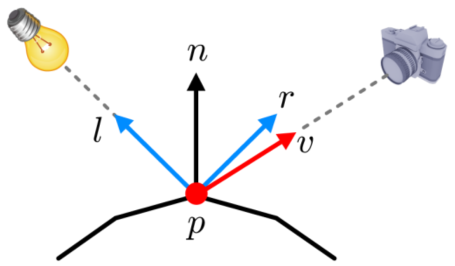
$$\begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\mathbf{EYE}_x \\ 0 & 1 & 0 & -\mathbf{EYE}_y \\ 0 & 0 & 1 & -\mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z & -\mathbf{EYE}\cdot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

- OpenGL view matrix

```
void gluLookAt(
   GLdouble Eye_x, GLdouble Eye_y, GLdouble Eye_z,
   GLdouble At_x, GLdouble At_y, GLdouble At_z,
   GLdouble Up_x, GLdouble Up_y, GLdouble Up_z
);
```

Per-vertex Lighting

- Light emitted from a light source is reflected by the object surface and then reaches the camera.



- The above figure describes what kinds of parameters are needed for computing lighting at vertex *p*. It is called *per-vertex lighting* and is done by the vertex program.
- Per-vertex lighting is old-fashioned. More popular is *per-fragment lighting*. It is performed by the fragment program and produces a better result.
- Understand that a vertex color can be computed at the vertex processing stage

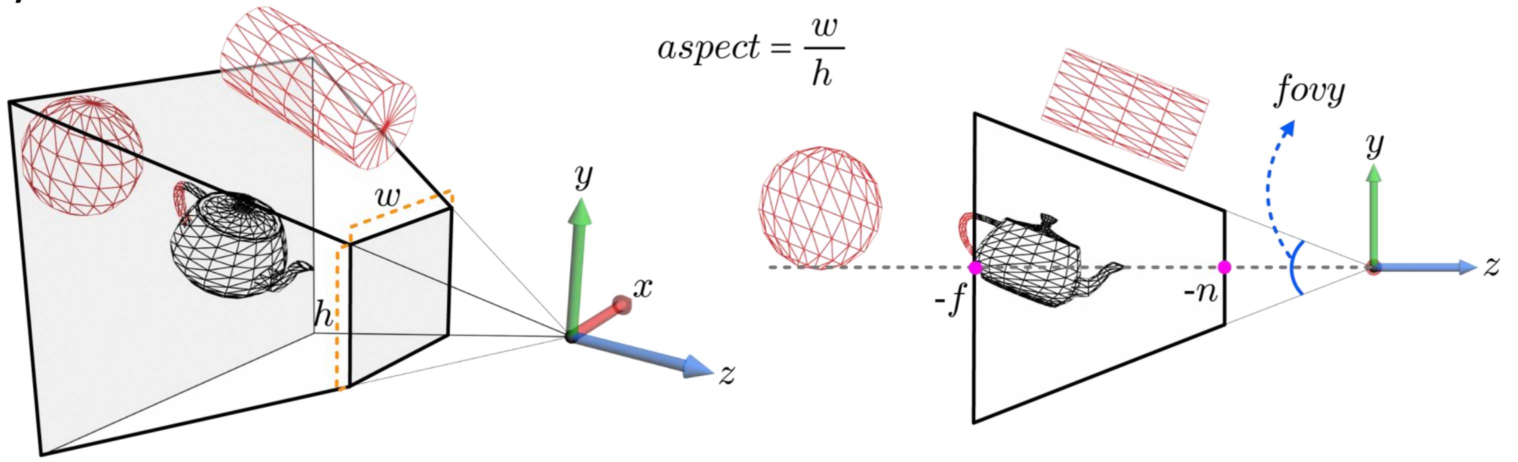
PROJECT TRANSFORM

To simulate how the real cameras capture the scene

View Frustum

$$\begin{tikzcd}\text{object space} \arrow[r, "\text{world transform}"] & \text{world space} \arrow[r, "\text{view transform}"] & \text{camera space} \arrow[r, "\text{projection transform}"] & \text{clip space}\end{tikzcd}$$

