Follow your vision

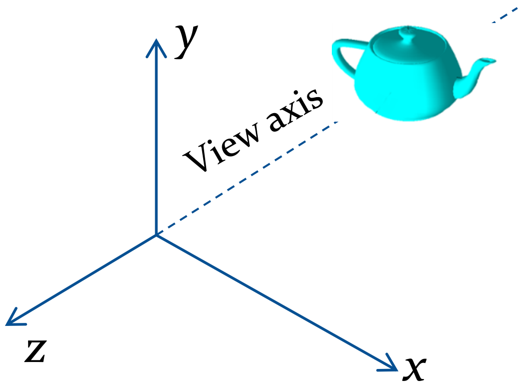
4 View Transform and Projection

Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

R. Mukundan (mukundan@canterbury.ac.nz)

OpenGL "Camera"

- OpenGL <u>always</u> displays the scene as seen from the origin, looking towards the **negative *z*-axis**.

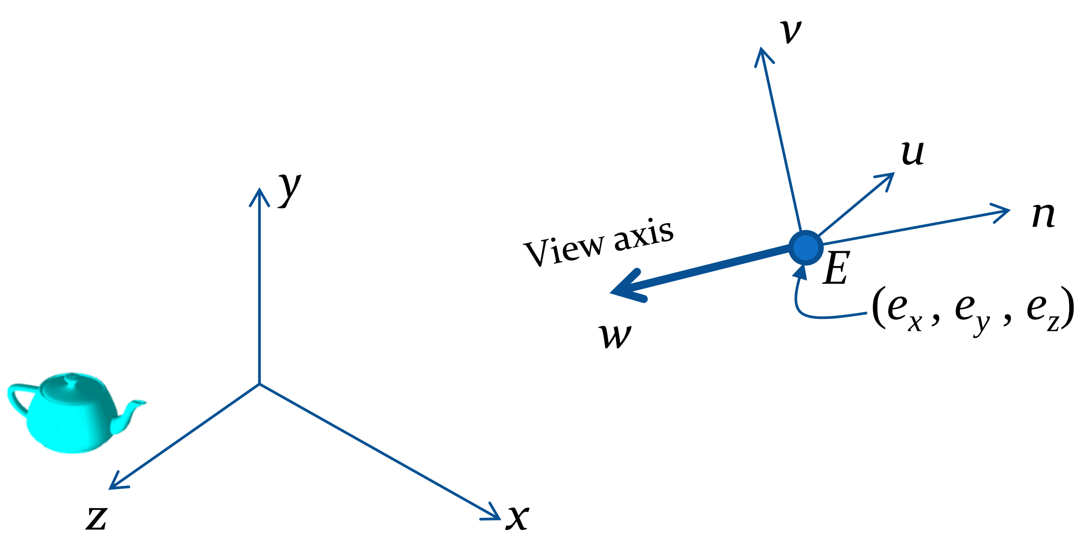



```
void display()
{
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0, 0, -4);
  glutSolidTeapot(1);
 ...
```

- If something needs to be displayed, it should be finally be (after all transformations) on the –z side.
  - How can I get the display of a teapot on the +z side?
  - How can I get the view from an arbitrary position and direction?

OpenGL "Camera"

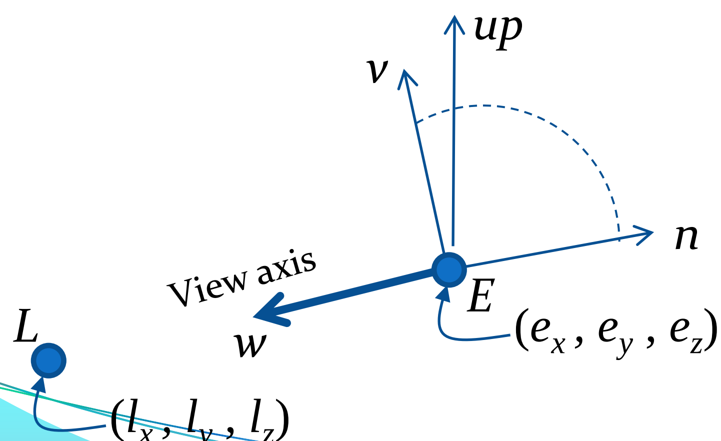
- If you require a display of the scene from a different view point $E(e_x, e_y, e_z)$ along the direction ***w***, then we have to transform the entire scene to a new coordinate frame so that in the new frame,
  - (e_x, e_y, e_z) is at the origin
  - ***w*** is along the –z axis in the transformed frame.



u: Camera's x-axis
v: Camera's y-axis
n: Camera's z-axis

A "Camera" Function

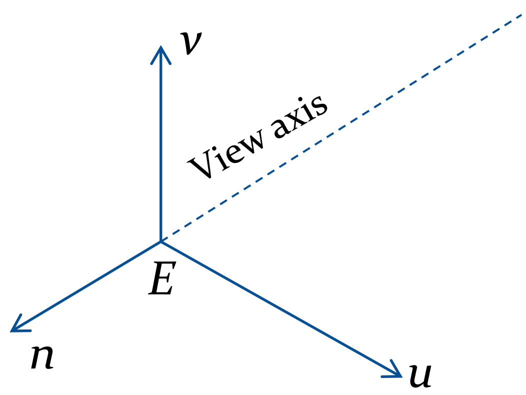
- The GLU library contains the function `gluLookAt(...)` that transforms the coordinates to the (*u*, *v*, *n*) frame such that
  - the eye position (e_x, e_y, e_z) is at the origin of the new frame
  - the view direction ***w*** is along the –z axis in the new frame. The view direction is specified using a "look point".
- The frame can still rotate about the view axis. In order to fix the frame, an approximate "up-vector" is also defined.



n, *up* vectors are known (user specified)
How can we get *u*, *v*?
u: $up \times n$
v: $n \times u$
Now we have two orthogonal systems, and can find the transformation between them.

