Computer Graphics (CS 4731)
Lecture 12: Viewing & Camera Control

Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

Finding Vector Reflected From a Surface

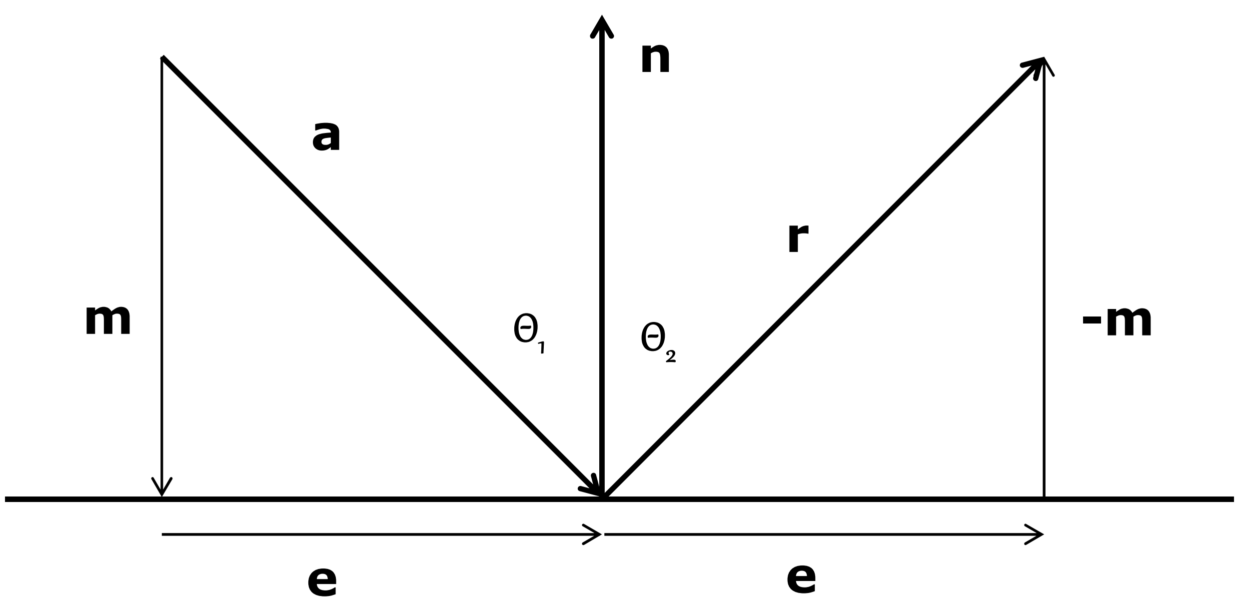
- **a** = original vector
- **n** = normal vector
- **r** = reflected vector
- **m** = projection of **a** along **n**
- **e** = projection of **a** orthogonal to **n**

Note: $\Theta_1 = \Theta_2$

$$\mathbf{e} = \mathbf{a} - \mathbf{m}$$

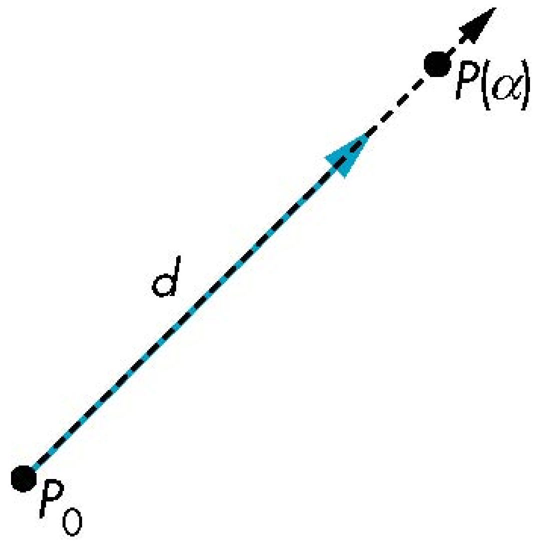
$$\mathbf{r} = \mathbf{e} - \mathbf{m}$$

$$=> \mathbf{r} = \mathbf{a} - 2\mathbf{m}$$



Lines

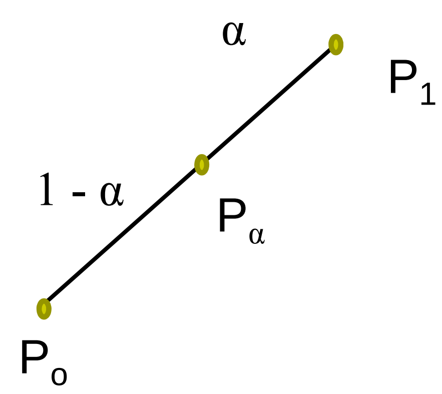
- Consider all points of the form
  - $P(\alpha)=P_0 + \alpha\, \mathbf{d}$
  - **Line:** Set of all points that pass through P_0 in direction of vector $\mathbf{d}$



Parametric Form

- Two-dimensional forms of a line
  - **Explicit:** $y = mx + h$
  - **Implicit:** $ax + by + c = 0$
  - **Parametric:**

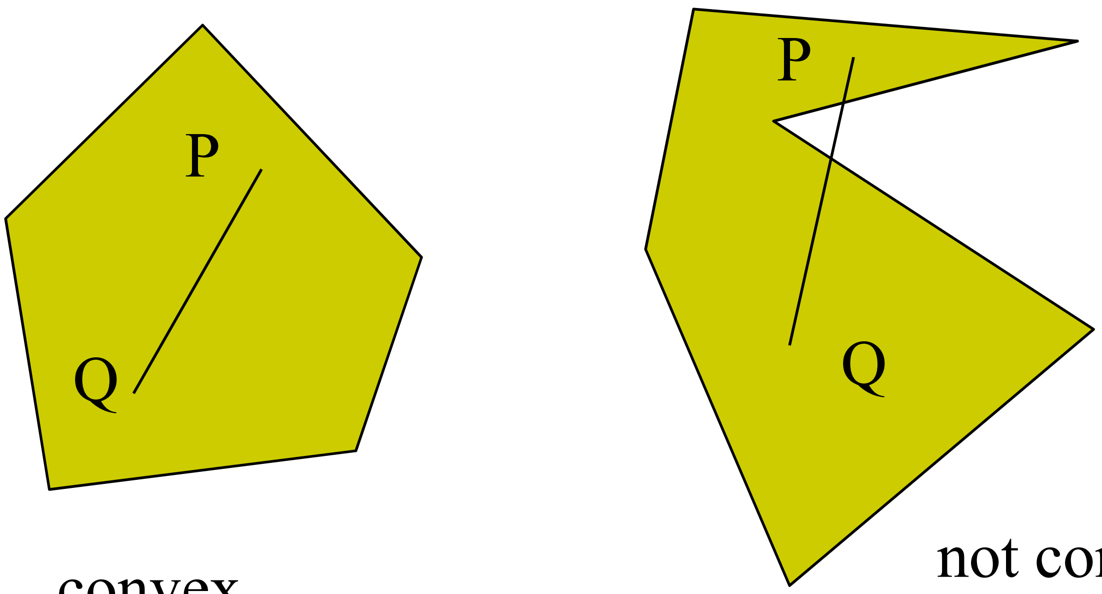
$$x(\alpha) = \alpha x_0 + (1-\alpha)x_1$$
$$y(\alpha) = \alpha y_0 + (1-\alpha)y_1$$



- Parametric form of line
  - More robust and general than other forms
  - Extends to curves and surfaces

Convexity

- An object is *convex* iff for any two points in the object all points on the line segment between these points are also in the object



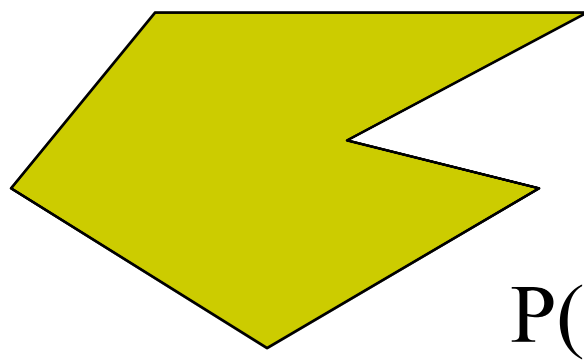
convex

not convex

Curves and Surfaces

- **Curves:** 1-parameter **non-linear** functions of the form $P(\alpha)$
- **Surfaces:** two-parameter functions $P(\alpha, \beta)$
  - Linear functions give planes and polygons

