CS380: Computer Graphics
Viewing Transformation

Sung-Eui Yoon
(윤성의)

Course URL:
http://sglab.kaist.ac.kr/~sungeui/CG/

KAIST

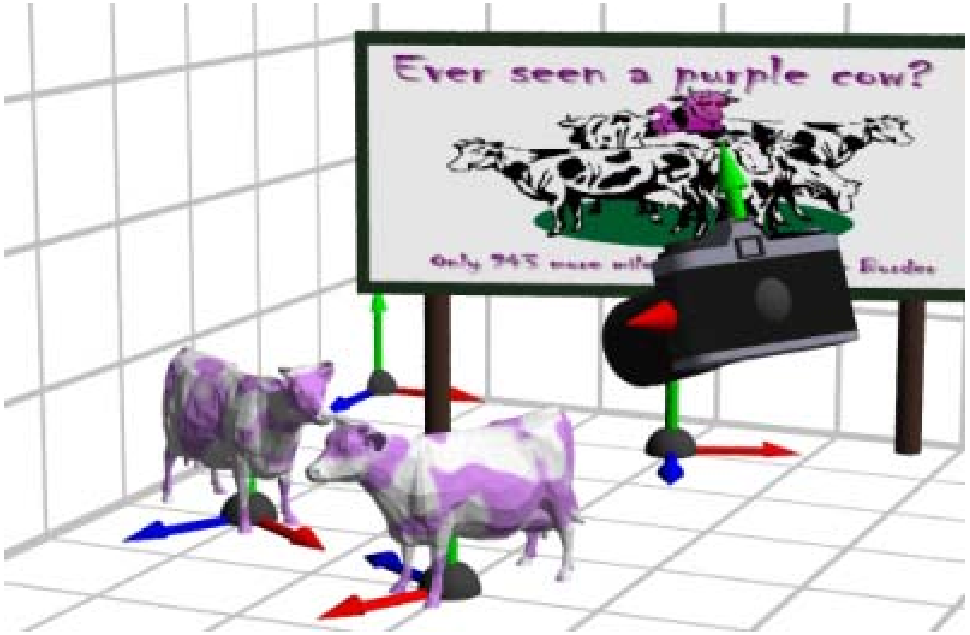
Class Objectives

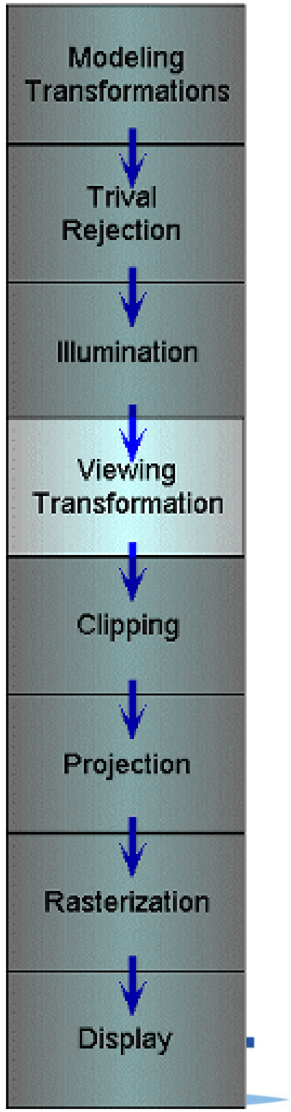
- Know camera setup parameters
- Understand viewing and projection processes

KAIST

Viewing Transformations

- **Map points from world spaces to eye space**
  - **Can be composed from rotations and translations**



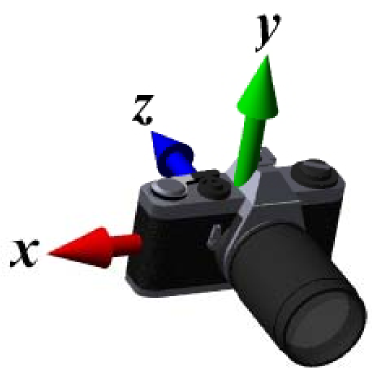


Viewing Transformations

- Goal: specify position and orientation of our camera
  - Defines a coordinate frame for eye space