A "Camera" Function

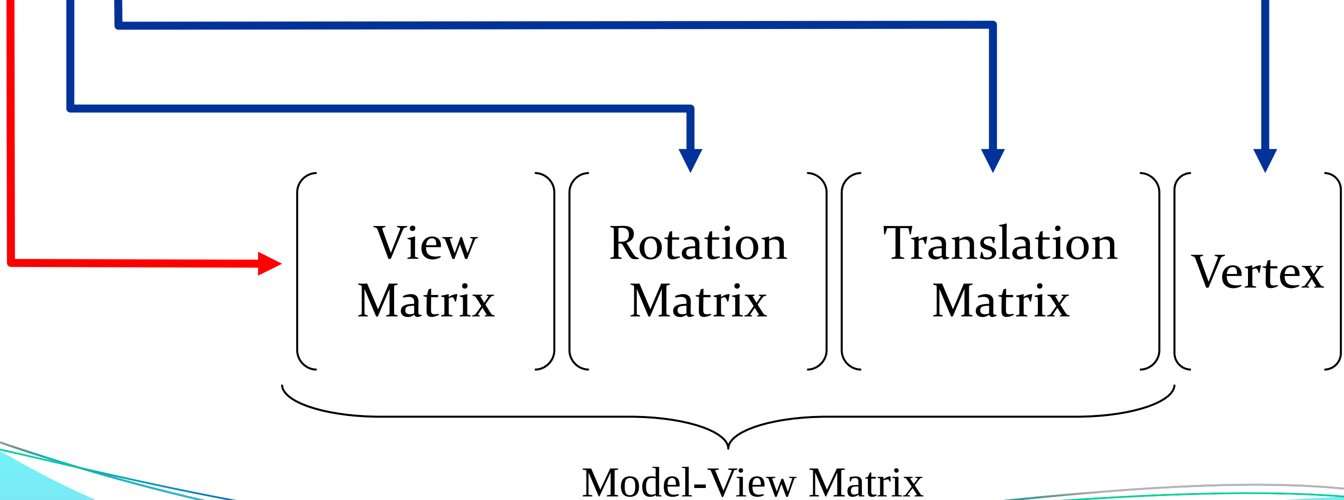
- The `gluLookAt(ex, ey, ez, lx, ly, lz, kx, ky, kz)` function creates a 4x4 transformation matrix for obtaining the new coordinates (u, v, n) from (x, y, z).
- This transformation must be applied to the entire scene.
- The gluLookAt(..) function must be called above all transformation functions that are applied to the scene.
- The eye position is at the origin of the transformed scene.



(u, v, n):
Camera coordinates
or
Eye coordinates

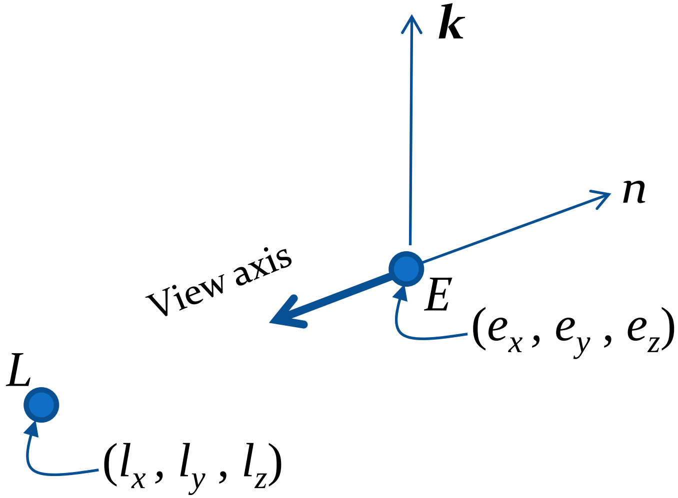
Model-View Transformation

```
void display()
{
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt (0, 5, 10, 0, 2, 0, 0, 1, 0);
  glRotatef(15.0, 0.0, 0.0, 1.0);
  glTranslatef(0.8, 0.2, -4.0);
  glutWireSphere(0.2, 10, 8);
 ...
```



Camera Coordinates

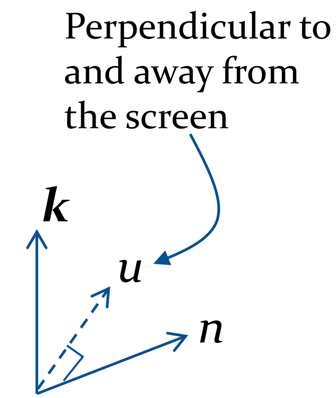
- The `gluLookAt(ex, ey, ez, lx, ly, lz, kx, ky, kz)` function specifies
  - Two points $E\,(e_x, e_y, e_z)$ and $L\,(l_x, l_y, l_z)$
  - A up-vector $\boldsymbol{k}\,(k_x, k_y, k_z)$