$P(\alpha)$



$P(\alpha, \beta)$

Computer Graphics (CS 4731)
Lecture 12: Viewing & Camera Control

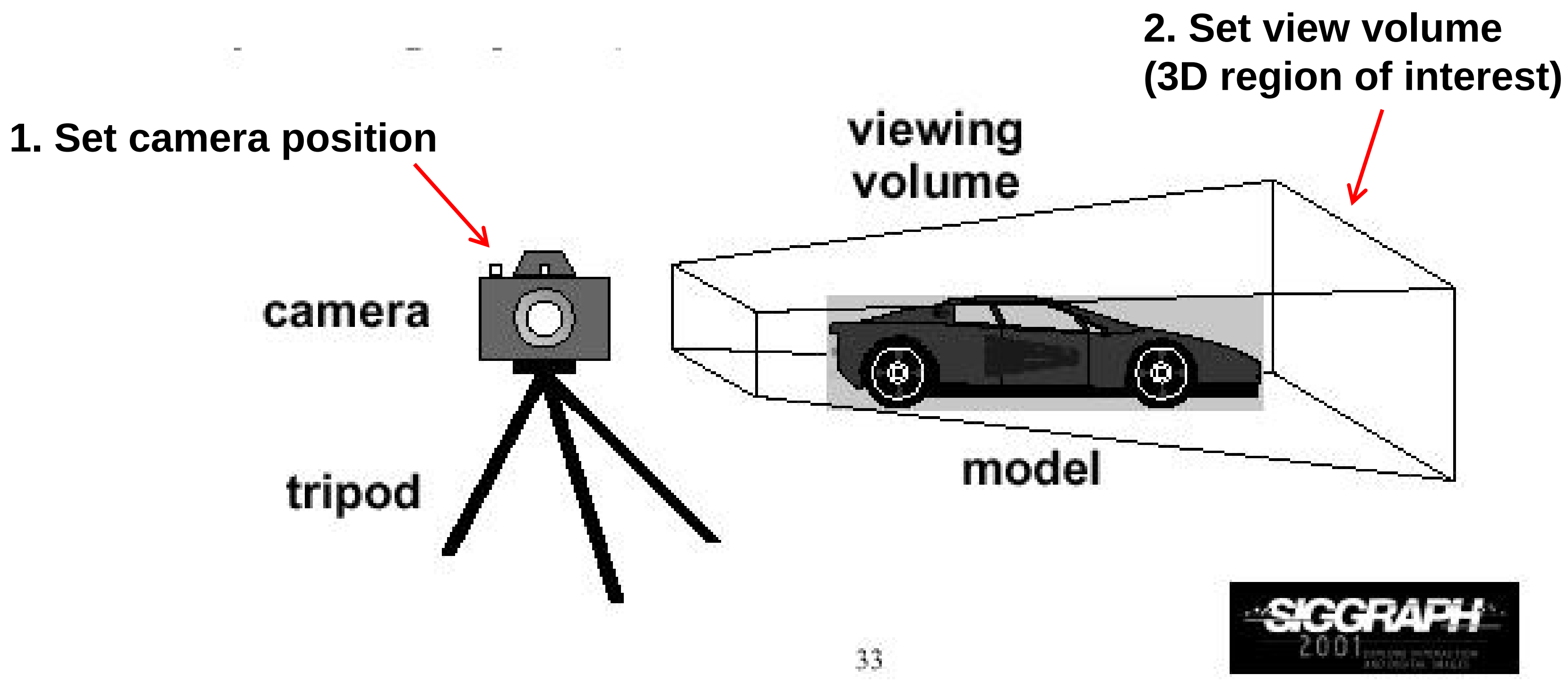
Prof Emmanuel Agu

Computer Science Dept.

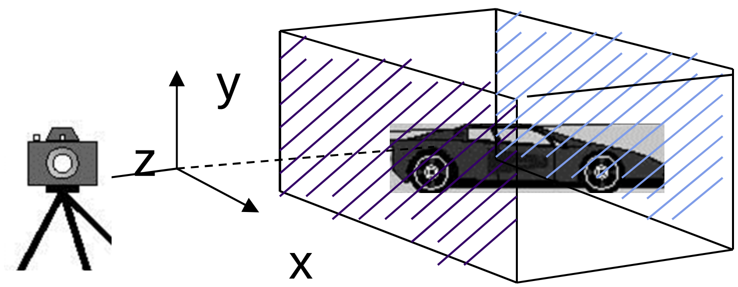
Worcester Polytechnic Institute (WPI)

3D Viewing?

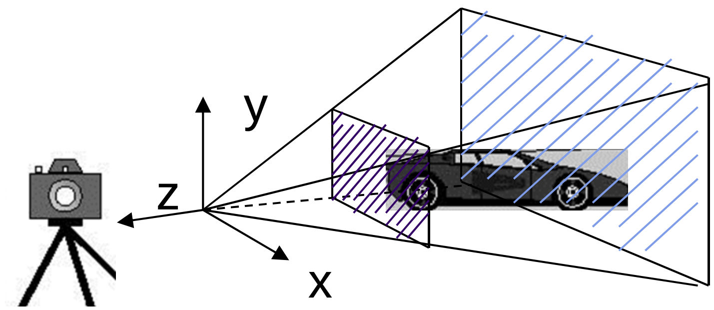
- Objects **inside** view volume show up on screen
- Objects outside view volume **clipped!**



Different View Volume Shapes



Orthogonal view volume

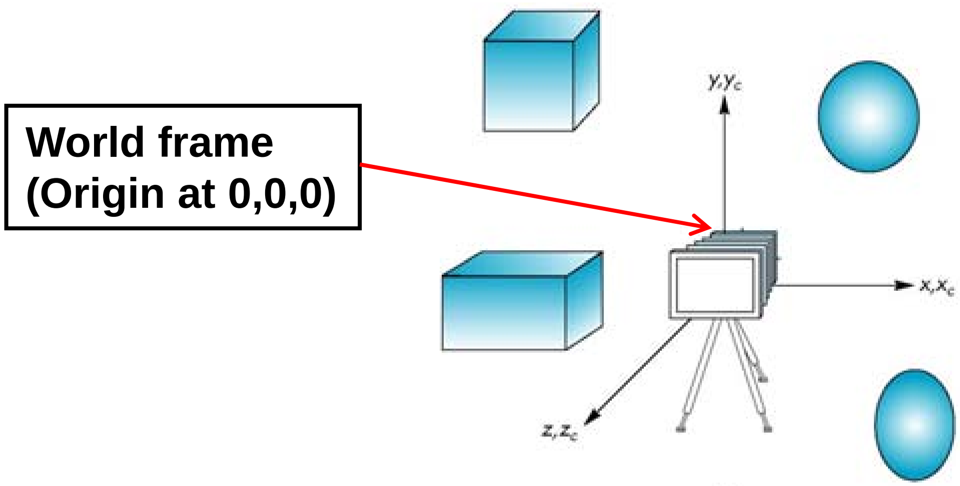


Perspective view volume

- Different view volume => different look
- **Foreshortening?** Near objects bigger
  - Perpective projection has **foreshortening**
  - Orthogonal projection: no foreshortening

The World Frame

- Objects/scene initially defined in **world frame**
- Objects positioned, transformations (translate, scale, rotate) applied to objects in **world frame**