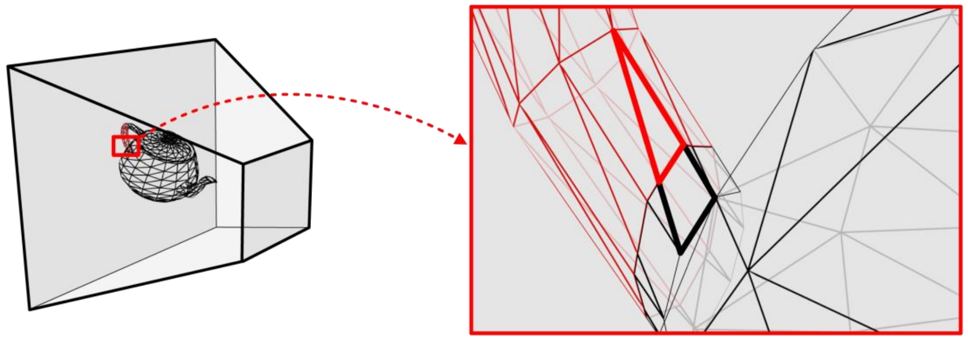
- Let us denote the basis of the camera space by {*x*, *y*, *z*} instead of {*u*, *v*, *n*}.
- Recall that, for constructing the view transform, we defined the external parameters of the camera, i.e., **EYE**, **AT**, and **UP**. Now let us control the camera's internals. It is analogous to choosing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



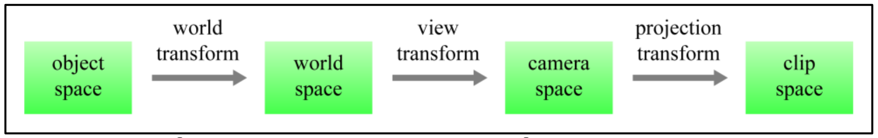
- The near and far planes run counter to the real-world camera or human vision system, but have been introduced for the sake of computational efficiency.

View Frustum (cont')

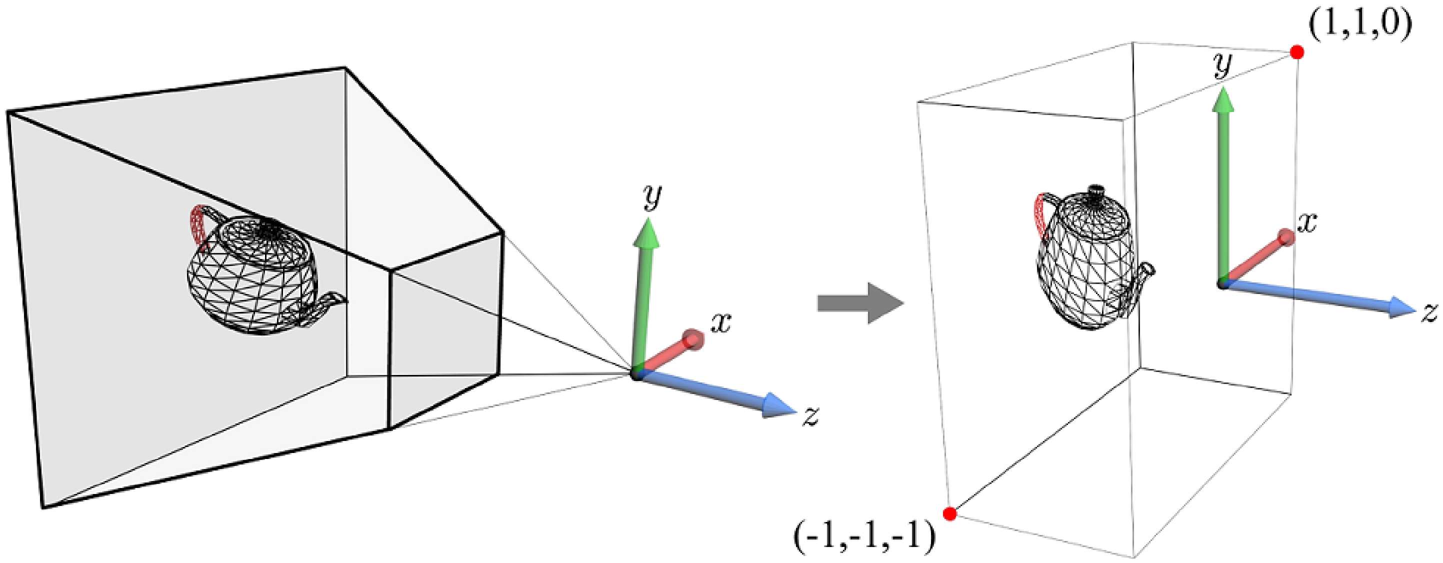
- View-frustum culling
  - A large enough box or sphere bounding a polygon mesh is computed at the preprocessing step, and then at run time a CPU program tests if the bounding volume is outside the view frustum. If it is, the polygon mesh is discarded and does not enter the rendering pipeline.
  - It can save a fair amount of GPU computing cost with a little CPU overhead.
- The cylinder and sphere would be discarded by the 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