“Framing” the Picture

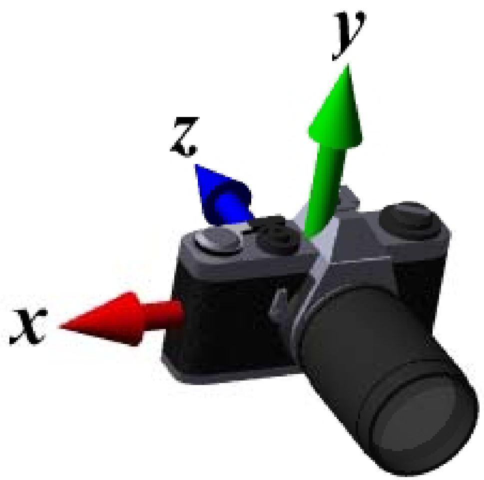
- A new camera coordinate
  - Camera position at the origin
  - Z-axis aligned with the view direction
  - Y-axis aligned with the up direction



- More natural to think of camera as an object positioned in the world frame

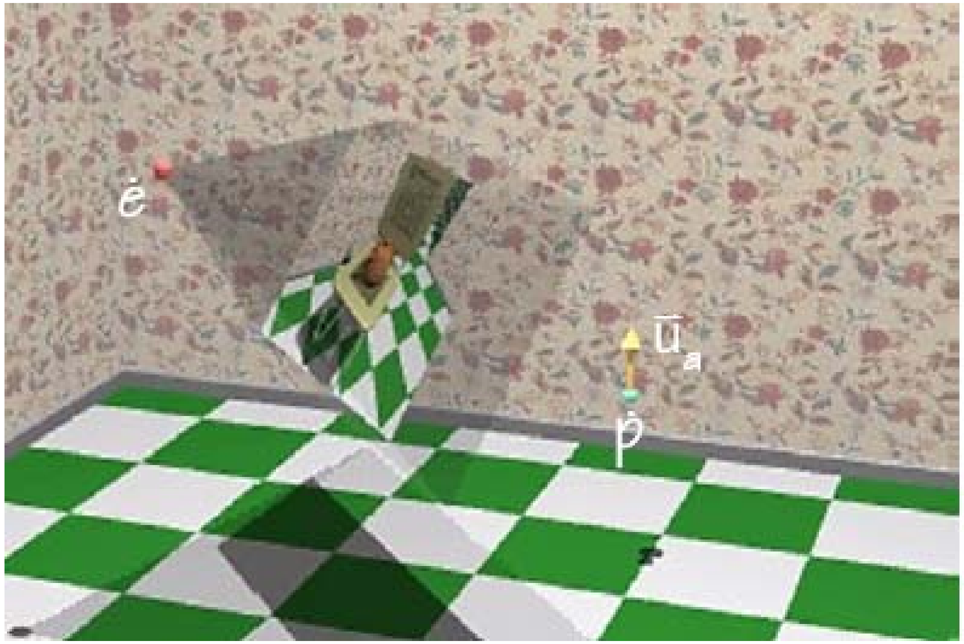
Viewing Steps

- Rotate to align the two coordinate frames and, then, translate to move world space origin to camera's origin



An Intuitive Specification

- Specify three quantities:
  - Eye point (e) - position of the camera
  - Look-at point (p) - center of the image
  - Up-vector ($\vec{u}_a$) - will be oriented upwards in the image



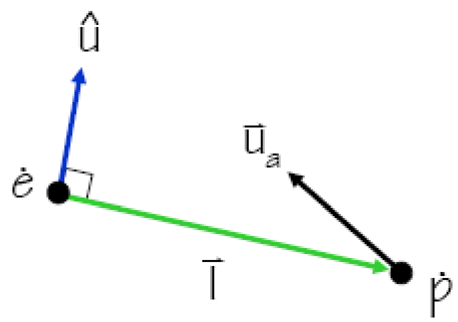
Deriving the Viewing Transformation

- **First compute the look-at vector and normalize**

$$\vec{l} = p - e \qquad \hat{l} = \frac{\vec{l}}{\left|\vec{l}\right|}$$

- **Compute right vector and normalize**
  - **Perpendicular to the look-at and up vectors**

$$\vec{r} = \vec{l} \times \vec{u}_a \qquad \hat{r} = \frac{\vec{r}}{\left|\vec{r}\right|}$$