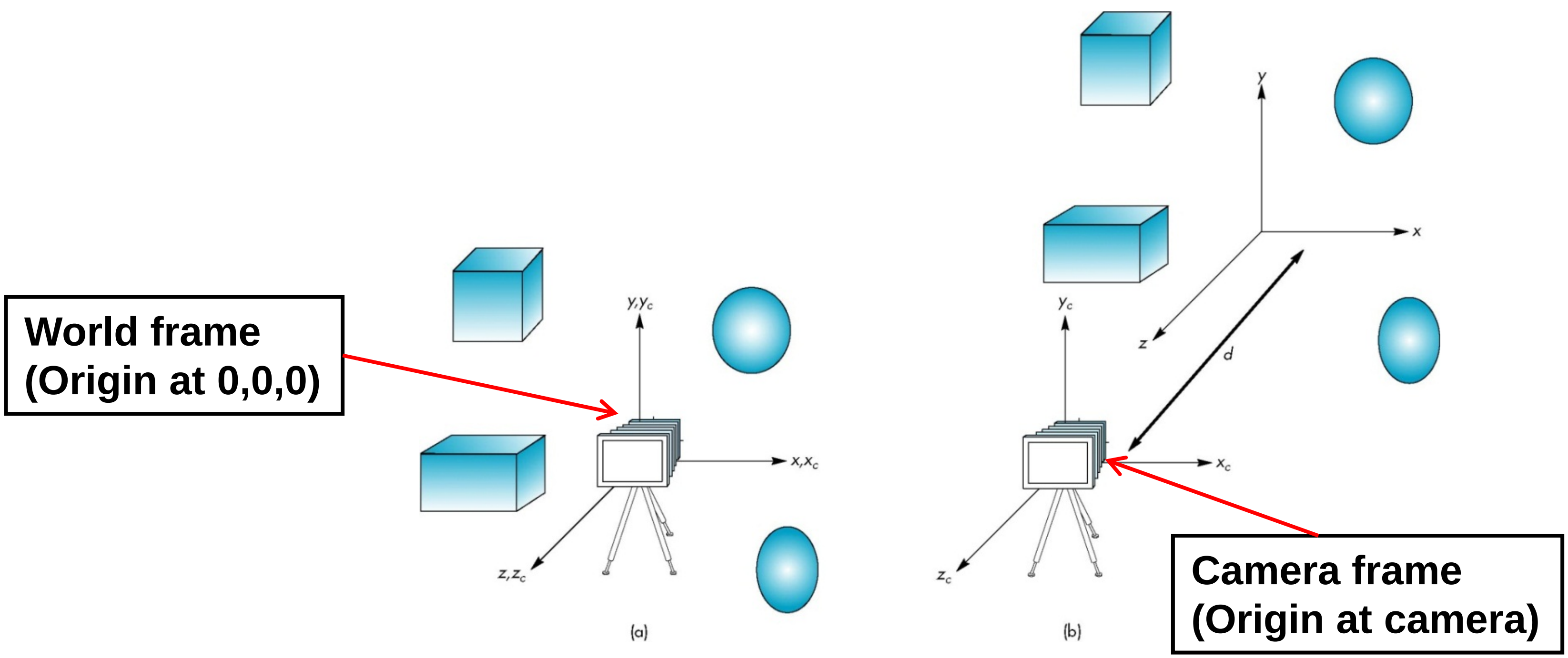
(a)

Camera Frame

- More natural to describe object positions relative to camera (eye)
- Think about
  - Our view of the world
  - First person shooter games

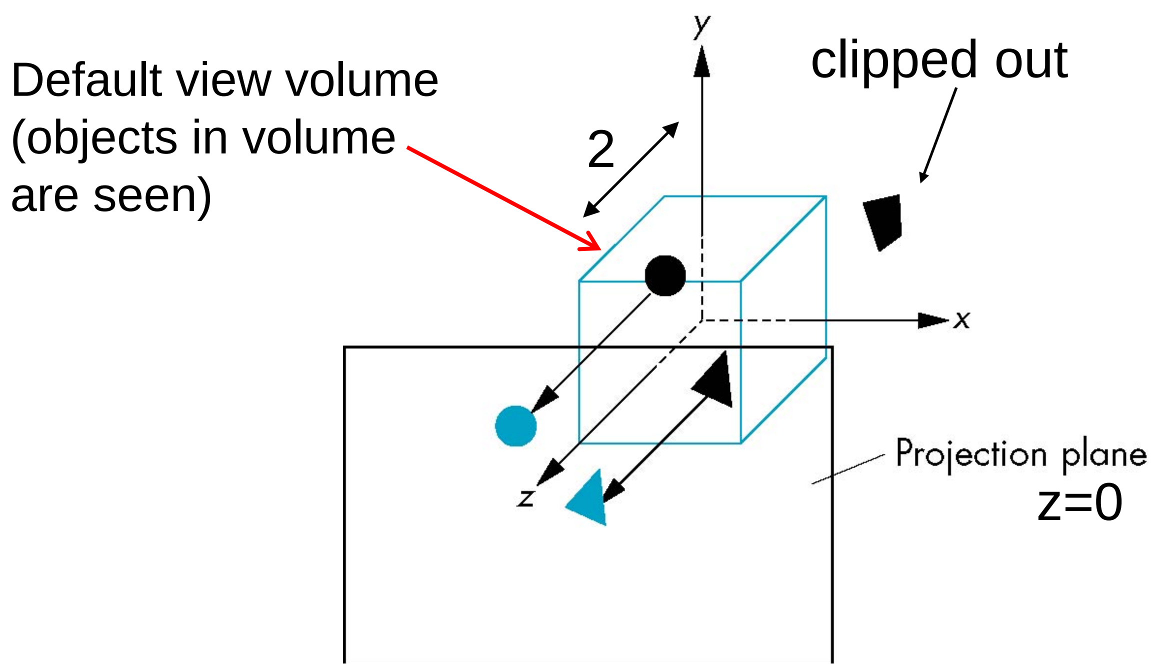
Camera Frame

- **Viewing:** After user sets camera (eye) position, represent objects in **camera frame** (origin at eye position)
- **Viewing transformation:** Changes object positions from world frame to positions in camera frame using **model-view matrix**



Default OpenGL Camera

- Initially Camera at origin: object and camera frames same
- Camera located at origin and points in negative z direction
- Default view volume is cube with sides of length 2



