Computer Graphics
- Transformations in OpenGL

Junjie Cao @ DLUT

Spring 2018

http://jjcao.github.io/ComputerGraphics/

Camera Analogy

- OpenGL coordinate system has different origin (lower-left corner) from the window system (upper-left corner)
- The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera
- The steps with a camera (or a computer) might be the following:
  - Arrange the scene to be photographed into the desired composition (**modelling** transformation)
  - Set up your tripod and pointing the camera at the scene (**viewing** transformation).
  - Choose a camera lens or adjust the zoom (**projection** transformation)
  - Determine how large you want the final photograph to be (**viewport** transformation)

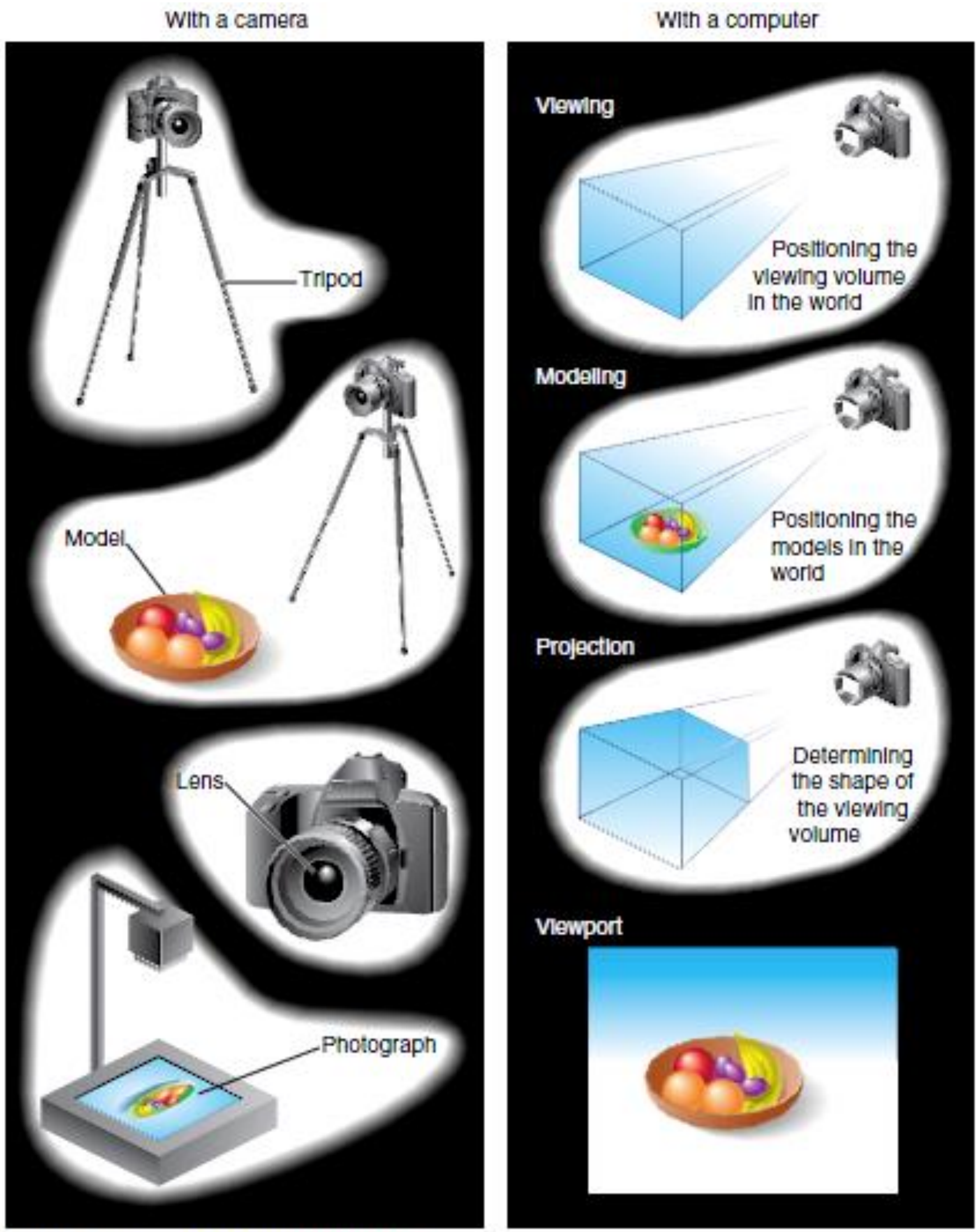
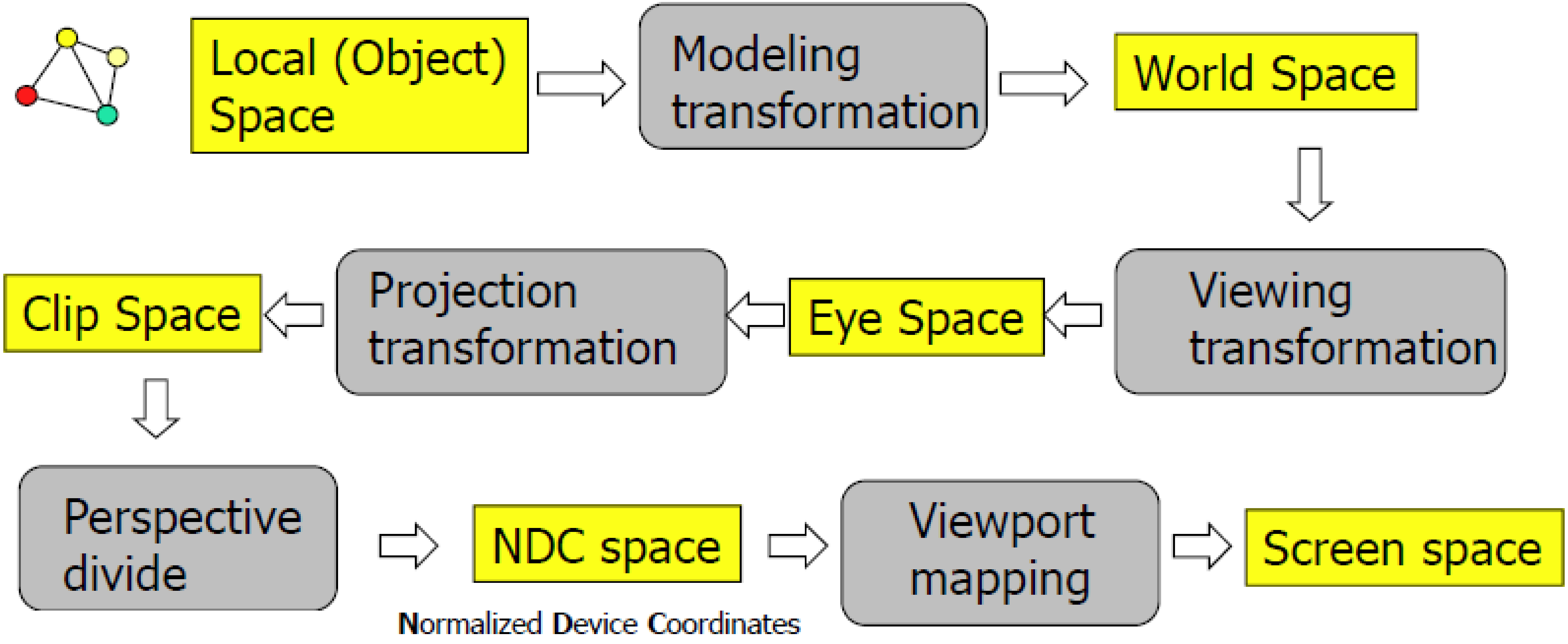


Figure 3-1 The Camera Analogy

Transformation Pipeline



Recall: Affine Transformations

- Given a point $[x\ y\ z]^\top$
- form homogeneous coordinates $[x\ y\ z\ 1]^\top$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The transformed point is $[x'\ y'\ z']^\top$

Transformation Matrices in OpenGL

- Transformation matrices in OpenGL are vectors of 16 values (**column-major** matrices)
- in glLoadMatrixf(GLfloat *m); $\mathbf{m}^\top = [m_1, m_2, \ldots, m_{16}]^\top$ represents

$$\begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

- OpenGL has 4 different types of matrices
  - **GL_MODELVIEW, GL_PROJECTION**, GL_TEXTURE, and GL_COLOR
  - Switch, e.g. **glMatrixMode(GL_MODELVIEW)**.

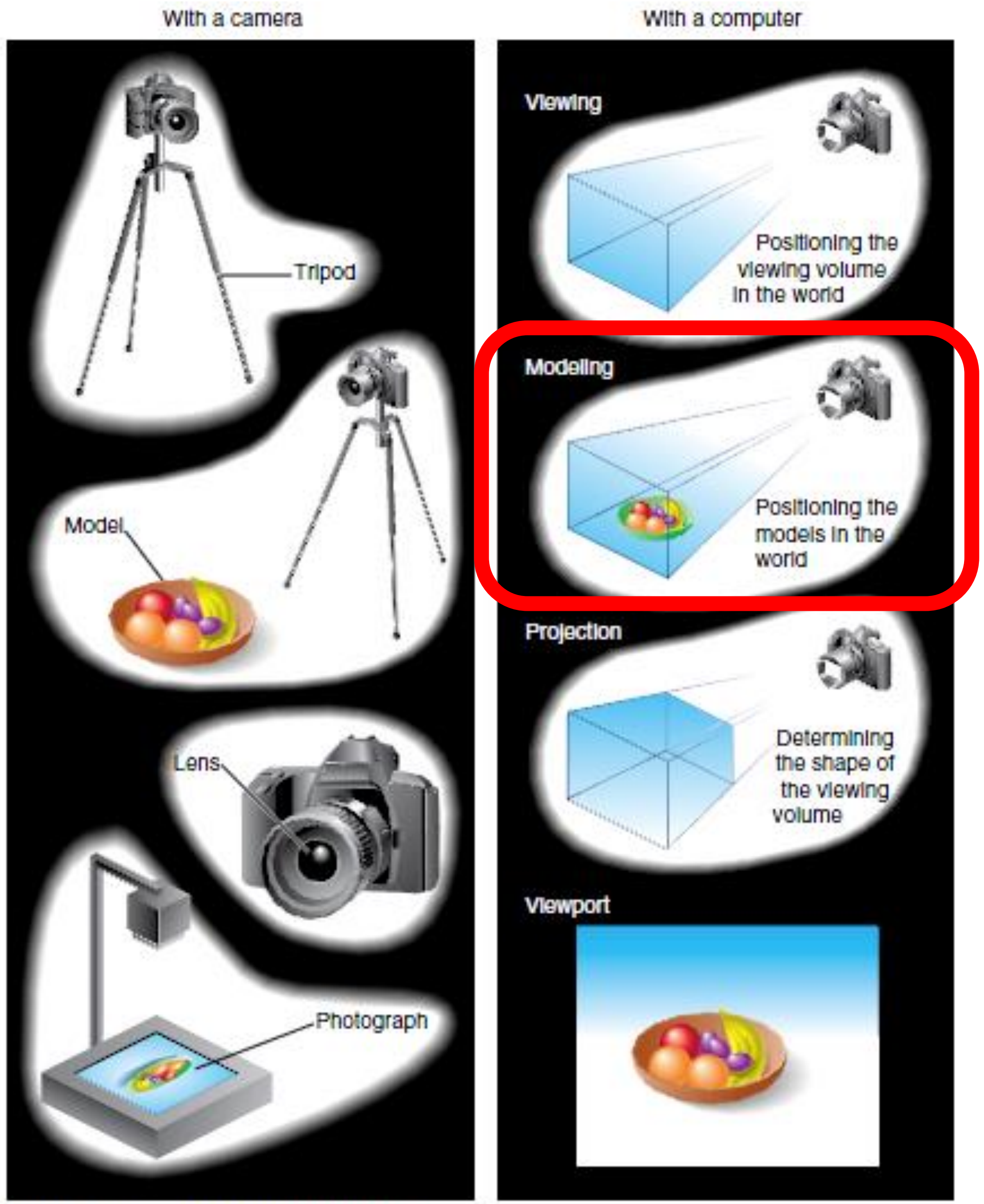
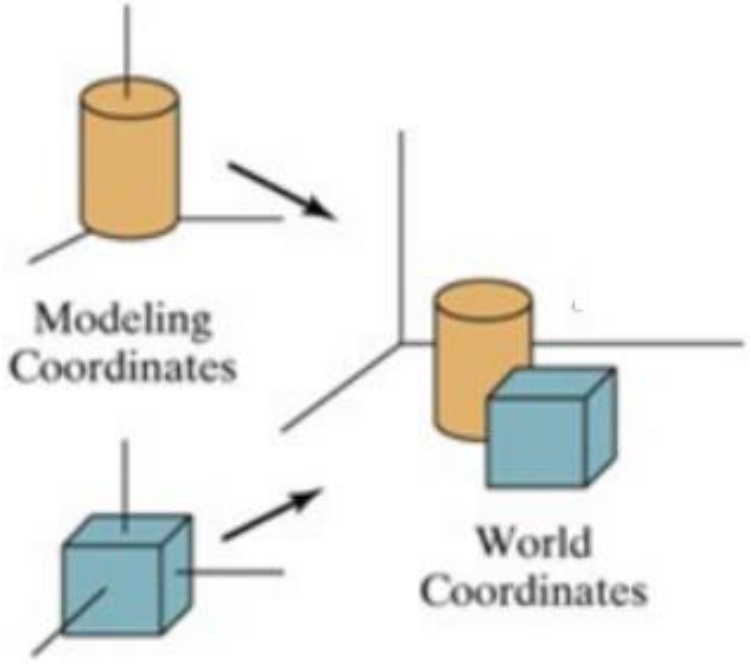


Figure 3-1 The Camera Analogy

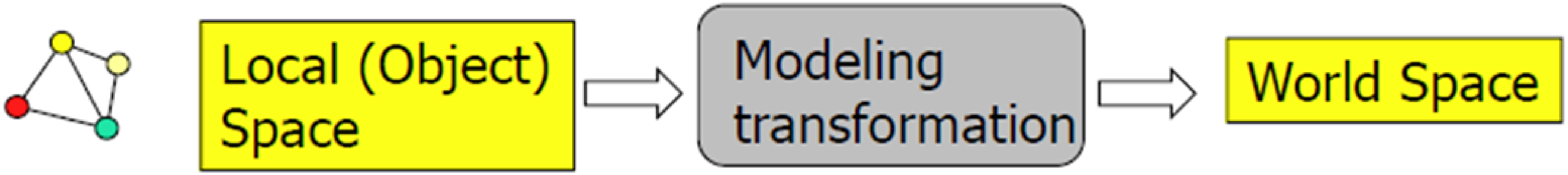
Local Coordinate System

- When you load a file containing a 3d object, its vertices stores coordinates in local CS.
- Assuming obj1, obj2 & obj3 are loaded.
  - Normally, their centers are the origins if they are actually created by code or hand.
  - Sometimes, their centers are not the origins of their local CS respectively if they are results of 3D scanning, etc.
  - Anyway, they are treated as local CS



World Coordinate System

- When the obj is just loaded, its local CS is used as WCS.
- To place multiple objs in your WCS, you need specify position, size, orientation of them
- Transformations need to be performed to position the object in WCS



- A modeling transformation is a sequence of translations, rotations, scalings (in arbitrary order) matrices multiplied together

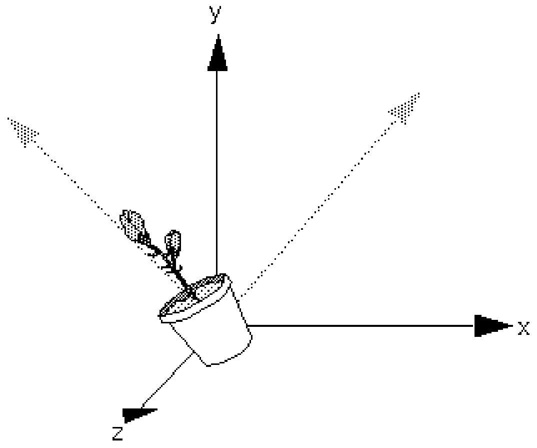
Modeling Transformations

- The three OpenGL routines for modeling transformations are:
  - glTranslate*(),
  - glScale*()
  - void **glRotate**{fd}(TYPE *angle*, TYPE *x*, TYPE *y*, TYPE *z*);
  - **glRotatef(**45.0, 0.0, 0.0, 1.0**)**

deprecated

- These routines **transform an object (or coordinate system**, if you're thinking of it that way) by moving, rotating, stretching it

- All three commands are **equivalent** to producing an appropriate translation, rotation, or scaling **matrix**, and then calling **glMultMatrix***() with that matrix as the argument



- OpenGL **automatically** computes the matrices for you

Modeling Transformations

- Each of these postmultiplies the *current matrix*
  - E.g., if current matrix is **C**, then **C=CS**
  - E.g., rotate then translate a vector x => T(Rx) = TRx not RTx

- The current matrix is either the **modelview** matrix or the projection matrix (also a texture matrix, won't discuss)
  - Set these with glMatrixMode(), e.g.:

    glMatrixMode(GL_MODELVIEW);

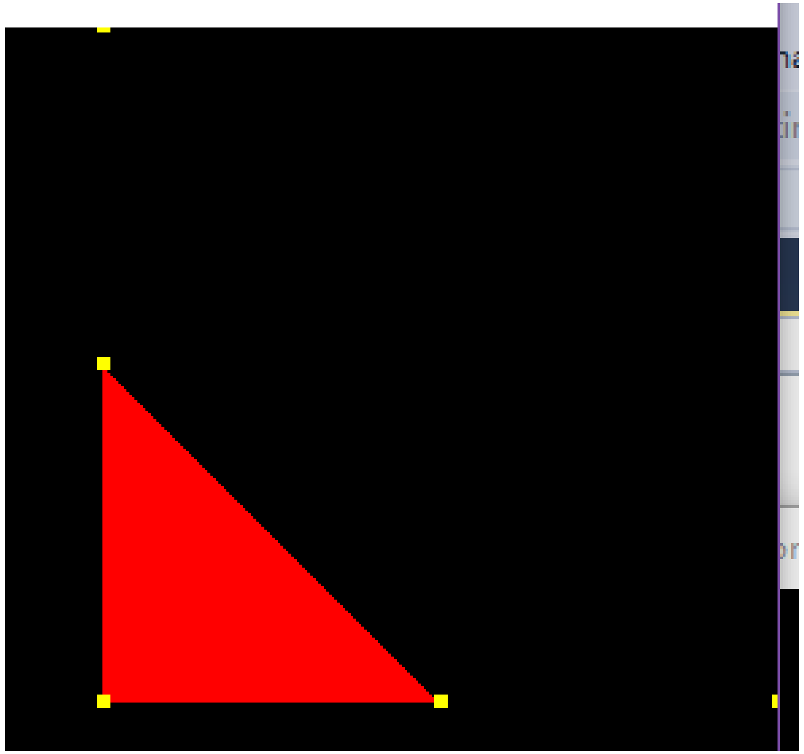
    glMatrixMode(GL_PROJECTION);