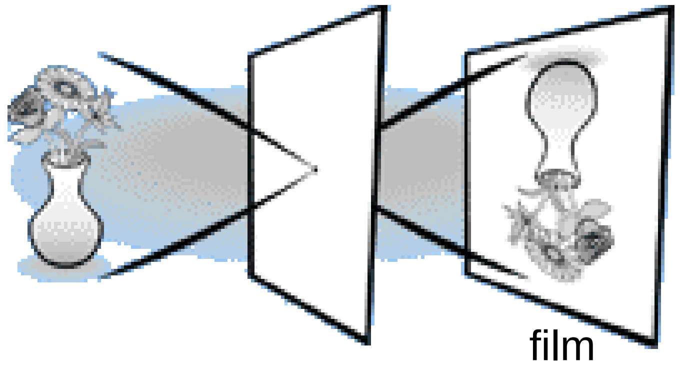


- It is not easy to clip the polygons with respect to the view frustum.
- If there is a transform that converts the view frustum to the axis-aligned box, and the transform is applied to the polygons of the scene, clipping the transformed polygons with respect to the box is much easier.



Projection Transform (cont')

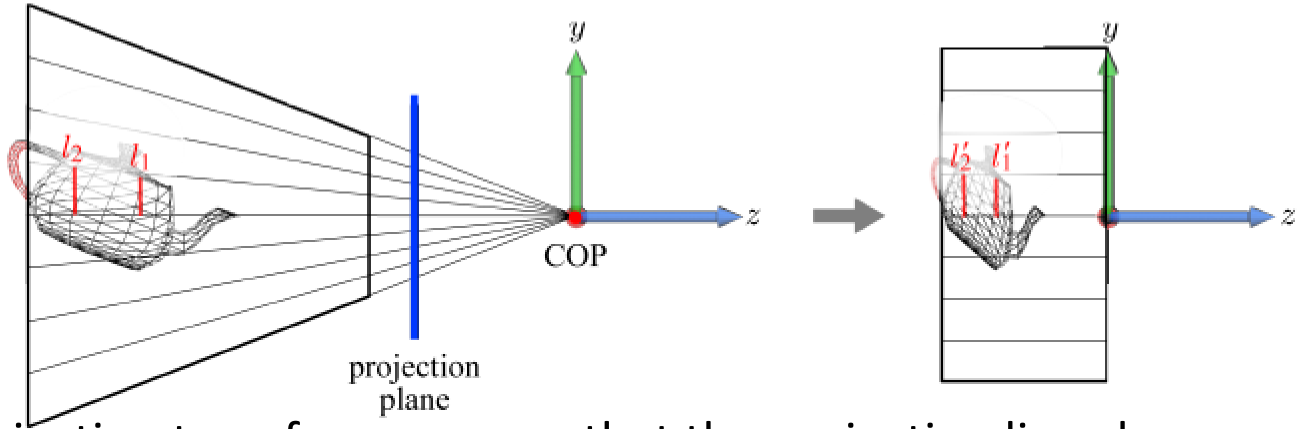
- Consider pinhole camera, which is the simplest imaging device with an infinitesimally small aperture.



- The convergent pencil of projection lines focuses on the aperture.
- The film corresponds to the projection plane.

Projection Transform (cont')

- 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. The origin is often called the center of projection (COP).
- 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.