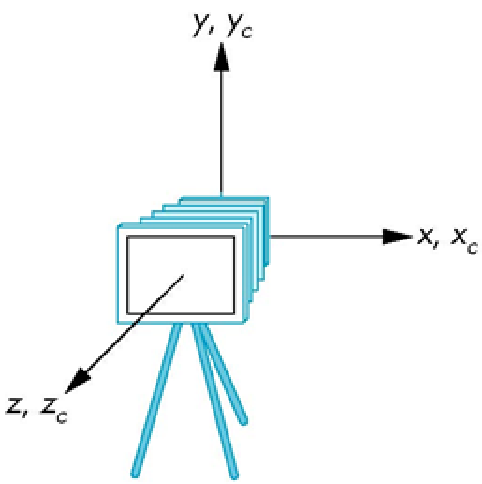
Moving Camera Frame

Same relative distance after
Same result/look

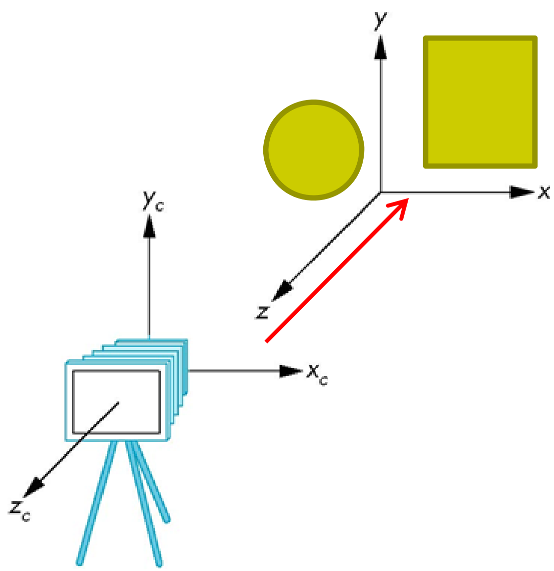
Translate objects +5
away from camera

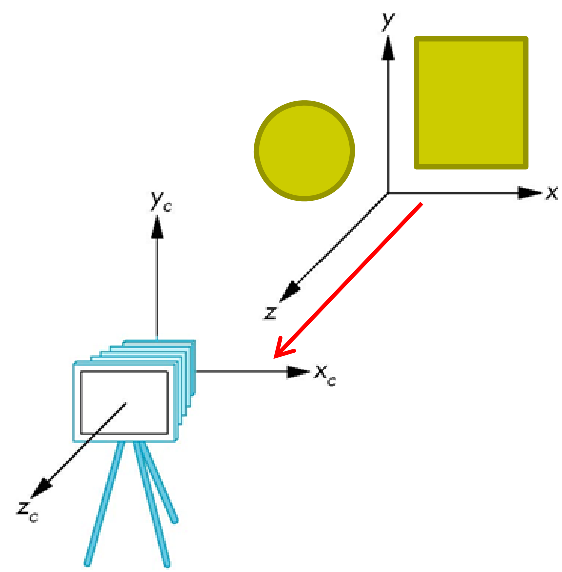
Translate camera -5
away from objects

default frames



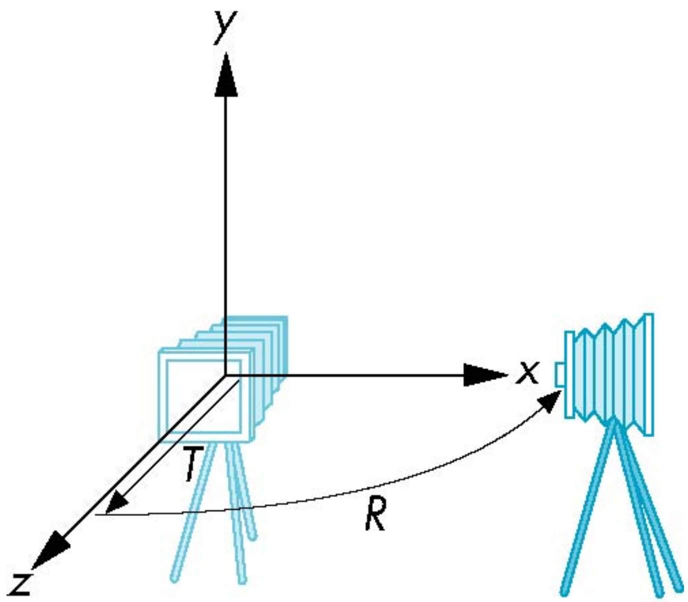
(a)





Moving the Camera

- We can move camera using sequence of rotations and translations
- Example: side view
  - Rotate the camera
  - Move it away from origin
  - Model-view matrix C = TR

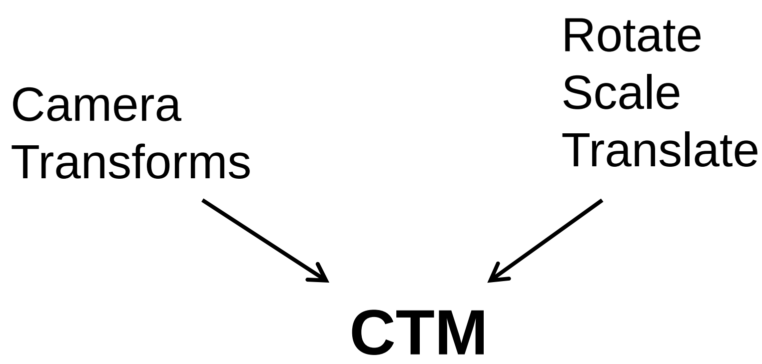



```
// Using mat.h

mat4 t = Translate (0.0, 0.0, -d);
mat4 ry = RotateY(90.0);
mat4 m = t*ry;
```

Moving the Camera Frame

- Object distances **relative to camera** determined by the model-view matrix
  - Transforms (scale, translate, rotate) go into **modelview matrix**
  - Camera transforms also go in **modelview matrix (CTM)**



The LookAt Function

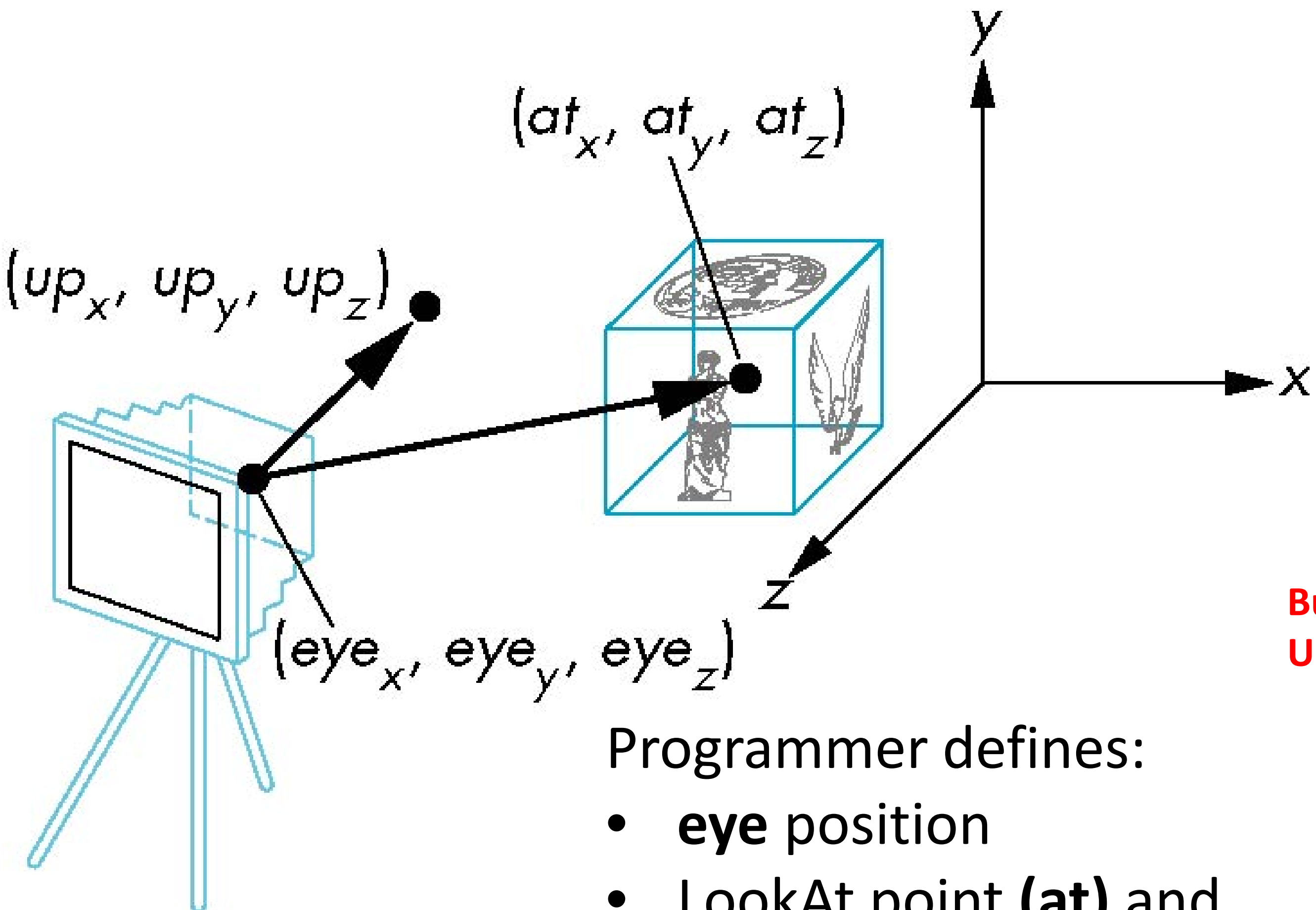
- Previously, command **gluLookAt** to position camera
- **gluLookAt** deprecated!!
- Homegrown mat4 method LookAt() in mat.h
  - Can concatenate with modeling transformations

```
void display( ){
    ………

    mat4 mv = LookAt(vec4 eye, vec4 at, vec4 up);
    …….. .
}
```