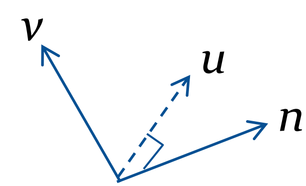


$$\boxed{n = \frac{E - L}{|E - L|}}$$

$$\boxed{u = \frac{k \times n}{|k \times n|}}$$

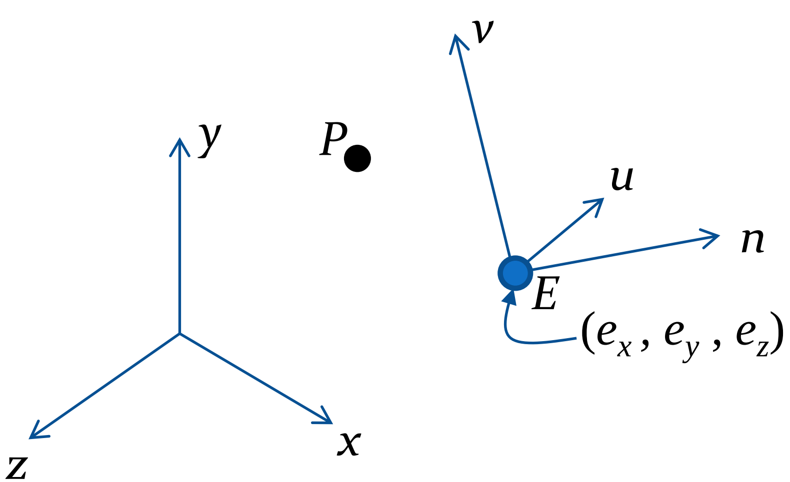
$$\boxed{v = n \times u}$$

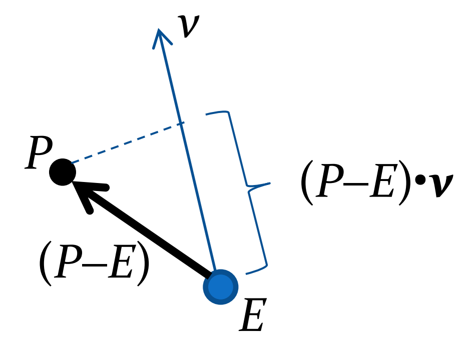




View Transformation

- Consider a point P (x, y, z) in the world coordinate space.
- The coordinate of P along the v-axis is the projection of the vector $P–E$ along that axis. This is $(P–E)\bullet\boldsymbol{v}$
- Similarly, the other coordinates of P in the new reference frame can be computed.





View Transformation

We can write the view transformation (the transformation of any point (x, y, z) into **eye coordinates** (x_e, y_e, z_e)) as

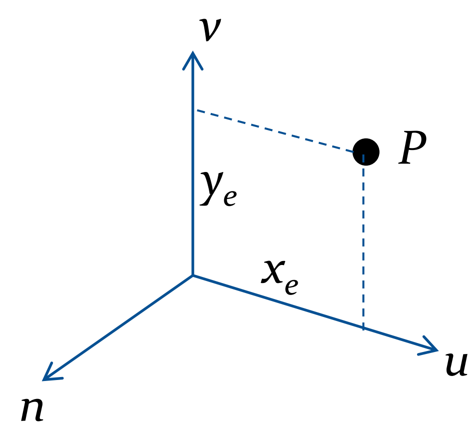
$$x_e = (P-E)\bullet\mathbf{u} = P\bullet\mathbf{u} - E\bullet\mathbf{u} = xu_x + yu_y + zu_z - E\bullet\mathbf{u}$$

$$y_e = (P-E)\bullet\mathbf{v} = P\bullet\mathbf{v} - E\bullet\mathbf{v} = xv_x + yv_y + zv_z - E\bullet\mathbf{v}$$

$$z_e = (P-E)\bullet\mathbf{n} = P\bullet\mathbf{n} - E\bullet\mathbf{n} = xn_x + yn_y + zn_z - E\bullet\mathbf{n}$$

Therefore,

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & -E\cdot\mathbf{u} \\ v_x & v_y & v_z & -E\cdot\mathbf{v} \\ n_x & n_y & n_z & -E\cdot\mathbf{n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



The transformation matrix generated by gluLookAt()

Camera Modes

A camera can be placed in a scene and transformed in different ways:

- Free-camera: The user can control the position and orientation of the camera irrespective of other object transformations in the scene.
- Camera attached to an object, eg. First Person View (see next slide)
- Fly-by camera: The camera transformed along a predefined path, usually without any user interaction.

Camera Modes

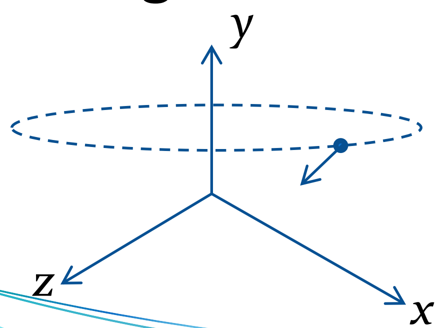
- First Person View (FPV): The view of the scene from the object/character being controlled. In a game, it is the view from the player's eye level.

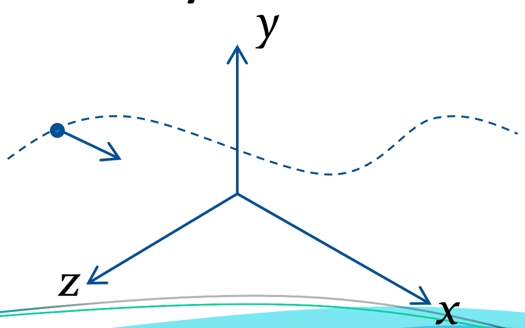
- Second Person View provides a view from a target, and is rarely used.

Camera Modes

- Third Person View: A view of the scene from a different perspective. This camera mode could either be a "free-camera" or dependent on other transformations.

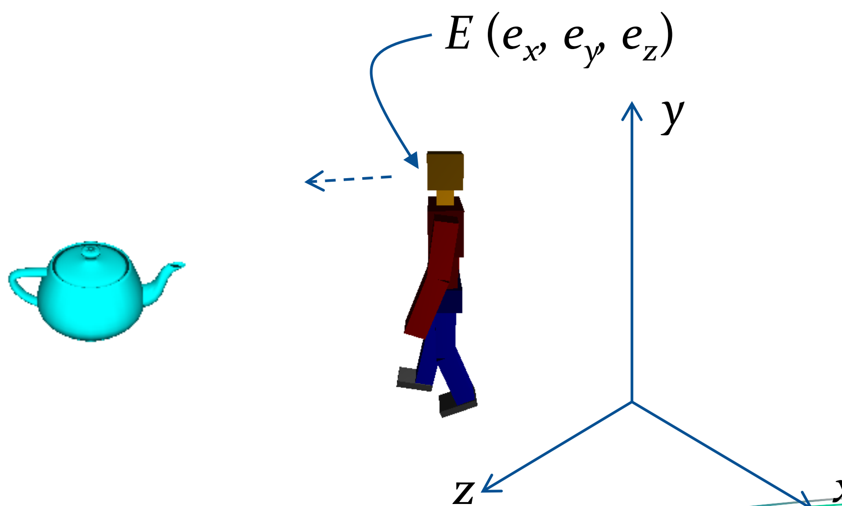
- Fly-by Camera: The camera moved along a predefined path through the scene, usually without any user interaction.