- **Compute up vector**
  - $\vec{u}_a$ **is only approximate direction**
  - **Perpendicular to right and look-at vectors**

$$\hat{u} = \hat{r} \times \hat{l}$$

KAIST

Rotation Component

- **Map our vectors to the cartesian coordinate axes**

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \hat{r} & \hat{u} & -\hat{l} \end{bmatrix} R_v$$

- **To compute** R_v **we invert the matrix on the right**
  - **This matrix M is orthonormal (or orthogonal) – its rows are orthonormal basis vectors: vectors mutually orthogonal and of unit length**
  - **Then,** $M^{-1} = M^T$
  - **So,**

$$\mathbf{R}_v = \begin{bmatrix} \hat{\mathbf{r}}^t \\ \hat{\mathbf{u}}^t \\ -\hat{\mathbf{l}}^t \end{bmatrix}$$

Translation Component

- **The rotation that we just derived is specified about the eye point in world space**
  - **Need to translate all world-space coordinates so that the eye point is at the origin**
  - **Composing these transformations gives our viewing transform,** V

$$\dot{w}^t = \dot{e}^t \mathbf{R}_v \mathbf{T}_{-\dot{e}}$$

$$\mathbf{V} = \mathbf{R}_v \mathbf{T}_{-\dot{e}} = \begin{bmatrix} \hat{r}_x & \hat{r}_y & \hat{r}_z & 0 \\ \hat{u}_x & \hat{u}_y & \hat{u}_z & 0 \\ -\hat{l}_x & -\hat{l}_y & -\hat{l}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} & \hat{r} & & -\hat{r} \cdot \vec{e} \\ & \hat{u} & & -\hat{u} \cdot \vec{e} \\ & -\hat{l} & & \hat{l} \cdot \vec{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transform a world-space point into a point in the eye-space

Viewing Transform in OpenGL

- **OpenGL utility (glu) library provides a viewing transformation function:**

gluLookAt (double eyex, double eyey, double eyez,
double centerx, double centery, double centerz,
double upx, double upy, double upz)

  - **Computes the same transformation that we derived and composes it with the current matrix**

Example in the Skeleton Codes of PA2

```
void setCamera ()
{ …
 // initialize camera frame transforms
   for (i=0; i < cameraCount; i++ )
   {
     double* c = cameras[i];
     wld2cam.push_back(FrameXform());
     glPushMatrix();
     glLoadIdentity();
     gluLookAt(c[0],c[1],c[2], c[3],c[4],c[5], c[6],c[7],c[8]);
     glGetDoublev( GL_MODELVIEW_MATRIX, wld2cam[i].matrix() );
     glPopMatrix();
     cam2wld.push_back(wld2cam[i].inverse());
   }
….
}
```

Projections

- Map 3D points in eye space to 2D points in image space

- Two common methods
  - Orthographic projection
  - Perspective projection

Orthographic Projection

- Projects points along lines parallel to z-axis
  - Also called parallel projection
  - Used for top and side views in drafting and modeling applications
- Appears unnatural due to lack of perspective foreshortening

Notice that the parallel lines of the tiled floor remain parallel after orthographic projection!

Orthographic Projection

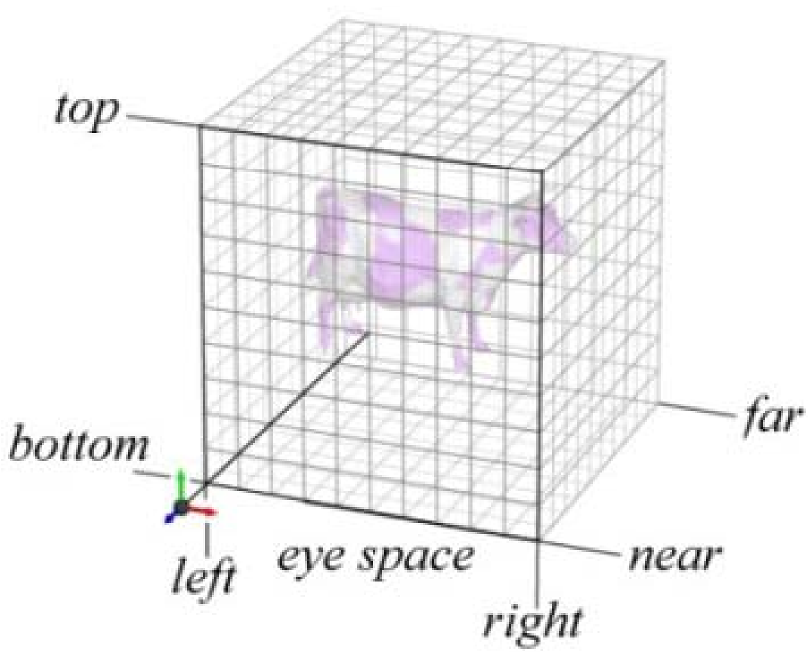
- **The projection matrix for orthographic projection is very simple**