LookAt

```
LookAt(eye, at, up)
```

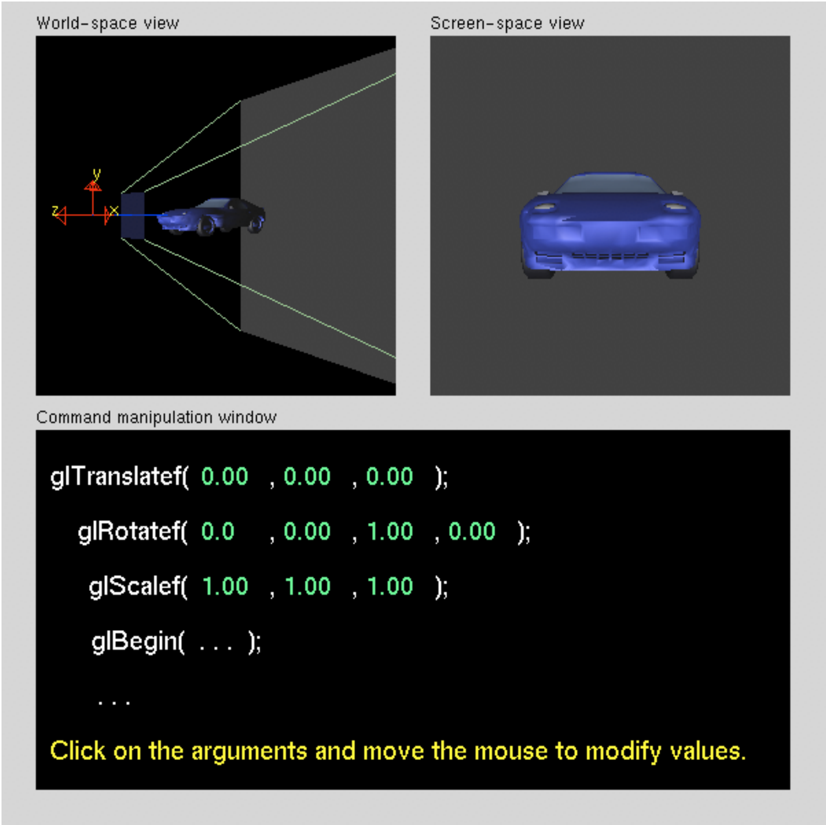


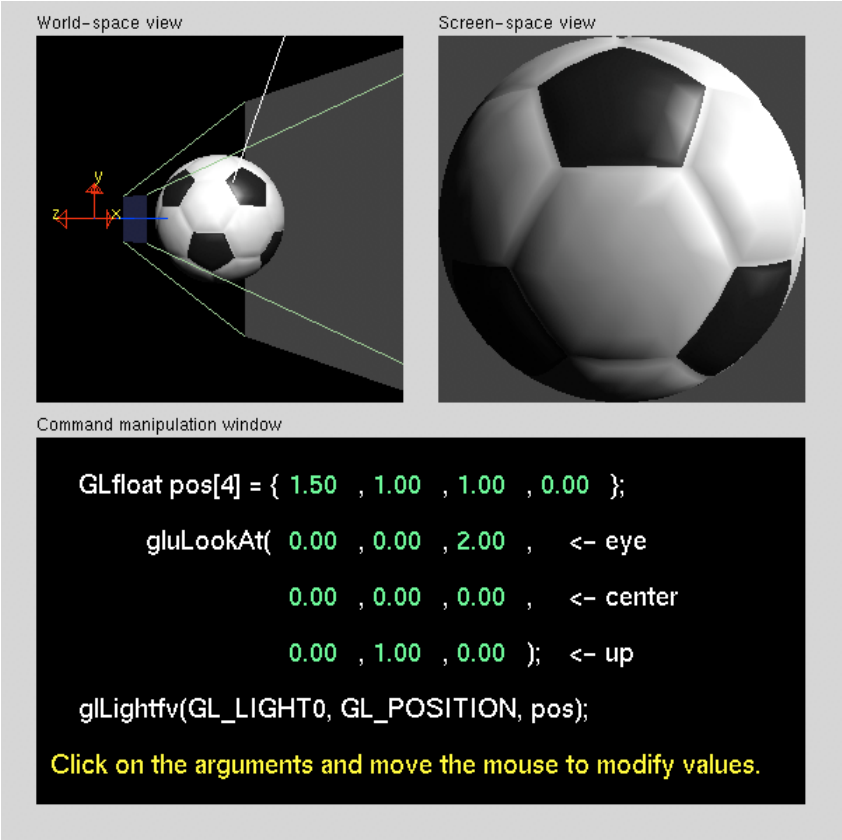
But Why do we set Up direction?

Programmer defines:

- **eye** position
- LookAt point **(at)** and
- **Up** vector (***Up*** direction usually (0,1,0))