Creating First Person Views (Method 1)

- Keep track of the object's position and orientation in world coordinate space, and update the camera position and the look vector.
  - You will need to compute the object's pose every frame, and reposition the camera on the transformed object.
  - Note: You cannot get the transformed vertex coordinates from OpenGL, you will have to compute them separately.



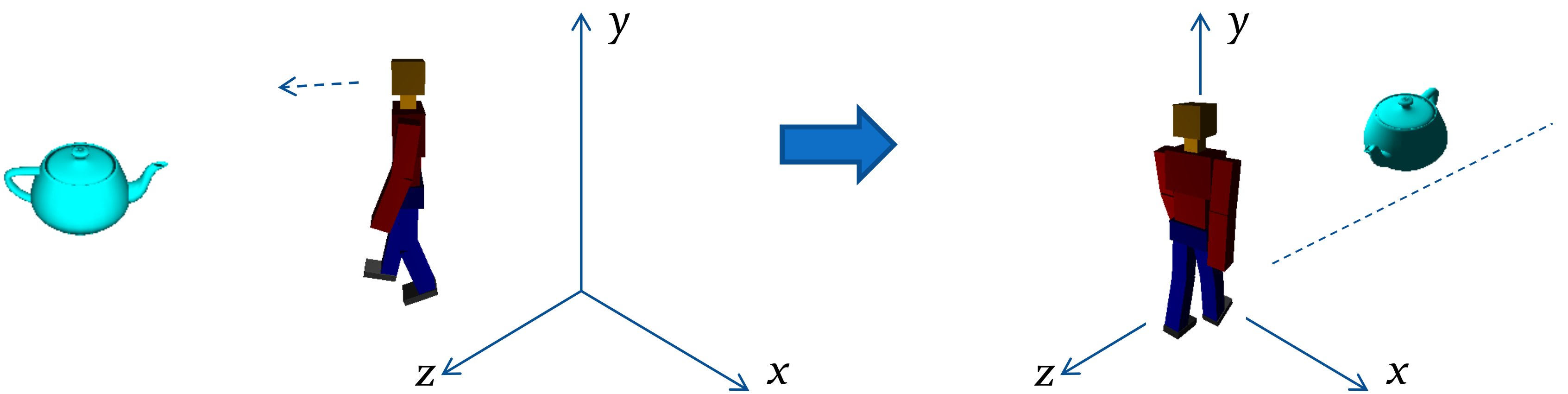
Creating First Person Views (Method 1)

```
void display()
{
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  ...  // compute camera parameters here
  ...  // by using character transformation
     gluLookAt (ex,ey,ez, lx,ly,lz, 0,1,0);
     ...   //common transforms
     glPushMatrix();
        //character transform
        drawCharacter();  //user defined
     glPopMatrix();

     glPushMatrix();
        //Teapot transform
        glutSolidTeapot(1);
     glPopMatrix();
```

Creating First Person Views (Method 2)

- This method does not use gluLookAt(...) which requires the transformed coordinates of a point on the character.
- Instead, the enitre scene is inverse-transformed so that the character goes back to the origin, looking towards the –z axis.



Creating First Person Views (Method 2)

```
void display()
{
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  ...  // Inverse of character transformation

     ...   //common scene transforms
     glPushMatrix();
        //character transform
        drawCharacter();  //user defined
     glPopMatrix();

     glPushMatrix();
        //Teapot transform
        glutSolidTeapot(1);
     glPopMatrix();
```

Character Transformation Example

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
  glRotatef(180, 0, 1, 0);    //Look towards -z
  glRotatef(-theta, 0, 1, 0);
  glTranslatef(-tx, -ty, -tz);

     ...   //common scene transforms
     glPushMatrix();
        glTranslatef(tx, ty, tz);
        glRotatef(theta, 0, 1, 0);
        drawCharacter();  //user defined
     glPopMatrix();