- **WARNING: common mistake ahead!**
  - Be sure that you are in **GL_MODELVIEW** mode before making modeling or viewing calls!
  - Ugly mistake because it can appear to work, at least for a while…, see https://sjbaker.org/steve/omniv/projection_abuse.html

Example for Modeling Transformation 1

```
void display() {
glClear(GL_COLOR_BUFFER_BIT);
glColor4f(1,1,0,1); //glColor* have been deprecated in OpenGL 3

// draw triangle 1
glBegin(GL_TRIANGLES);
glColor4f(1.0,0.0,0.0,1.0);glVertex3f(0.0, 0.0, -10.0);
glVertex3f(1.0, 0.0, -10.0); glVertex3f(0.0, 1.0, -10.0);
glEnd();
…
```



Example for Modeling Transformation 2

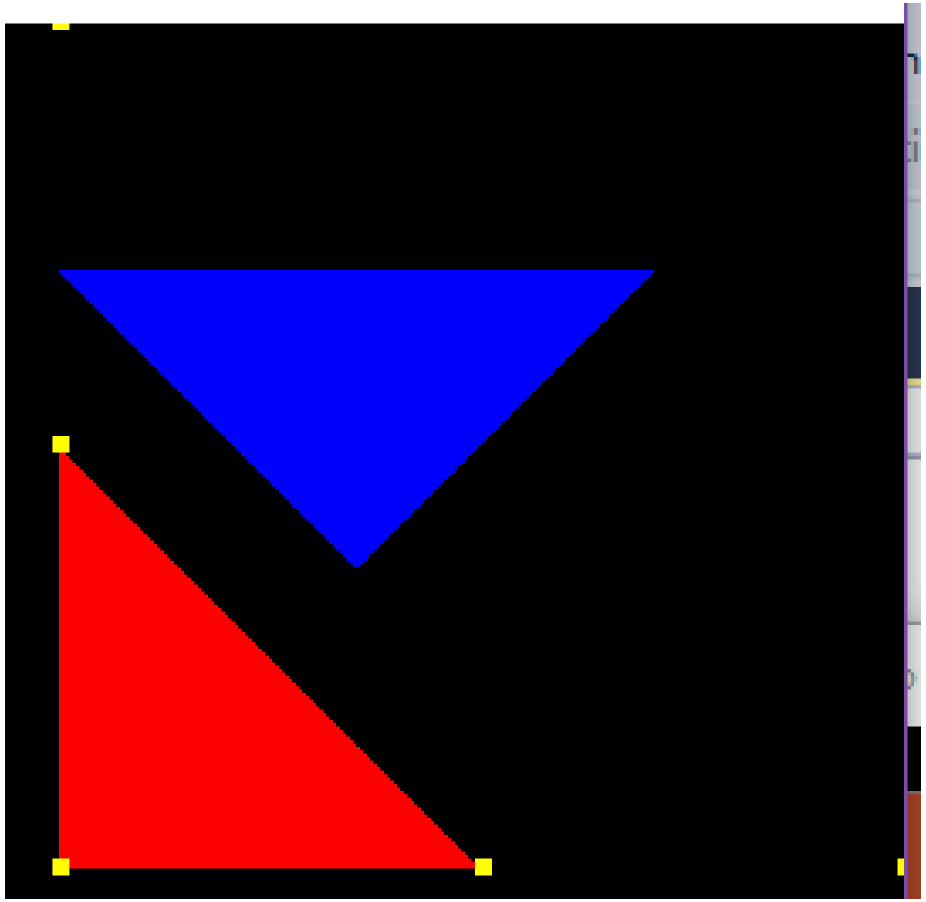
```
// draw triangle 3
glMatrixMode(GL_MODELVIEW);
glPushMatrix(); glLoadIdentity(); //More details will be explained
glRotatef(45, 0, 0, 1);
glTranslatef(1, 0, 0);
glBegin(GL_TRIANGLES);
glColor4f(0.0, 1.0, 0.0, 1.0);glVertex3f(0.0, 0.0, -10.0);
glVertex3f(1.0, 0.0, -10.0);glVertex3f(0.0, 1.0, -10.0);
glEnd();
glPopMatrix();

glutSwapBuffers();
}
```

Could you draw the two triangles on some paper?

Example for Modeling Transformation 2

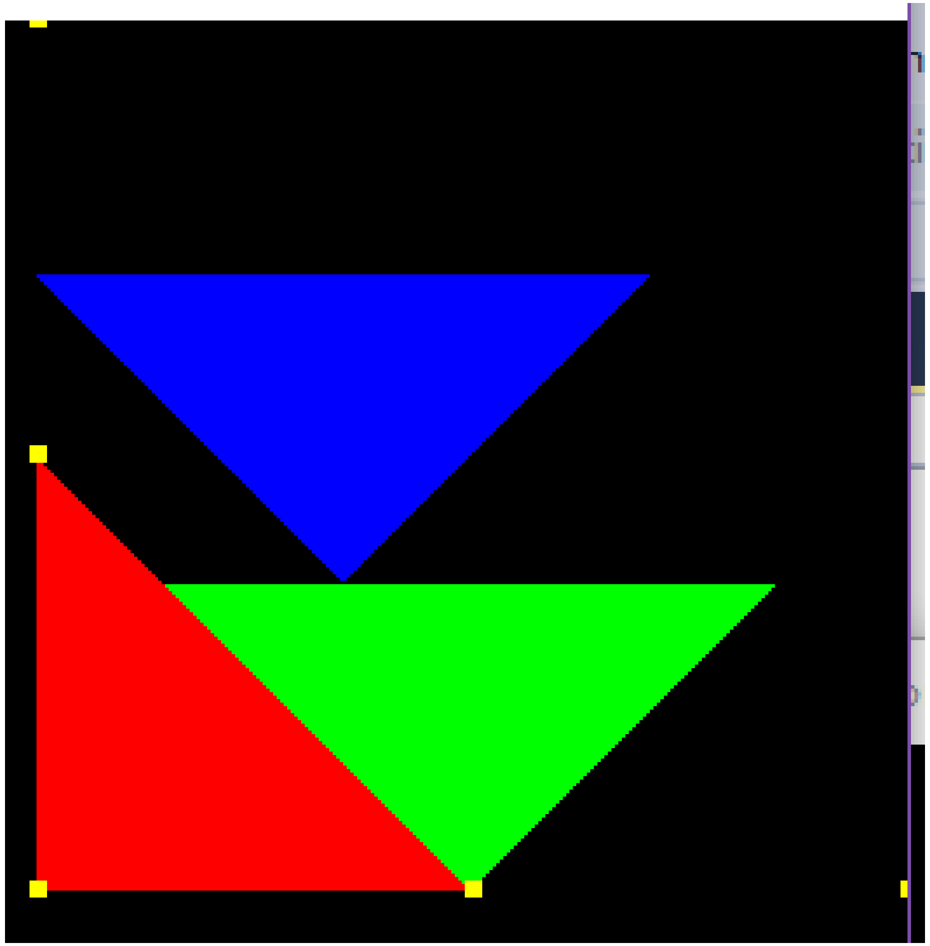
```
// draw triangle 3
glMatrixMode(GL_MODELVIEW);
glPushMatrix(); glLoadIdentity();
glRotatef(45, 0, 0, 1);
glTranslatef(1, 0, 0);
glBegin(GL_TRIANGLES);
glColor4f(0.0, 1.0, 0.0, 1.0);
glVertex3f(0.0, 0.0, -10.0);
glVertex3f(1.0, 0.0, -10.0);
glVertex3f(0.0, 1.0, -10.0);
glEnd();
glPopMatrix();
glutSwapBuffers();
}
```



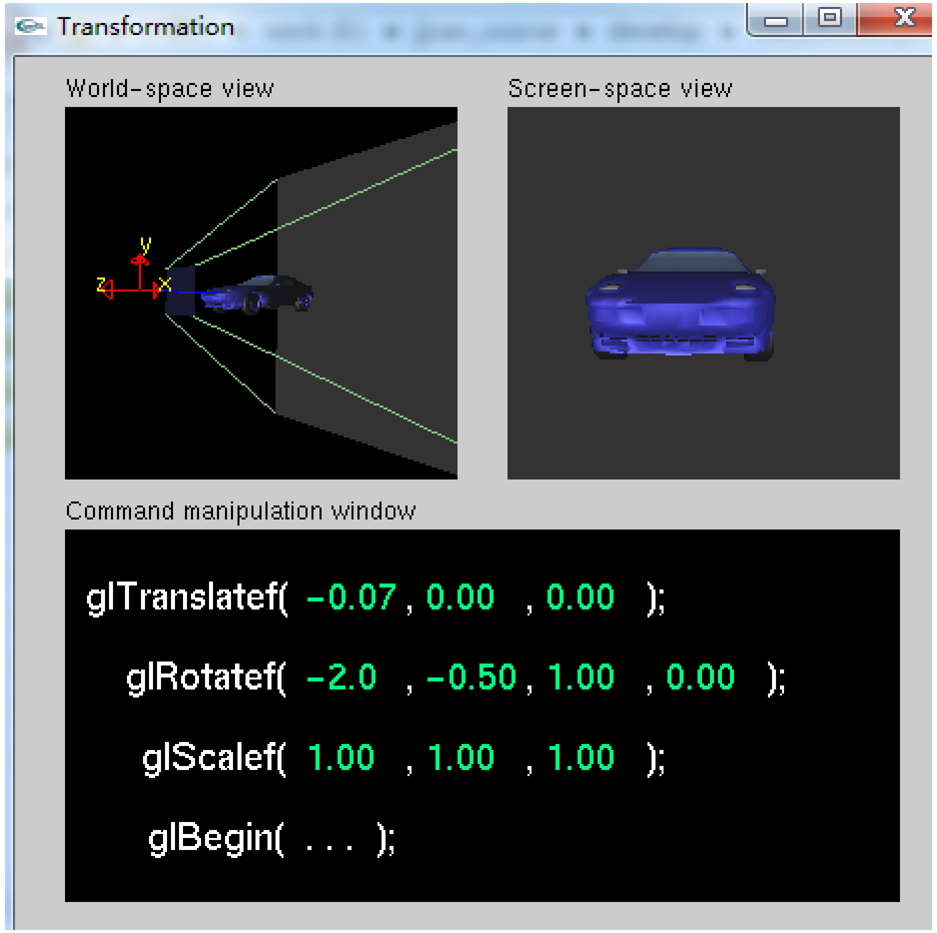
Example for Modeling Transformation 2

```
// draw triangle 2
glPushMatrix(); glLoadIdentity();
glTranslatef(1, 0, 0);
glRotatef(45, 0, 0, 1);
glColor4f(0.0, 1.0, 0.0, 1.0);
glBegin(GL_TRIANGLES); … glEnd();
glPopMatrix();

// draw triangle 3
glPushMatrix(); glLoadIdentity();
glRotatef(45, 0, 0, 1);
glTranslatef(1, 0, 0);
glColor4f(0.0, 1.0, 0.0, 1.0);
glBegin(GL_TRIANGLES);  … glEnd();
glPopMatrix();
```

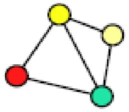


Modeling Transformations (cont)



Nate_Robins_tutorials: Transformation

Viewing transformation



Local (Object) Space ⇨ Modeling transformation ⇨ World Space

⇩

Viewing transformation ⇨ Eye Space ⇨ Projection transformation ⇨ Clip Space

⇩

Perspective divide ⇨ NDC space ⇨ Viewport mapping ⇨ Screen space

Normalized **D**evice **C**oordinates

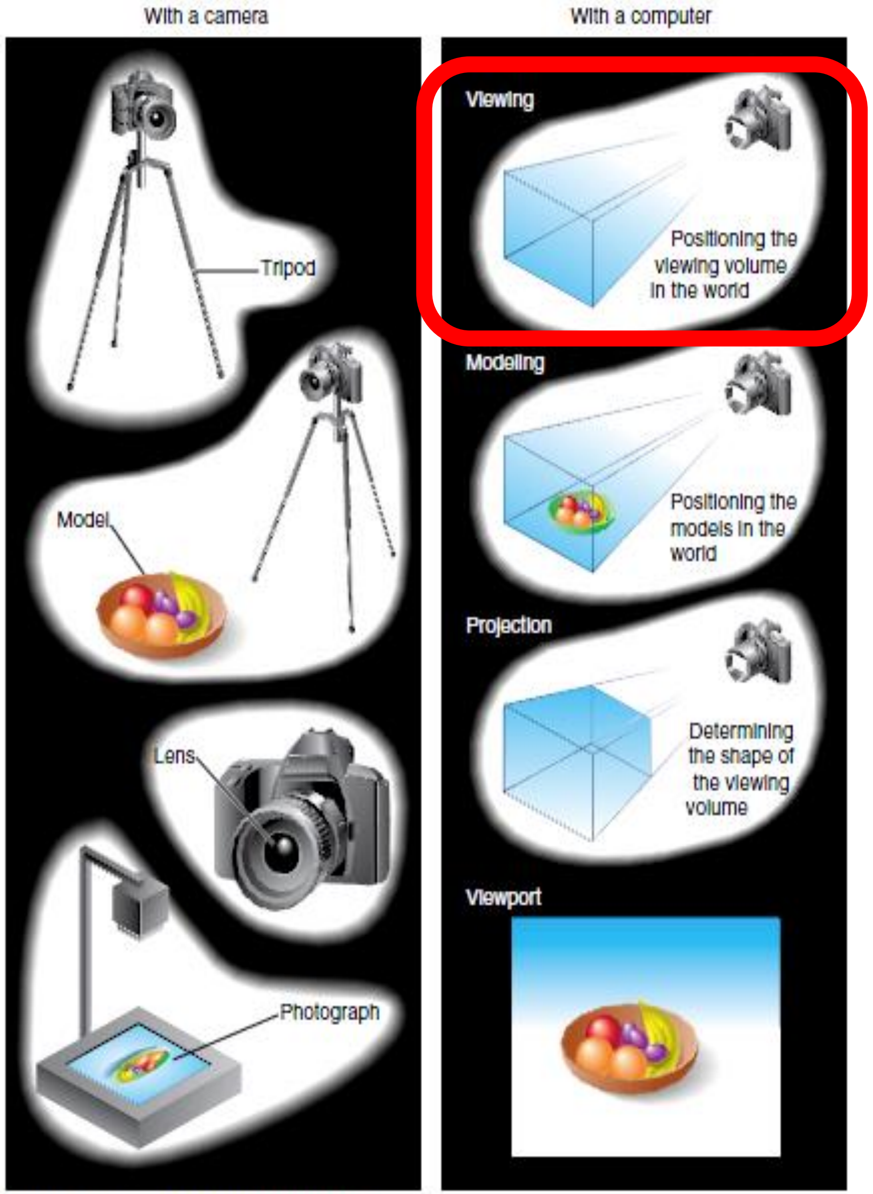


Figure 3-1 The Camera Analogy

Viewing Transformation

- Convert from WCS to the camera (eye) coordinate sys

- The camera position is the origin initially.
- The objs are also in the origin mostly. Or have been placed well in WCS

- Anyway, we need move the camera to see what we wish to see (may see nothing using default camera/viewing transformation)

