# Computer Graphics (CS 543)
# Lecture 6a: Viewing & Camera Control

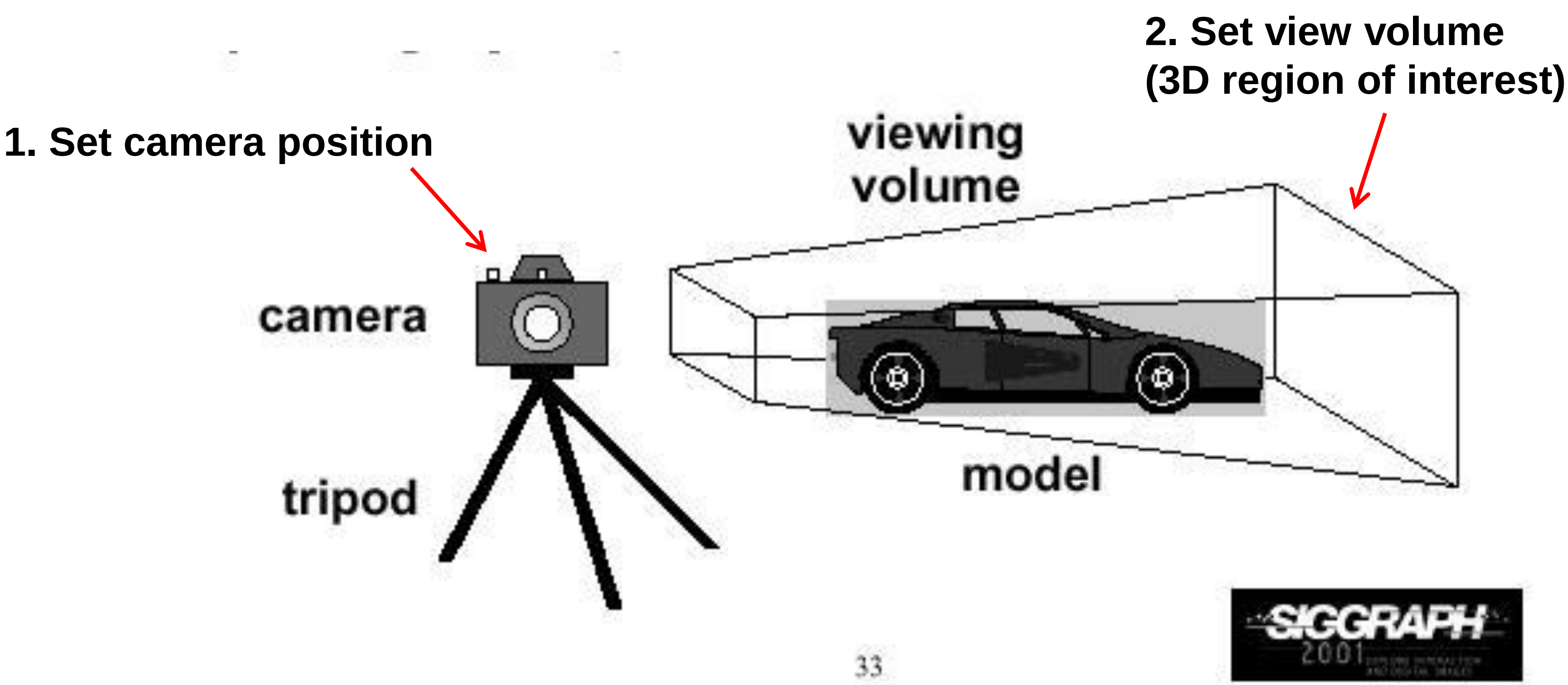
Prof Emmanuel Agu

*Computer Science Dept.*

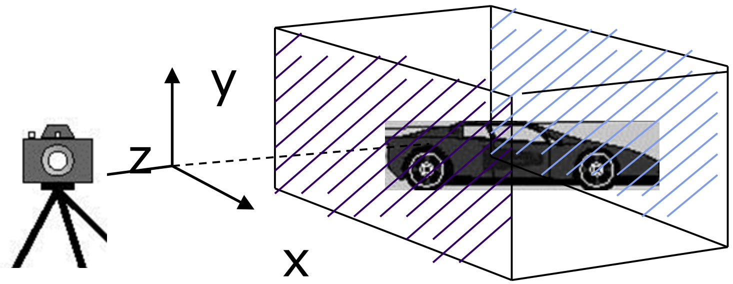
*Worcester Polytechnic Institute (WPI)*

# 3D Viewing?

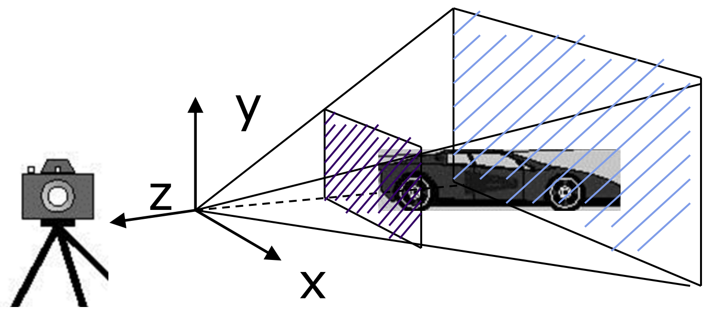
- Specify a view volume
- Objects **inside** view volume drawn to viewport (screen)
- Objects outside view volume **clipped** (not drawn)**!**



# Different View Volume Shapes




Orthogonal view volume

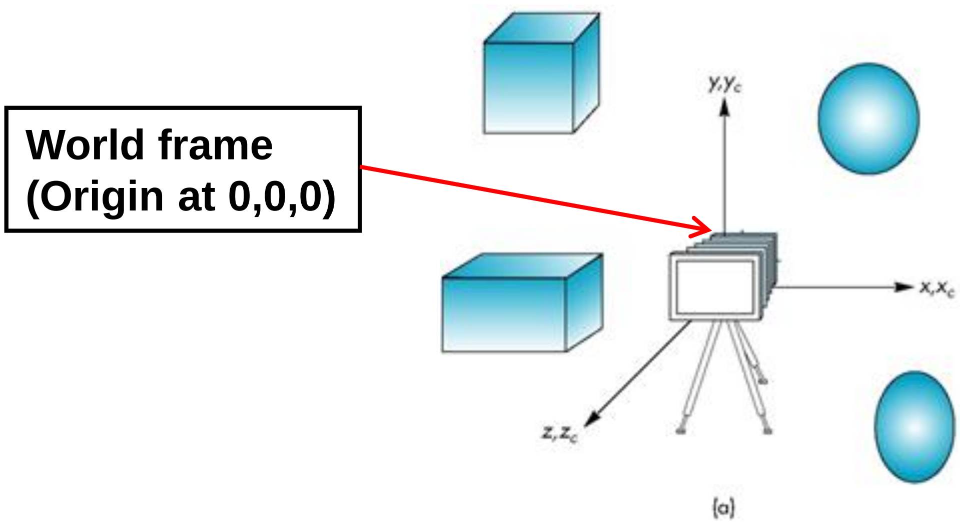



Perspective view volume

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

# The World Frame

- Object positions initially defined in **world frame**
- **World Frame origin** at (0,0,0)
- Objects positioned, oriented (translate, scale, rotate transformations) applied to objects in **world frame**

