COSC363 Computer Graphics

7

Mathematics of Lighting and Viewing

There's more to a scene than meets the eye

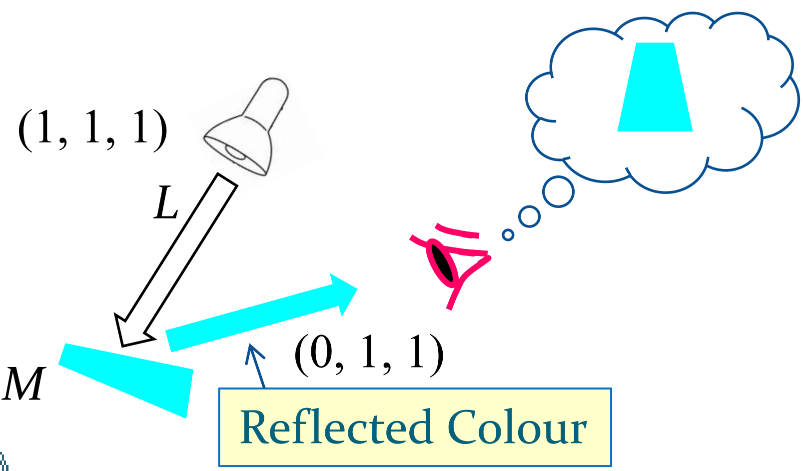
R. Mukundan (mukundan@canterbury.ac.nz)
Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

Local Illumination Model

- **OpenGL uses a local illumination model** where the colour value at each vertex is computed using
  - The position of the light and the vertex
  - Light and material properties
  - The surface normal orientation at the vertex
  - The position of the viewer
- The local illumination model does not take into account any other geometrical or colour information in the scene (Eg. light reflected from other objects)
- The illumination model is highly suitable for per-vertex lighting (computation inside the vertex processor of the rendering pipeline).

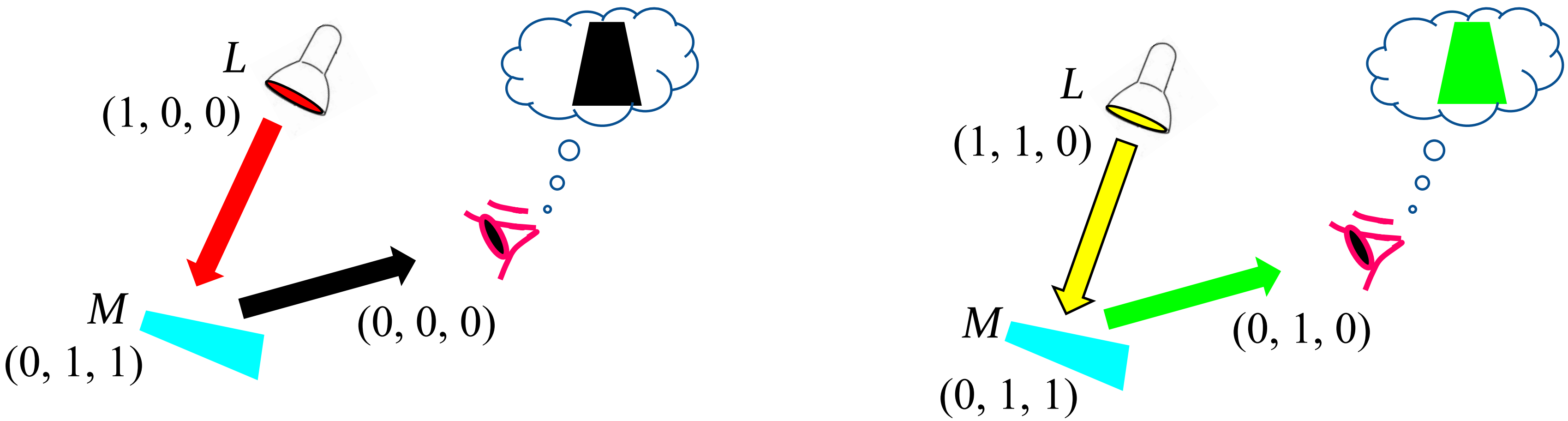
Reflected Light at a Vertex

- The colour at a vertex of an object is the colour of light reflected from the vertex towards the direction of the viewer.
- The color of the reflected light depends on the color of the incident light *L* and the color of the material *M*. This light-material interaction is modelled using a simple modulation of color values as shown on the next slide.

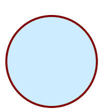


Side note: The incident light has a white color. The material reflects only cyan colored light (because it appears in that color!). The material thus absorbs the remaining component (red) of white light. If the light itself is colored red, then this material will appear black (see next slide).

Reflected Color

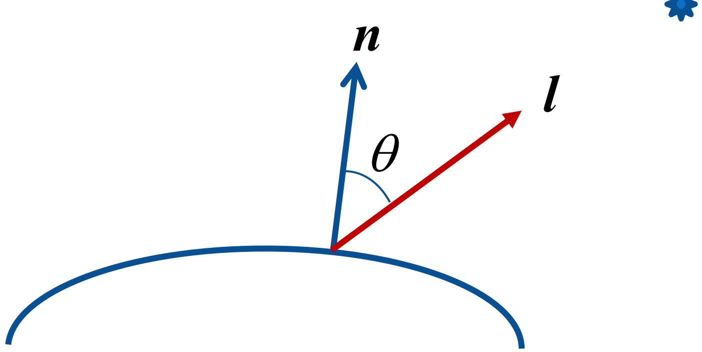


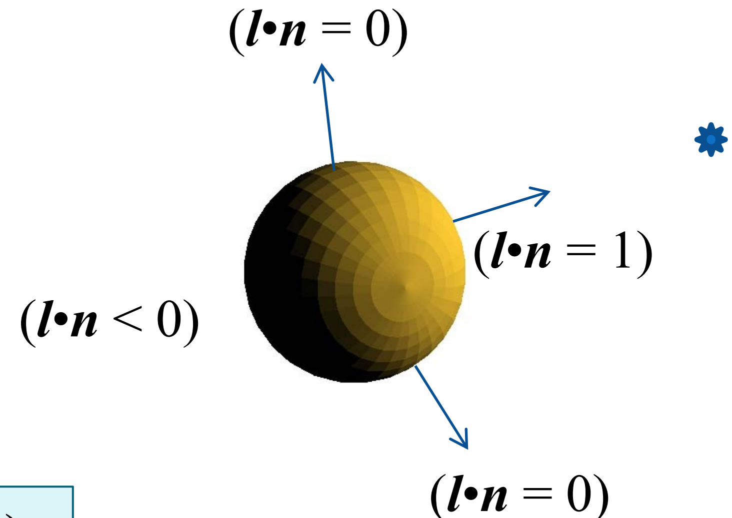
In OpenGL, the interaction between the light and the material properties is simply modelled as the **component-wise product** of the light colour and the material colour.

Light colour		Material colour		Perceived colour of the object
(0., 1., 1.)	*	(1., 0., 1.)	=	(0., 0., 1.)
				
(l_1, l_2, l_3)	*	(m_1, m_2, m_3)	=	$(l_1 m_1,\ l_2 m_2,\ l_3 m_3)$

Diffuse Reflections

The diffuse reflection from a surface varies as the cosine of the angle between the **normal vector *n*** and the **light source vector *l*** (Lambert's law): $\boldsymbol{I}_d = L_d M_d \cos\theta$





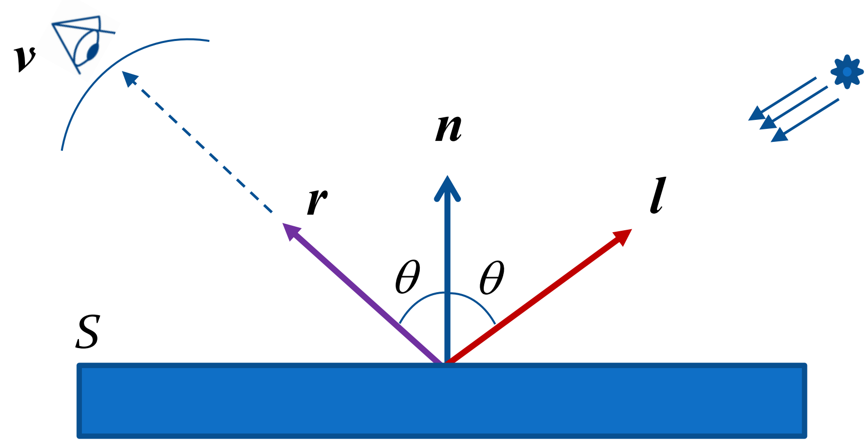
$$\boxed{\boldsymbol{I}_d = L_d M_d \max(\boldsymbol{l} \bullet \boldsymbol{n}\ , 0)}$$

L_d: Light's diffuse color
M_d: Material's diffuse color

Note: $\boldsymbol{l}$, $\boldsymbol{n}$ must be normalized to unit vectors.