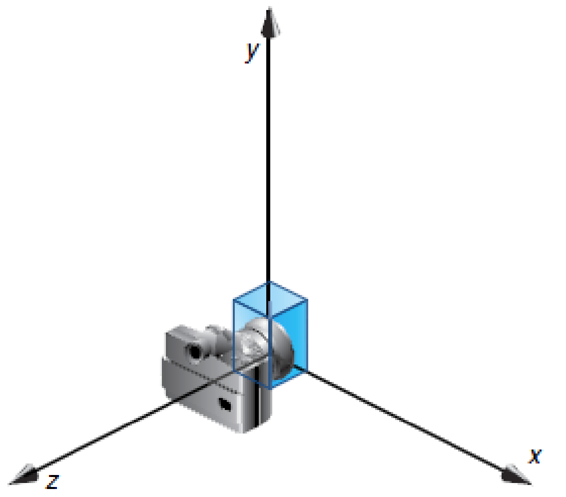
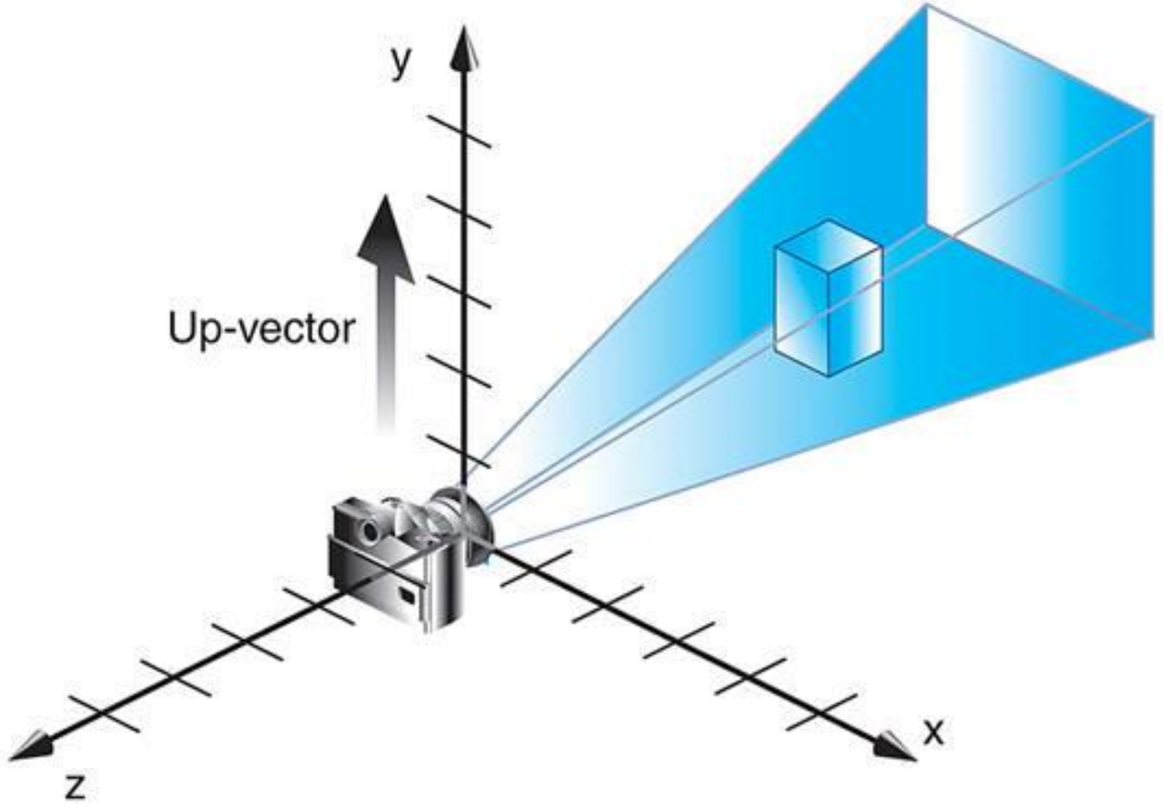
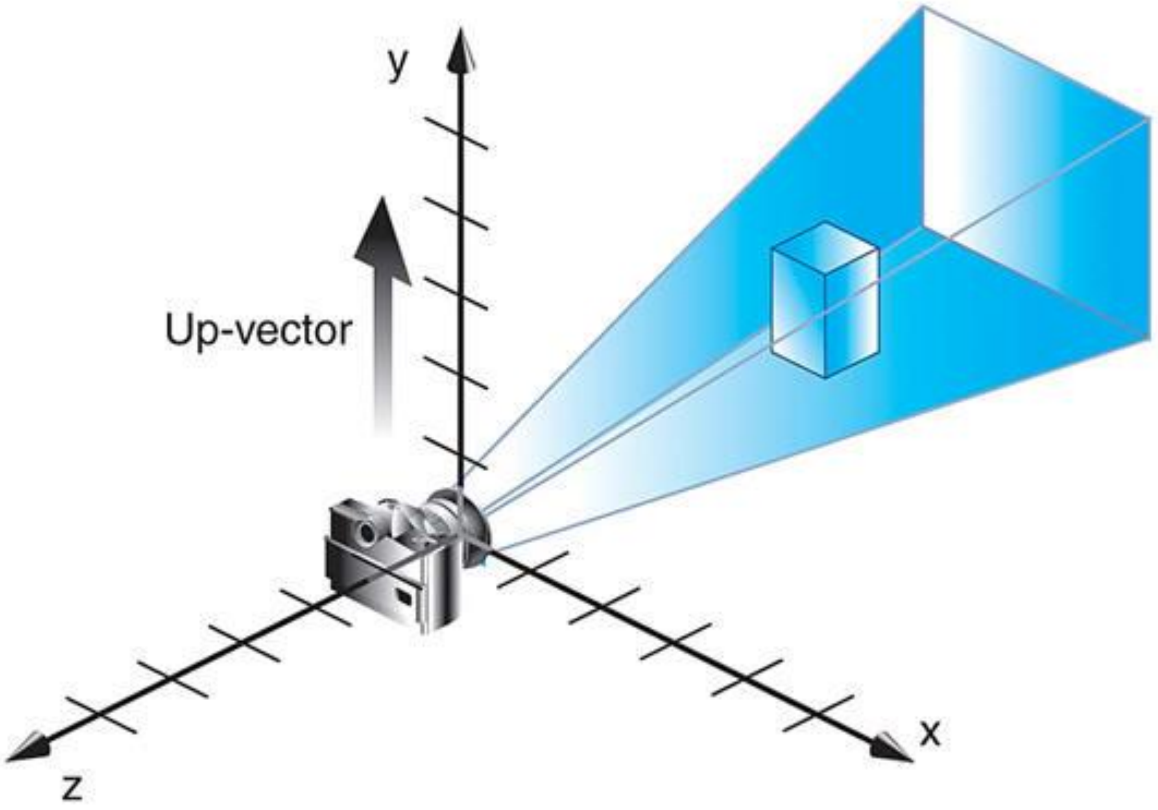


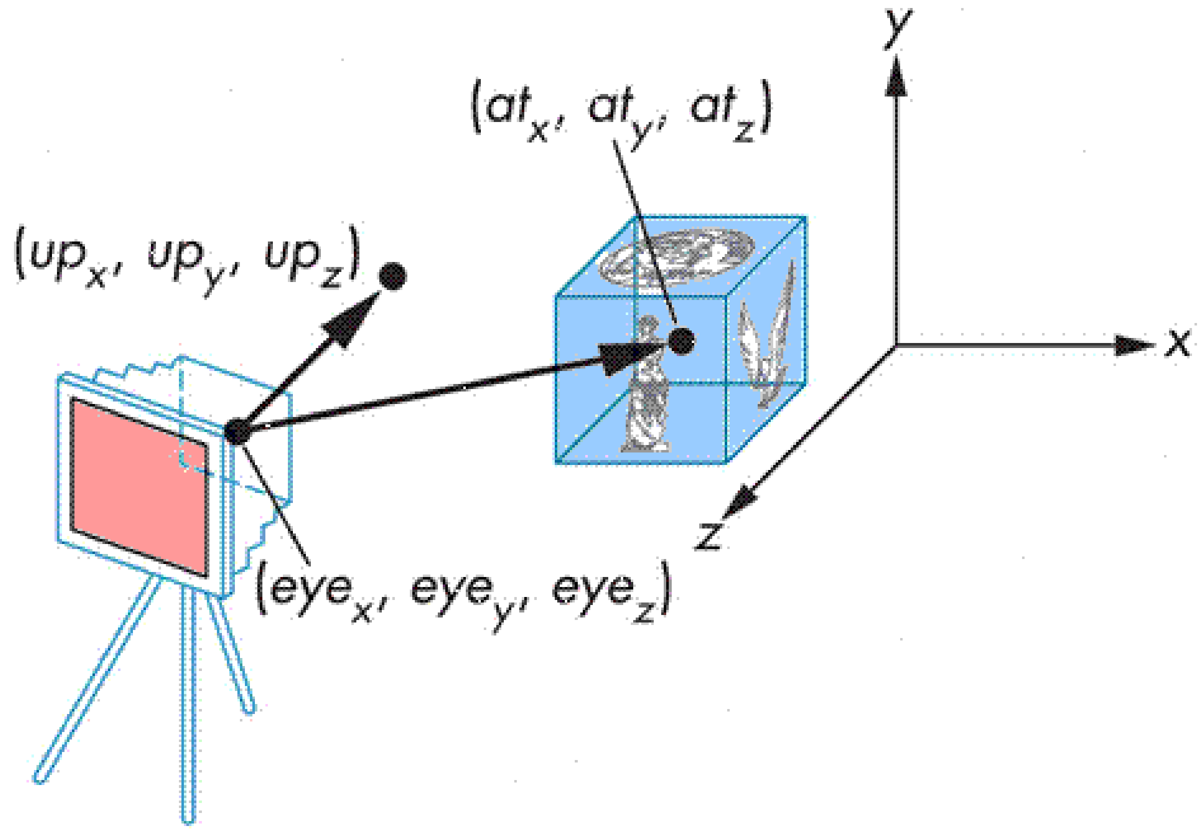
Figure 3-9 Object and Viewpoint at the Origin



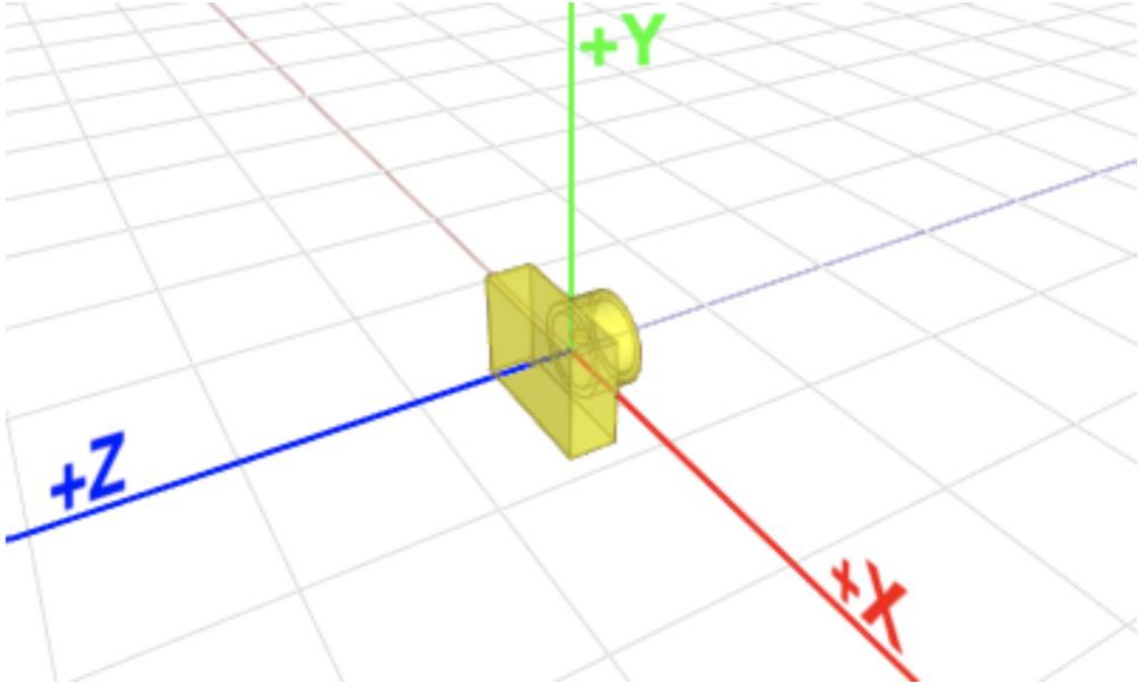
Viewing Transformation

- void **gluLookAt**(*eyeX*, *eyeY*, *eyeZ*, *centerX*, *centerY*, *centerZ*, *upX*, *upY*, *upZ*);

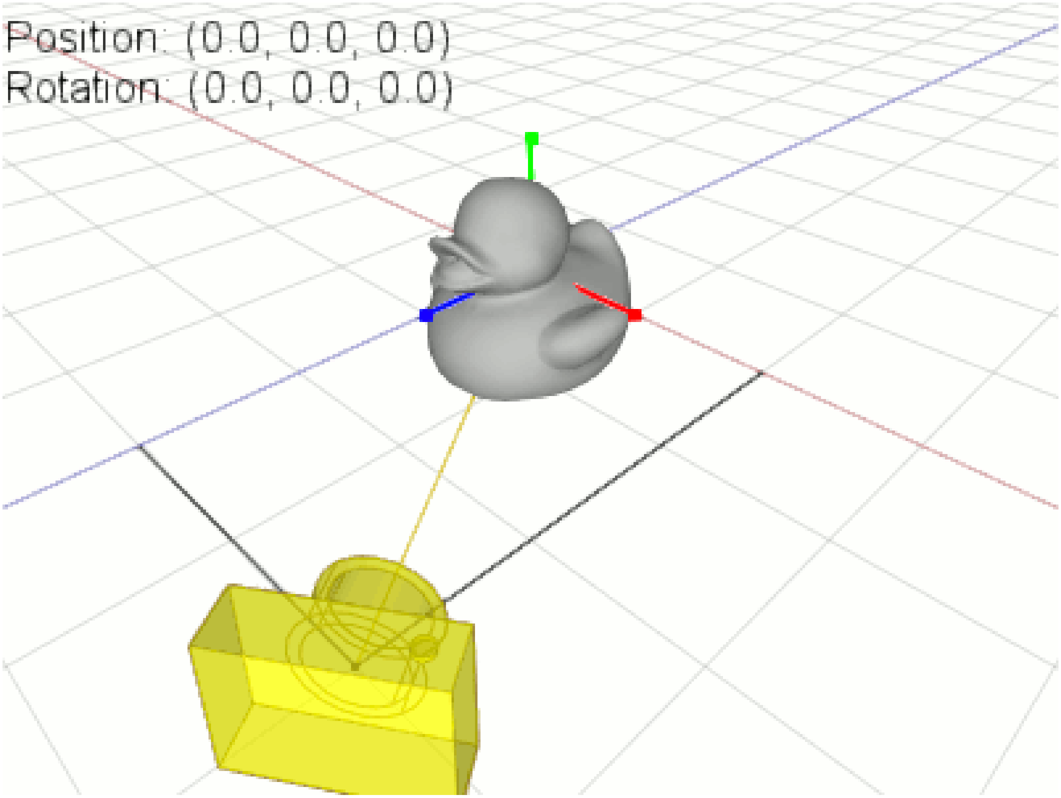




void **gluLookAt**(*eyeX*, *eyeY*, *eyeZ*, *centerX*, *centerY*, *centerZ*, *upX*, *upY*, *upZ*);



OpenGL camera is always at origin and facing to -Z in eye space



$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modelView} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix} = M_{view} \cdot M_{model} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix}$$

Example: modeling + viewing transformation

- With all this, we can give an outline for a typical display routine for drawing an image of a 3D scene with OpenGL 1.1:

```
// possibly set clear color here, if not set elsewhere
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

// possibly set up the projection here, if not done elsewhere
glMatrixMode( GL_MODELVIEW ); glLoadIdentity();
gluLookAt( eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ ); // Viewing transform

glRotatef(45, 0, 0, 1);
glTranslatef(1, 0, 0);
glBegin(GL_TRIANGLES);
glColor4f(0.0, 1.0, 0.0, 1.0);glVertex3f(0.0, 0.0, -10.0);
glVertex3f(1.0, 0.0, -10.0);glVertex3f(0.0, 1.0, -10.0);
glEnd();
…
```

Where are we drawing actually?

We are drawing in the eye coord

Implementing the Look-At Function

- gluLookAt(0.0, 0.0, 2.0,   0.0, 0.0, 0.0,   0.0, 1.0, 0.0);对应的M

1， 0， 0， 0；

0， 1， 0， 0；

0， 0， 1， -2；

0， 0， 0， 1；

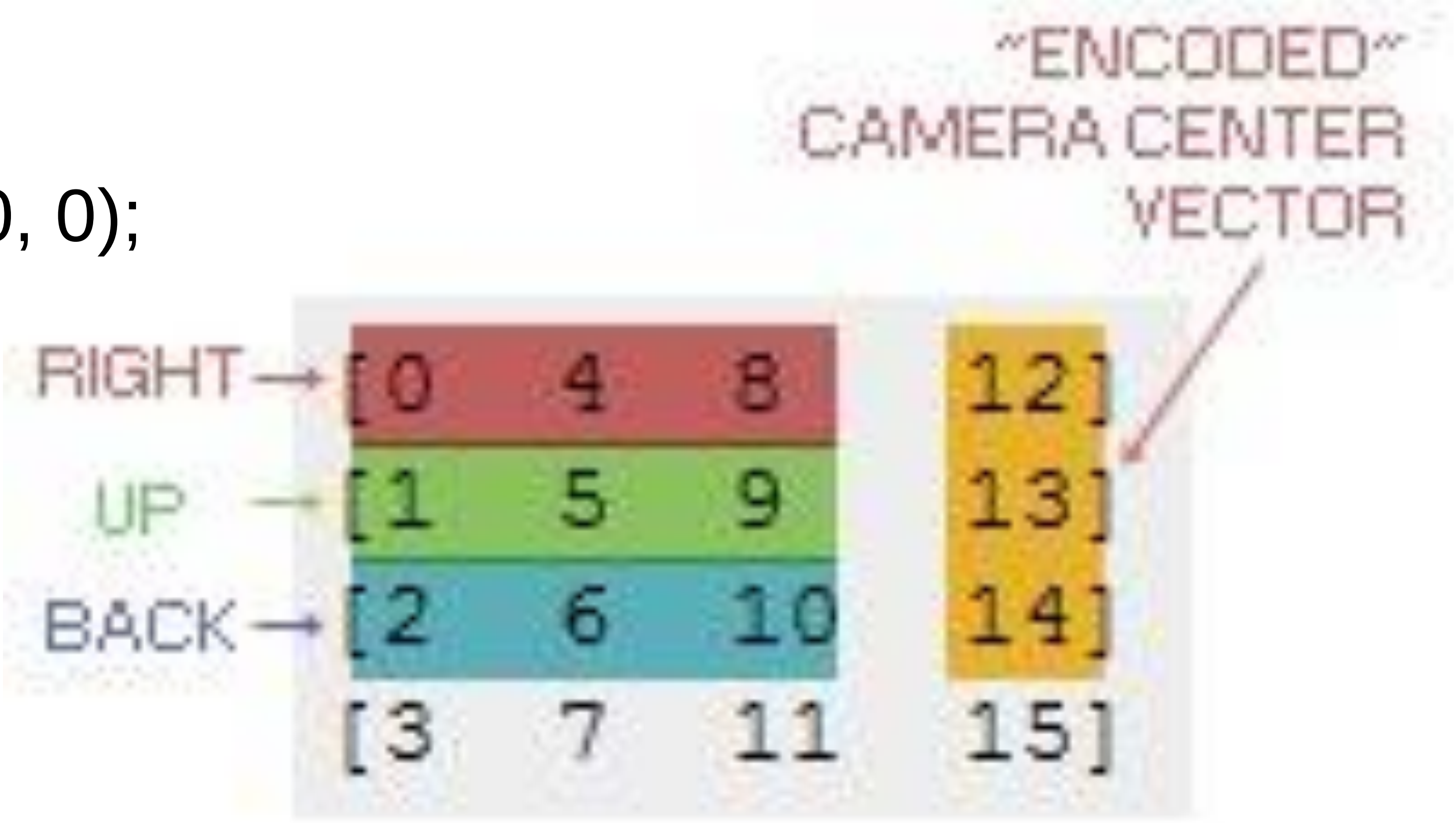
- gluLookAt(0, 0, 2, 0, 0, 0, 1, 0, 0);