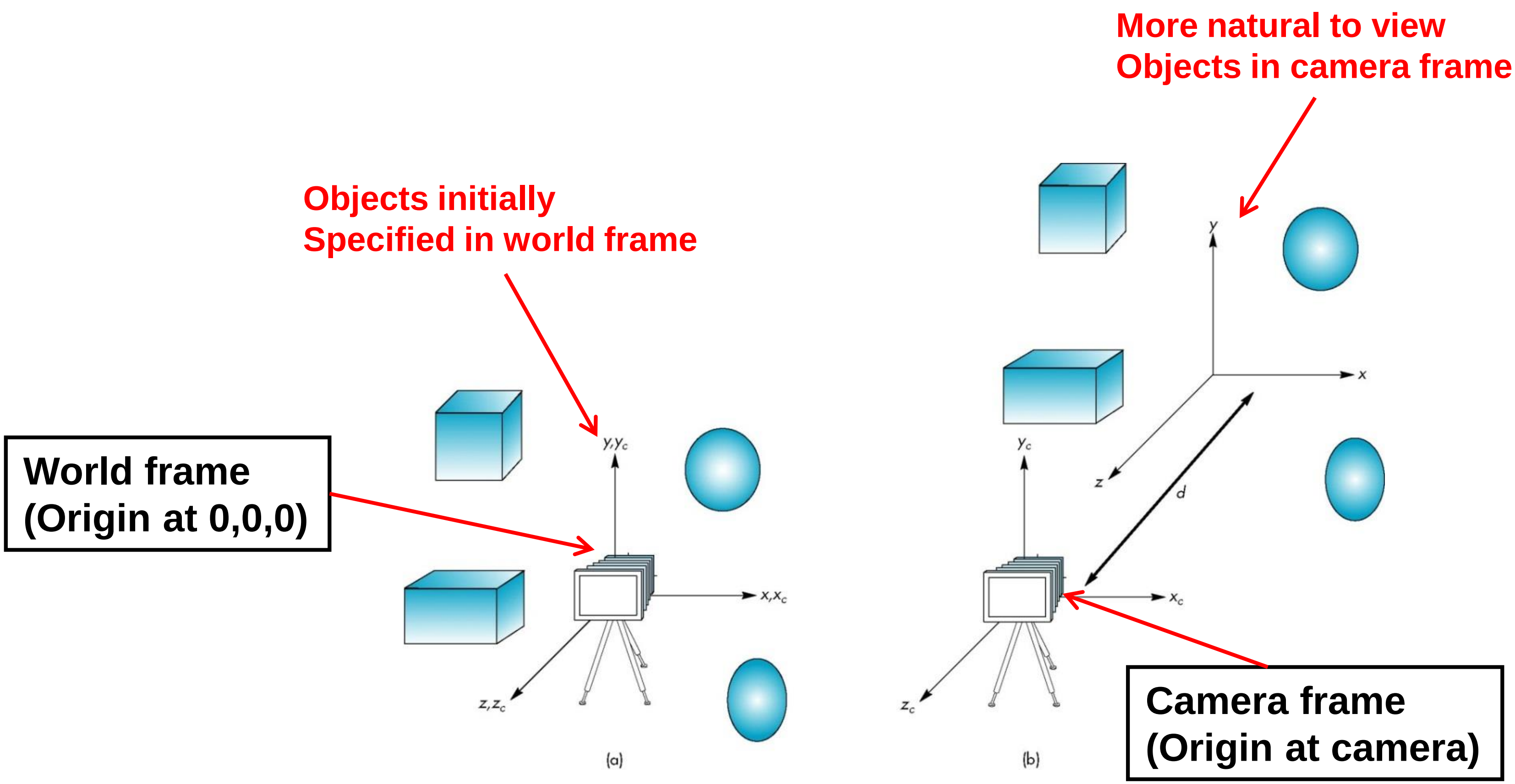


# Camera Frame

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

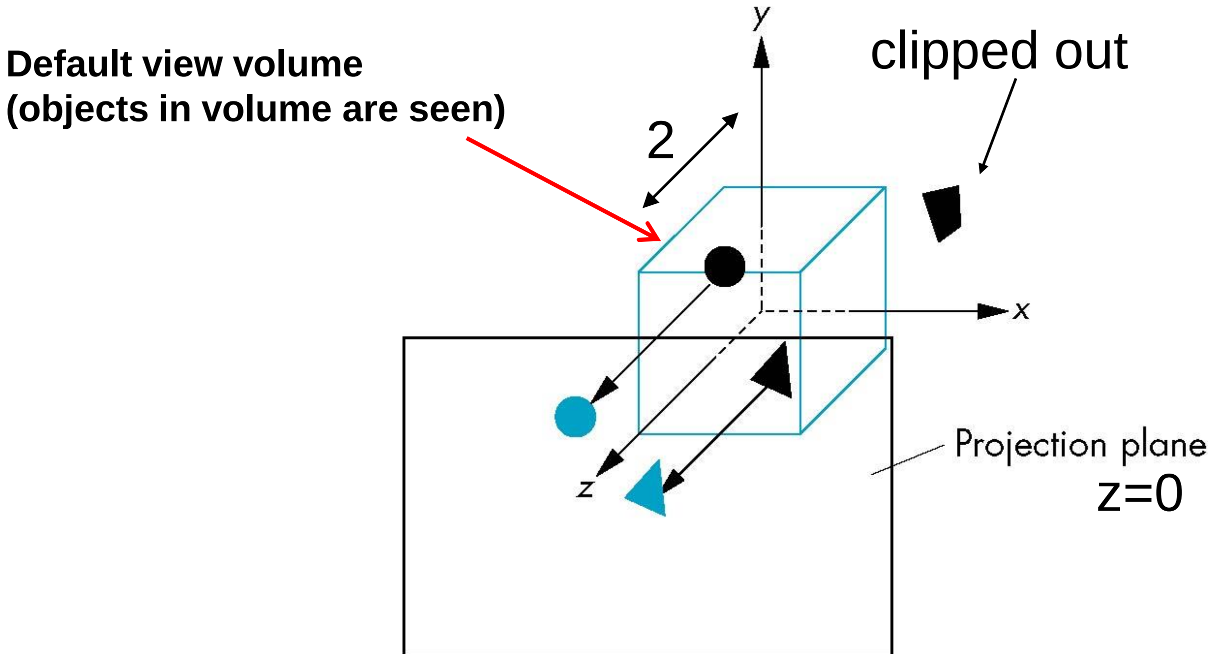
# Camera Frame

- **Viewing:** After user chooses camera (eye) position, represent objects in **camera frame** (origin at eye position)
- **Viewing transformation:** Converts object (x,y,z) positions in world frame to positions in camera frame



# Default OpenGL Camera, View Volume

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



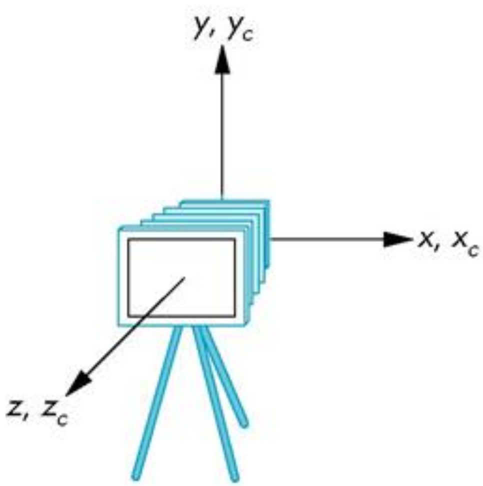