     ...  //other objects in the scene
```

Inverse of

If **A** and **B** are matrices, $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\,\mathbf{A}^{-1}$

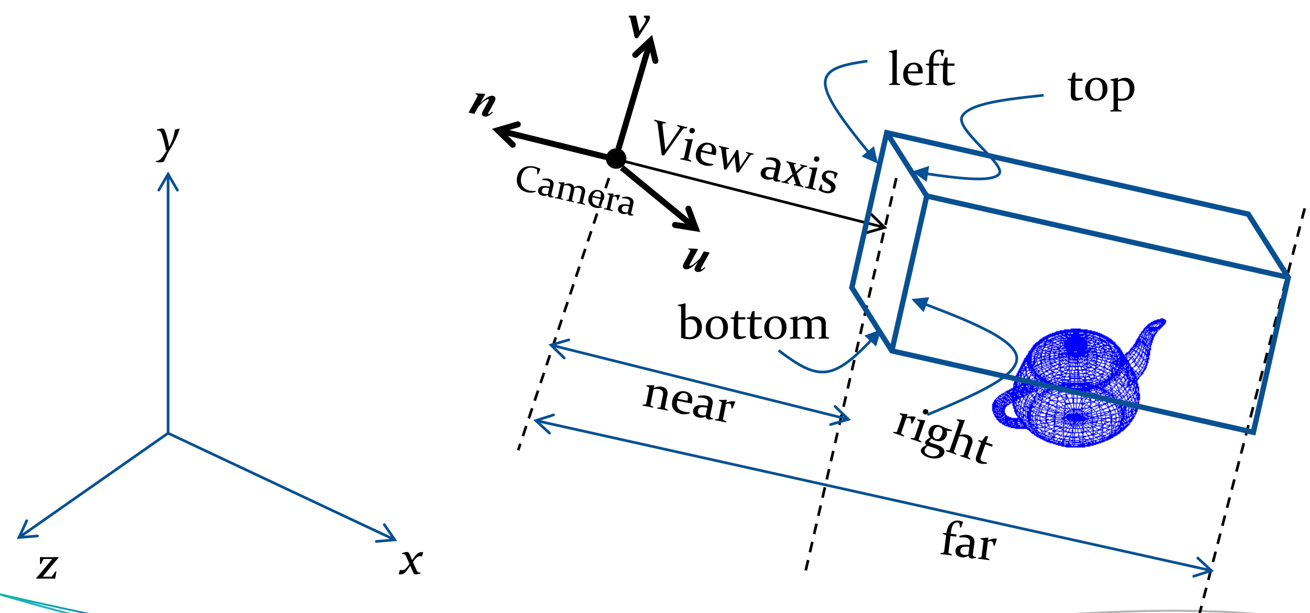
View Volumes

- The view transformation only transforms the world coordinates of points into the camera's coordinate frame.
- We need to specify "how much" the camera actually sees. That is, we require a view volume that contains the part of the scene that is visible to the camera. In other words, the view volume acts as a **clipping volume**.
- We further require a projection model to simulate the way in which the 3D scene is viewed.
- The view volume is attached to the camera and is always defined in the camera-coordinate space. Therefore, all points inside the view volume are represented using **eye coordinates** (x_e, y_e, z_e).

Orthographic View Volume

- The orthographic view volume is a rectangular region defined in the camera coordinate space.
- OpenGL function:

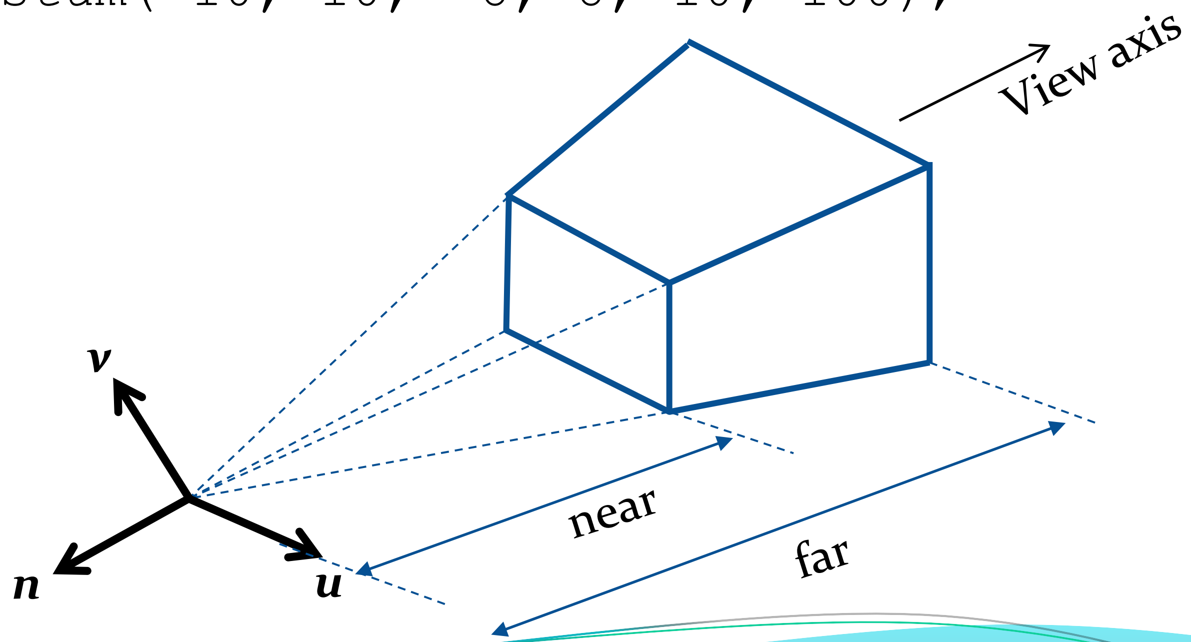
```
    glOrtho(left, right, bottom, top, near, far);
Eg: glOrtho(-10, 10, -8, 8, 10, 100);
```



Perspective View Volume

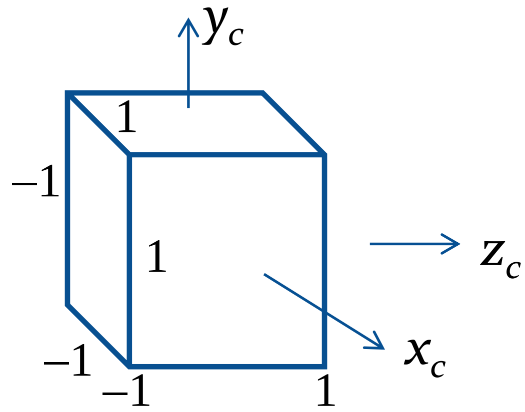
- The perspective view volume is defined by a frustum that has its vertex at the eye position. The near-plane acts as the plane of projection.
- OpenGL function:

```
     glFrustum(left, right, bottom, top, near, far);
Eg:  glFrustum(-10, 10, -8, 8, 10, 100);
```



The Canonical View Volume

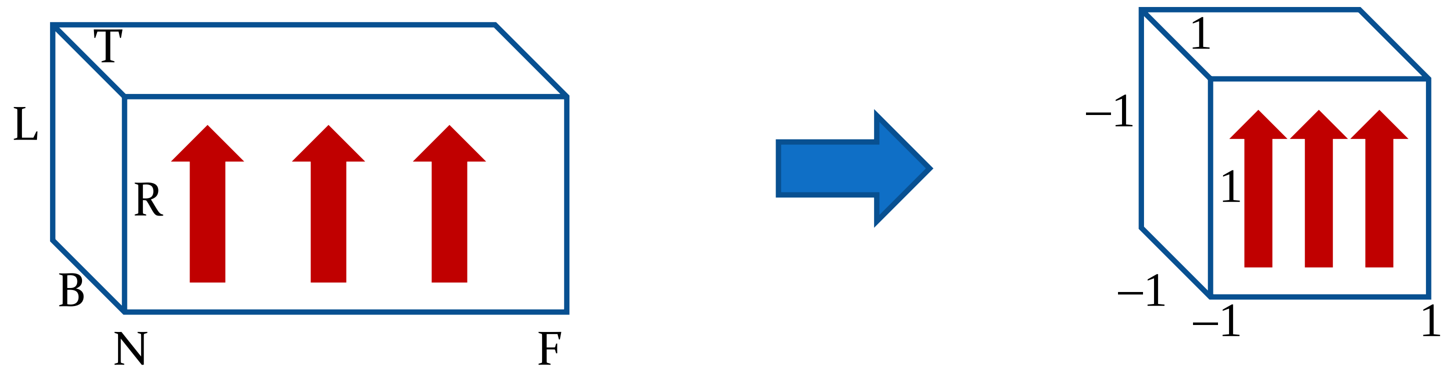
- All view volumes are mapped to a canonical view volume which is an axis-aligned cube with sides at a distance of 1 unit from the centre.
- The coordinates of a point inside the canonical view volume are called clip coordinates.
- The canonical view volume facilitates clipping of the primitives with its sides.
- A point is visible only if it has clip coordinates between -1 and +1.



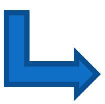
Clip Coordinate Axes

glOrtho(...)

The function glOrtho(...) transforms points inside the orthographic view volume into points inside the canonical view volume, where the coordinates have the range [-1, 1].



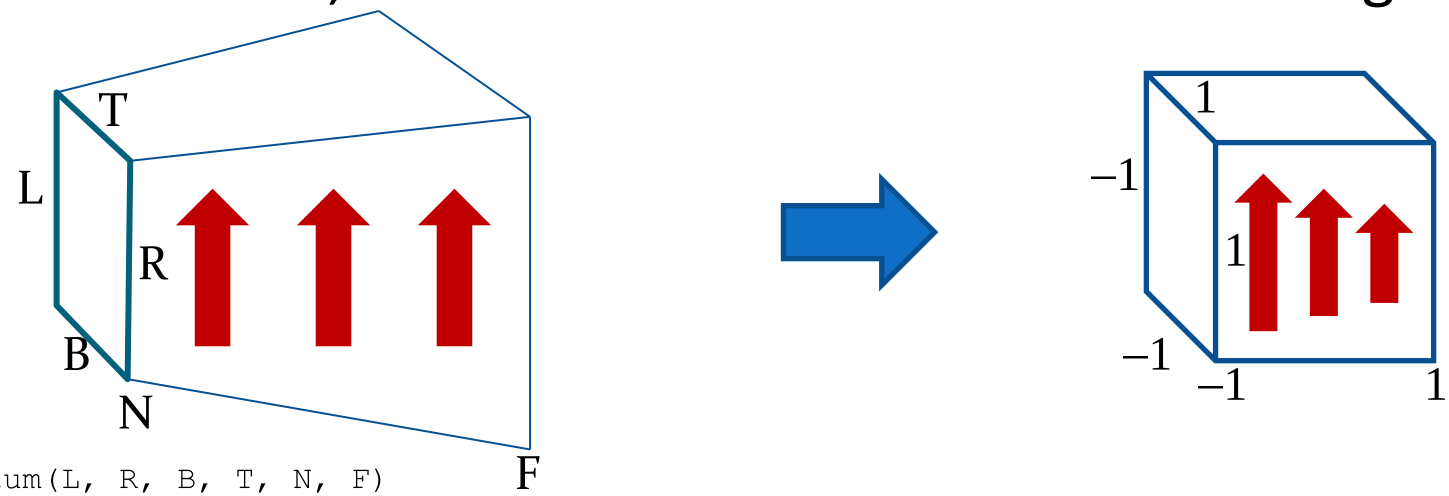