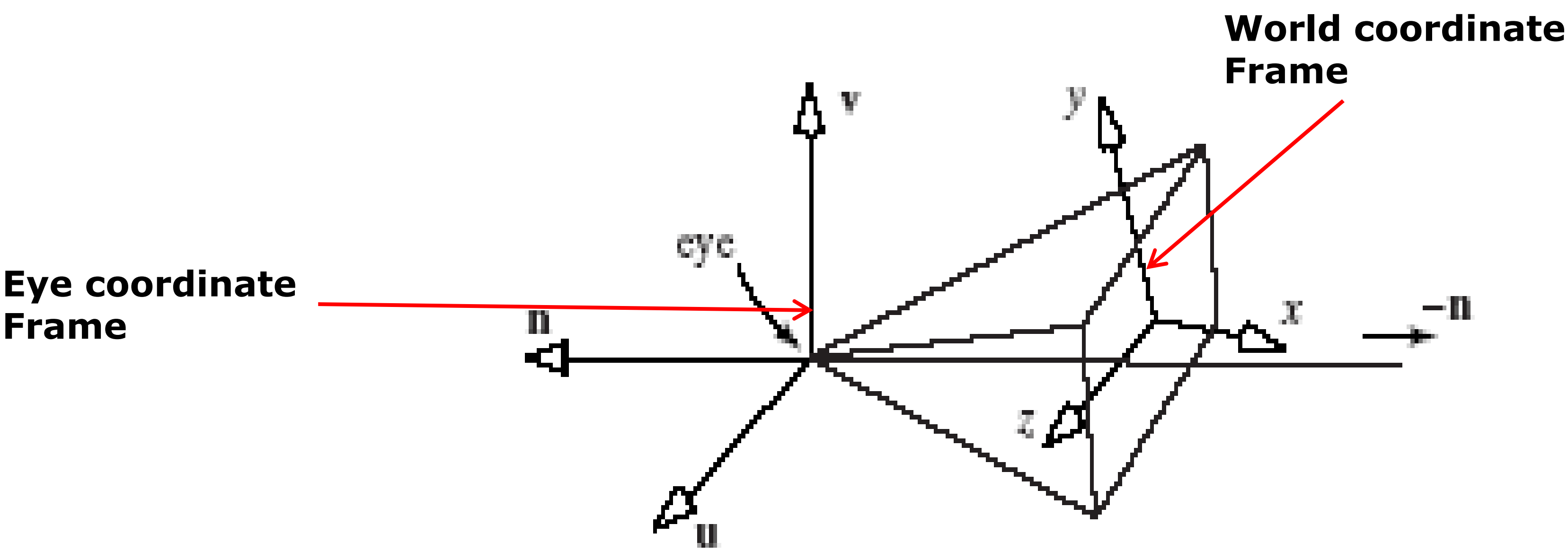
Nate Robbins LookAt Demo





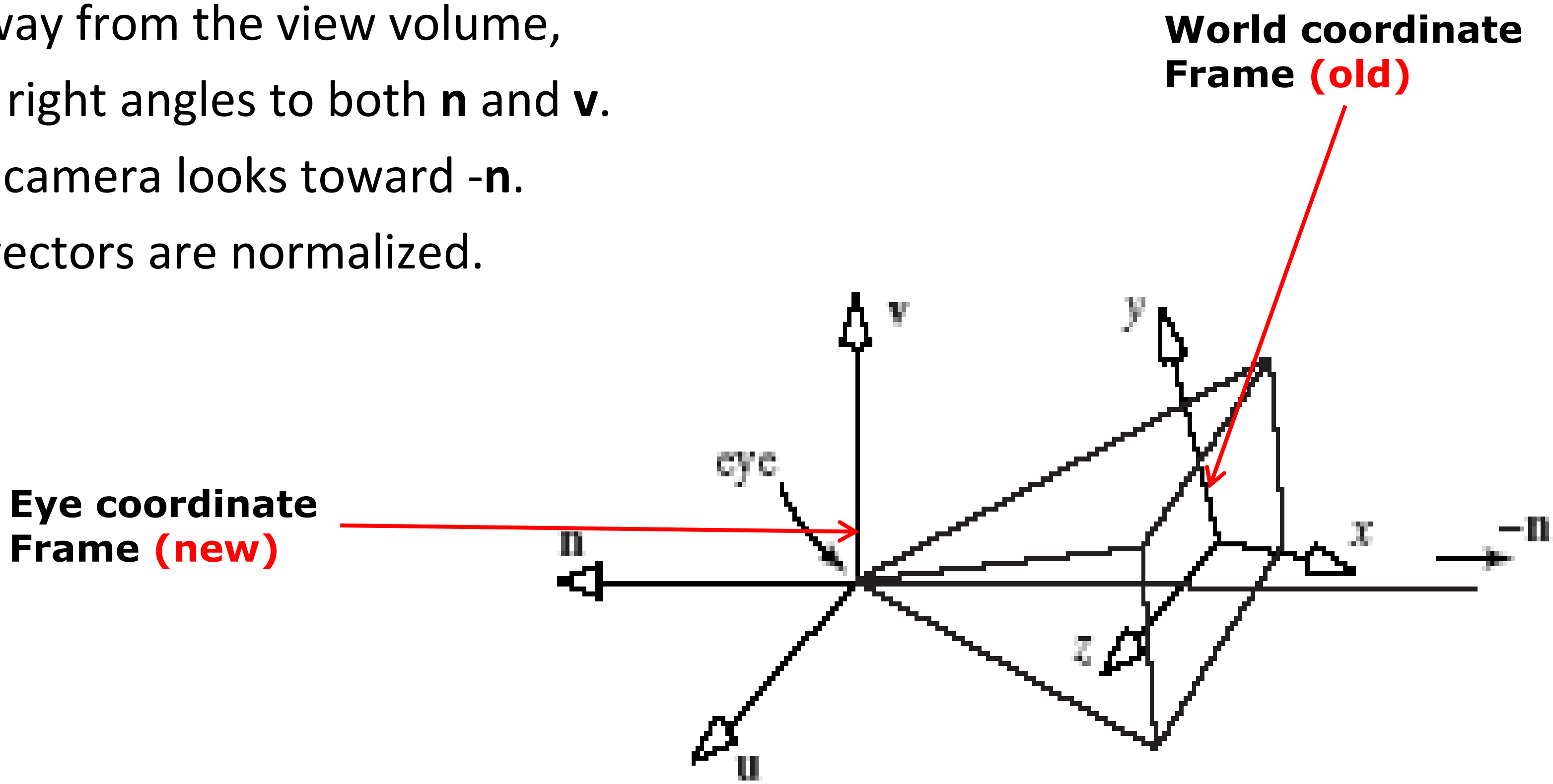
What does LookAt do?

- Programmer defines eye, lookAt and Up
- **LookAt method:**
  - Form new axes (u, v, n) at camera
  - Transform objects from world to eye camera frame



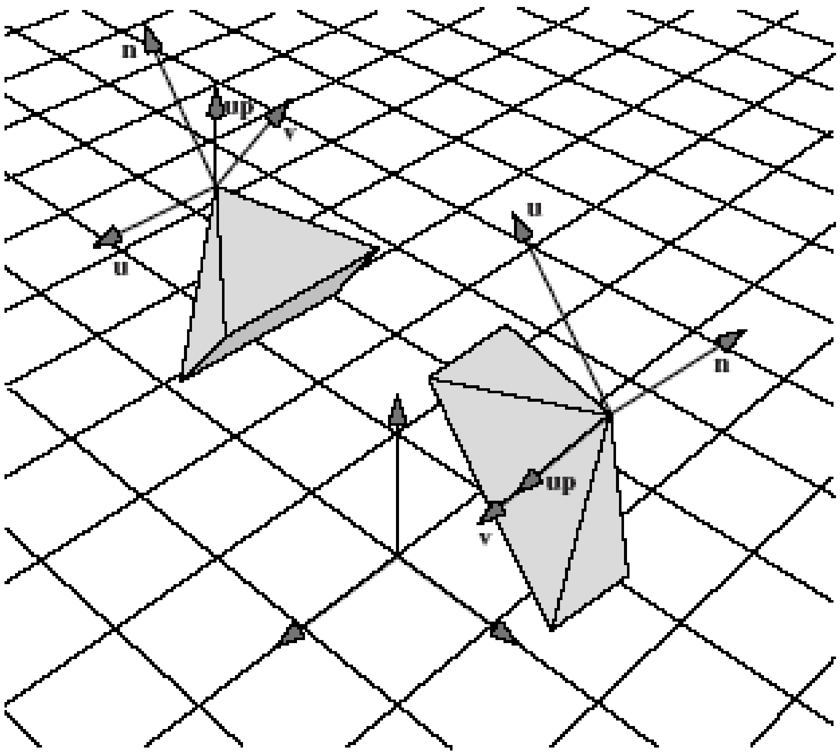
Camera with Arbitrary Orientation and Position

- Define new axes (u, v, n) at eye
  - **v** points vertically upward,
  - **n** away from the view volume,
  - **u** at right angles to both **n** and **v**.
  - The camera looks toward -**n**.
  - All vectors are normalized.



LookAt: Effect of Changing Eye Position or LookAt Point

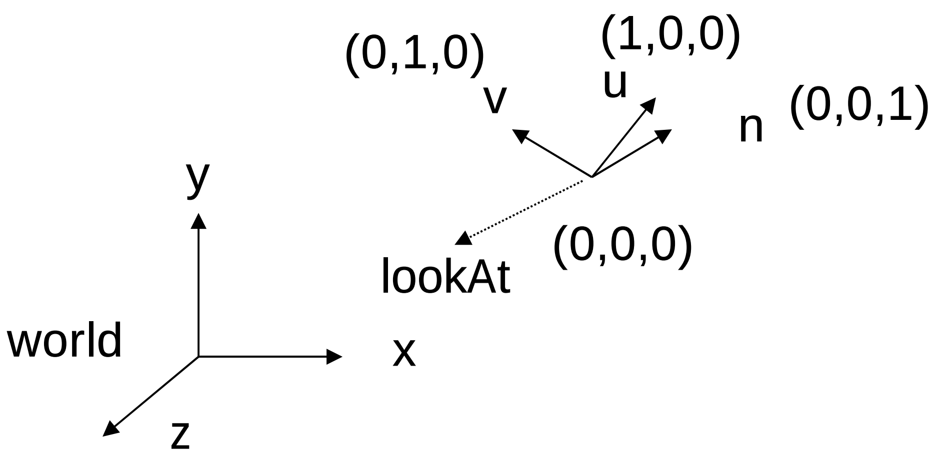
- Programmer sets **`LookAt(eye, at, up)`**
- If **eye**, **lookAt** point changes => **u,v,n** changes