# Moving Camera Frame

Same RELATIVE distance after
Same result/look

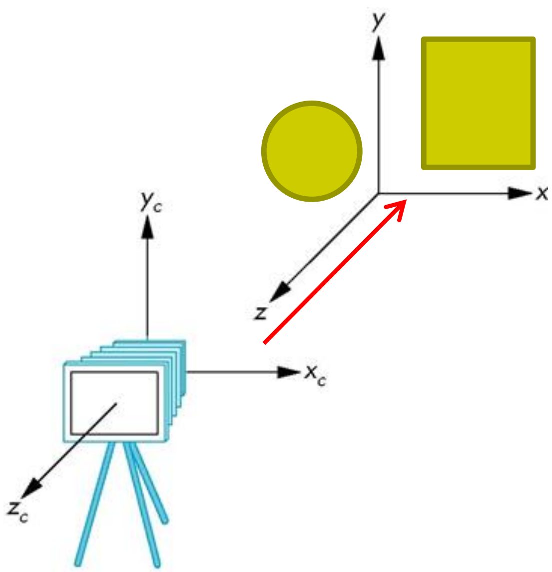
Translate objects -5
away from camera

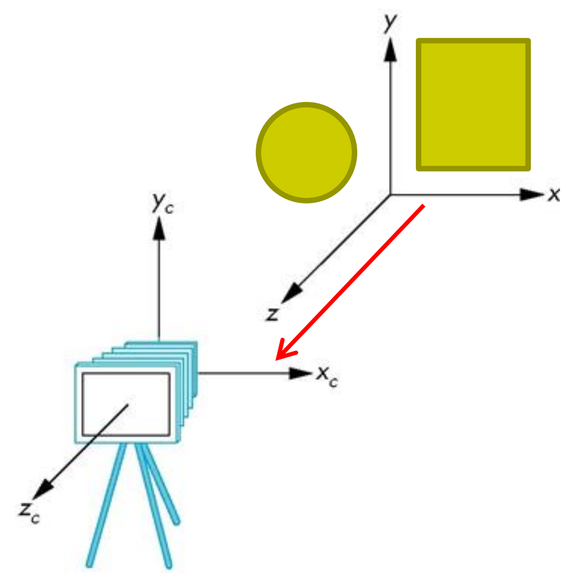
Translate camera +5
away from objects

default frames




(a)



# Moving the Camera 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)**
  - Why? Combination of object + camera transforms = relative transform

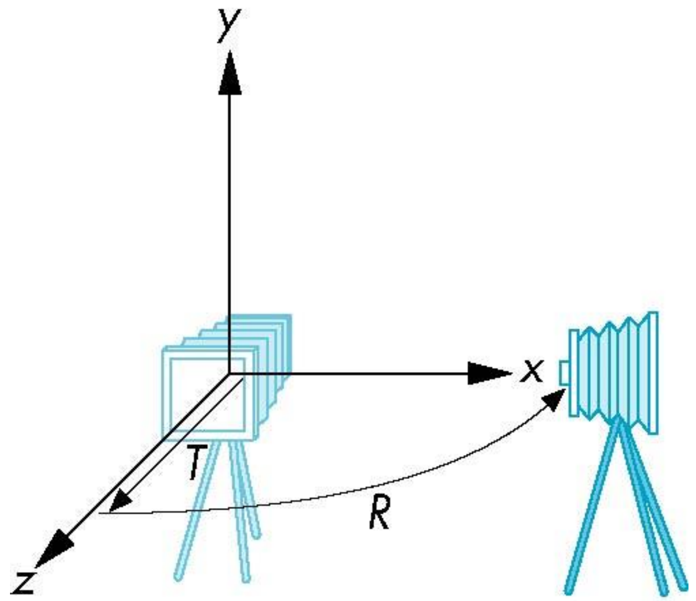
Camera
Transforms

Object transforms
(Rotate, Scale, Translate)