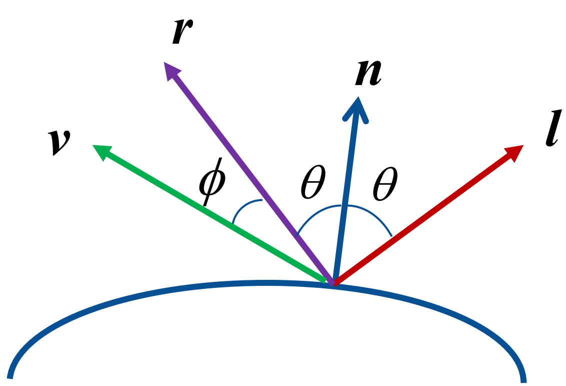
Specular Reflections

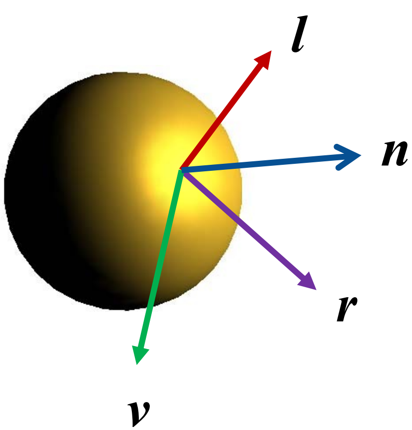
- Consider a highly polished (mirror-like) surface *S*. In the following figure, ***l*** is the light source vector and ***n*** the normal vector.
- A viewer along the direction of reflection ***r***, where the angle of reflection is the same as the angle of incidence (θ) of light, will observe maximum specular reflection from *S*.
- The intensity of specular highlight reduces as the viewer moves away from ***r***.



Specular Reflections

Similar to diffuse reflection, we can write, $\boldsymbol{I_s} = L_s M_s \cos\phi$, where ϕ is the angle between the view vector ***v*** and the reflection vector ***r***.





$$\boxed{\boldsymbol{I_s} = L_s M_s \max(\boldsymbol{r} \bullet \boldsymbol{v}\ ,\ 0)}$$

L_s: Light's specular color

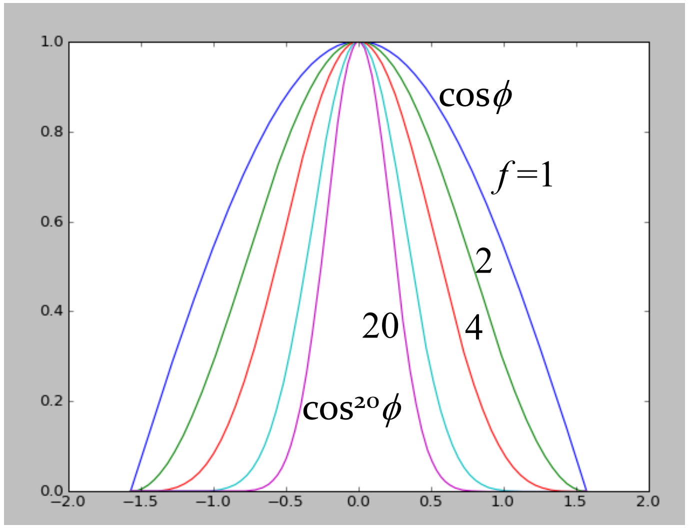
M_s: Material's specular color

Note: ***r***, ***v*** are unit vectors.

Specular Reflections

We also include the Phong's constant (shininess term) f to control the diameter of the specular highlight

$$\boldsymbol{I_s} = L_s M_s \{\max(\boldsymbol{r} \bullet \boldsymbol{v}, 0)\}^f$$

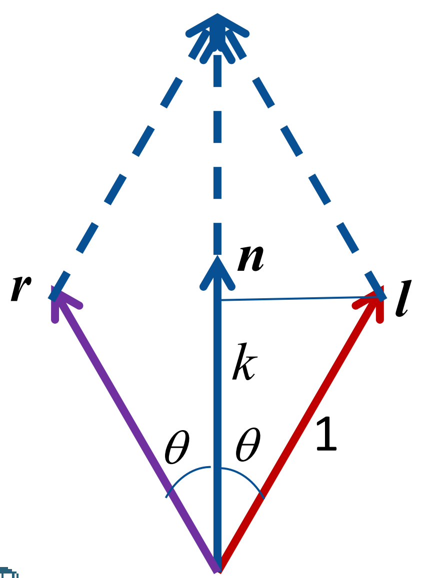


Large values of the exponent f gives highly concentrated specular highlights.

Computation of Reflection Vector

The computation of the specular component of lighting requires the reflection vector $\boldsymbol{r}$. It has the following properties:

- Vectors $\boldsymbol{l}$ and $\boldsymbol{r}$ make equal angles with the normal vector $\boldsymbol{n}$
- Vectors $\boldsymbol{l}$, $\boldsymbol{r}$ and $\boldsymbol{n}$ are on the same plane.



Let k be the length of projection of the unit vector $\boldsymbol{l}$ on vector $\boldsymbol{n}$.

$$k = \cos\theta = \boldsymbol{l} \bullet \boldsymbol{n}$$

The projection of $\boldsymbol{r}$ on the unit vector $\boldsymbol{n}$ also has the same length k.

$$\boldsymbol{r} + \boldsymbol{l} = 2k\,\boldsymbol{n}$$

Therefore,

$$\boldsymbol{r} = 2\,(\boldsymbol{l} \bullet \boldsymbol{n})\,\boldsymbol{n} - \boldsymbol{l}$$

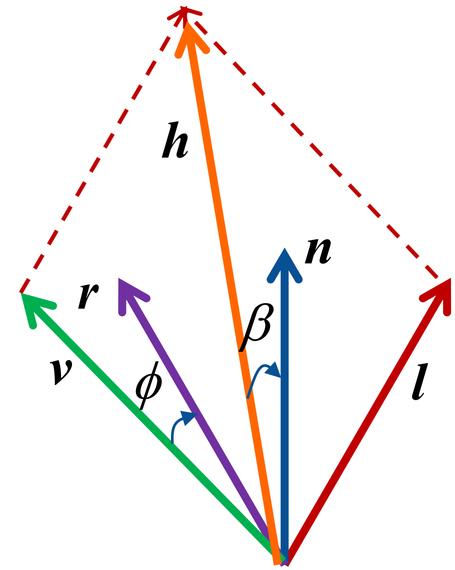
Half-way Vector

Consider the vector $\boldsymbol{h} = (\boldsymbol{l} + \boldsymbol{v})$ normalized.