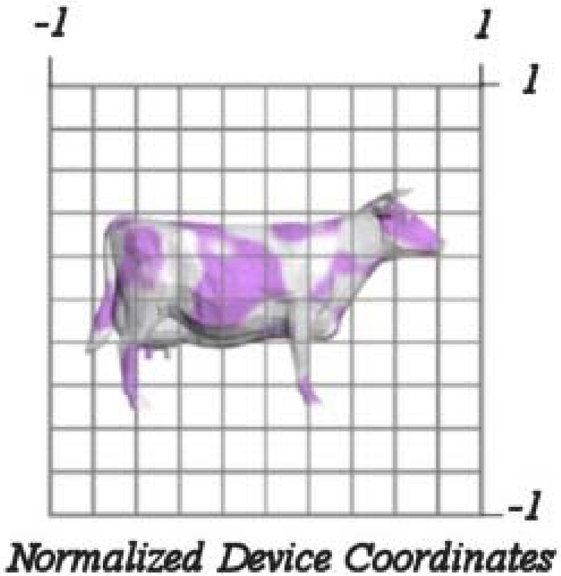
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- **Next step is to convert points to NDC**

View Volume and Normalized Device Coordinates

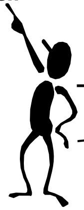
- Define a view volume
- Compose projection with a scale and a translation that maps eye coordinates to normalized device coordinates





Orthographic Projections to NDC

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & \frac{-(right+left)}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & \frac{-(top+bottom)}{top-bottom} \\ 0 & 0 & \frac{2}{far-near} & \frac{-(far+near)}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



Scale the z coordinate in exactly the same way .Technically, this coordinate is not part of the projection. But, we will use this value of z for other purposes

Some sanity checks:

$$x = left \Rightarrow x' = \frac{2 \cdot left}{right-left} - \frac{right+left}{right-left} = -\frac{right-left}{right-left} = -1$$

$$x = right \Rightarrow x' = \frac{2 \cdot right}{right-left} - \frac{right+left}{right-left} = \frac{right-left}{right-left} = 1$$

KAIST

Orthographic Projection in OpenGL

- **This matrix is constructed by the following OpenGL call:**

```
void glOrtho(double left, double right,
             double bottom, double top,
             double near, double far );
```

- **2D version (another GL utility function):**

```
void gluOrtho2D( double left, GLdouble right,
                 double bottom, GLdouble top);
```

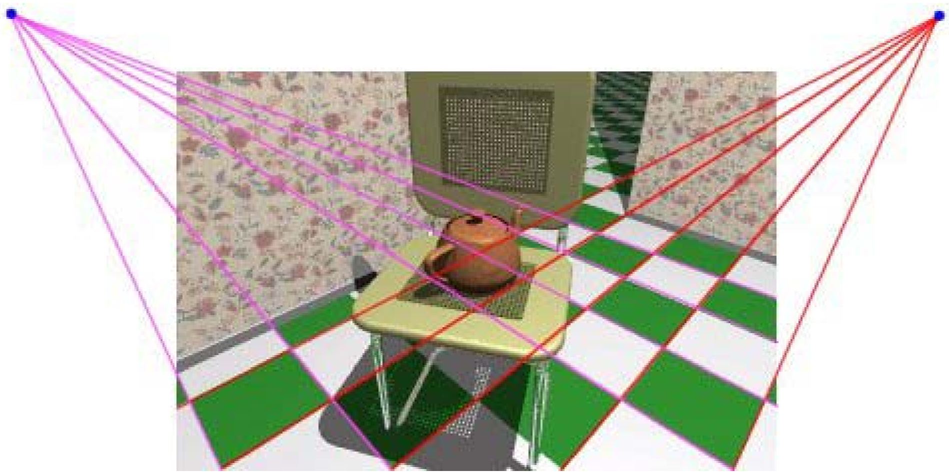
, which is just a call to glOrtho( ) with near = -1 and far = 1