0， -1， 0， 0；

1， 0， 0， 0；

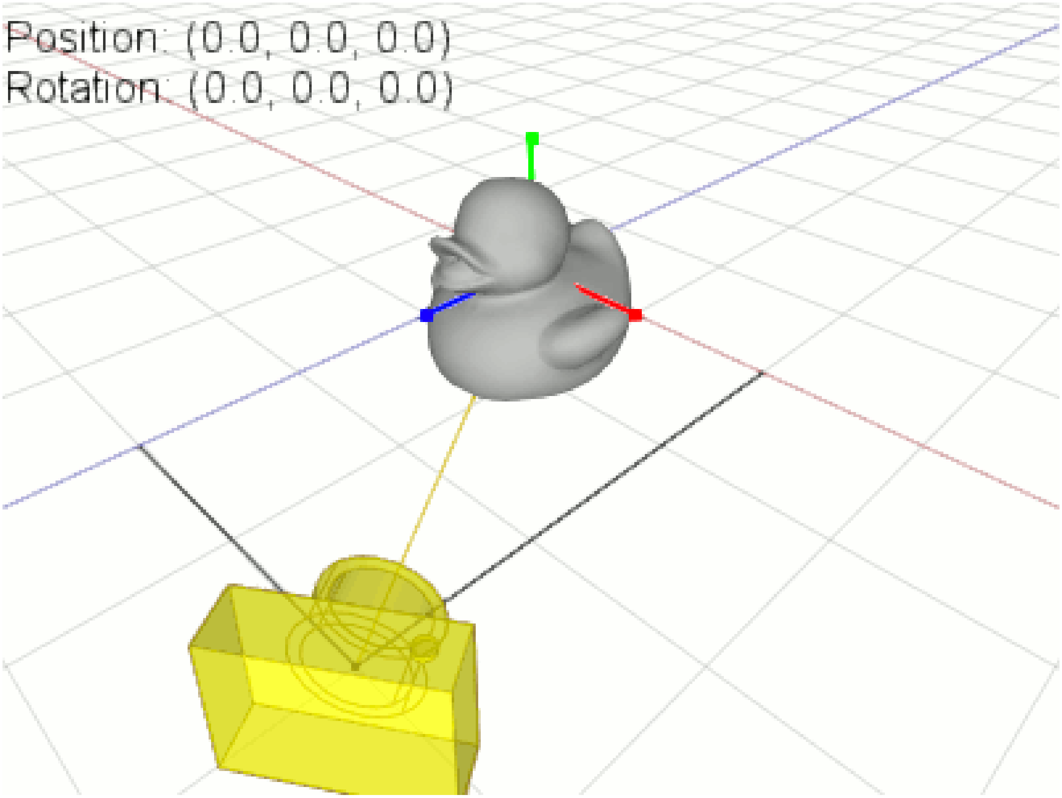
0， 0， 1， -2；

0， 0， 0， 1；



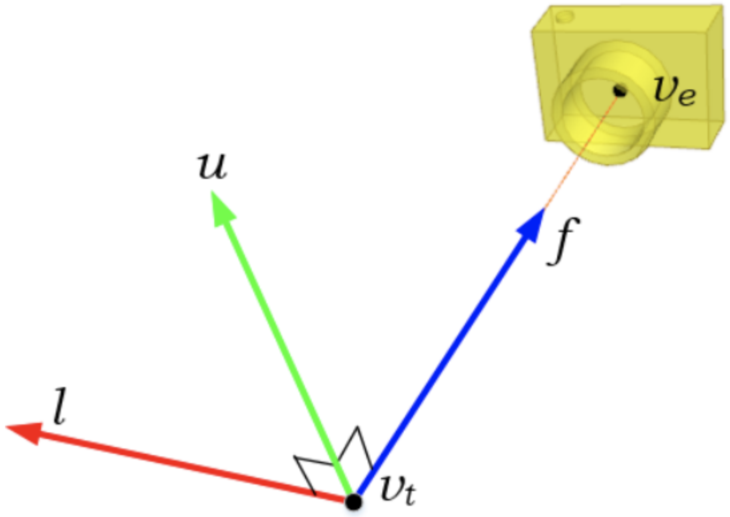
Implementing the Look-At Function

$$M_{\text{view}} = M_{\text{R}} M_{\text{T}} = \begin{pmatrix} r_0 & r_4 & r_8 & 0 \\ r_1 & r_5 & r_9 & 0 \\ r_2 & r_6 & r_{10} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} r_0 & r_4 & r_8 & r_0 t_x + r_4 t_y + r_8 t_z \\ r_1 & r_5 & r_9 & r_1 t_x + r_5 t_y + r_9 t_z \\ r_2 & r_6 & r_{10} & r_2 t_x + r_6 t_y + r_{10} t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$M_{\text{T}} = \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Implementing the Look-At Function



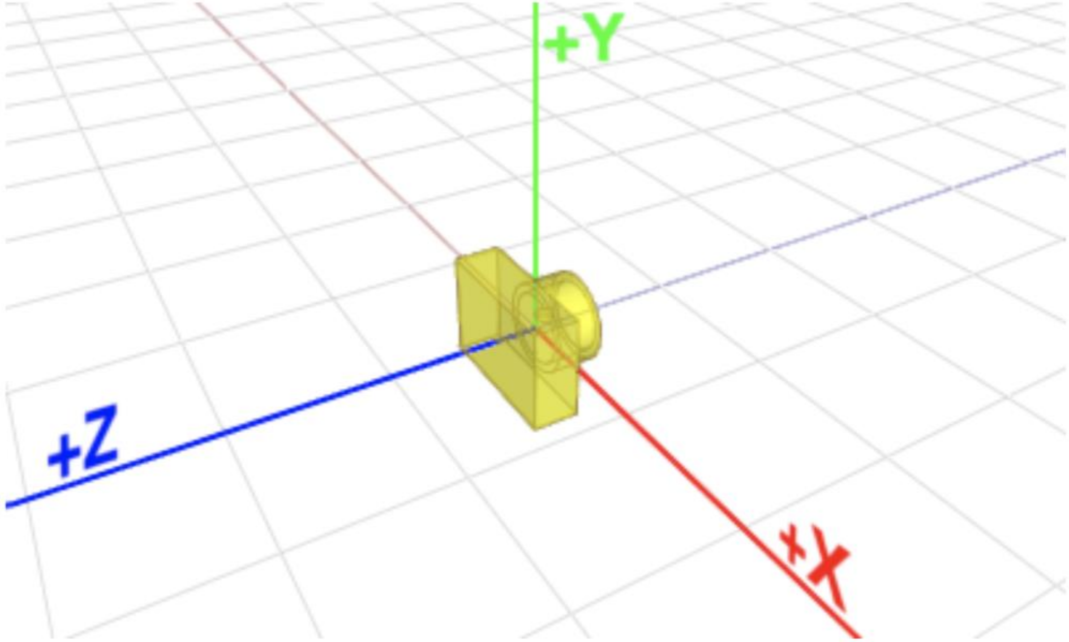
Left, Up and Forward vectors from target to eye

$$f = \frac{v_e - v_t}{\|v_e - v_t\|} \qquad left = up \times f \qquad u = f \times l$$

$$l = \frac{left}{\|left\|}$$

$$M_{\mathrm{R}} = \begin{pmatrix} l_x & u_x & f_x & 0 \\ l_y & u_y & f_y & 0 \\ l_x & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} l_x & u_x & f_x & 0 \\ l_y & u_y & f_y & 0 \\ l_x & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} l_x & l_y & l_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

void **gluLookAt**(*eyeX*, *eyeY*, *eyeZ*, *centerX*, *centerY*, *centerZ*, *upX*, *upY*, *upZ*);

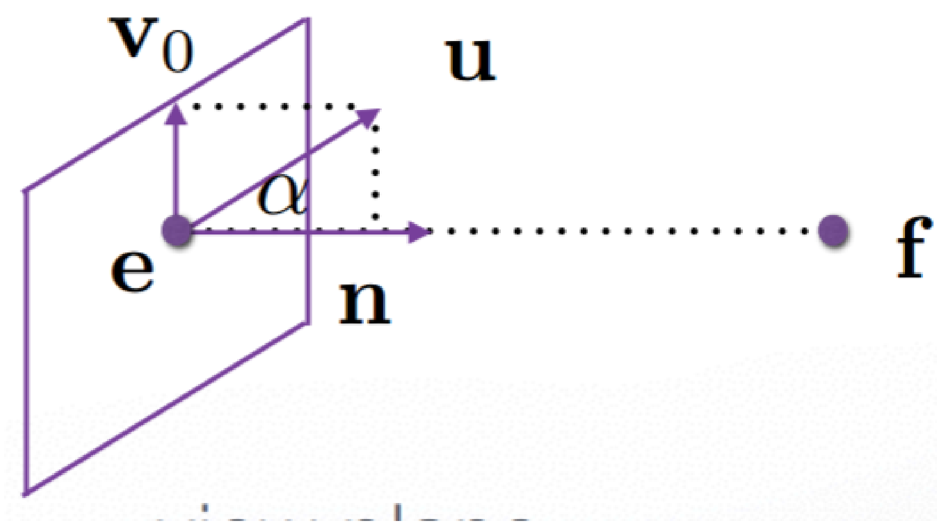


OpenGL camera is always at origin and facing to -Z in eye space

$$M_{\text{view}} = M_{\text{R}} M_{\text{T}} = \begin{pmatrix} l_x & l_y & l_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} l_x & l_y & l_z & -l_x x_e - l_y y_e - l_z z_e \\ u_x & u_y & u_z & -u_x x_e - u_y y_e - u_z z_e \\ f_x & f_y & f_z & -f_x x_e - f_y y_e - f_z z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Camera Frame: n v w – use just one cross

- gluLookAt(ex, ey, ez, fx, fy, fz, ux, uy, uz);
- n = (f-e)/||f-e|| is unit normal to view plane

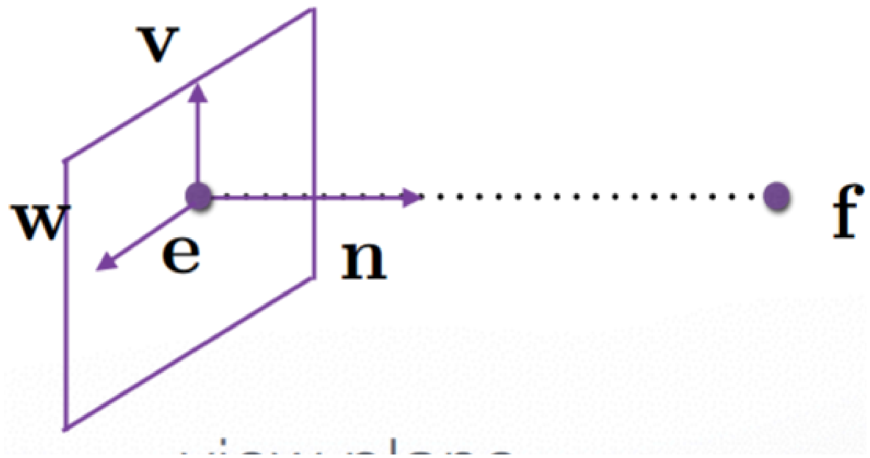


$$\alpha = \mathbf{u}^\top \mathbf{n}/\|\mathbf{n}\| = \mathbf{u}^\top \mathbf{n}$$

$$\mathbf{v}_0 = \mathbf{u} - \alpha\mathbf{n}$$

$$\mathbf{v} = \mathbf{v}_0/\|\mathbf{v}_0\|$$

- w = cross(n,v); w = w/||w||

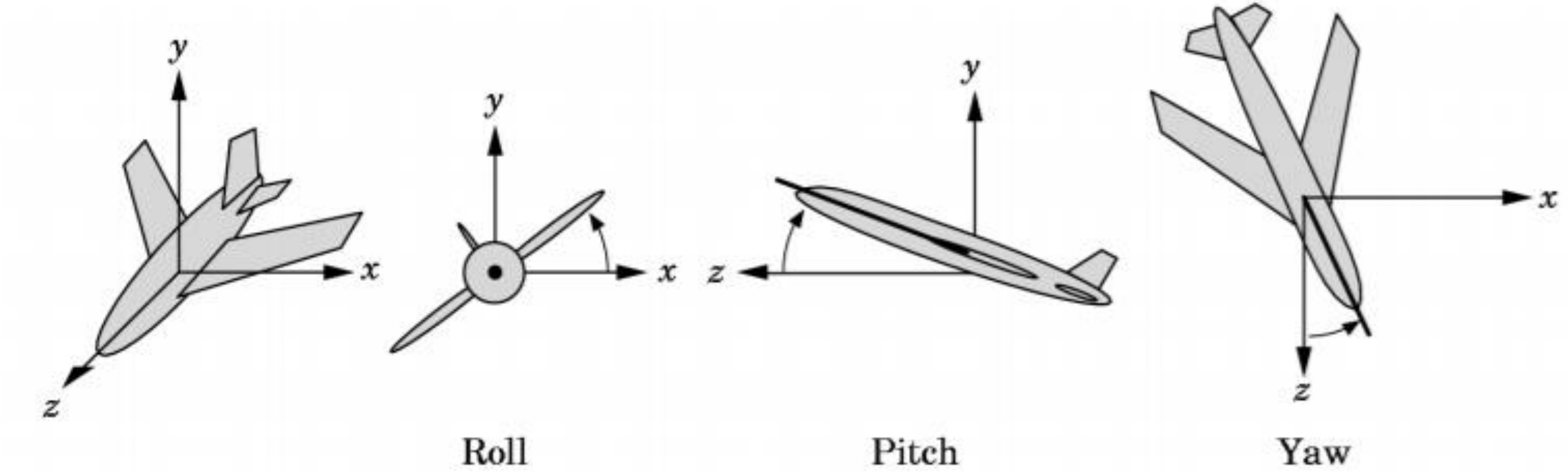


- w v –n is right-handed

gluLookAt does not require that the up vector you provide be perpendicular to the direction you're looking.

Other Viewing Functions

- A pilot wants:



- Polar coord

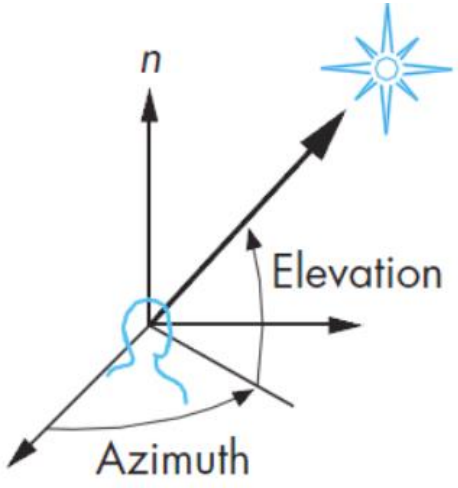
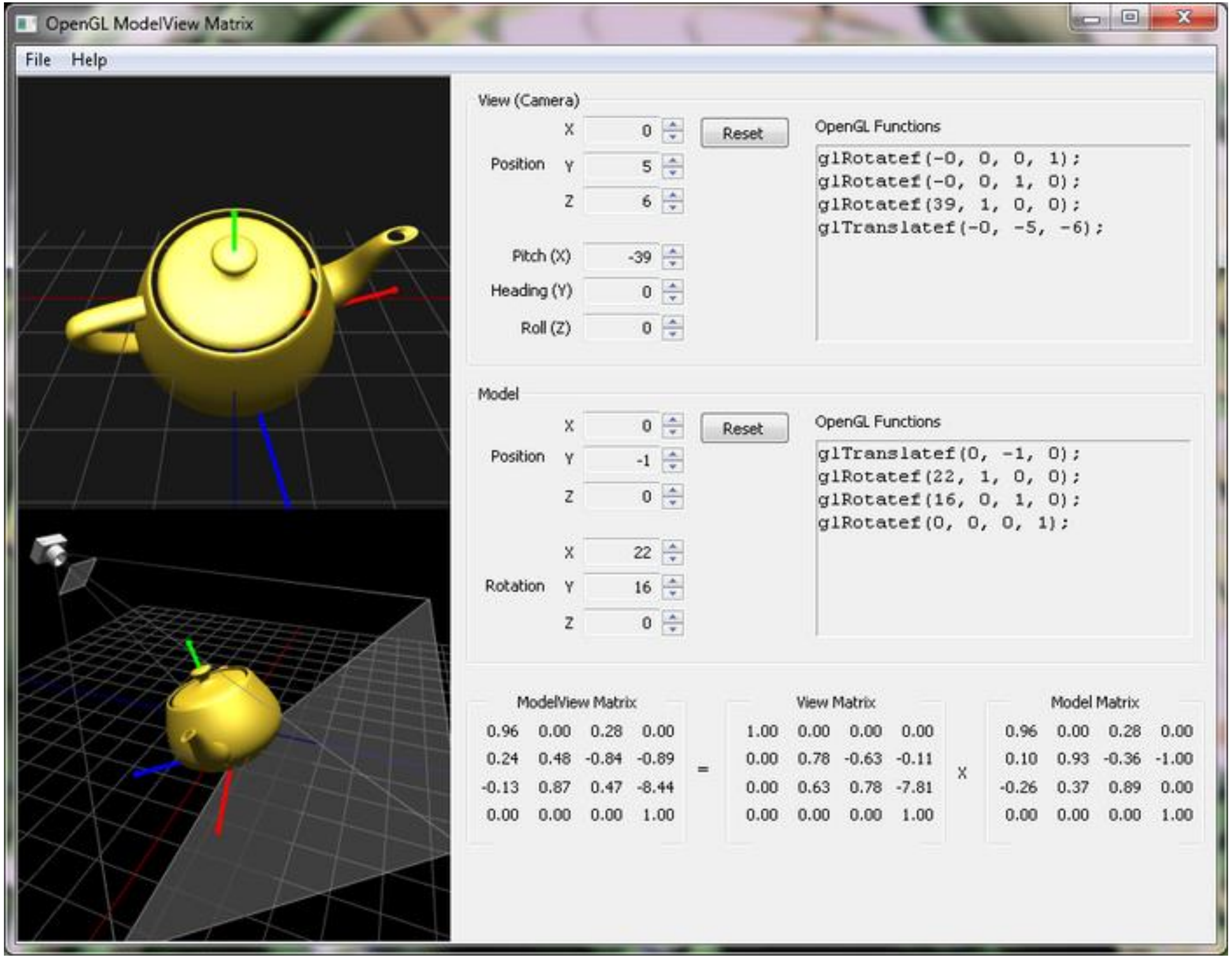


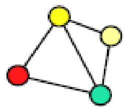
FIGURE 4.20 Elevation and azimuth.