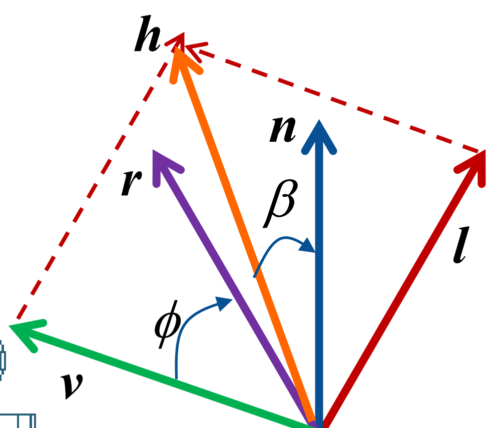
This vector is called the "half-way vector"

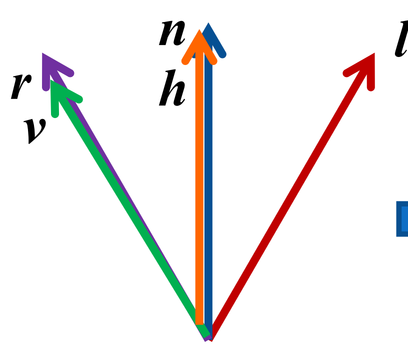
Let β be the angle between ***h*** and ***n***.

We observe the following facts:



When ***v*** is moved away from ***r***, the vector ***h*** moves away from ***n***.
i.e., β increases with ϕ.
When ϕ becomes 0, β also becomes 0.





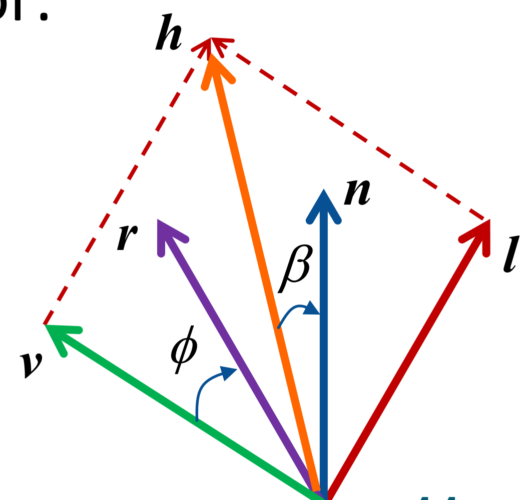
β varies with ϕ, and could be used as a substitute for ϕ.

Phong-Blinn Model

- OpenGL uses an approximation of $(\boldsymbol{r} \bullet \boldsymbol{v})$ by the term $(\boldsymbol{h} \bullet \boldsymbol{n})$ in the computation of specular reflections, where $\boldsymbol{h}$ is the **Half-way Vector**, computed as $\boldsymbol{h} = (\boldsymbol{l} + \boldsymbol{v})$ normalized.
- We can now rewrite the formula for specular reflection:

$$\boldsymbol{I}_s = L_s M_s \{\max(\boldsymbol{h} \bullet \boldsymbol{n}), 0\}^f$$

- The lighting equation with the above approximation is called the **Phong-Blinn** model.
- The advantages of using the half-way vector:
  1. Easier to compute 'h' compared to 'r'.
  2. If $\boldsymbol{l}$ is a directional source, and the view direction is constant (viewer at infinity), then $\boldsymbol{h}$ needs to be computed only once for the whole scene.



Lighting Equation: Phong-Blinn Model

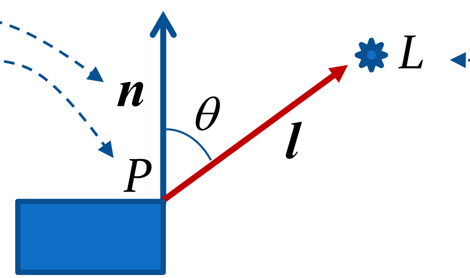
Vertex Colour

$$V_C = \underbrace{L_a M_a}_{\text{Ambient Term}} + \underbrace{L_d M_d \max(\boldsymbol{l} \bullet \boldsymbol{n}\ ,\ 0)}_{\text{Diffuse Term}} + \underbrace{L_s M_s \{\max(\boldsymbol{h} \bullet \boldsymbol{n}\ ,\ 0)\}^f}_{\text{Specular Term}}$$

Lighting Equation: Phong-Blinn Model

```
void drawCube() {
   glBegin(GL_QUADS);
      glNormal3f(0, 0, 1);
      glVertex3f(-10, 0, 10);
      glVertex3f(10, 0, 10);
        ...
```

```
glLightfv(GL_LIGHT0, GL_POSITION, lgt_pos);
```



$\boldsymbol{l} = L - P$

$\boldsymbol{h} = \boldsymbol{l} + \boldsymbol{v}$

```
glLightfv(GL_LIGHT0, GL_AMBIENT, grey);
glLightfv(GL_LIGHT0, GL_DIFFUSE, white);
glLightfv(GL_LIGHT0, GL_SPECULAR, white);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white);
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,cyan);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 100);
```

L_a

L_d

L_s

M_s

M_a, M_d

f

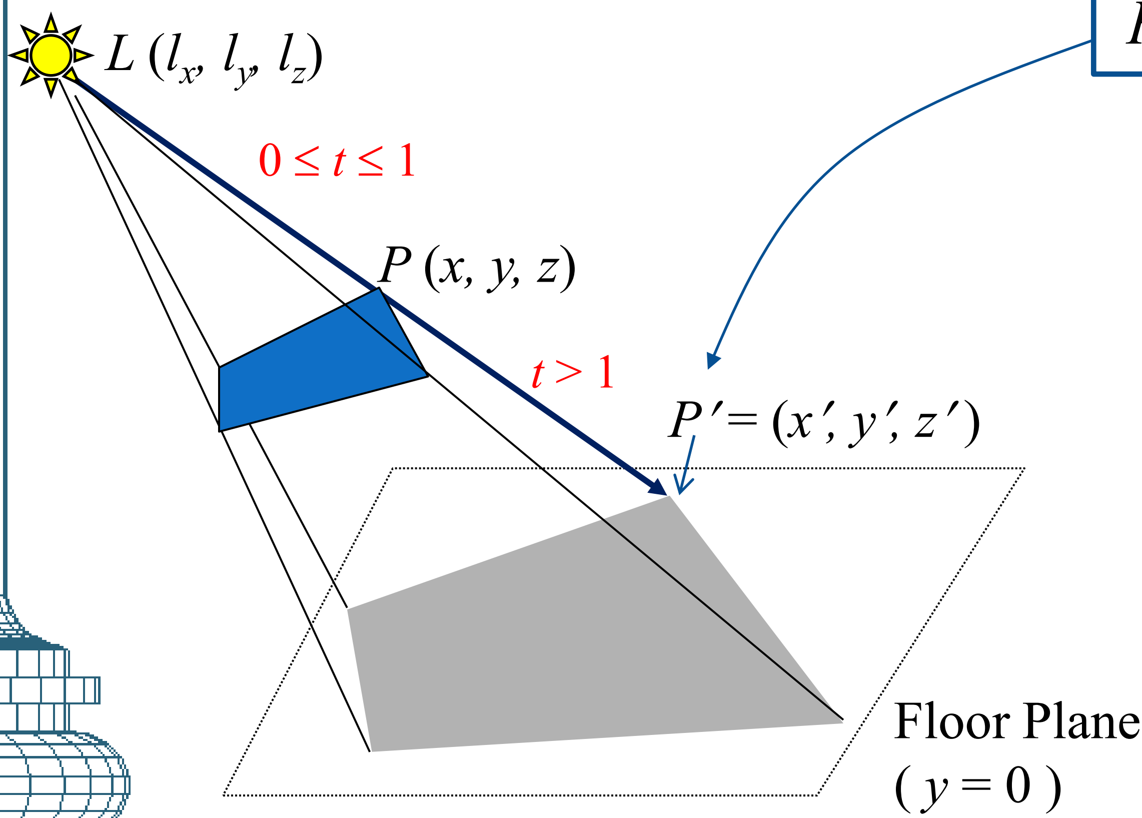
$$V_C = L_a M_a + L_d M_d \max(\boldsymbol{l} \bullet \boldsymbol{n}, 0) + L_s M_s \{\max(\boldsymbol{h} \bullet \boldsymbol{n}, 0)\}^f$$

Planar Shadows

(*See also slides* [3]:19-21)

- Project each of the polygonal faces onto the floor plane, using the light source (*L*) as the centre of projection.
- Use only the ambient light to draw the projected object.