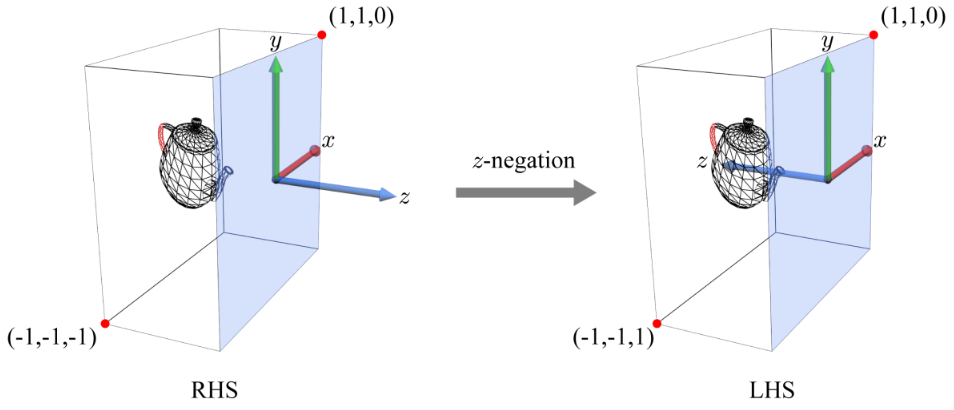
- The projection transform ensures that the projection lines become parallel, i.e., we have a *universal* projection line. Now viewing is done along the universal projection line. It is called the *orthographic projection*. The projection transform brings the effect of perspective projection "within a 3D space."

Projection Transform (cont')

- Projection transform matrix

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

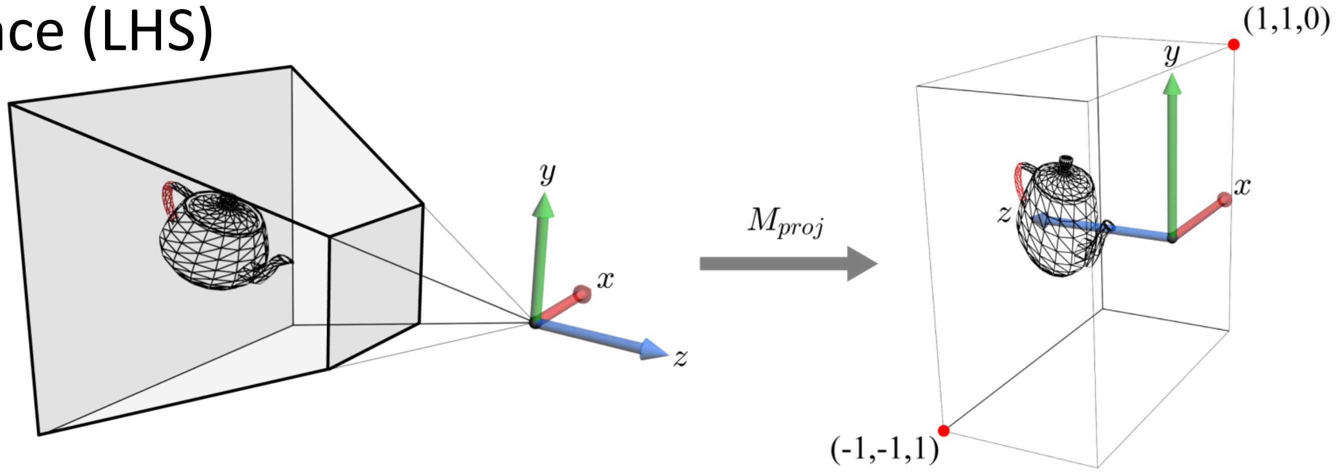
- The projection-transformed objects will enter the rasterization stage.
- Unlike the vertex processing stage, the rasterization stage is implemented in hardware, and assumes that the clip space is left-handed. In order for the vertex processing stage to be compatible with the hard-wired rasterization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



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

Projection Transform (cont')

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



- D3DXMatrixPerspectiveFovRH builds the projection transform matrix.

```
D3DXMATRIX *WINAPI D3DXMatrixPerspectiveFovRH(
    D3DXMATRIX *pOut,
    FLOAT fovy,              // in radians
    FLOAT Aspect,            // width divided by height
    FLOAT zn,
    FLOAT zf );
```

Projection Transform (cont')