Example: ModelView Matrix



- http://www.songho.ca/opengl/gl_transform.html#example1

Transformation Pipeline



Local (Object) Space ⇨ Modeling transformation ⇨ World Space ⇩

Viewing transformation ⇨ Eye Space ⇨ Projection transformation ⇨ Clip Space ⇩

Perspective divide ⇨ NDC space ⇨ Viewport mapping ⇨ Screen space

Normalized **D**evice **C**oordinates

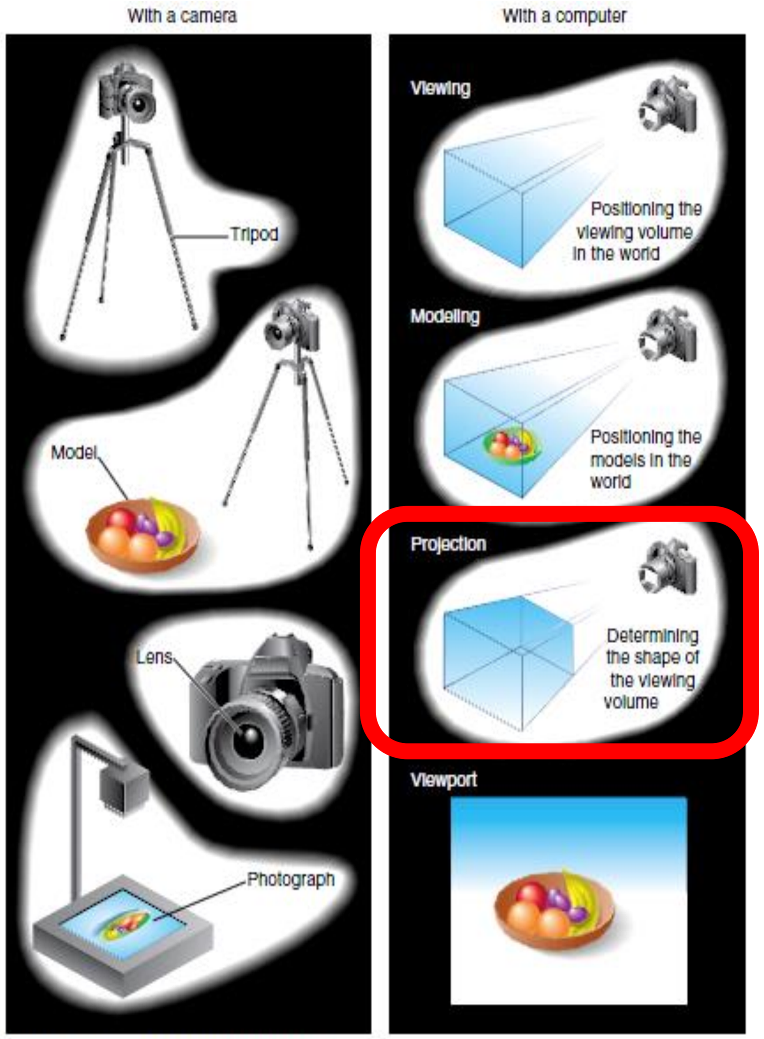


Figure 3-1 The Camera Analogy

Perspective projection

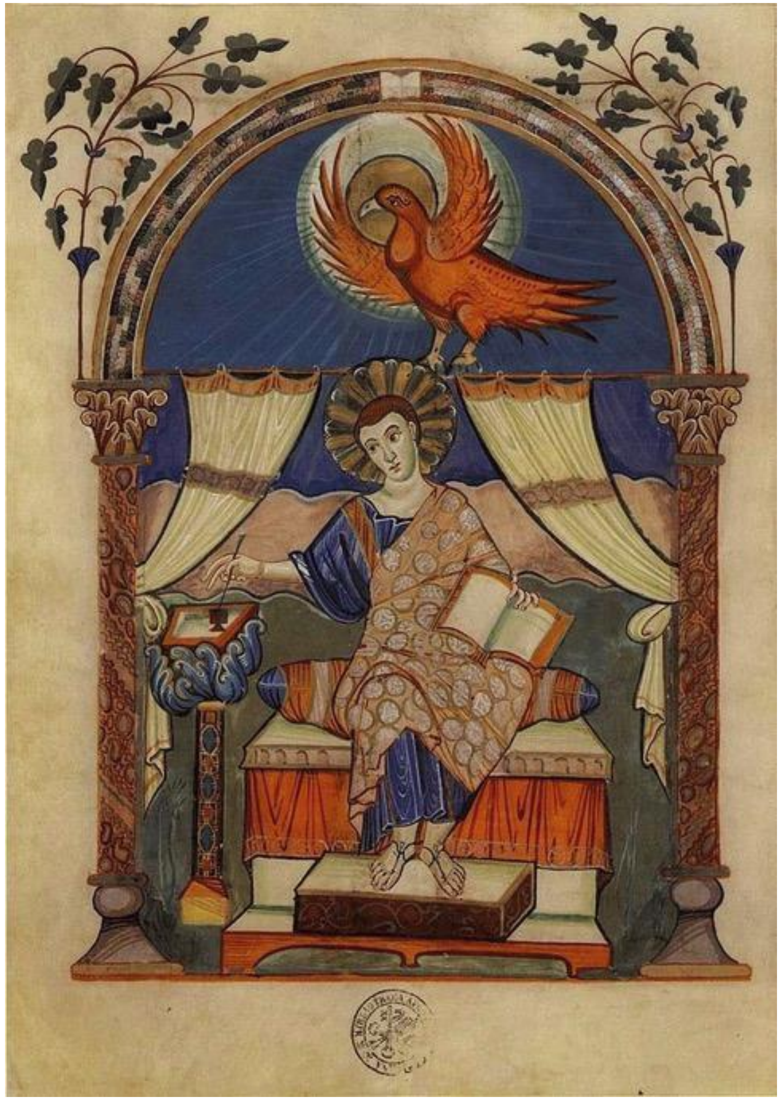
Rudimentary perspective in cave drawings

Lascaux, France source: Wikipedia

Painting in middle ages: incorrect perspective

- Art in the service of religion
- Perspective abandoned or forgotten

Ottonian manuscript, ca. 1000

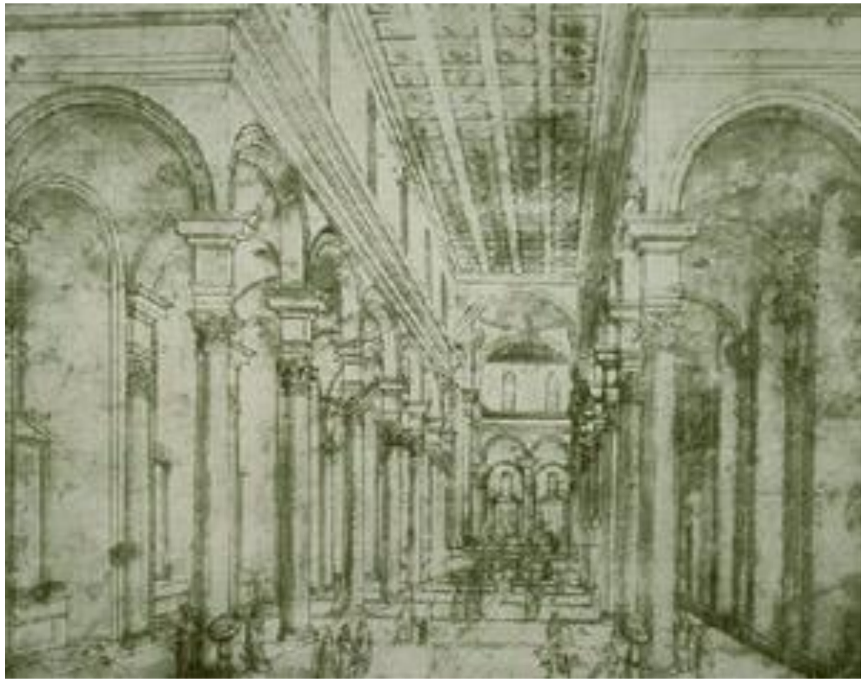


8-9th century painting

Renaissance

- Rediscovery, systematic study of perspective

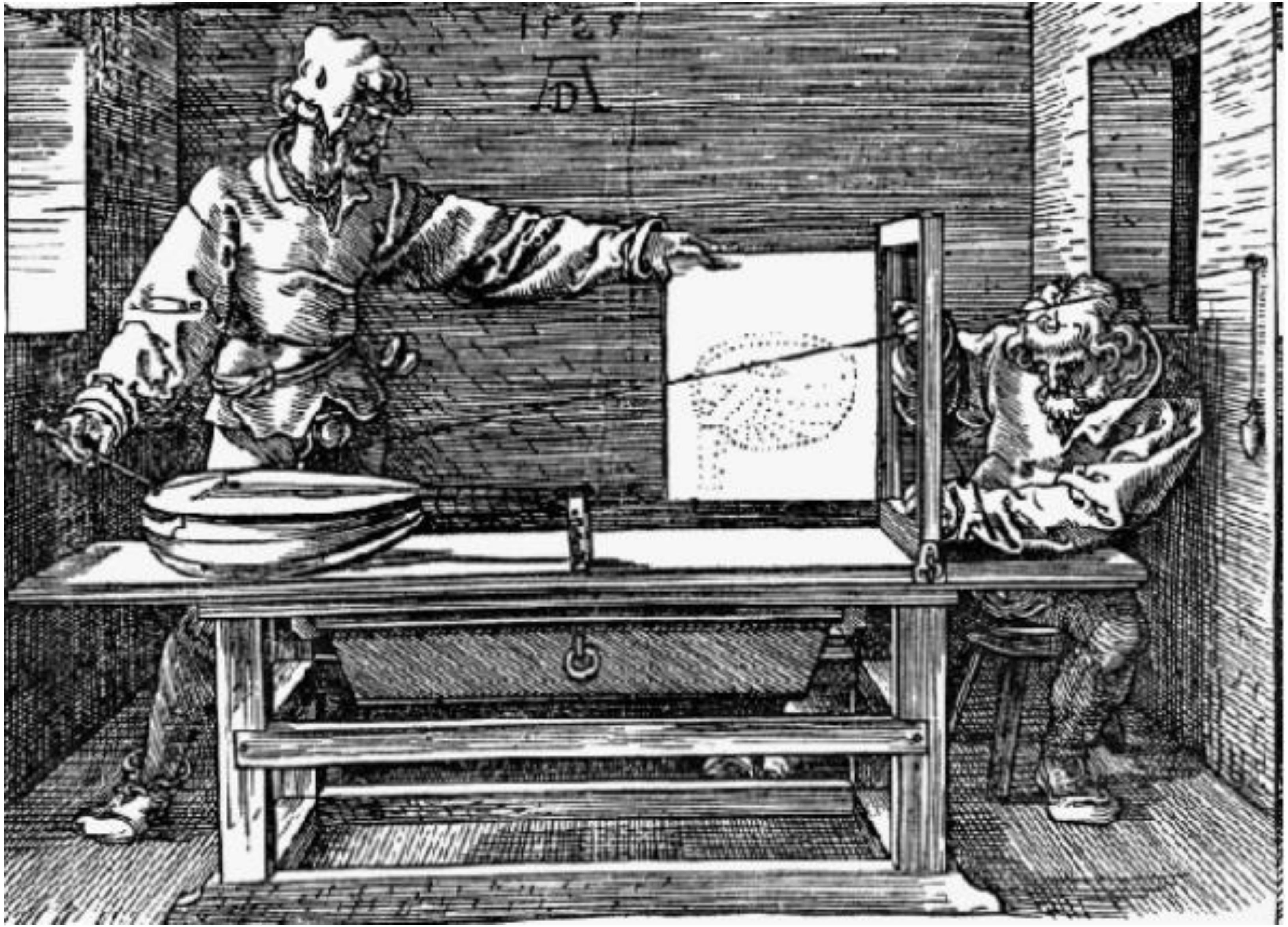
Filippo Brunelleschi Florence, 1415



Brunelleschi, elevation of Santo Spirito, 1434-83, Florence

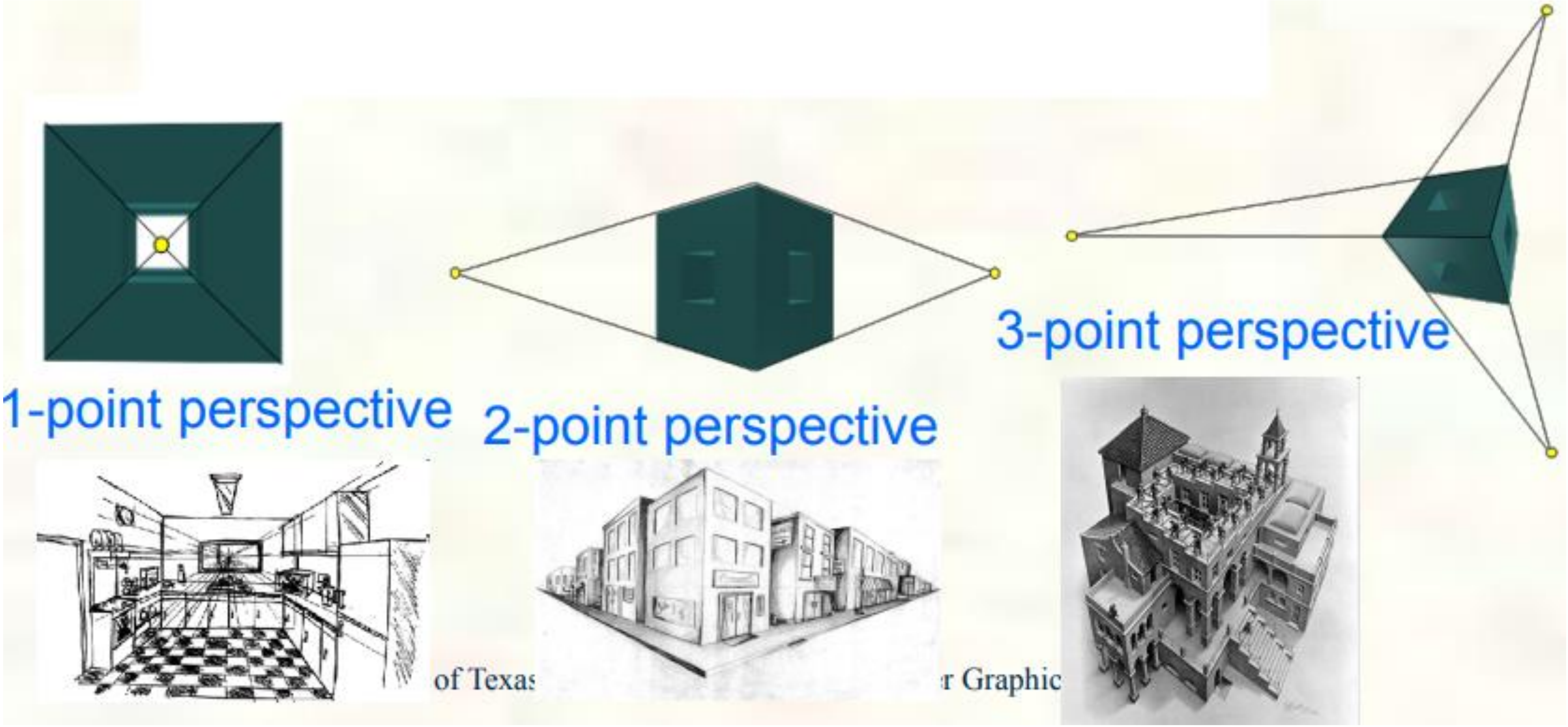
Masaccio – The Tribute Money c.1426-27
Fresco, The Brancacci Chapel, Florence

Humanist Analysis of Perspective



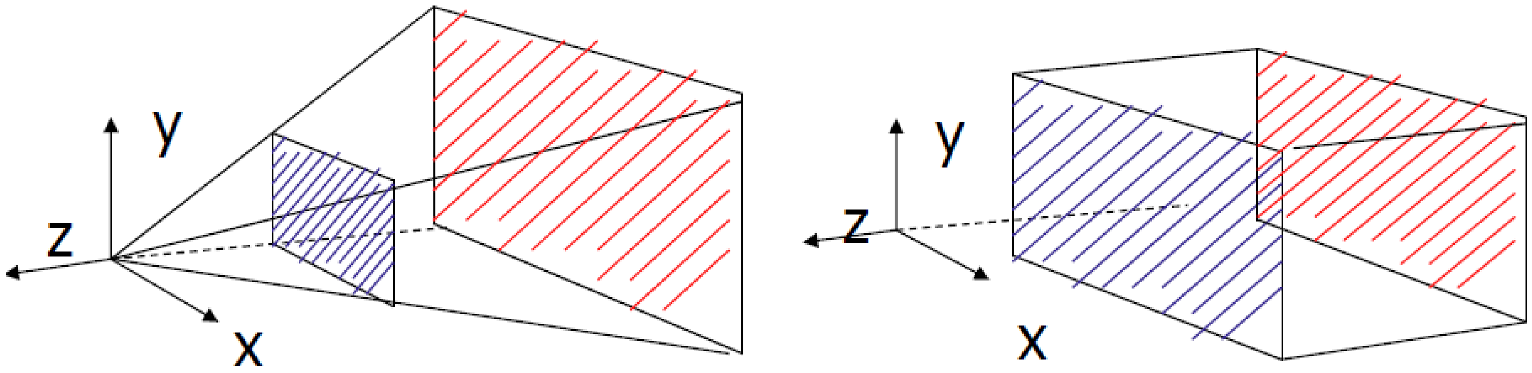
[Albrecht Dürer, 1471]