```
glOrtho(L, R, B, T, N, F)
```



$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{R-L} & 0 & 0 & -\left(\frac{R+L}{R-L}\right) \\ 0 & \frac{2}{T-B} & 0 & -\left(\frac{T+B}{T-B}\right) \\ 0 & 0 & \frac{-2}{F-N} & -\left(\frac{F+N}{F-N}\right) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

glFrustum(..)

- The function glFrustum(...) transforms points inside the perspective view volume into points inside the canonical view volume, where the coordinates have the range [-1, 1].

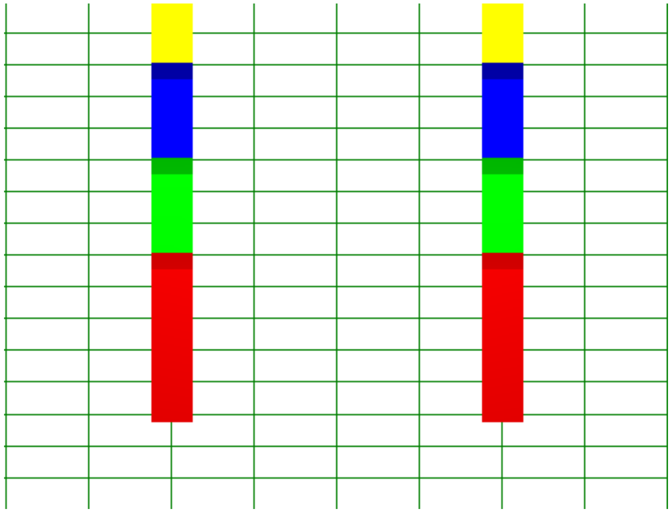


`glFrustum(L, R, B, T, N, F)`

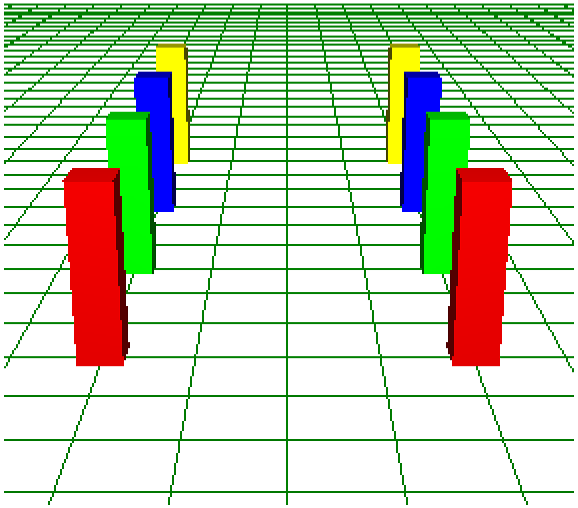
Perspective division term

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w \end{bmatrix} = \begin{bmatrix} \frac{2N}{R-L} & 0 & \frac{R+L}{R-L} & 0 \\ 0 & \frac{2N}{T-B} & \frac{T+B}{T-B} & 0 \\ 0 & 0 & -\left(\frac{F+N}{F-N}\right) & \frac{-2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

Projection Examples



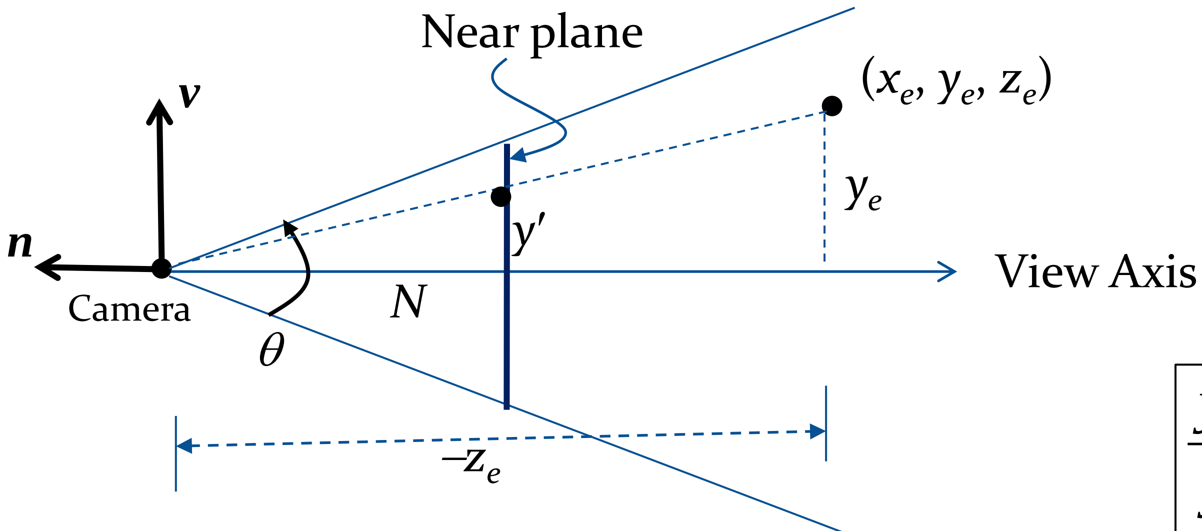
Orthographic Projection
`glOrtho(...)`



Perspective Projection
`glFrustum(...)`

Perspective Projection

- A perspective projection of a point (x_e, y_e, z_e) inside the view frustum is a point (x', y') on the near plane.



Pinhole Camera Model

$$\frac{y_e}{y'} = \frac{-z_e}{N}$$

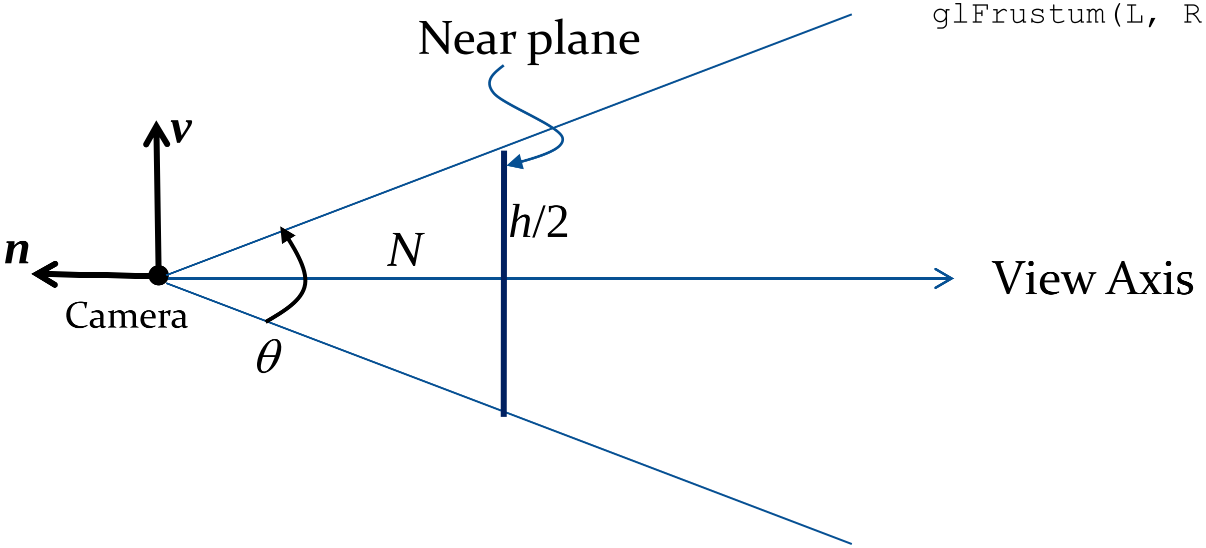
$$\therefore \; y' = \frac{N\,y_e}{-z_e}$$

$$(x', y') \rightarrow \left(\frac{Nx_e}{-z_e}, \frac{Ny_e}{-z_e} \right)$$

Perspective Projection

- The field of view of the view frustum is a useful parameter that can be conveniently adjusted to cover a region in front of the camera.

`glFrustum(L, R, B, T, N, F)`



$h = T - B$

- Field of view along the *y*-axis of the eye-coordinate space fov = θ.

$$\tan\left(\frac{\theta}{2}\right) = \frac{h}{2N}$$

gluPerspective(..)

- The GLU library provides another function for perspective transformation in the form