Perspective Projection

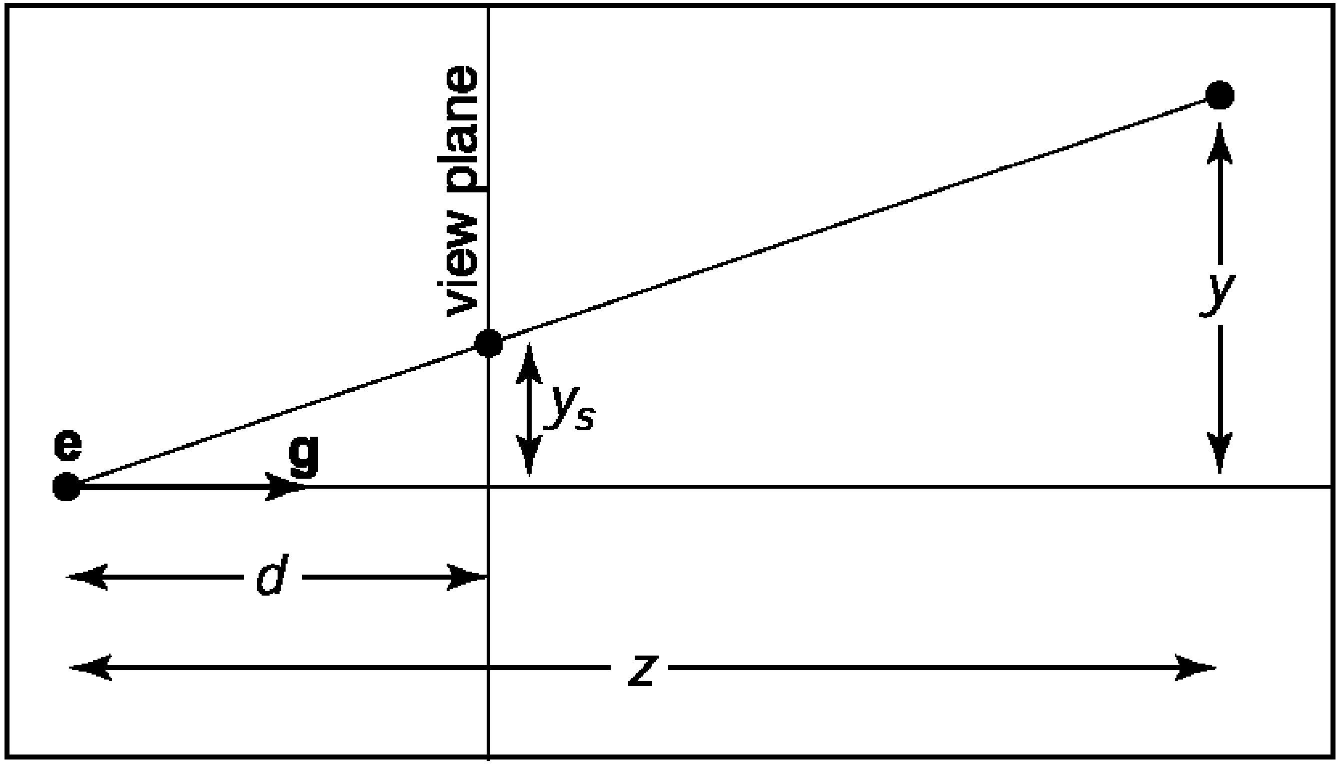
- Artists (Donatello, Brunelleschi, Durer, and Da Vinci) during the renaissance discovered the importance of perspective for making images appear realistic
- Perspective causes objects nearer to the viewer to appear larger than the same object would appear farther away
- Homogenous coordinates allow perspective projections using linear operators