*
$$\boxed{P' = (1-t)L + tP, \qquad t > 1.}$$



$$x' = (1-t)l_x + tx$$

$$y' = (1-t)l_y + ty = 0$$

$$z' = (1-t)l_z + tz$$

$$\therefore t = \frac{l_y}{l_y - y}$$

* Linear interpolation: See [6]-27

Planar Shadows

- The projection P' of the vertex (x, y, z) on the floor- plane is given by the following coordinates:

$$\left.\begin{aligned} x' &= \frac{-l_x y + l_y x}{l_y - y} \\ y' &= 0 \\ z' &= \frac{-l_z y + l_y z}{l_y - y} \end{aligned}\right\} \xrightarrow{\text{Homogeneous Coordinates}} \begin{aligned} s_x &= -l_x y + l_y x \\ s_y &= 0 \\ s_z &= -l_z y + l_y z \\ w &= l_y - y \end{aligned}$$

- The above equations can be written as a transformation:

$$\begin{bmatrix} s_x \\ s_y \\ s_z \\ w \end{bmatrix} = \begin{bmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Planar Shadows: Code

(*See also slide* [3]:21)

```
// Light source position = (lx, ly, lz)
float shadowMat[16] = { ly,0,0,0, -lx,0,-lz,-1,
                        0,0,ly,0,  0,0,0,ly };
// Draw object
 glEnable(GL_LIGHTING);
 glPushMatrix();      //Draw Actual Object
    /* Transformations */
    drawObject();
 glPopMatrix();

//  Draw shadow
 glDisable(GL_LIGHTING);
 glPushMatrix();      //Draw Shadow Object
   glMultMatrixf(shadowMat);
   /* Transformations */
   glColor4f(0.2, 0.2, 0.2, 1.0);
   drawObject();
 glPopMatrix();
```

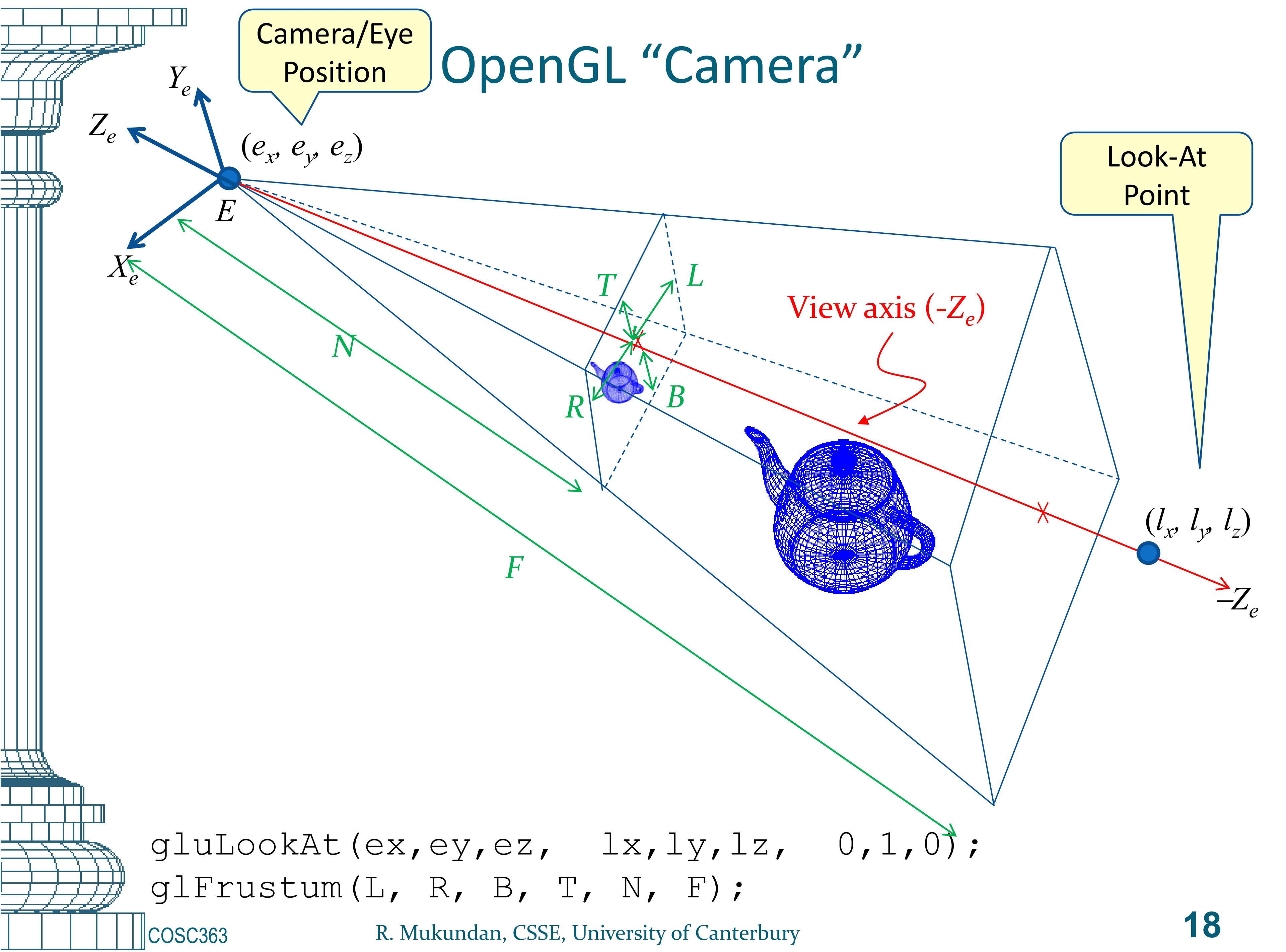
Camera View and Projection

- Camera **View**
  - Depends on the camera's position and orientation
  - Specified by a view transformation matrix ***V*** generated by the function gluLookAt(ex, ey, ez, lx, ly, lz, ux, uy, uz);
  - Note: gluLookAt() represents a view matrix, not the camera's position.
- Camera **Projection**
  - Depends on the field of view and focal length.
  - Specified by a projection matrix ***P*** generated by

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

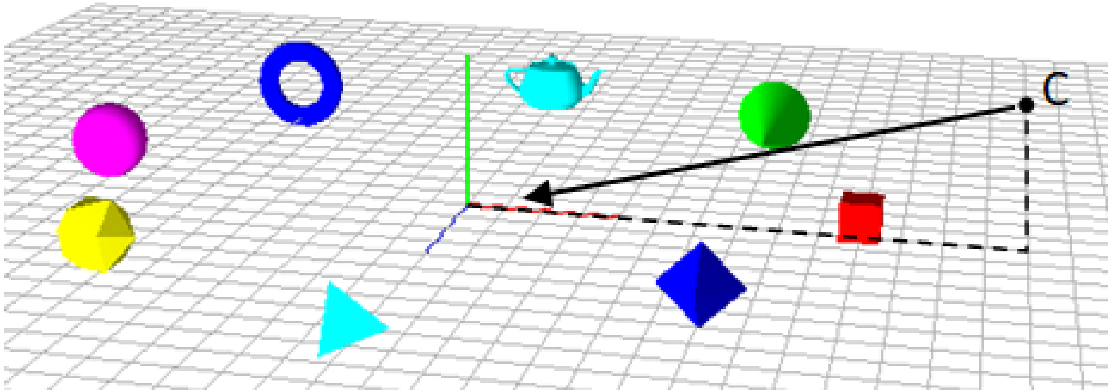
    or, gluPerspective(fov, ar, N, F)

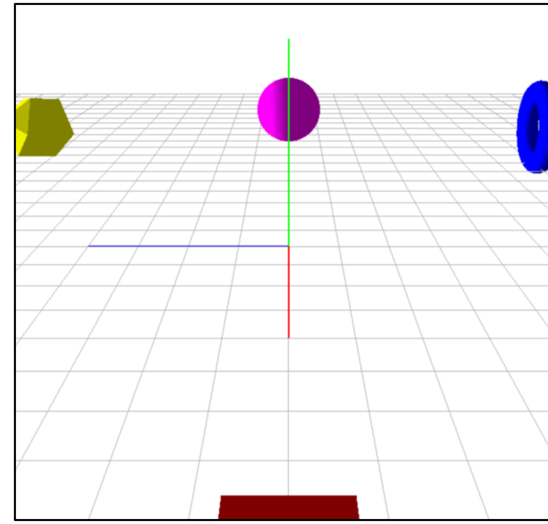
OpenGL "Camera"