**CTM**

# 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;
```

# The LookAt Function

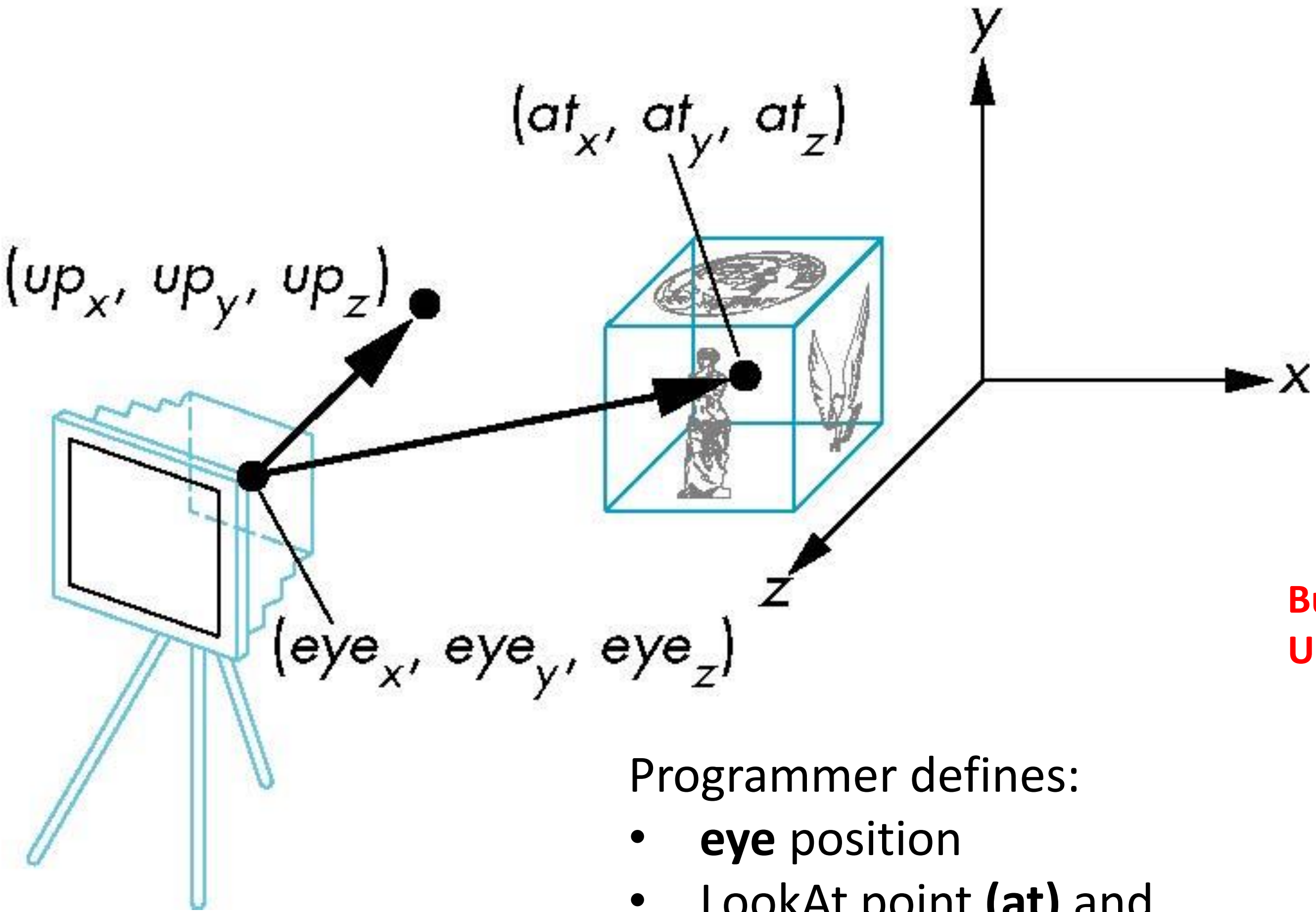
- Previously, command **gluLookAt** to position camera
- **gluLookAt** deprecated!!
- Homegrown mat4 method LookAt() in mat.h
  - Functionality: sets camera position, transforms object distances to camera frame

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

Builds 4x4 matrix for positioning, orienting Camera and puts it into variable **mv**

# The LookAt Function

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

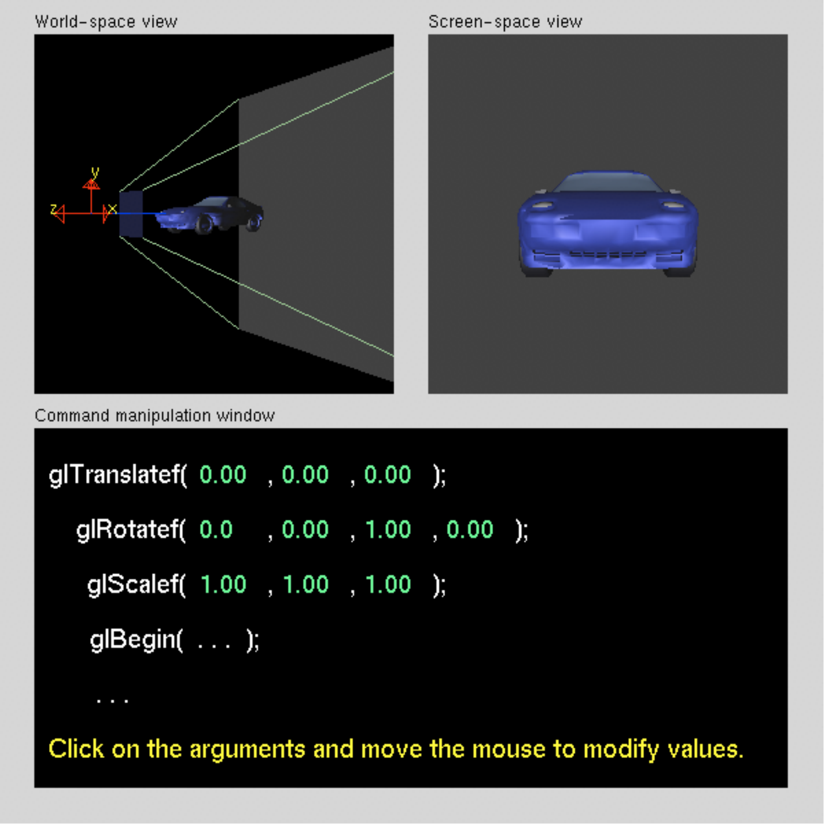


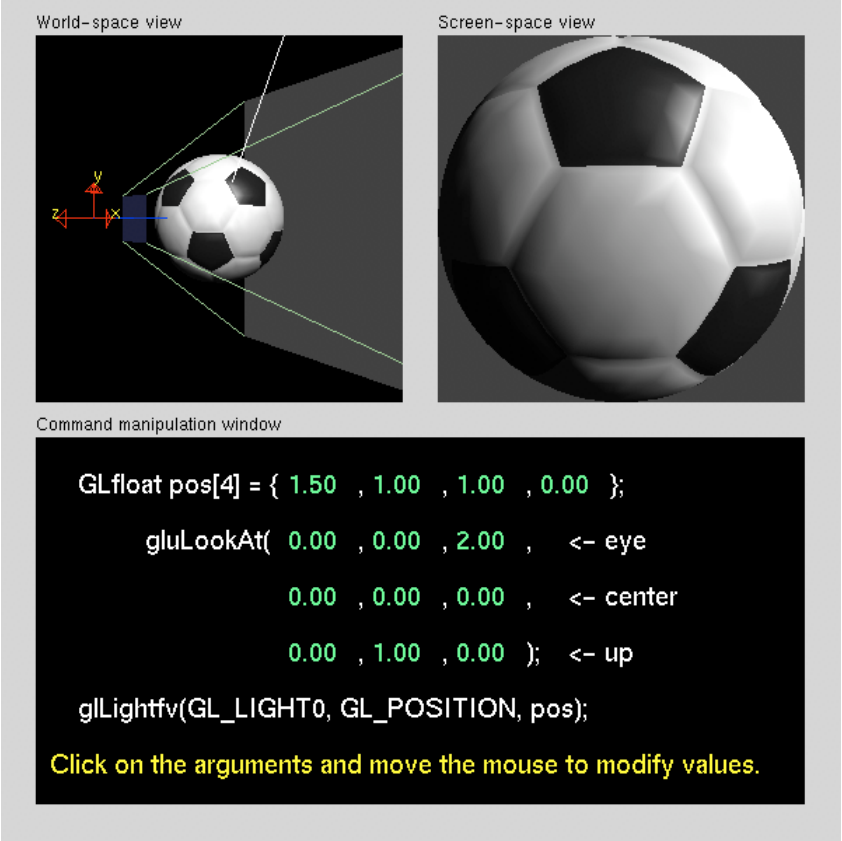

**But Why do we set Up direction?**

Programmer defines:

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

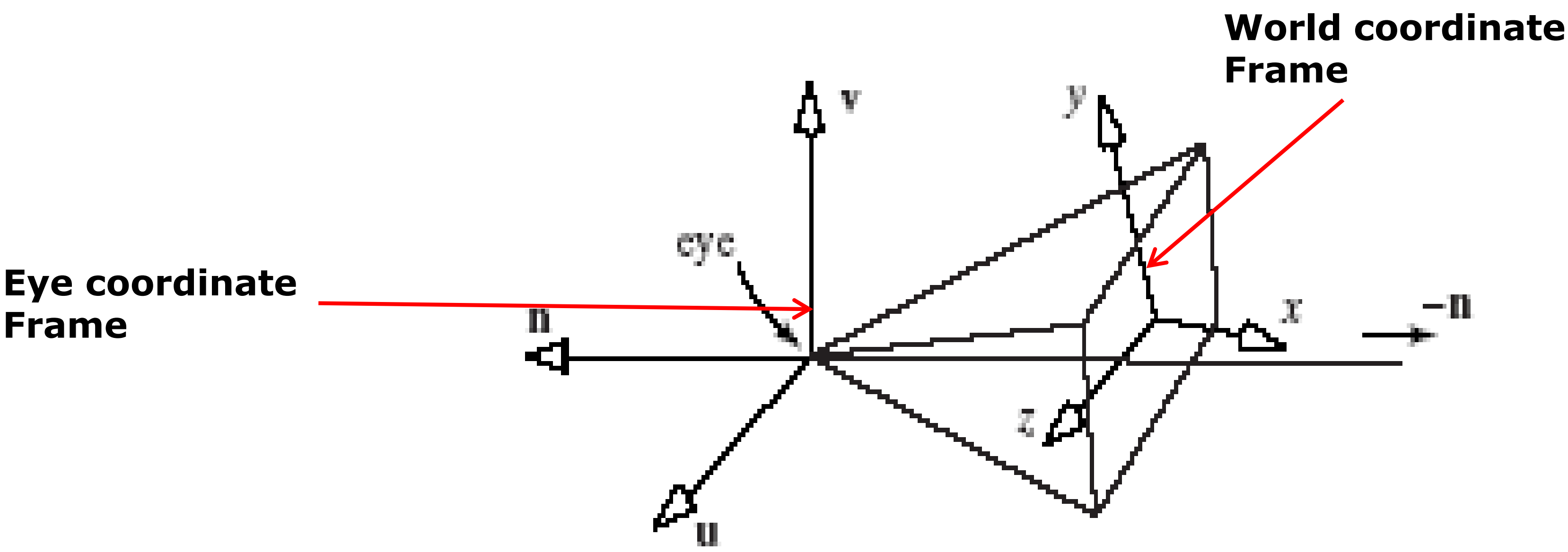
# Nate Robbins LookAt Demo



# What does LookAt do?

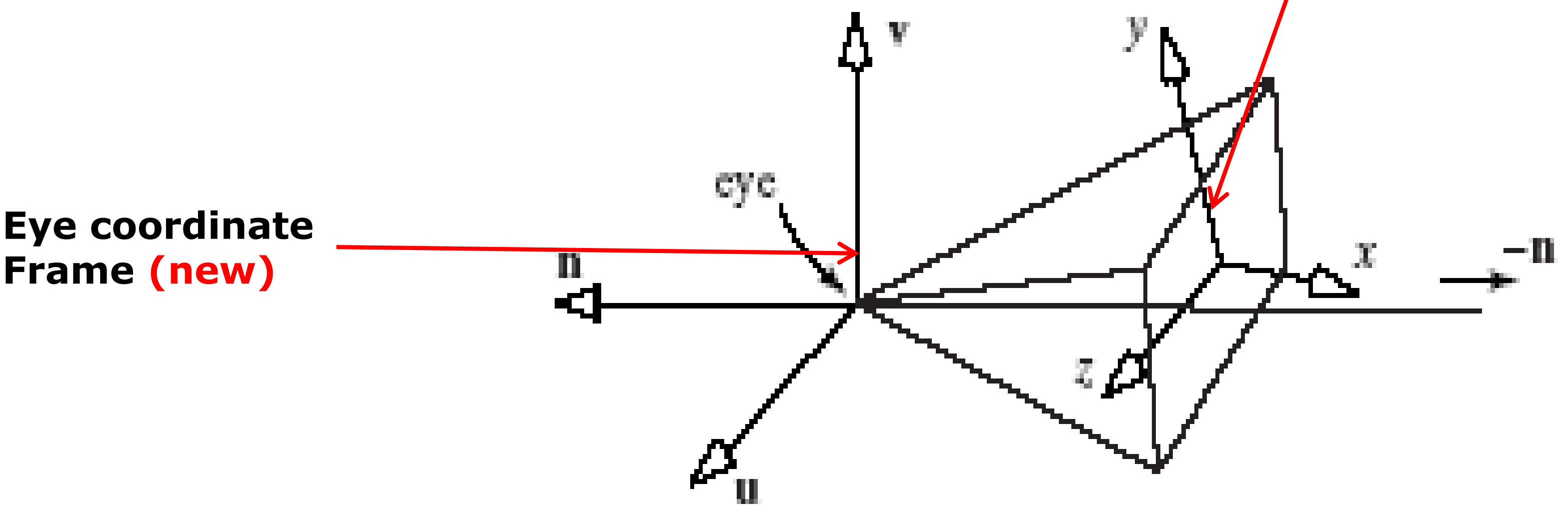
- Programmer defines eye, lookAt and Up
- **LookAt method:**
  - Forms new axes (u, v, n) at camera
  - Transform objects from world to eye camera frame



# Camera with Arbitrary Orientation and Position

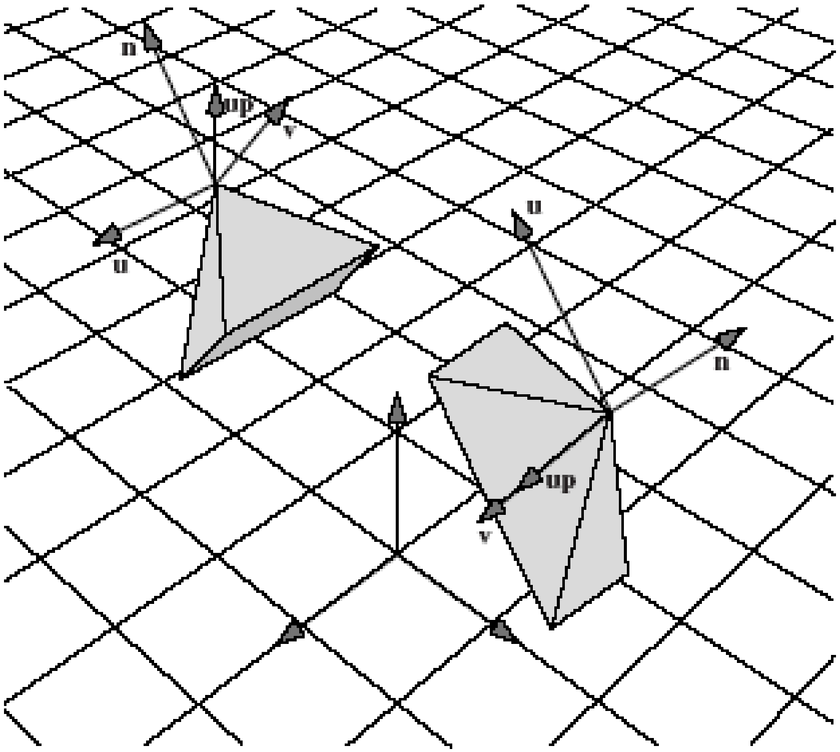
**Ref: Computer Graphics using OpenGL (3rd edition) by F.S. Hill and Kelley**

- 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

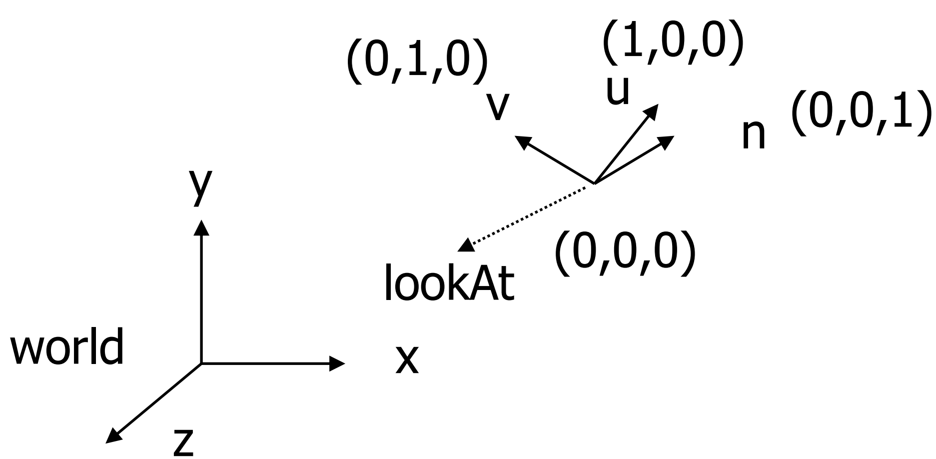
- Programmer sets `LookAt(eye, at, up)`
- If **eye**, **lookAt** point changes => **u,v,n** changes