Signs of Perspective

- **Lines in projective space always intersect at a point**



Perspective Projection



$$y_s = d\frac{y}{z}$$

Perspective Projection Matrix

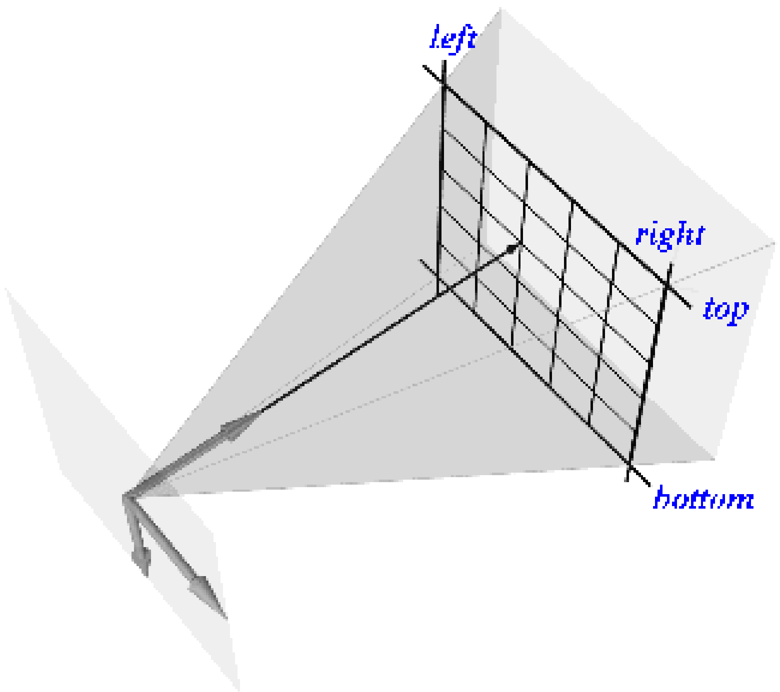
- **The simplest transform for perspective projection is:**

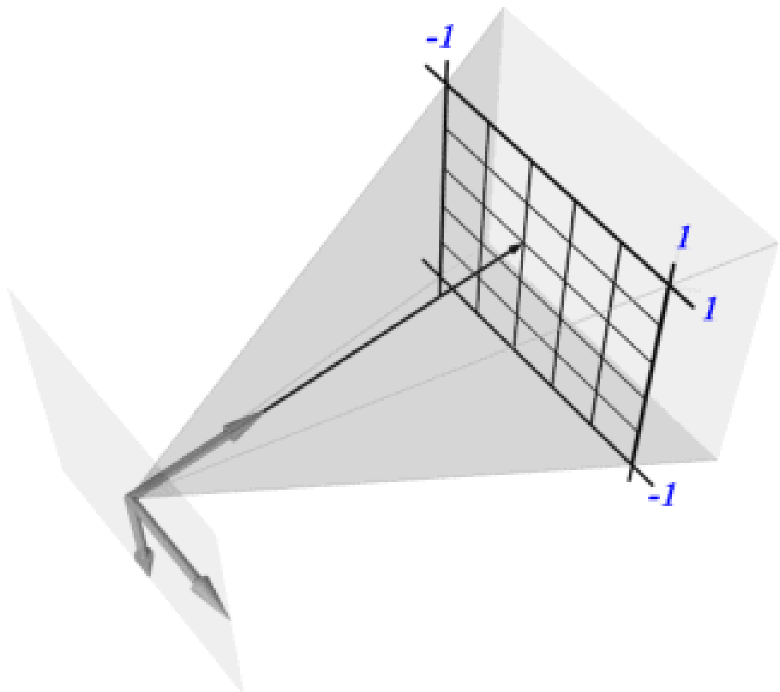
$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- **We divide by w to make the fourth coordinate 1**
  - **In this example, w = z**
  - **Therefore, x' = x / z, y' = y / z, z' = 0**

Normalized Perspective

- **As in the orthographic case, we map to normalized device coordinates**





KAIST

NDC Perspective Matrix

$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} \frac{2 \cdot near}{right - left} & 0 & \frac{-(right + left)}{right - left} & 0 \\ 0 & \frac{2 \cdot near}{top - bottom} & \frac{-(top + bottom)}{top - bottom} & 0 \\ 0 & 0 & \frac{far + near}{far - near} & \frac{-2 \cdot far \cdot near}{far - near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- **The values of left, right, top, and bottom are specified at the near depth. Let's try some sanity checks:**

$$\begin{matrix} x = left \\ z = near \end{matrix} \Rightarrow x' = \frac{\frac{2 \cdot near \cdot left}{right - left} - \frac{near(right + left)}{right - left}}{near} = \frac{-near}{near} = -1$$

$$\begin{matrix} x = right \\ z = near \end{matrix} \Rightarrow x' = \frac{\frac{2 \cdot near \cdot right}{right - left} - \frac{near(right + left)}{right - left}}{near} = \frac{near}{near} = 1$$