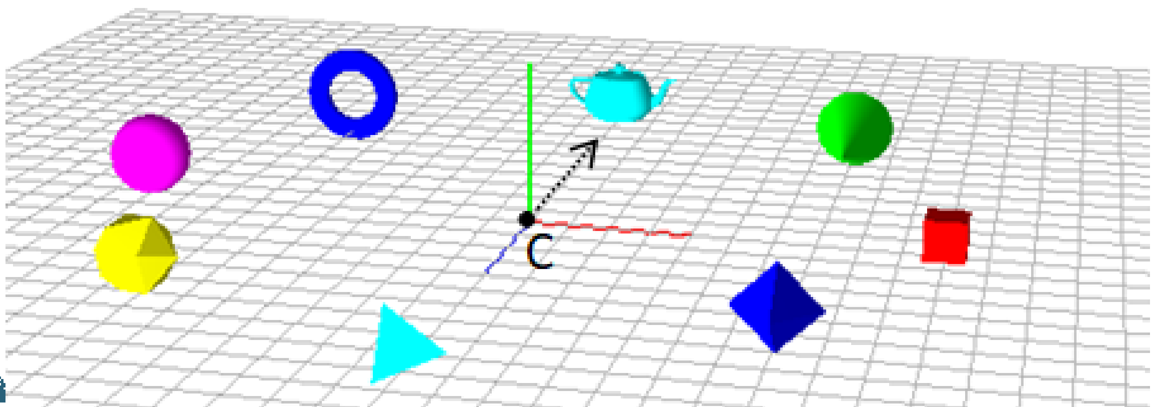

```
gluLookAt(ex,ey,ez,  lx,ly,lz,  0,1,0);
glFrustum(L, R, B, T, N, F);
```

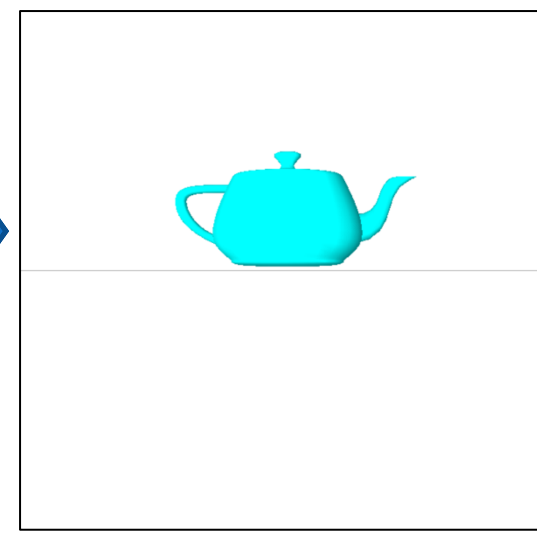
The Default Camera





```
gluLookAt(14,5,0,   0,0,0   0,1,0);
```



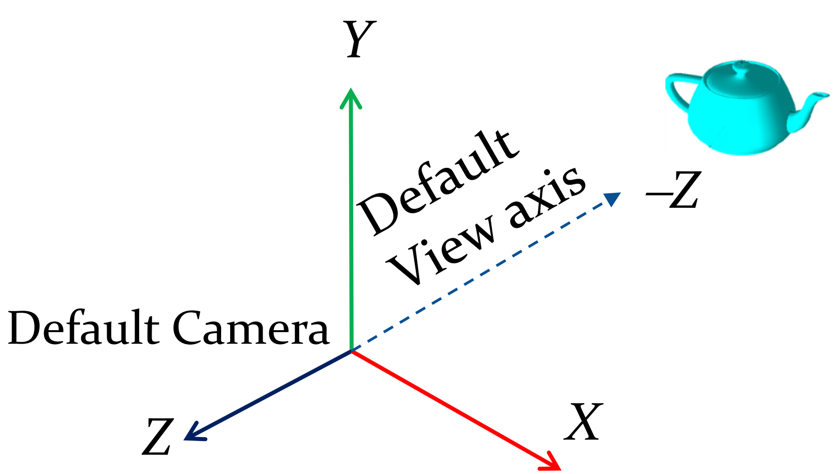


Default Camera

Camera View

If gluLookAt() represents a transformation from the world coordinate space to the camera-centered coordinate system, where the camera is at the origin, and the view axis is along the negative Z_e direction.

If gluLooAt() function is not used, then the camera axes (X_e, Y_e, Z_e) coincide with (X, Y, Z) . This corresponds to the **default camera view**, where the camera is at the origin, looking towards the **negative *z*-axis of the world space**.

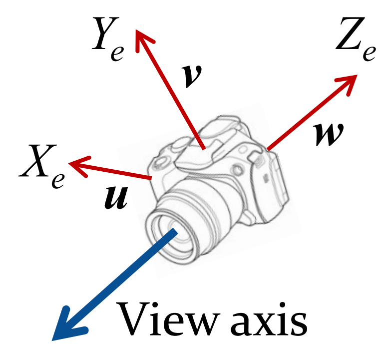


View Transformation

The view transformation matrix generated by the function `gluLookAt(...)` is given below:

Eye Coordinates

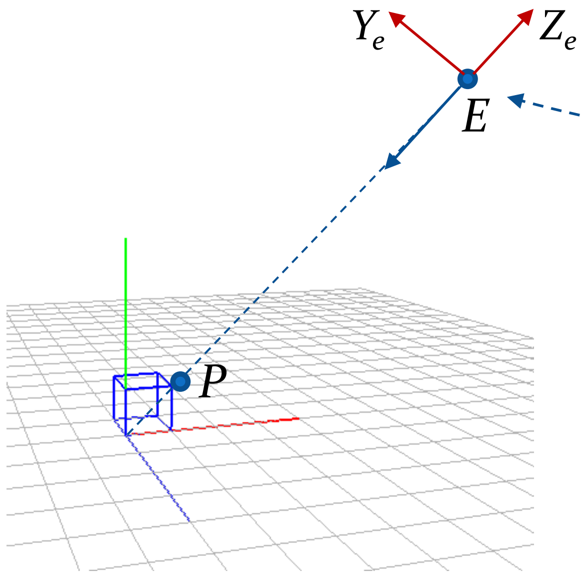
$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & -e_x u_x - e_y u_y - e_z u_z \\ v_x & v_y & v_z & -e_x v_x - e_y v_y - e_z v_z \\ w_x & w_y & w_z & -e_x w_x - e_y w_y - e_z w_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



- ***u*, *v*, *w*** are unit vectors along camera axes X_e, Y_e, Z_e respectively.
- The view matrix transforms points from world-coordinate space to the eye-coordinate space (see example on next slide)
- Obtaining the view matrix:

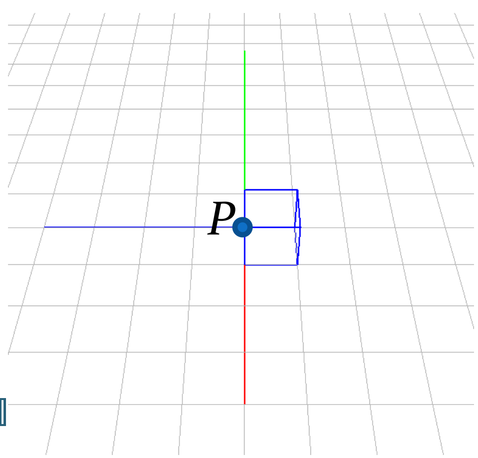
```
float mat[16];
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(ex, ey, ez, lx, ly, lz,  0.,1.,0.);
glGetFloatv(GL_MODELVIEW_MATRIX, mat);
```

View Transformation Example




```
gluLookAt(10,10,0,  0,0,0,  0,1,0);
```

The camera is placed at (10, 10, 0), looking at the origin. The point P on the cube has coordinates (1, 1, 0). This point is along the view axis of the camera.
The view of the scene from the camera is given below.
In this view, the point P can be seen along the view axis. Therefore, for this point, $x_e = 0$, $y_e = 0$. z_e will have a negative value, giving the distance of the point P from E. Every point in the field of view of the camera will have a negative value for z_e.