1-, 2-, and 3-point Perspective



Projection Transformation

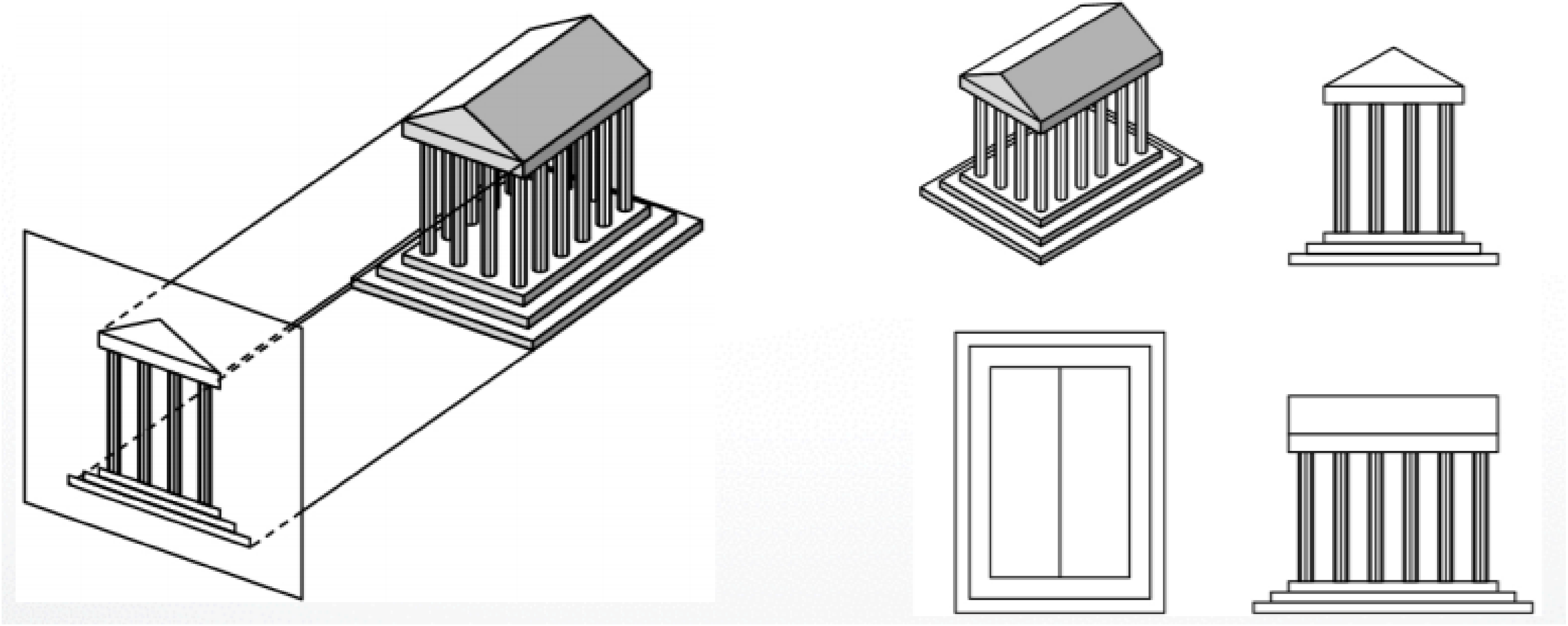
- Specifying PT is like choosing a **lens for a camera**
- The purpose of PT is to define a **viewing volume**, which is used in two ways.
  - The viewing volume determines **how an object is projected** onto the screen (that is, by using a perspective or an orthographic projection), and
  - Defines **which objects or portions of objects are clipped out** of the final image
- Need to establish the appropriate mode for constructing the viewing transformation, or in other words select the projection mode
  - **glMatrixMode(GL_PROJECTION);**
- This designates the projection matrix as the current matrix, which is originally set to the identity matrix



Perspective: **gluPerspective()**   Parallel: **glOrtho()**

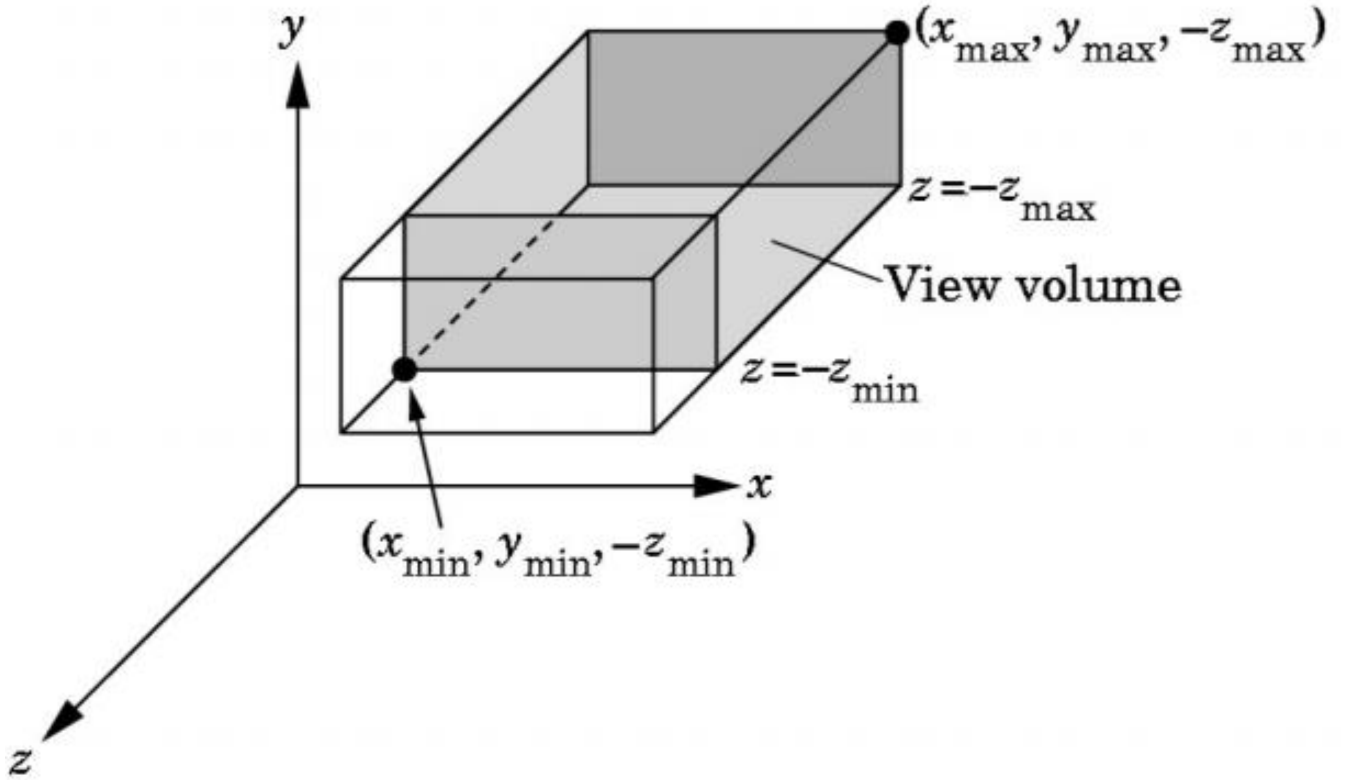
Orthographic Projection

- A special kind of parallel projection:
- projectors perpendicular to projection plane
- Simple, but not realistic
- Used in blueprints (multiview projections)



Orthographic Projection

- void **glOrtho** (left, right, bottom, top, near, far);

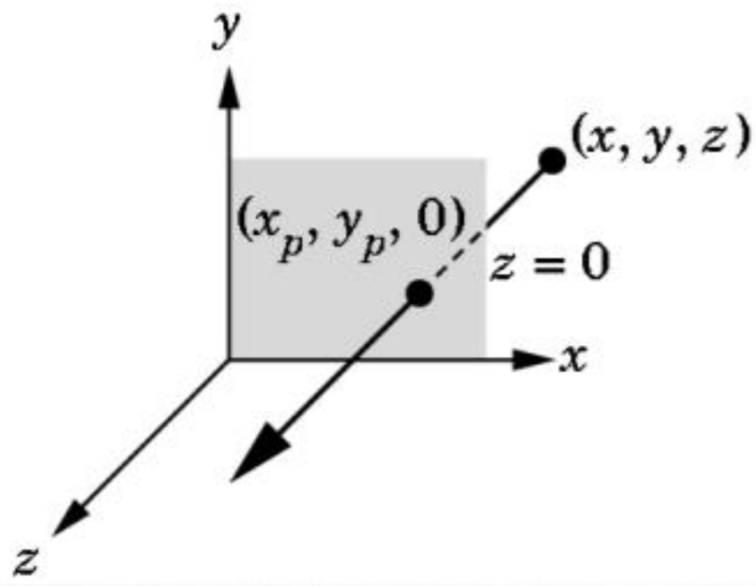


$$z_{\text{min}} = \text{near}, z_{\text{max}} = \text{far}$$

Maps (projects) everything in the visible volume into a canonical view volume

Orthographic Projection Matrix

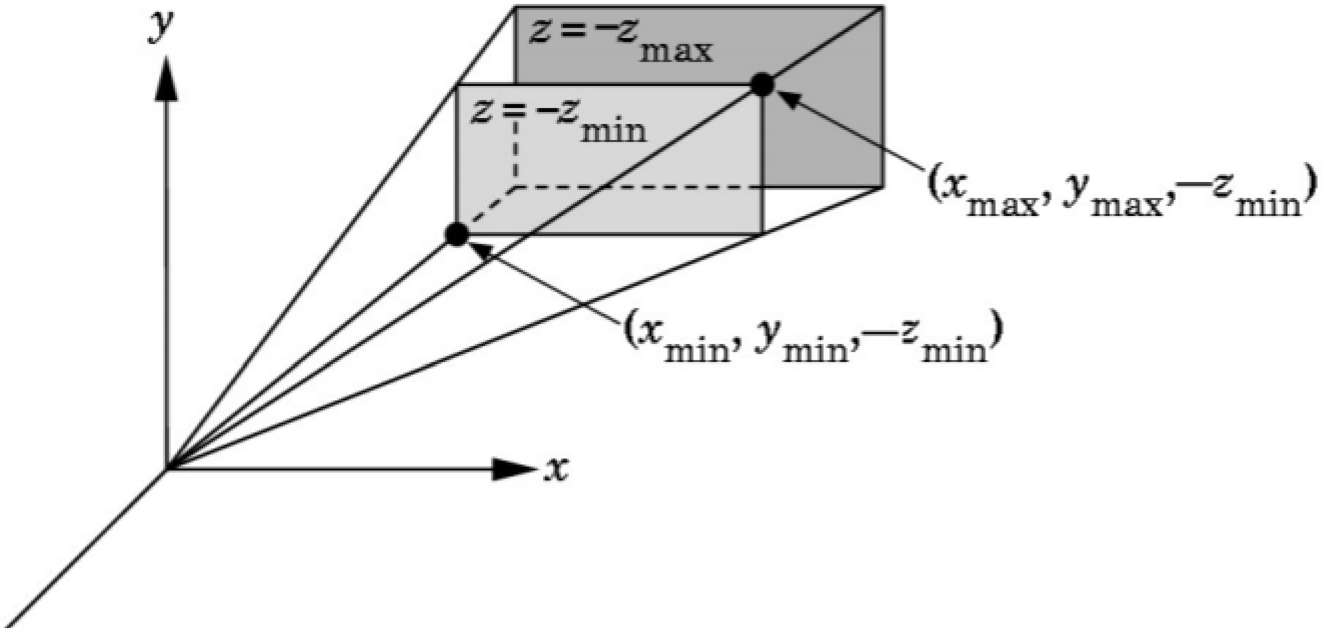
- Project onto $z = 0$
- $x_p = x$, $y_p = y$, $z_p = 0$
- In homogenous coordinates



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

Perspective Projection

glFrustum(xmin, xmax, ymin, ymax, N, F); N = near plane, F = far plane

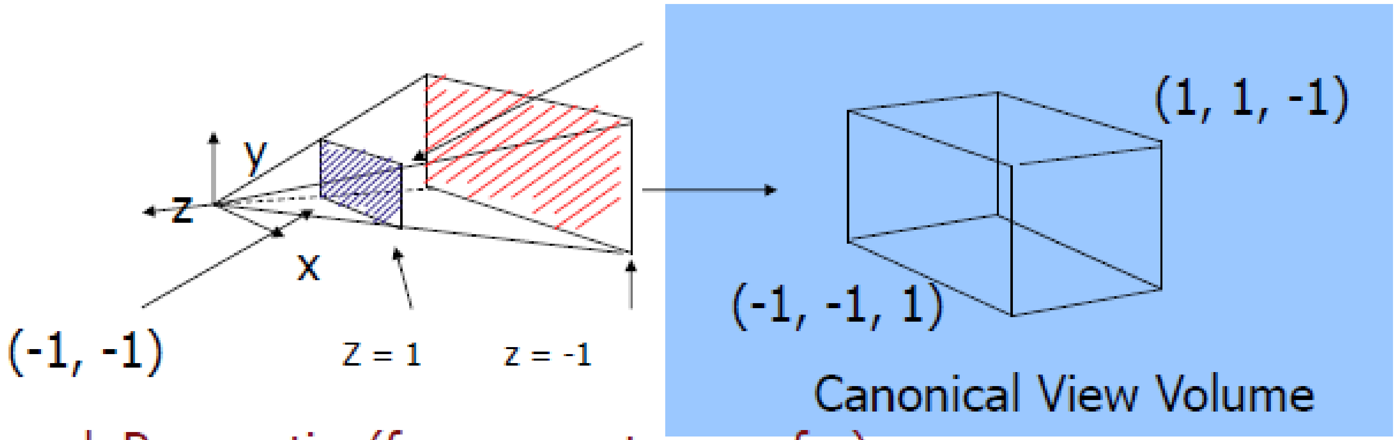


Projection Matrix

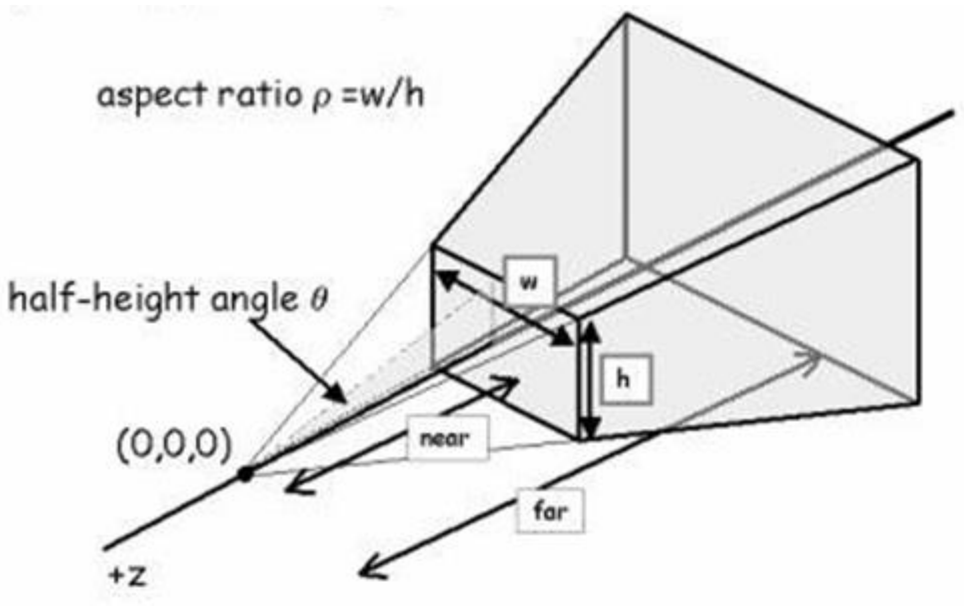
$$\begin{array}{c} x' \\ y' \\ z' \\ w' \end{array} = \left|\begin{array}{cccc} 2N/(xmax-xmin) & 0 & (xmax+xmin)/(xmax-xmin) & 0 \\ 0 & 2N/(ymax-ymin) & (ymax+ymin)/(ymax-ymin) & 0 \\ 0 & 0 & -(F+N)/(F-N) & -2F*N/(F-N) \\ 0 & 0 & -1 & 0 \end{array}\right| \begin{array}{c} x \\ y \\ z \\ 1 \end{array}$$

gluPerspective

- glFrustum() isn't intuitive to use so can use **gluPerspective** to specify
  - Fovy: the angle of the field of view in the y direction
  - Aspect: the aspect ratio of the width to height (x/y)
  - Near & far: distance between the viewpoint and the near and far clipping planes
- Note that gluPerspective() is limited to creating frustums that are symmetric in both the x- and y-axes along the line of sight

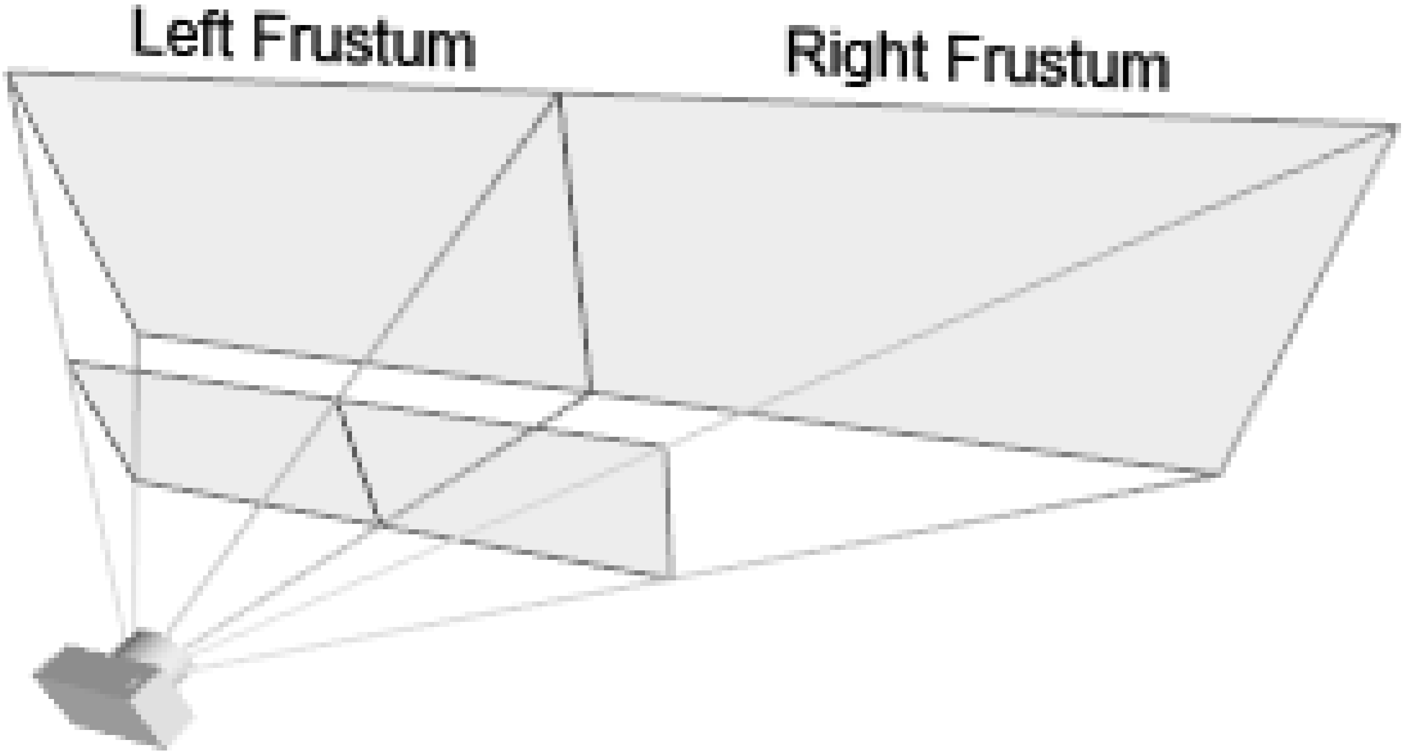


gluPerspective(fovy, aspect, near, far)