KAIST

NDC Perspective Matrix

$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} \frac{2 \cdot near}{right - left} & 0 & \frac{-(right + left)}{right - left} & 0 \\ 0 & \frac{2 \cdot near}{top - bottom} & \frac{-(top + bottom)}{top - bottom} & 0 \\ 0 & 0 & \frac{far + near}{far - near} & \frac{-2 \cdot far \cdot near}{far - near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- **The values of left, right, top, and bottom are specified at the near depth. Let's try some sanity checks:**

$$z = far \Rightarrow z' = \frac{far \frac{far + near}{far - near} + \frac{-2 \cdot far \cdot near}{far - near}}{far} = \frac{\frac{far\,(far - near)}{far - near}}{far} = 1$$

$$z = near \Rightarrow z' = \frac{near \frac{far + near}{far - near} + \frac{-2 \cdot far \cdot near}{far - near}}{near} = \frac{\frac{near(near - far)}{far - near}}{near} = -1$$

Perspective in OpenGL

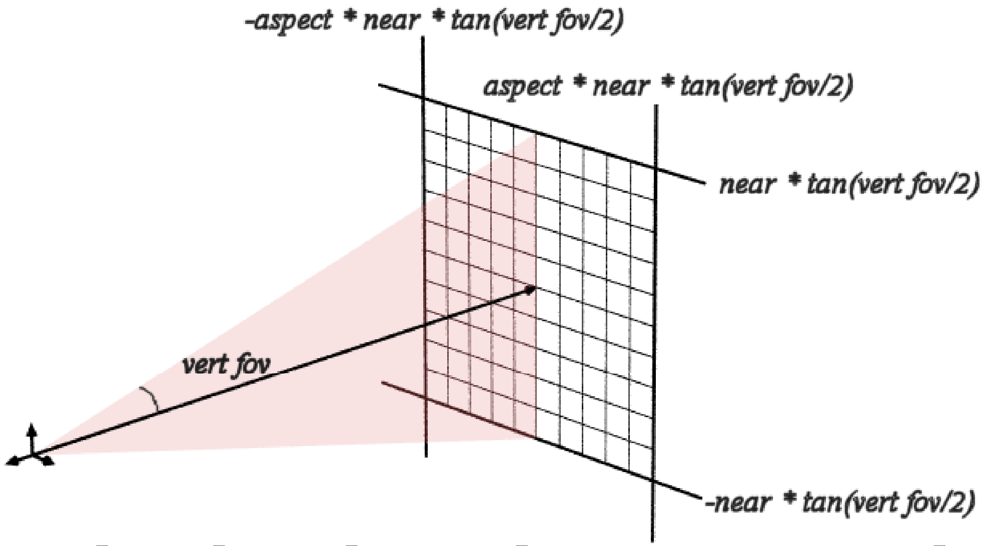
- **OpenGL provides the following function to define perspective transformations:**

void glFrustum(double *left*, double *right*,
double *bottom*, double *top*,
double *near*, double *far*);

- **Some think that using glFrustum( ) is nonintuitive. So OpenGL provides a function with simpler, but less general capabilities**

void gluPerspective(double *vertfov*, double *aspect*,
double *near*, double *far*);

gluPerspective()