Viewing Transformation Steps

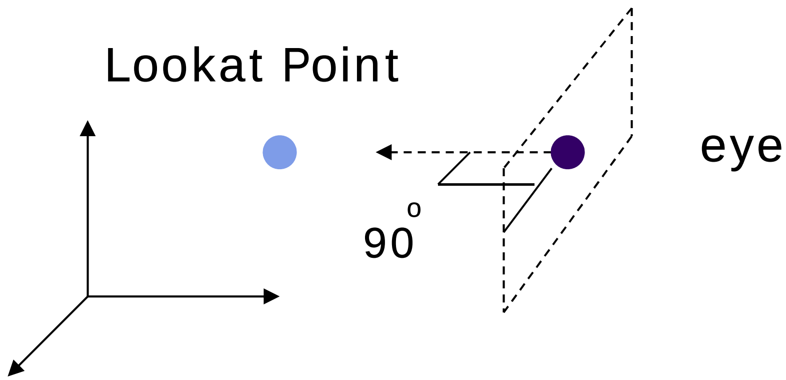
1. Form camera (u,v,n) frame
2. Transform objects from world frame (Composes matrix for coordinate transformation)

- Next, let's form camera (u,v,n) frame



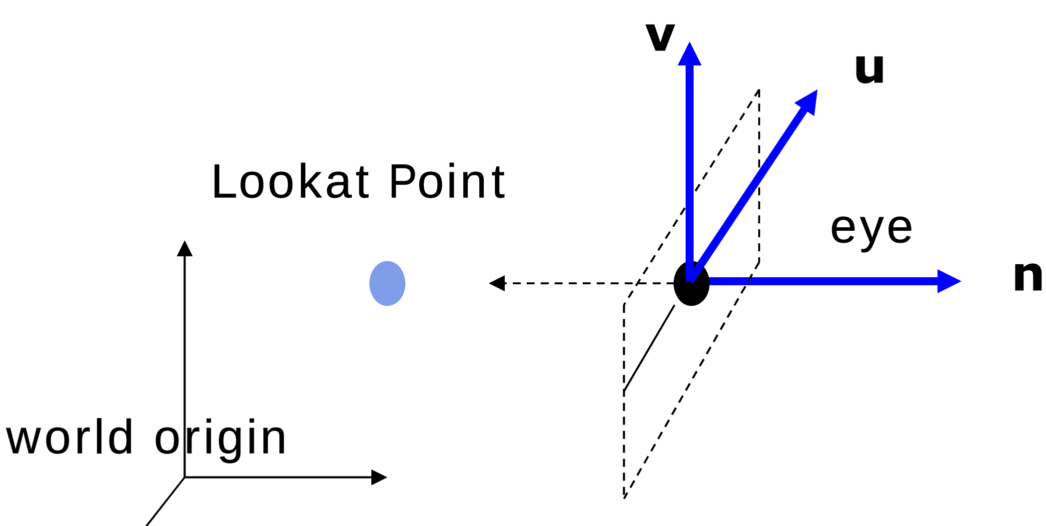
Constructing U,V,N Camera Frame

- Lookat arguments: **`LookAt(eye, at, up)`**
- **Known:** eye position, LookAt Point, up vector
- **Derive:** new origin and three basis (u,v,n) vectors



Eye Coordinate Frame

- **New Origin:** eye position (that was easy)
- 3 basis vectors:
  - one is the normal vector (**n**) of the viewing plane,
  - other two (**u** and **v**) span the viewing plane



(u,v,n should all be orthogonal)

n is pointing away from the world because we use left hand coordinate system

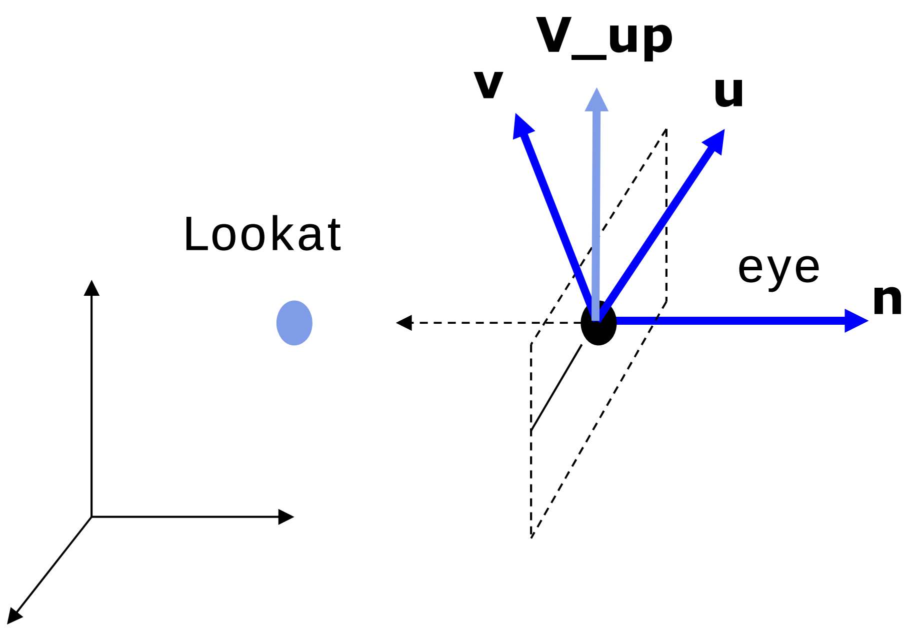
N = eye – Lookat Point

n = N / | N |

Remember **u,v,n** should be all unit vectors

Eye Coordinate Frame

- How about u and v?



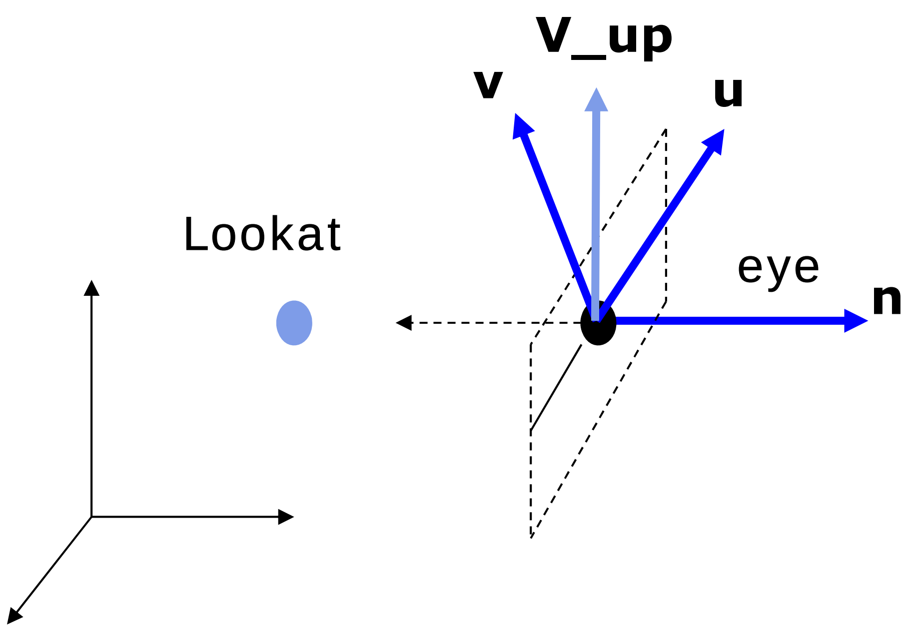
- We can get u first -
  - u is a vector that is perp to the plane spanned by N and view up vector (V_up)

U = V_up x **n**

u = U / | U |

Eye Coordinate Frame

- How about v?