```
gluPerspective(fov, aspect, near, far);
```

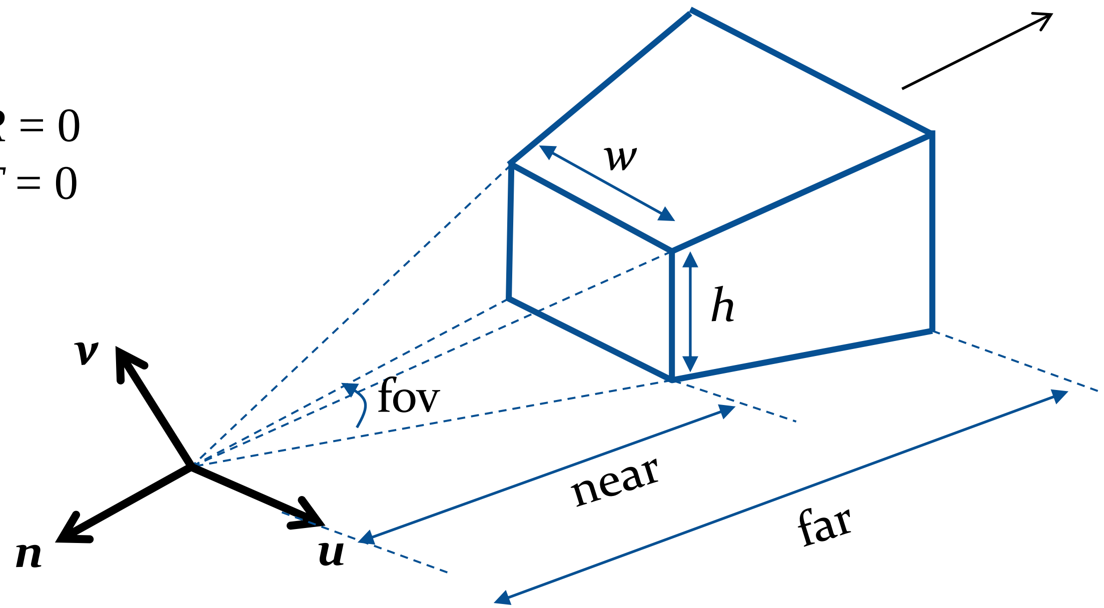
- In this case, the view axis passes through the centre of the near plane.
- Aspect Ratio = *w*/*h*

$L = -w/2 \qquad R = w/2 \qquad L + R = 0$

$B = -h/2 \qquad T = h/2 \qquad B + T = 0$

$R - L = w, \qquad T - B = h$

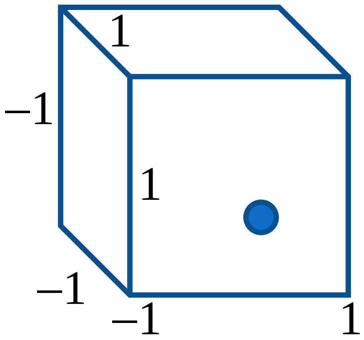
How do you get w, h ?

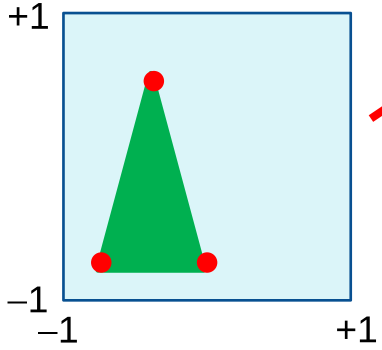


Clip Coordinates

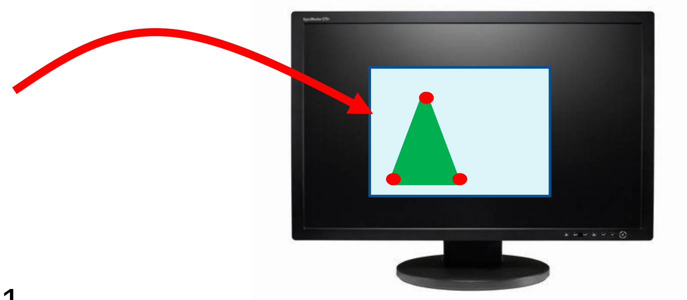
Suppose a point has clip coordinates (x_c, y_c, z_c).

- The z_c value is called the point's pseudo-depth. It has a value between -1 and +1.
- The pseudo-depth is converted into a depth buffer value in the range [0, 1] using the equation $z_{\text{depth}} = (z_c + 1)/2$
- If the point passes the **depth test**, then its clip coordinates (x_c, y_c) are mapped to the display viewport.





Clip Coordinates



An Overview of Transformations