Camera View

View Matrix: $\begin{bmatrix} 0 & 0 & -1 & 0 \\ -0.707 & 0.707 & 0 & 0 \\ 0.707 & 0.707 & 0 & -14.1421 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The above matrix converts the coordinates of P

from $\begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}$ (World coordinates) to $\begin{bmatrix} 0 \\ 0 \\ -12.727 \\ 1 \end{bmatrix}$ (Eye coordinates)

View Volumes

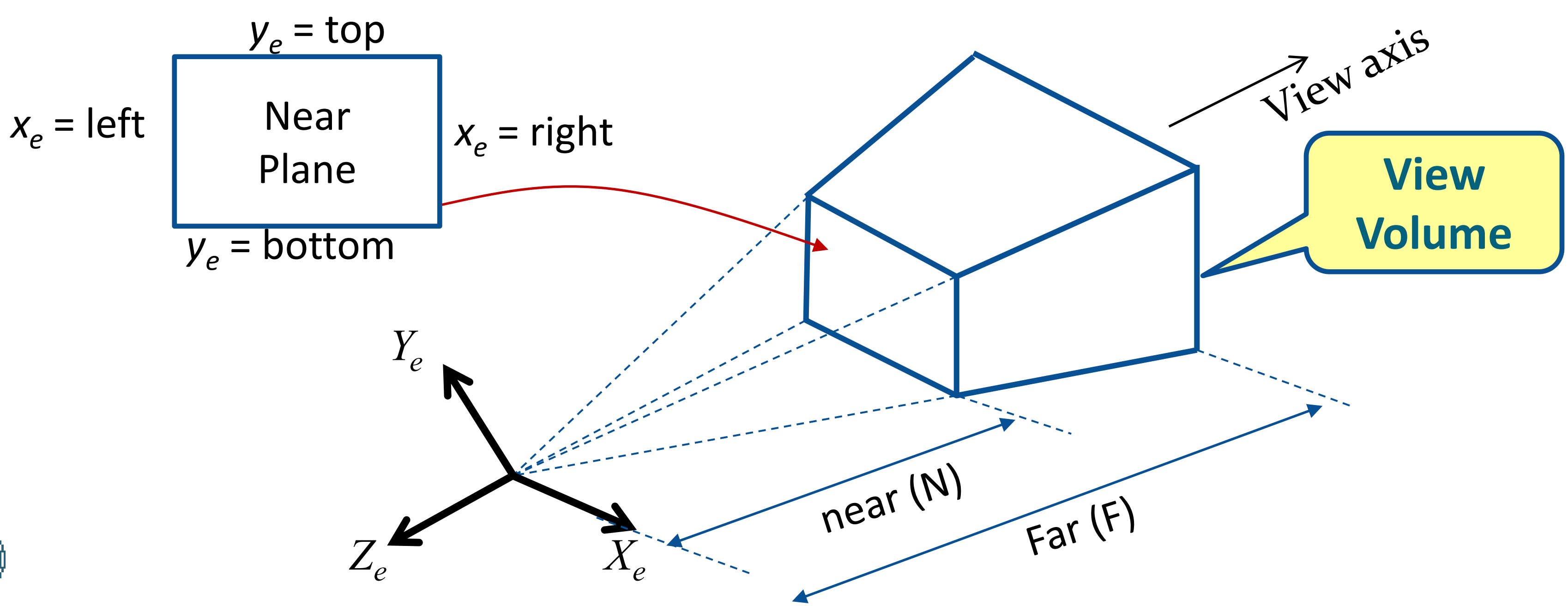
- The view transformation only transforms the world coordinates of points into the camera's coordinate frame.
- We need to specify "how much" the camera actually sees. That is, we require a view volume that contains the part of the scene that is visible to the camera. In other words, the view volume acts as a **clipping volume**.
- We further require a projection model to simulate the way one would "see" the 3D scene through the camera.
- The **projection matrix** is defined using the frustum parameters. It transforms eye coordinates to **clip coordinates.** The clip coordinates of a point will have values in the range [-1, +1] if the point is inside the view frustum.

Perspective View Volume

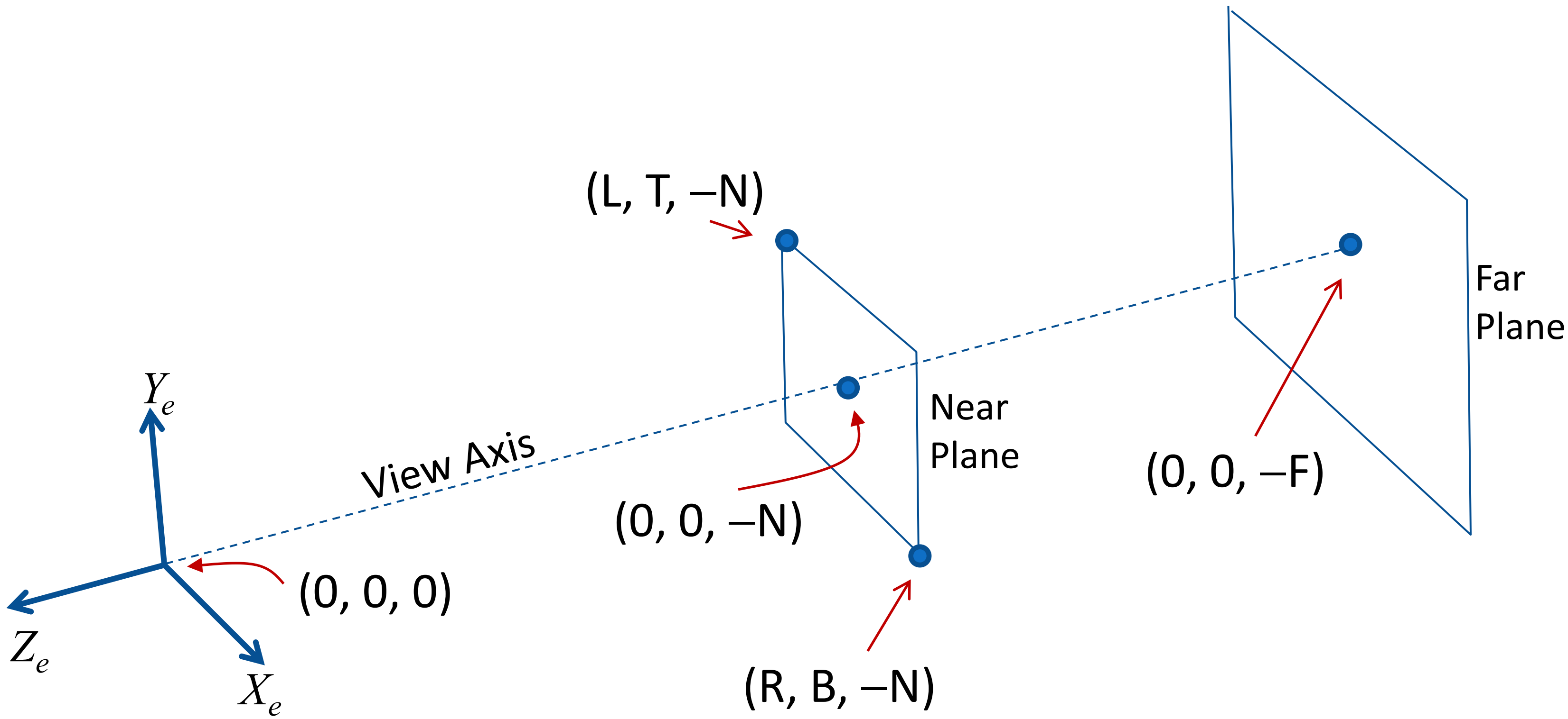
- The perspective view volume is defined by a frustum that has its vertex at the eye position. The near-plane acts as the plane of projection.
- OpenGL function:

Both values are positive

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



View Volume: Eye Coordinates



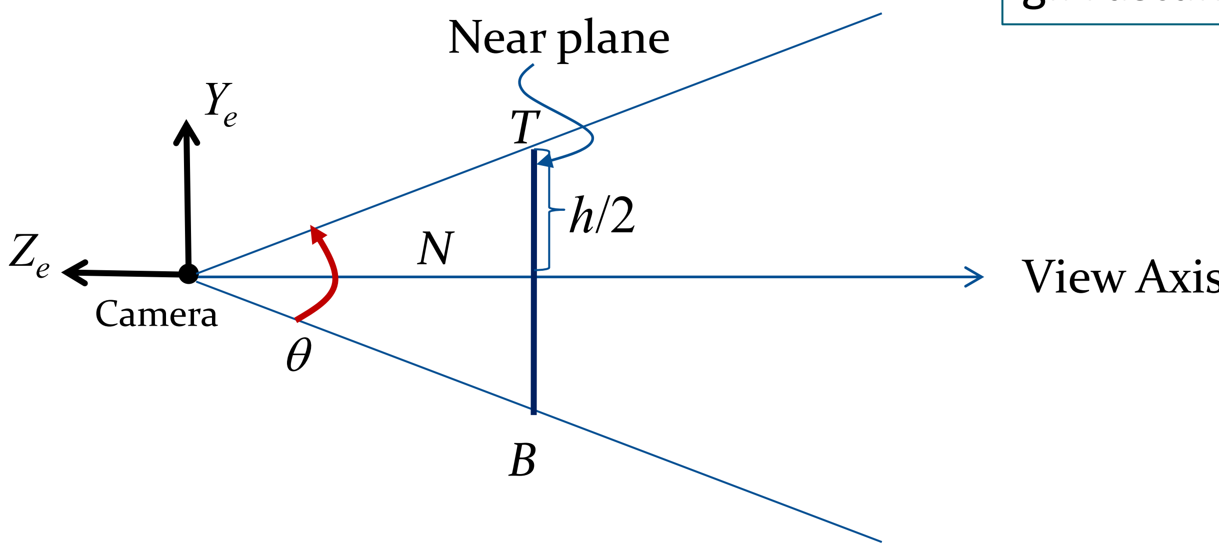

```
glFrustum(L, R, B, T, N, F);
```

Eye coordinates of a few points in the camera's view volume

Perspective View

- 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)



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

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

$h = T - B$

gluPerspective

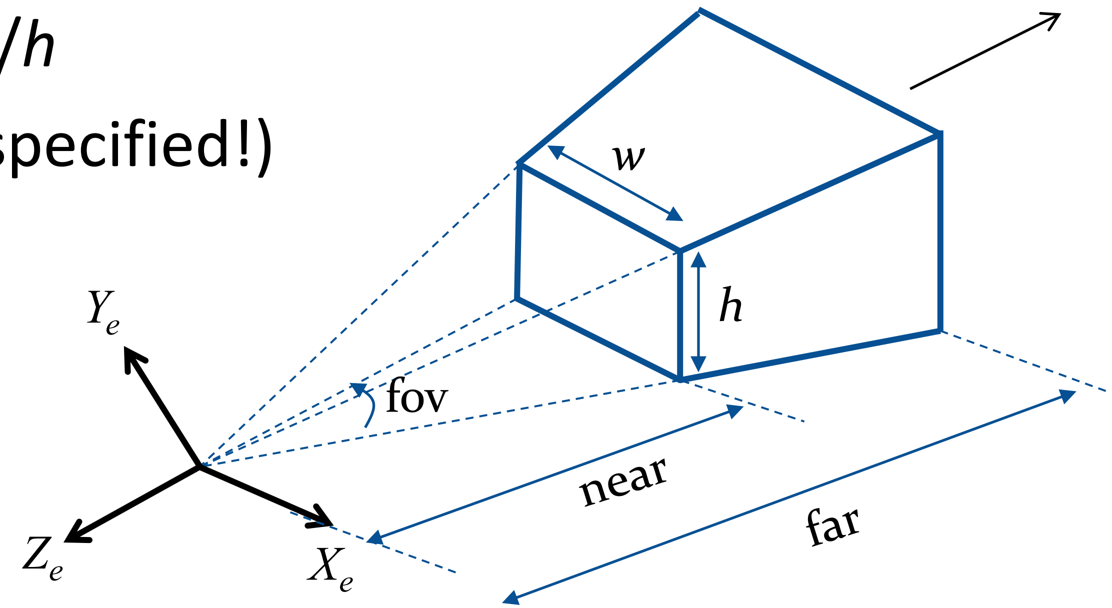
- The GLU library provides another function for perspective transformation in the form

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

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

(Note: *w*, *h* are not specified!)

- fov = θ

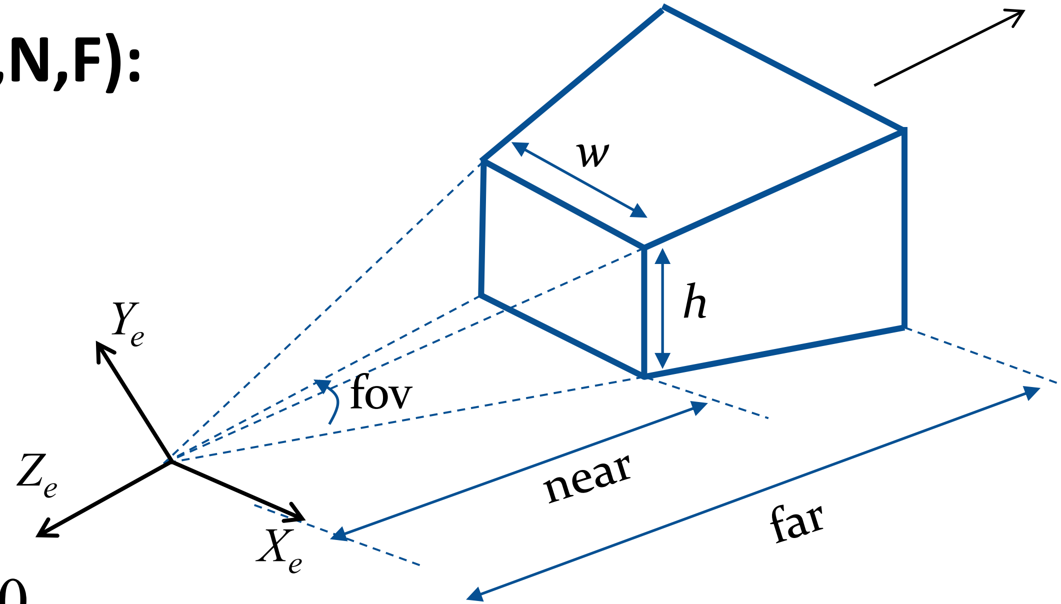


gluPerspective vs. glFrustum

- **$(\theta, a, N, F) \rightarrow (L,R,B,T,N,F)$:**

$$h = 2N\tan\left(\frac{\theta}{2}\right)$$ (Slide 26)

$$w = a.\,h$$
$$L = -w/2 \qquad R = w/2$$
$$B = -h/2 \qquad T = h/2,$$
$$\text{Note: } L + R = 0, \quad B + T = 0$$

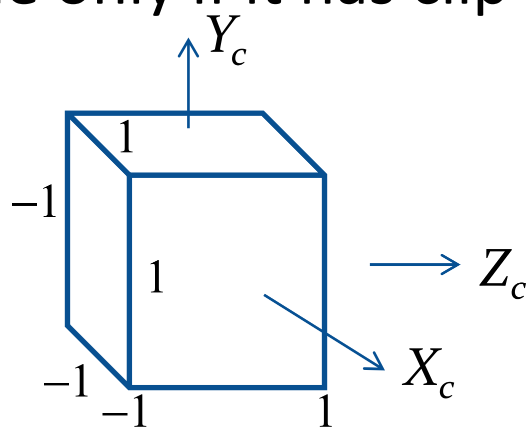


- **$(L,R,B,T,N,F) \rightarrow (\theta, a, N, F)$:**

$$w = R - L\,, \qquad h = T - B$$
$$a = w/h$$
$$\theta = 2\tan^{-1}\left(\frac{h}{2N}\right)$$

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.