# Viewing Transformation Steps

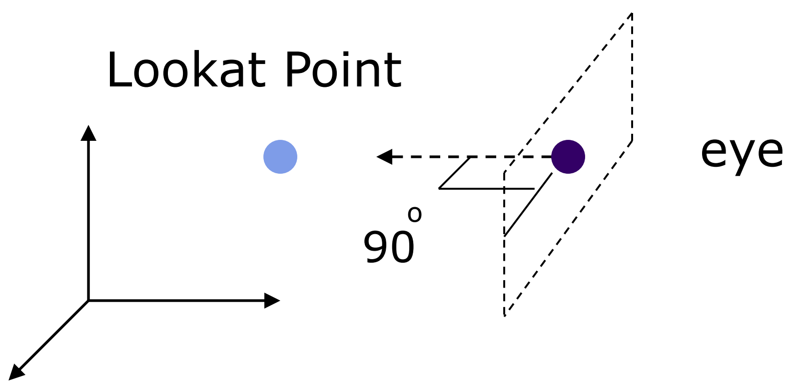
1. Form camera (u,v,n) frame
2. Transform objects from world frame (Compose matrix to transform coordinates)

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



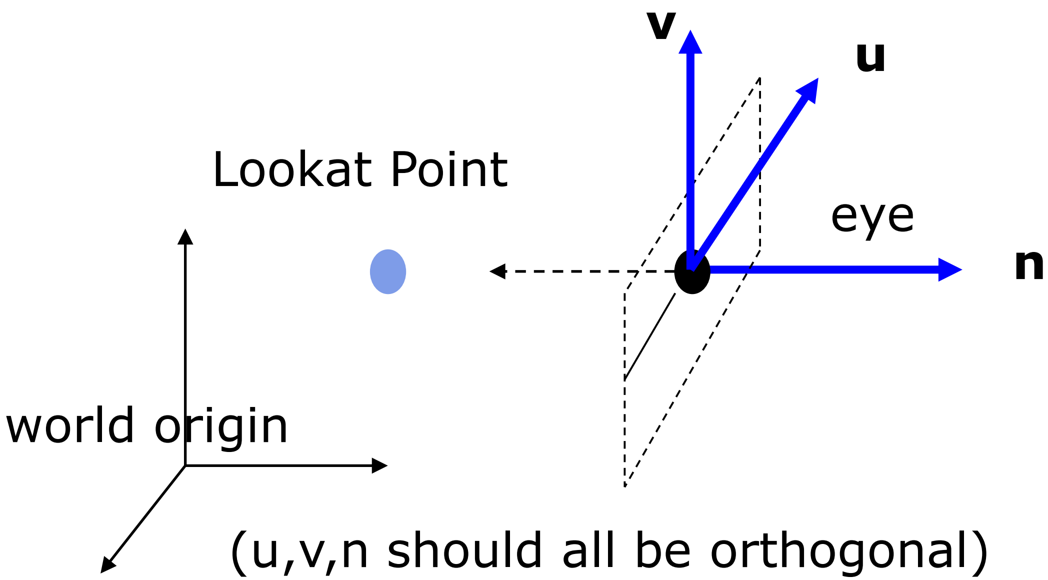
# Constructing U,V,N Camera Frame

- 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)

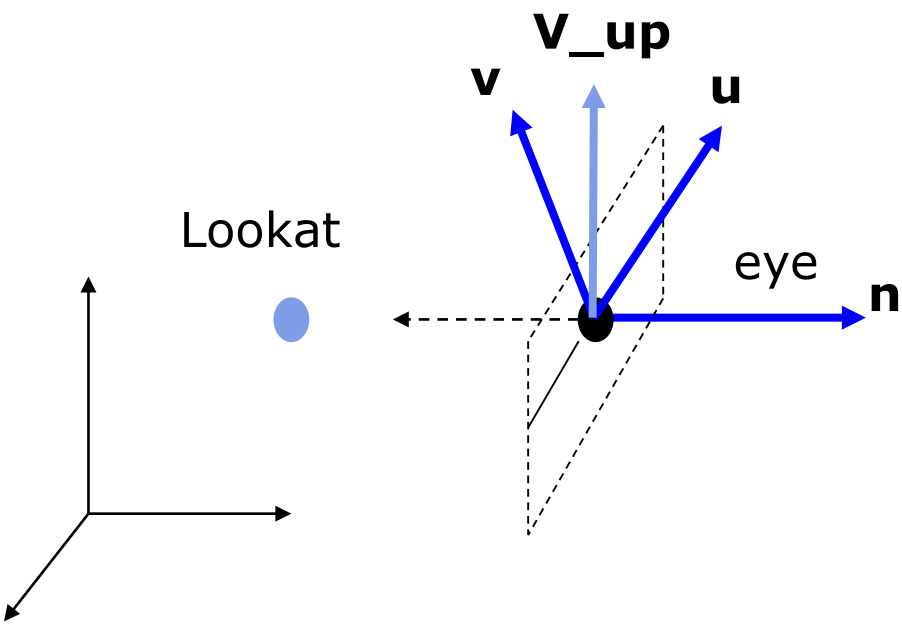
**n** is pointing away from the world because we use left hand coordinate system

**N** = eye – Lookat Point
**n = N / | N |**

Remember **u,v,n** should be all unit vectors
So… Normalize vectors!!!!!

# Eye Coordinate Frame

- How about u and v?



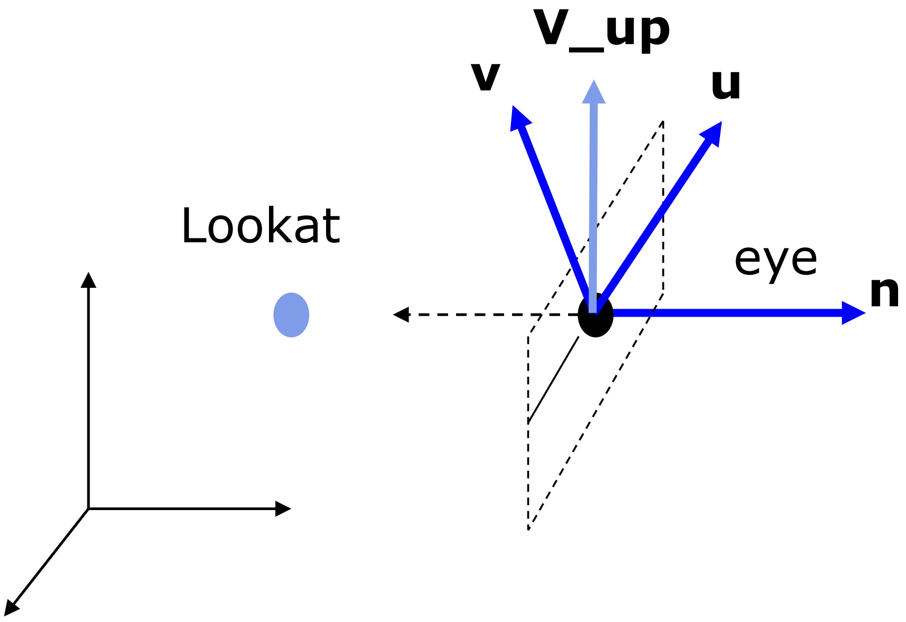