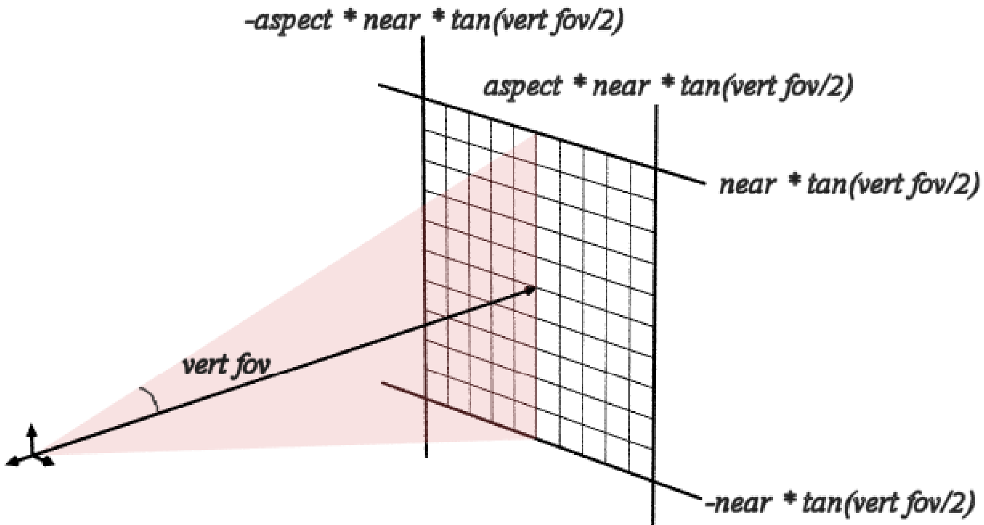


Simple "camera-like" model

Can only specify symmetric frustums

- **Substituting the extents into glFrustum()**

gluPerspective()



Simple "camera-like" model

Can only specify symmetric frustums

- **Substituting the extents into glFrustum()**

$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} \frac{COT(\frac{vertfov}{2})}{aspect} & 0 & 0 & 0 \\ 0 & COT(\frac{vertfov}{2}) & 0 & 0 \\ 0 & 0 & \frac{far+near}{far-near} & \frac{-2 \cdot far \cdot near}{far-near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Example in the Skeleton Codes of PA2

```
void reshape( int w, int h)
{
  width = w;    height = h;
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);          // Select The Projection Matrix
  glLoadIdentity();                     // Reset The Projection Matrix
  // Define perspective projection frustum
  double aspect = width/double(height);

  gluPerspective(45, aspect, 1, 1024);
  glMatrixMode(GL_MODELVIEW);           // Select The Modelview Matrix

  glLoadIdentity();                     // Reset The Projection Matrix
}
```

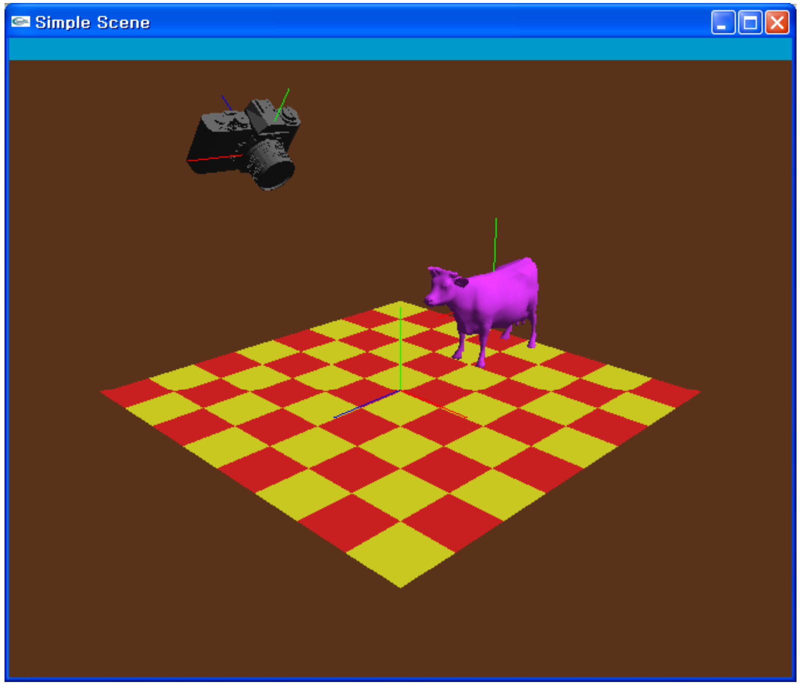
Class Objectives were:

- Know camera setup parameters
- Understand viewing and projection processes

Reading Assignment

- **Read the chapter "Data Structure for Graphics"**

PA3



- **PA2: perform the transformation at the modeling space**
- **PA3: perform the transformation at the viewing space**

Next Time

- **Interaction**