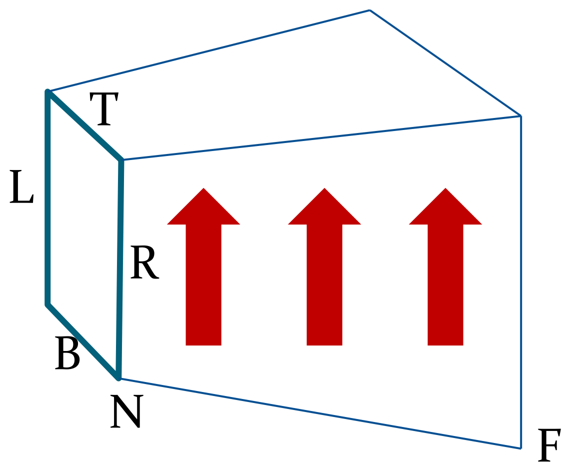


Clip Coordinate Axes

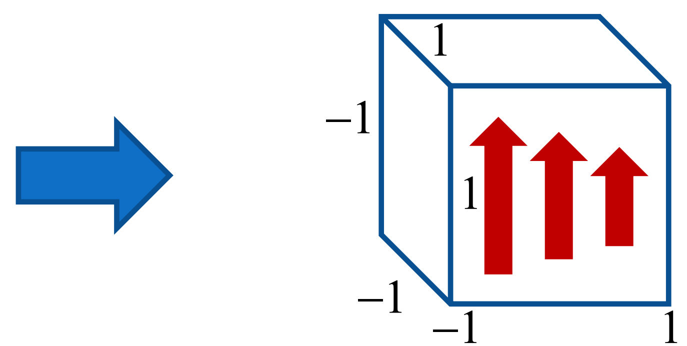
Left handed system

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

- 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].

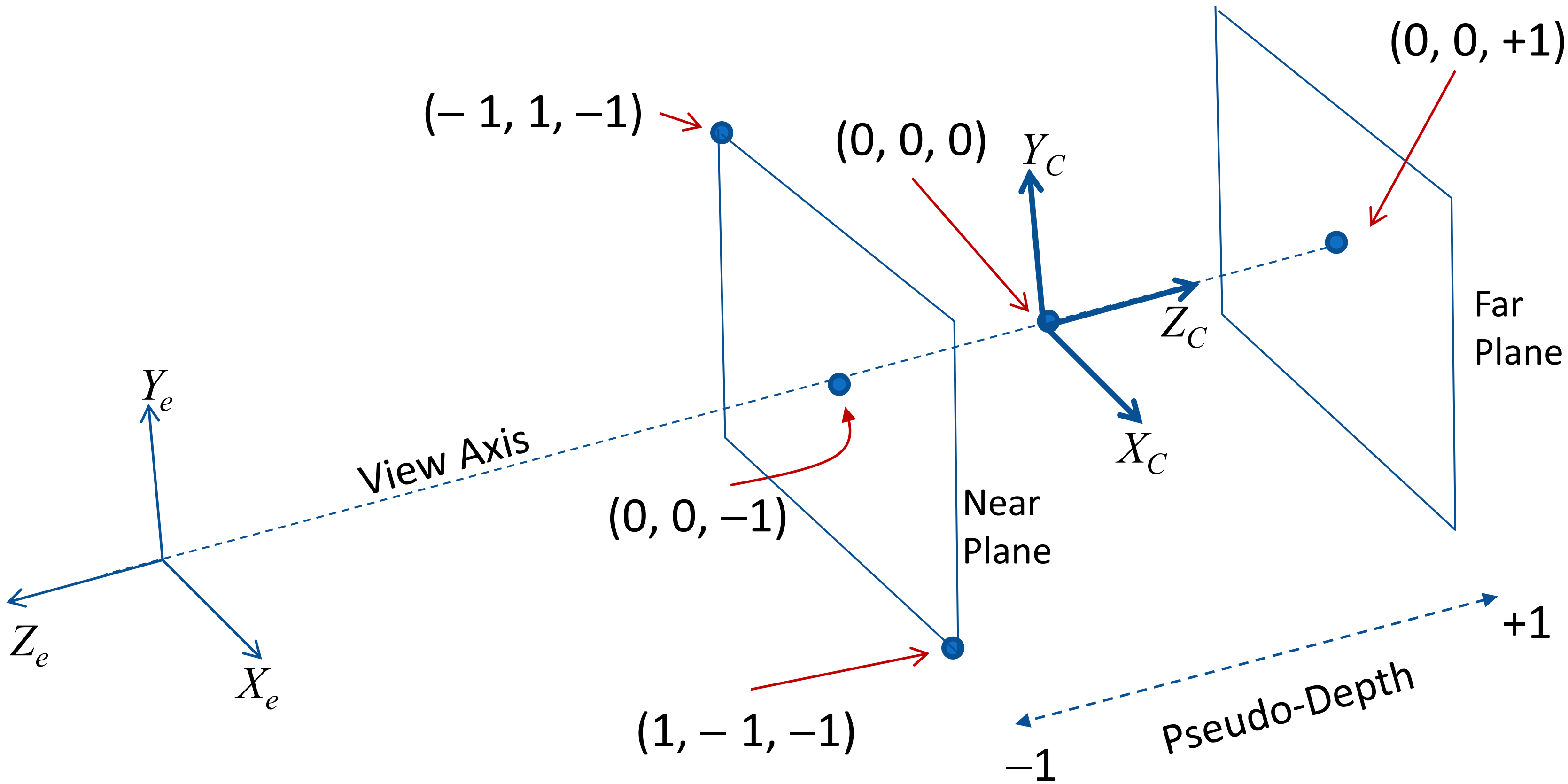


Eye coordinates



Clip coordinates

View Volume: **Clip** Coordinates

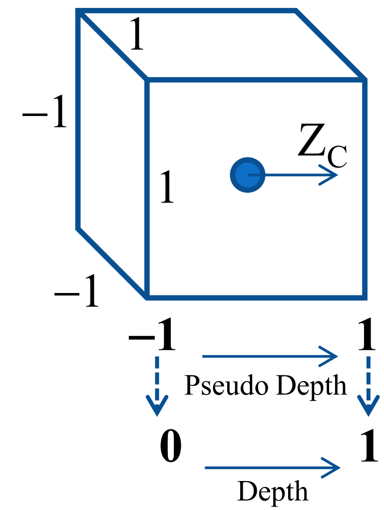


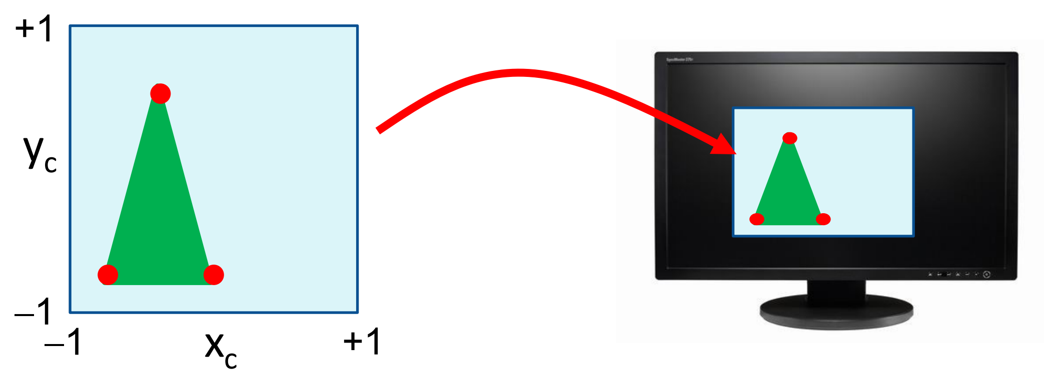
Clip coordinates of a few points in the camera's view volume

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 $z_{depth} = (z_c + 1)/2$
- 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