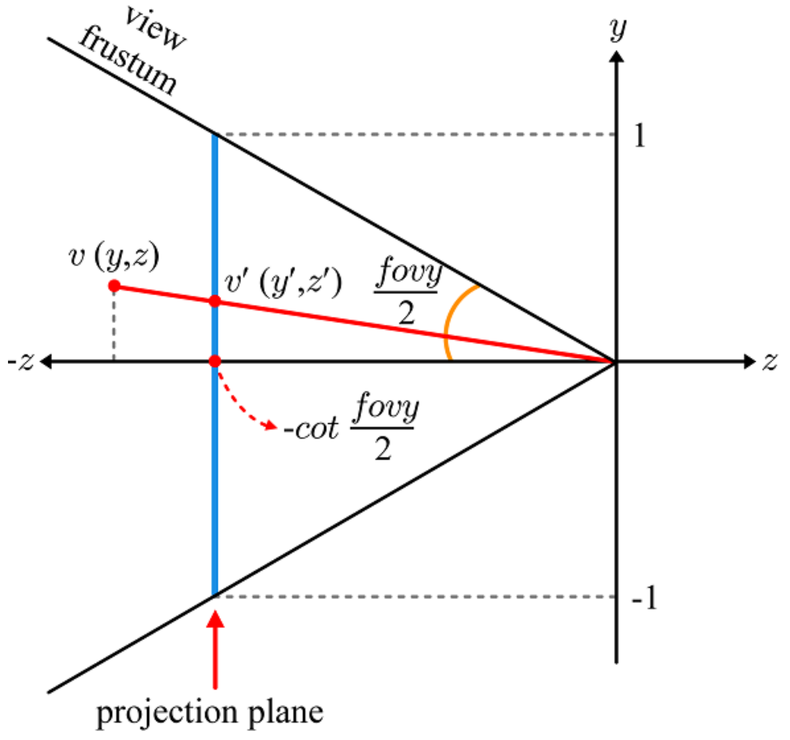
- OpenGL function for projection matrix

```
void gluPerspective(
  GLdouble fovy,
  GLdouble aspect,
  GLdouble n,
  GLdouble f
);
```

- In OpenGL, the clip-space cuboid has a different dimension, and consequently the projection transform is different. See the book.

Deriving Projection Transform

- Based on the fact that projection-transformed *y* coordinate (*y*´) is in the range of [-1,1], we can compute the general representation of *y*´.



$$y : z = y' : \text{-}cot\,\frac{fovy}{2}$$

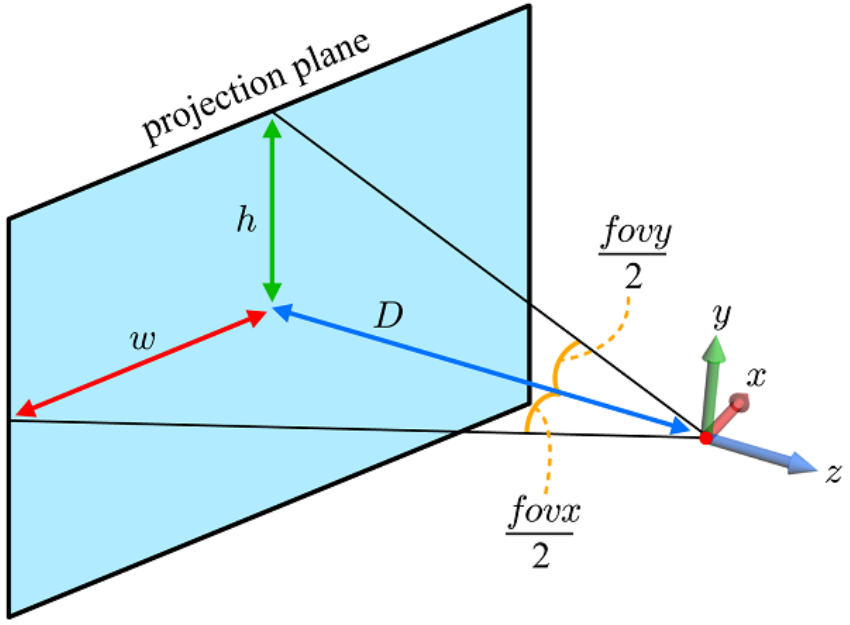
$$\rightarrow \quad y' = -cot\frac{fovy}{2} \cdot \frac{y}{z}$$

$$x' = -cot\frac{fovx}{2} \cdot \frac{x}{z}$$

- As shown above, we could compute *x*´ in a similar way if *fovx* were given.