- Derive u first -
  - u is a vector that is perp to the plane spanned by N and view up vector (V_up)

$$U = V\_up \times \mathbf{n}$$

$$\mathbf{u} = U \ / \ |U|$$

# Eye Coordinate Frame

- How about v?




To derive v from n and u

**v = n x u**

**v is already normalized**

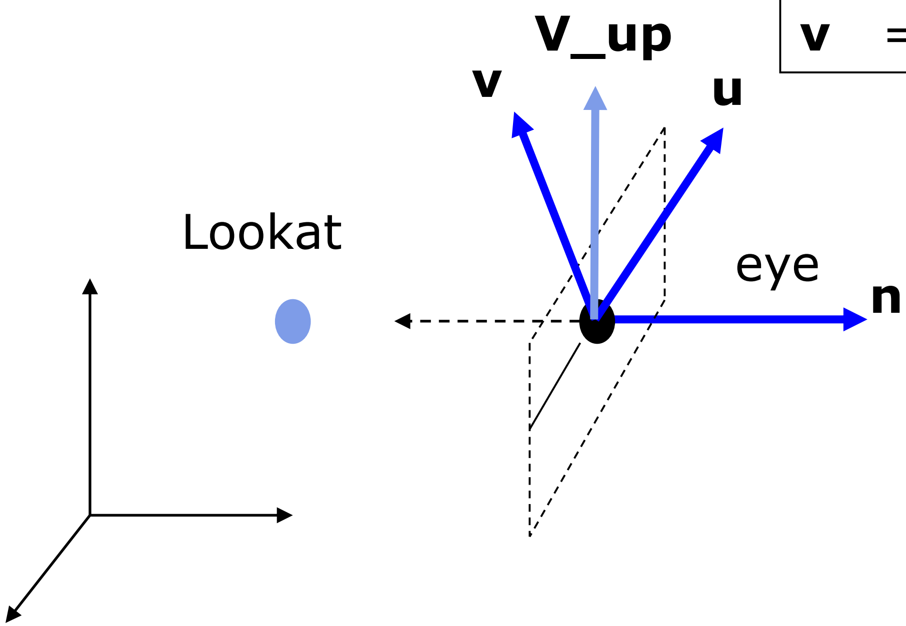
# Eye Coordinate Frame

- Put it all together

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

Basis vectors:

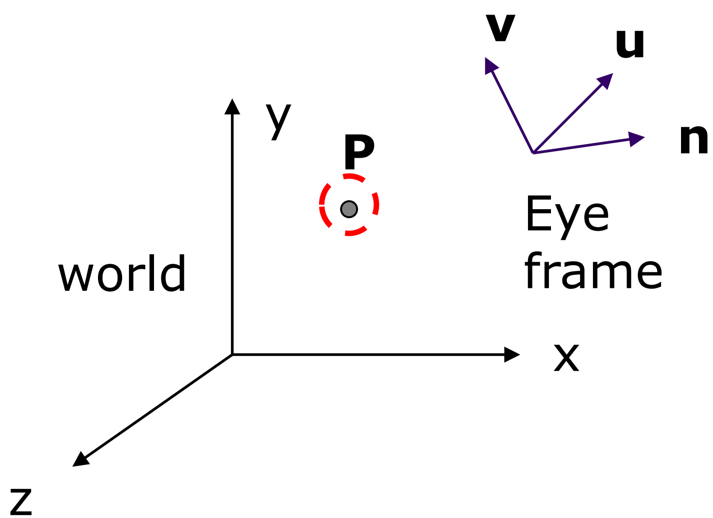
**n** = (eye – Lookat) / | eye – Lookat|
**u** = (V_up x **n**) / | V_up x **n** |
**v** = **n** x **u**



# Step 2: World to Eye Transformation

- Next, use u, v, n to compose LookAt matrix
- Transformation matrix ($M_{w2e}$) ?