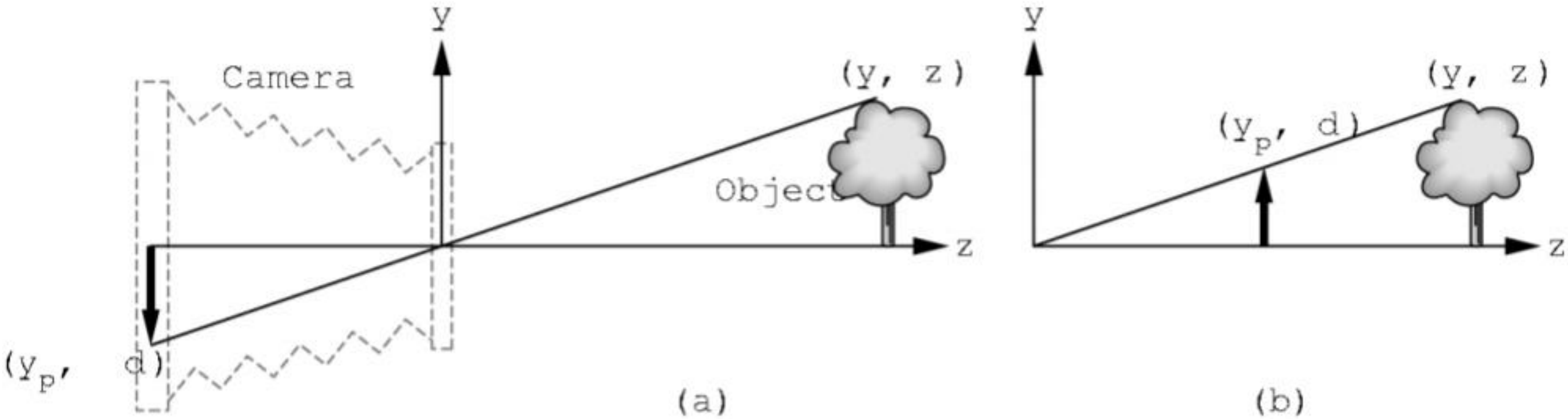
Maps (projects) everything in the visible volume into
a canonical view volume

Render a wide scene into 2 adjoining screens



- have to use glFrustum() directly if you need to create a non-ymmetrical viewing volume

Perspective Viewing Mathematically



- d = focal length
- $y/z = y_p/d$ so $y_p = y/(z/d) = yd/z$
- Note that y_p is non-linear in the depth z!

homogeneous coordinates

Perspective projection is not affine:

$$M\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$ has no solution for M

Idea: exploit homogeneous coordinates

$$p = w\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$ for arbitrary $w \neq 0$

Perspective Projection Matrix

- Use multiple of point

$$(z/d)\begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix}$$

$$M\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

No solution

- Solve

$$M\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} \quad \text{with} \quad M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

Projection Algorithm

- **Input**: 3D point $[x\ y\ z]^\top$ to project

$$M\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} \quad \text{with} \quad M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

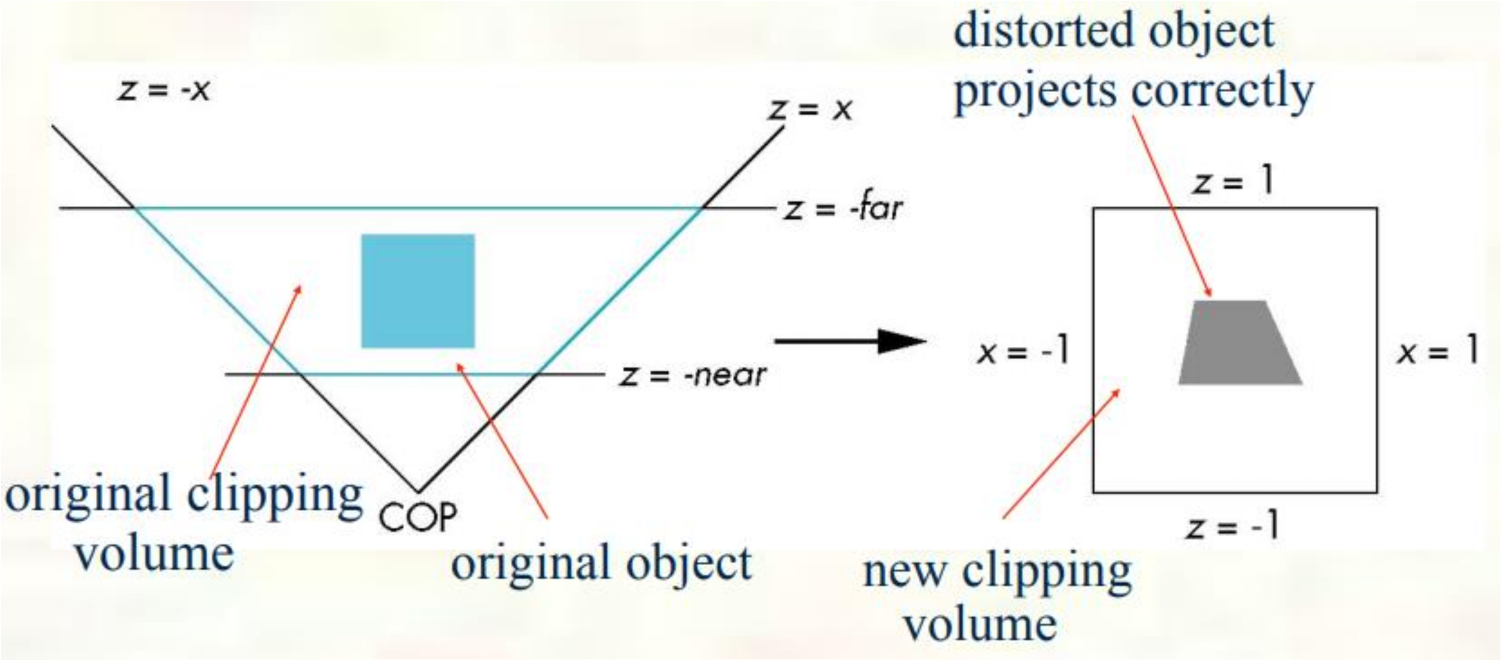
- Form $[x\ y\ z\ 1]^\top$
- Multiply M with $[x\ y\ z\ 1]^\top$; obtaining $[X\ Y\ Z\ W]^\top$
- Perform perspective division:

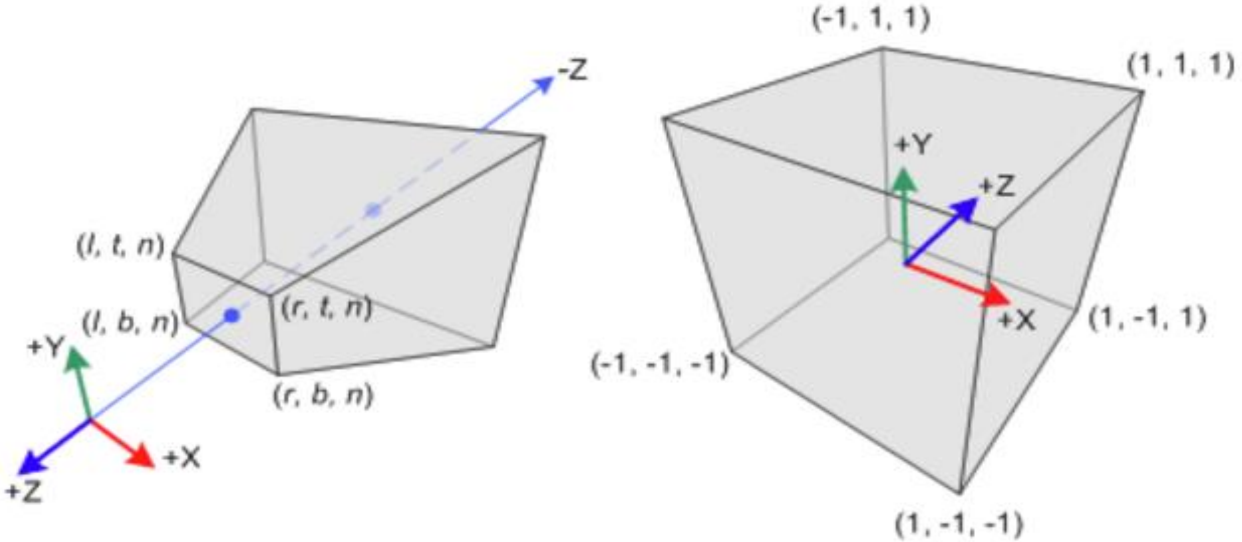
  X/W , Y/W , Z/W

$$\begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

- **Output**: $[X/W, Y/W, Z/W]^\top$
- (last coordinate will be d)

Clip Coordinates

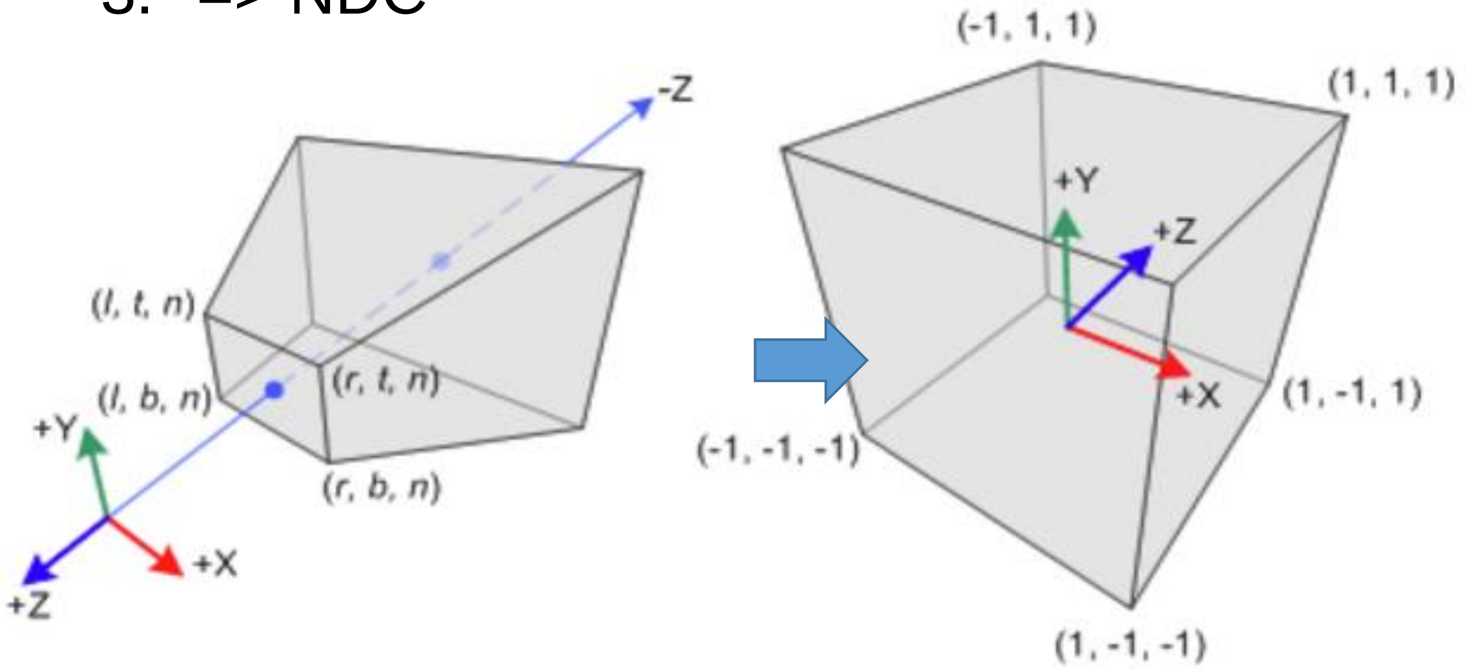




Both projection & clipping are integrated into GL_PROJECTION matrix.

Normalized Device Coordinates (NDC)

- Eye space => Clip space => NDC (a cube)
  1. Matrix multiplication: Mx
  2. Perspective division: Mx/w
  3. => NDC



$$\begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

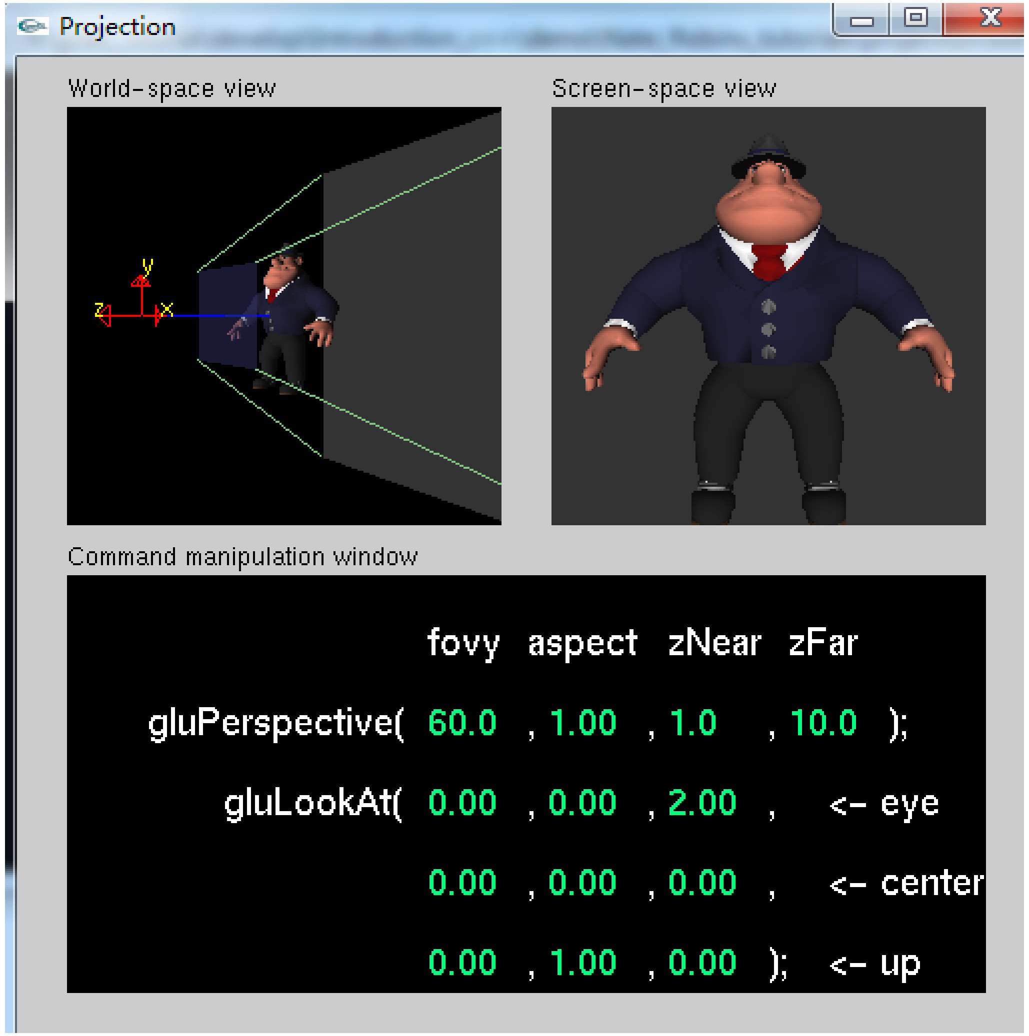
What we derived, but not really happened in GL

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

Both projection & clipping are integrated into GL_PROJECTION matrix.

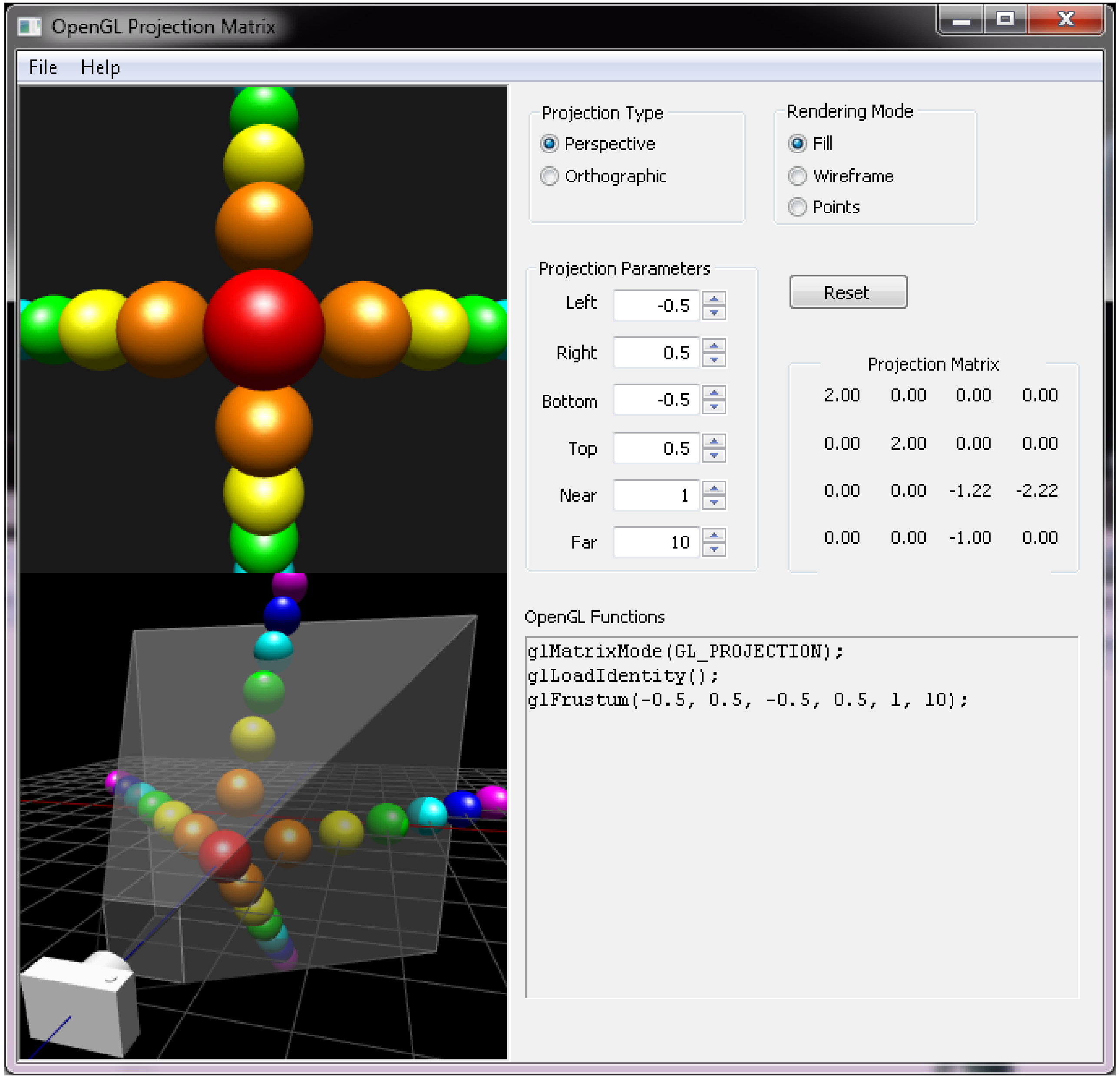
http://www.songho.ca/opengl/gl_projectionmatrix.html