Knowing n and u, getting v is easy

v = n x u

v is already normalized

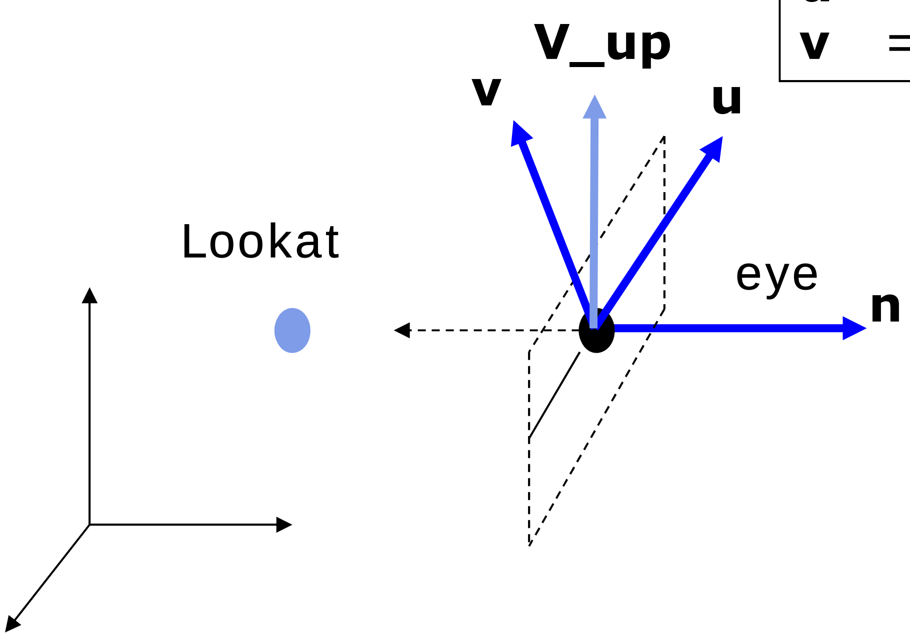
Eye Coordinate Frame

- Put it all together

Eye space **origin: (Eye.x , Eye.y,Eye.z)**

Basis vectors:

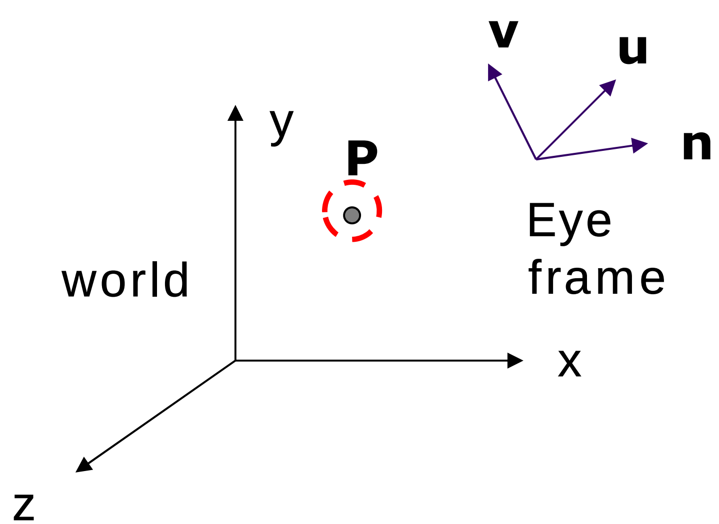
n = (eye – Lookat) / | eye – Lookat|
u = (V_up x **n**) / | V_up x **n** |
v = **n** x **u**



Step 2: World to Eye Transformation

- Next, use u, v, n to compose LookAt matrix
- Transformation matrix (M_{w2e}) ?

$P' = M_{w2e} \times P$

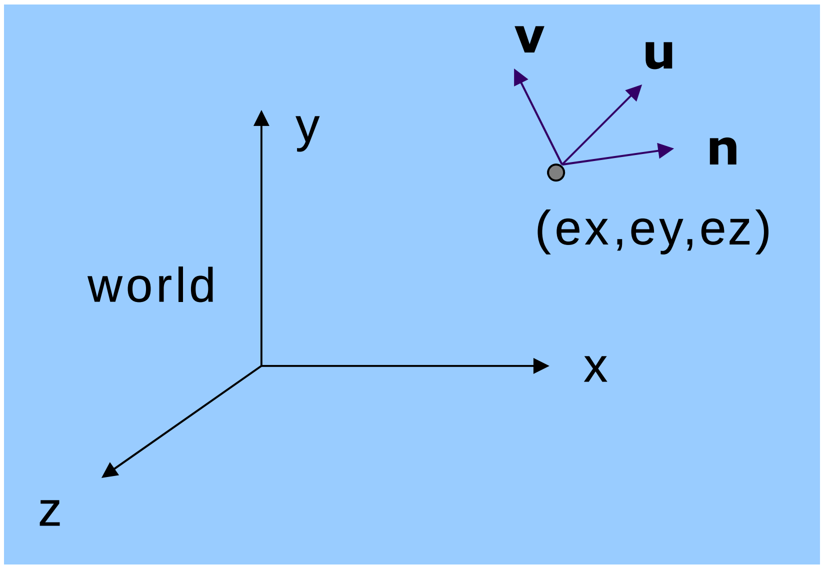


1. Come up with transformation sequence that lines up eye frame with world frame

2. Apply this transform sequence to point **P** in reverse order

World to Eye Transformation

1. Rotate eye frame to "align" it with world frame
2. Translate (-ex, -ey, -ez) to align origin with eye



Rotation:

$$\begin{vmatrix} ux & uy & uz & 0 \\ vx & vy & vz & 0 \\ nx & ny & nz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

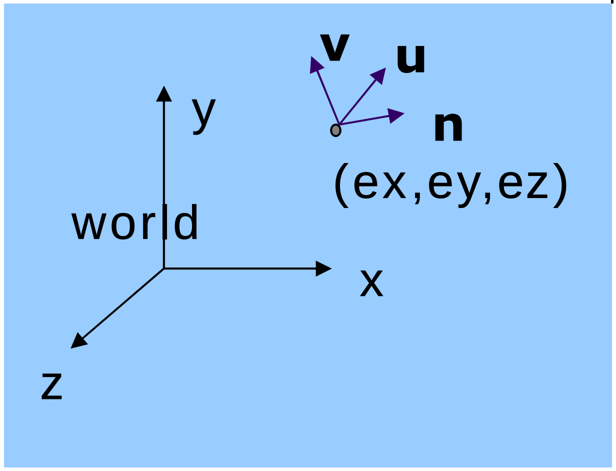
Translation:

$$\begin{vmatrix} 1 & 0 & 0 & -ex \\ 0 & 1 & 0 & -ey \\ 0 & 0 & 1 & -ez \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

World to Eye Transformation

- Transformation order: apply the transformation to the object in reverse order - translation first, and then rotate

$$
Mw2e = \underbrace{\begin{vmatrix} ux & uy & ux & 0 \\ vx & vy & vz & 0 \\ nx & ny & nz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}}_{\textbf{Rotation}} \; \underbrace{\begin{vmatrix} 1 & 0 & 0 & -ex \\ 0 & 1 & 0 & -ey \\ 0 & 0 & 1 & -ez \\ 0 & 0 & 0 & 1 \end{vmatrix}}_{\textbf{Translation}}
$$



$$
= \begin{vmatrix} ux & uy & uz & -\mathbf{e}\,.\,\mathbf{u} \\ vx & vy & vz & -\mathbf{e}\,.\,\mathbf{v} \\ nx & ny & nz & -\mathbf{e}\,.\,\mathbf{n} \\ 0 & 0 & 0 & 1 \end{vmatrix}
$$

Multiplied together = lookAt transform

Note: $\mathbf{e.u}$ = ex.ux + ey.uy + ez.uz

lookAt Implementation (from mat.h)

Eye space **origin: (Eye.x , Eye.y,Eye.z)**

Basis vectors:

n = (eye – Lookat) / | eye – Lookat|
u = (V_up x **n**) / | V_up x **n** |
v = **n** x **u**

$$\begin{vmatrix} ux & uy & uz & -\mathbf{e}\cdot\mathbf{u} \\ vx & vy & vz & -\mathbf{e}\cdot\mathbf{v} \\ nx & ny & nz & -\mathbf{e}\cdot\mathbf{n} \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

```
mat4 LookAt( const vec4& eye, const vec4& at, const vec4& up )
{
    vec4 n = normalize(eye - at);
    vec4 u = normalize(cross(up,n));
    vec4 v = normalize(cross(n,u));
    vec4 t = vec4(0.0, 0.0, 0.0, 1.0);
    mat4 c = mat4(u, v, n, t);
    return c * Translate( -eye );
}
```

References

- Interactive Computer Graphics, Angel and Shreiner, Chapter 4
- Computer Graphics using OpenGL (3rd edition), Hill and Kelley