Deriving Projection Transform (cont')

- Unfortunately *fovx* is not given, and therefore let's define *x´* in terms of *fovy* and *aspect*.



$$x' = -cot\frac{fovx}{2} \cdot \frac{x}{z}$$

$$\downarrow$$

$$\boxed{x' = -cot\frac{fovx}{2} \cdot \frac{x}{z} = -\frac{cot\frac{fovy}{2}}{aspect} \cdot \frac{x}{z}}$$

$$\uparrow$$

$$cot\frac{fovx}{2} = \frac{cot\frac{fovy}{2}}{aspect}$$

$$\uparrow$$

$$\frac{w}{D} = tan\frac{fovx}{2} \rightarrow w = D \cdot tan\frac{fovx}{2}$$

$$\frac{h}{D} = tan\frac{fovy}{2} \rightarrow h = D \cdot tan\frac{fovy}{2}$$

$$\rightarrow \frac{w}{h} = \frac{tan\frac{fovx}{2}}{tan\frac{fovy}{2}} = \frac{cot\frac{fovy}{2}}{cot\frac{fovx}{2}}$$

Deriving Projection Transform (cont')

- We have found *x´* and *y´*.

$$x' = -\cot\frac{fovx}{2}\cdot\frac{x}{z} = -\frac{\overbrace{\boxed{\cot\frac{fovy}{2}}}^{D}}{\underbrace{\boxed{aspect}}_{A}}\cdot\frac{x}{z}$$

$$y' = -\underbrace{\boxed{\cot\frac{fovy}{2}}}_{D}\cdot\frac{y}{z}$$

- Homogeneous coordinates representation

$$v' = (x', y', z', 1) = (-\frac{D}{A}\cdot\frac{x}{z}, -D\frac{y}{z}, z', 1) \rightarrow (\frac{D}{A}x, Dy, -zz', -z)$$

- Then, we have the following projection matrix. Note that *z´* and *z´´* are independent of *x* and *y*, and therefore m_1 and m_2 are 0s.

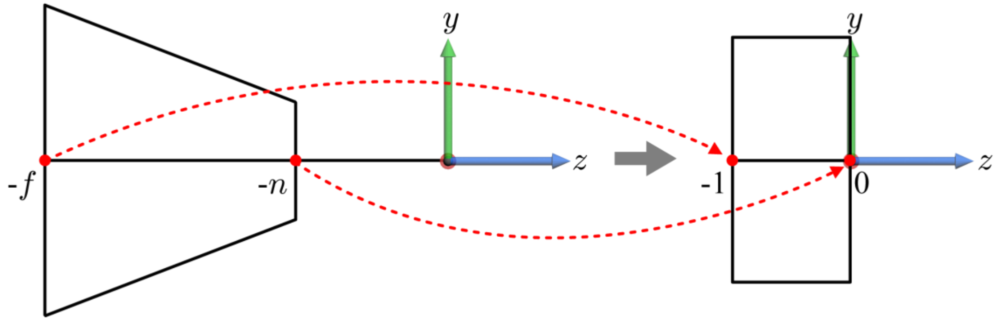
$$\begin{pmatrix} \frac{D}{A}x \\ Dy \\ z'' \\ -z \end{pmatrix} = \begin{pmatrix} \frac{D}{A} & 0 & 0 & 0 \\ 0 & D & 0 & 0 \\ \underbrace{\boxed{m_1 \;\; m_2}}_{\text{just } 0} & & m_3 & m_4 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Deriving Projection Transform (cont')

- Let's apply the projection matrix.

$$\begin{pmatrix} \frac{D}{A} & 0 & 0 & 0 \\ 0 & D & 0 & 0 \\ 0 & 0 & m_3 & m_4 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{D}{A}x \\ Dy \\ m_3 z + m_4 \\ -z \end{pmatrix} \rightarrow \begin{pmatrix} -\frac{D}{A} \cdot \frac{x}{z} \\ -D\frac{y}{z} \\ -m_3 - \frac{m_4}{z} \\ 1 \end{pmatrix} = v'$$

- In projection transform, observe that *–f* and *-n* are transformed to -1 and 0, respectively.



- Using the fact, the projection matrix can be completed.

$$z' = -m_3 - \frac{m_4}{z} \rightarrow \begin{matrix} -1 = -m_3 + \frac{m_4}{f} \\ 0 = -m_3 + \frac{m_4}{n} \end{matrix} \rightarrow \begin{matrix} m_3 = \frac{f}{f-n} \\ m_4 = \frac{nf}{f-n} \end{matrix} \rightarrow \begin{pmatrix} \frac{cot(\frac{fovy}{2})}{aspect} & 0 & 0 & 0 \\ 0 & cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & \frac{nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$