Projection & Viewpoint (cont)



Nate_Robins_tutorials: Projection

Example: Projection Matrix

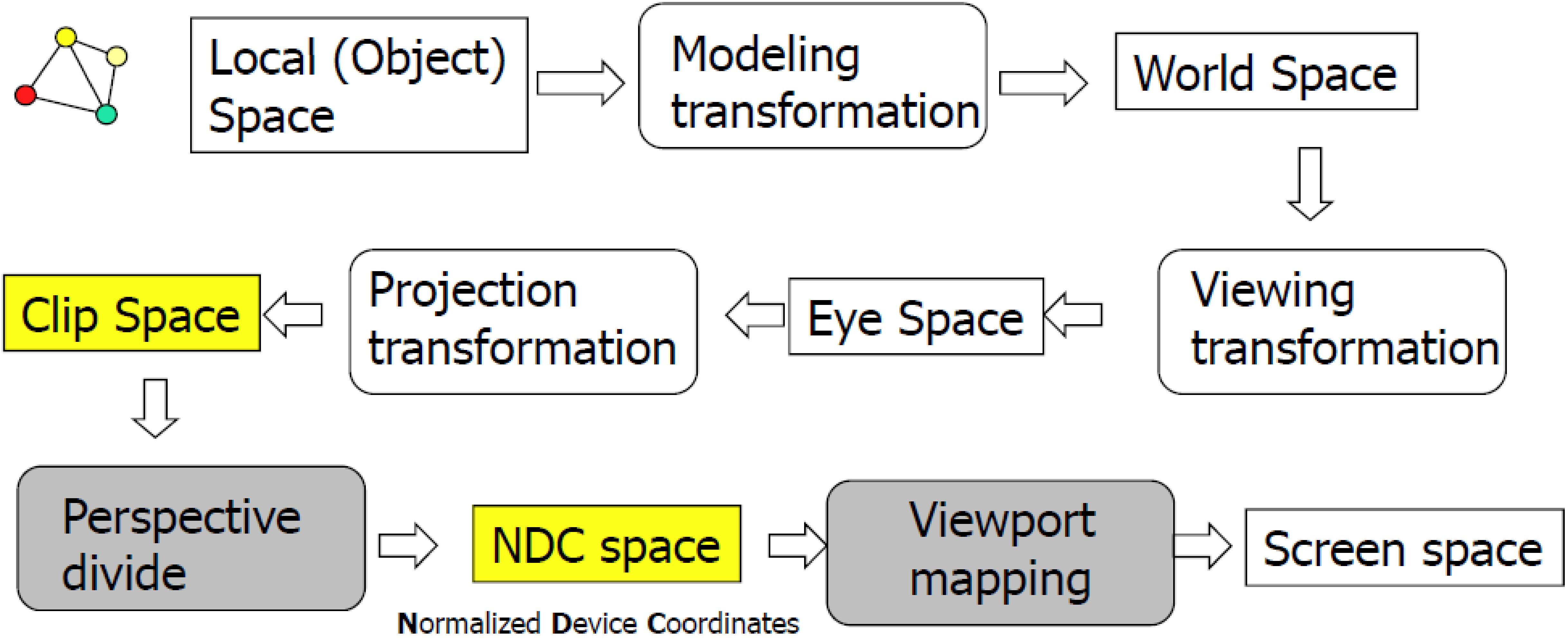


- http://www.songho.ca/opengl/gl_transform.html#projection

The Golden Rule

- Modeling transformation
  - glMatrixMode(GL_MODELVIEW); glRotate3f?
- Viewing transformation
  - glMatrixMode(GL_MODELVIEW); gluLookAt()
- Projection transformation
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity - to initialize current matrix.
  - **gluPerspective/glFrustum/glOrtho/gluOrtho2 - to set the appropriate projection onto the stack.**

Transformation Pipeline

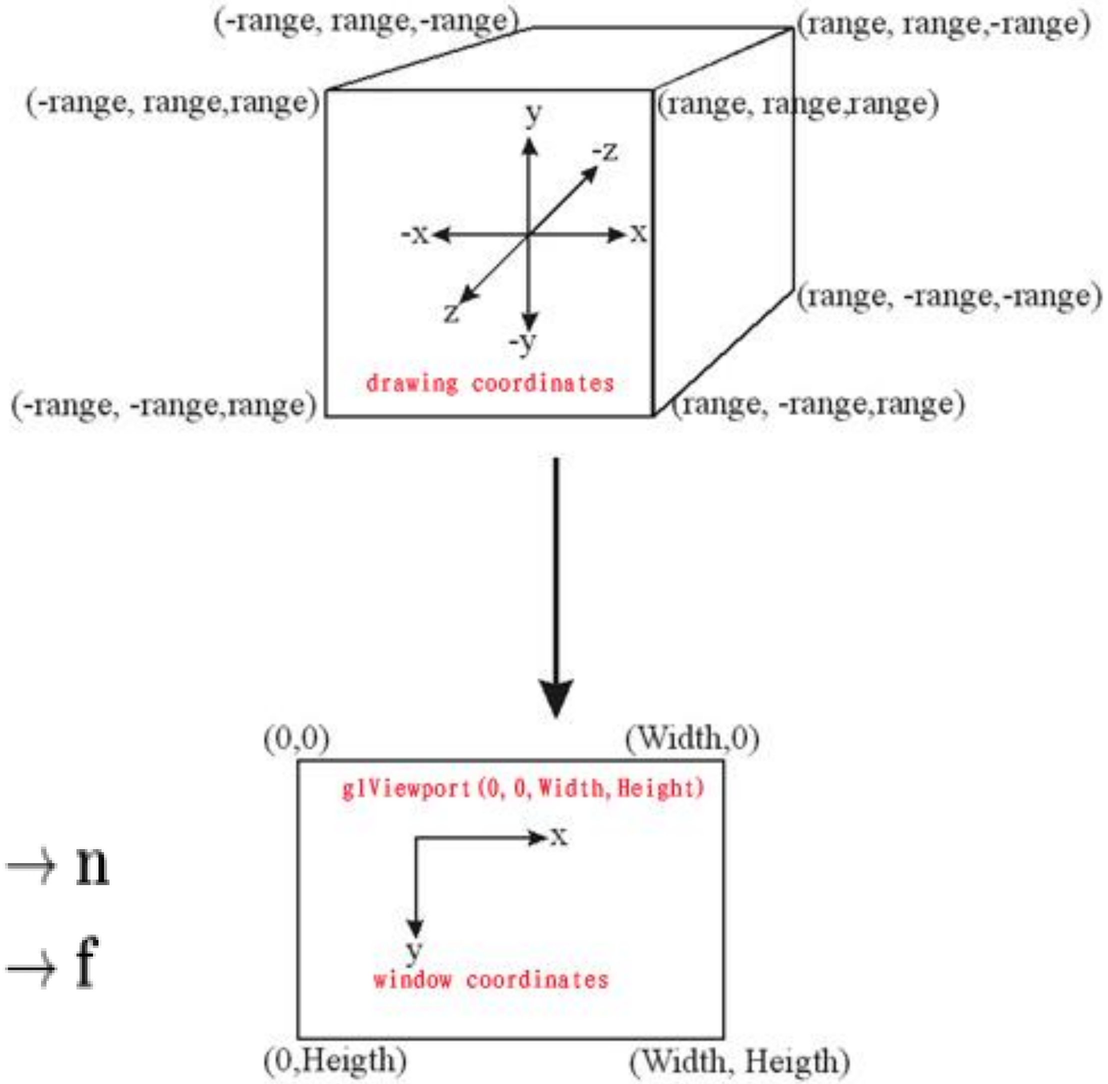


Viewport and Depth Range

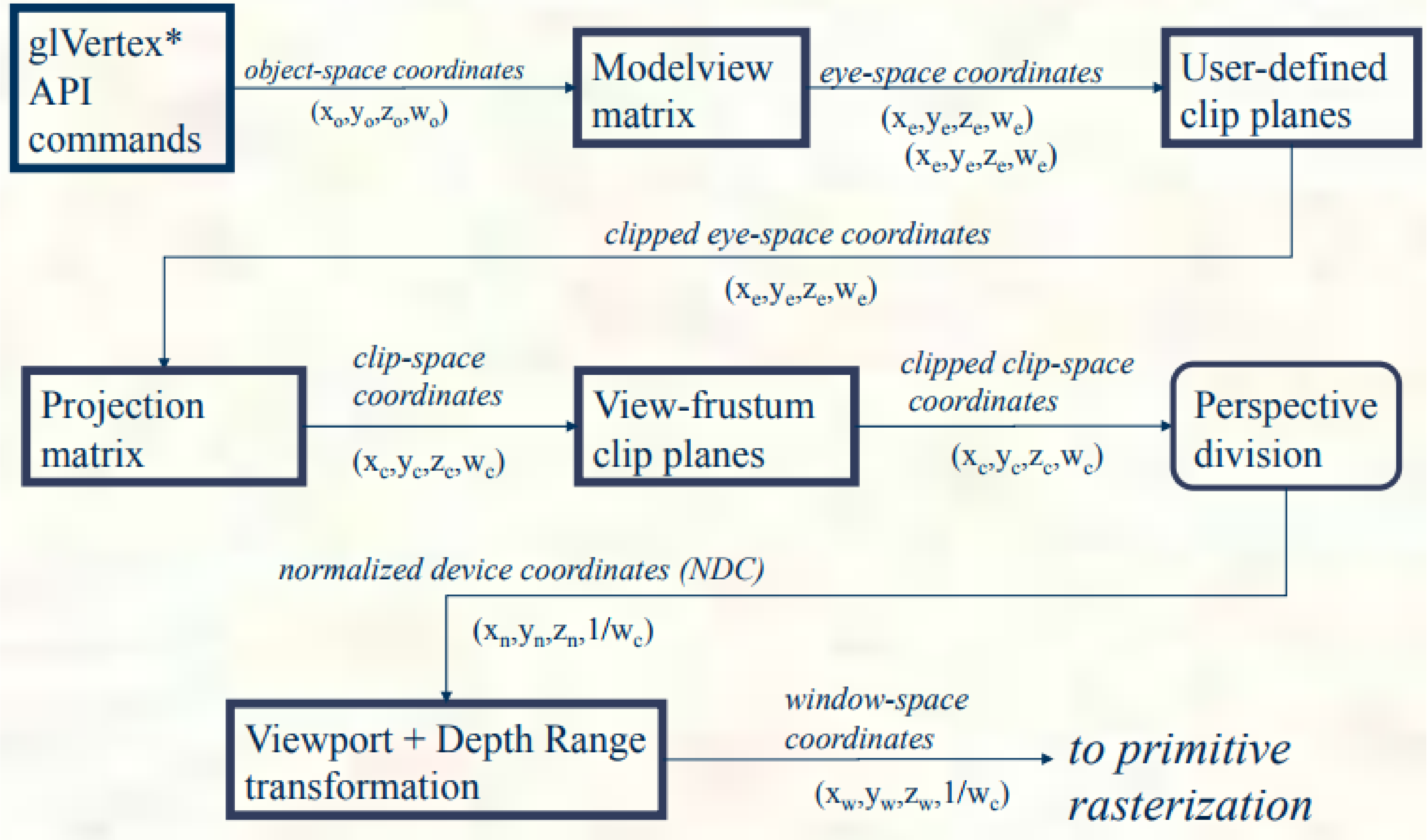
- **glViewport(int left, bottom, w, h)** maps NDC to **window coordinates**
  - If the user resizes the window, we have to adjust the **viewport** and correct the aspect ratio.
- **glDepthRange(n, f);**

$$\begin{pmatrix} x_w \\ \\ y_w \\ \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{\mathbf{w}}{2} x_{ndc} + \left(\mathbf{x} + \frac{\mathbf{w}}{2}\right) \\ \\ \frac{\mathbf{h}}{2} y_{ndc} + \left(\mathbf{y} + \frac{\mathbf{h}}{2}\right) \\ \\ \frac{\mathbf{f} - \mathbf{n}}{2} z_{ndc} + \frac{(\mathbf{f} + \mathbf{n})}{2} \end{pmatrix}$$

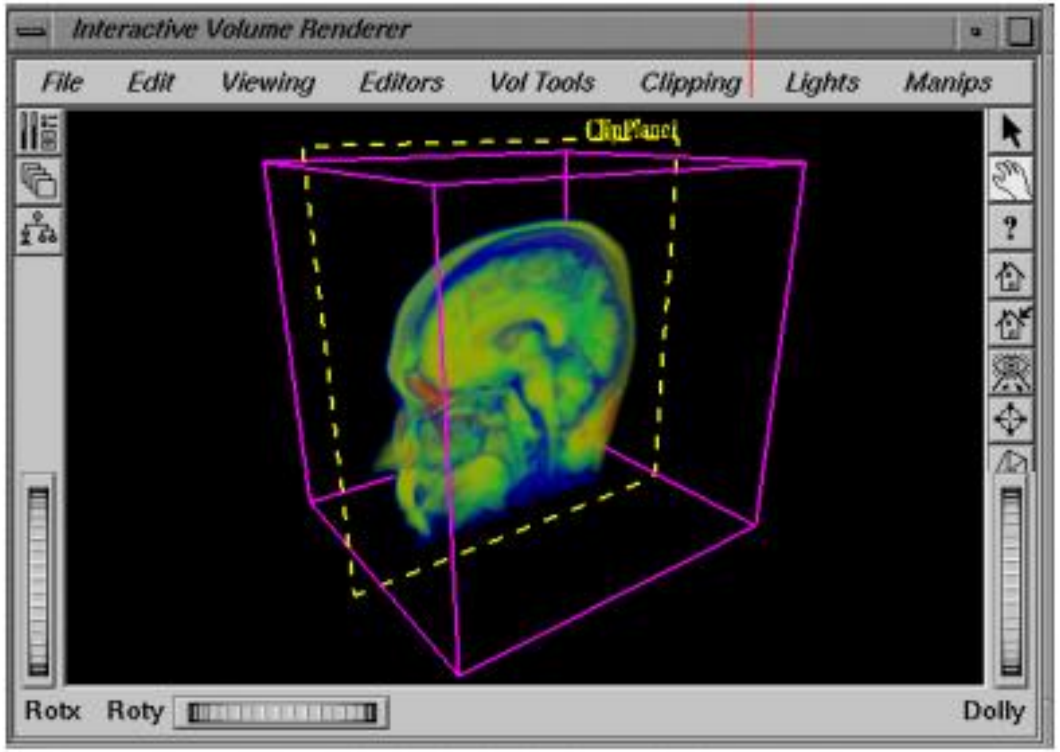
$$\begin{cases} -1 & \rightarrow \mathbf{x} \\ 1 & \rightarrow \mathbf{x} + \mathbf{w} \end{cases}, \quad \begin{cases} -1 & \rightarrow \mathbf{y} \\ 1 & \rightarrow \mathbf{y} + \mathbf{h} \end{cases}, \quad \begin{cases} -1 & \rightarrow \mathbf{n} \\ 1 & \rightarrow \mathbf{f} \end{cases}$$



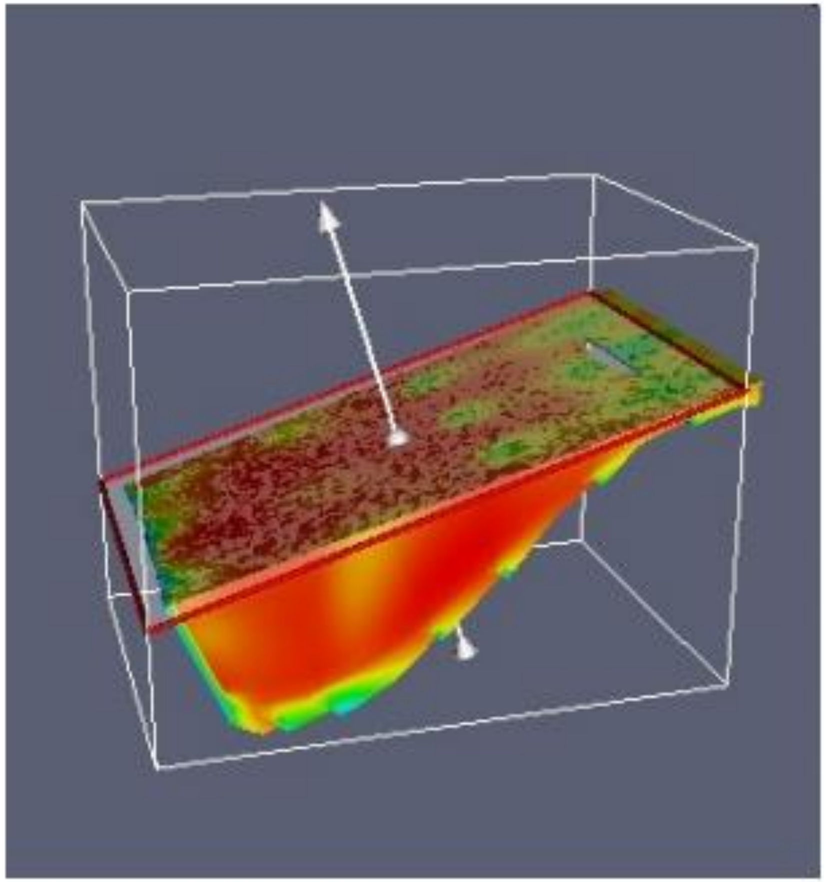
Conceptual Vertex Transformation



User Clip Planes in Practice



[IVoR]



[ParaView]

Primarily used scientific visualization
and Computer Aided Design (CAD) applications

Thanks