  - Matrix that transforms a point P in world frame to P' in eye frame

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

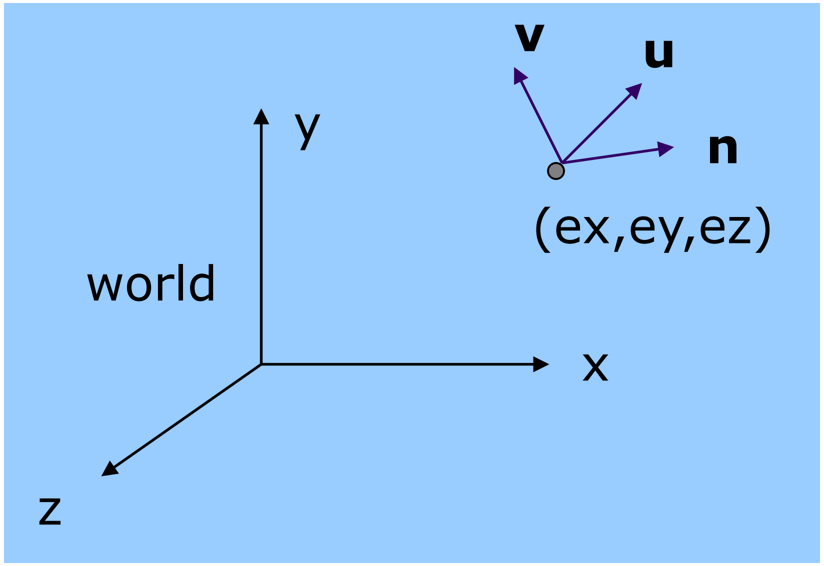



1. Come up with transformation sequence that aligns eye frame with world frame

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

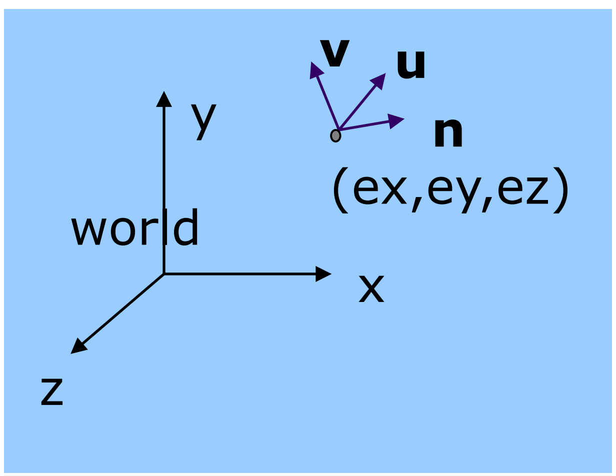
- Transformation order: apply the transformation to the object in reverse order - translation first, and then rotate

**Order**

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

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

**Multiplied together = lookAt transform**




Note: **e.u** = ex.ux + ey.uy + ez.uz

**e.v** = ex.vx + ey.vy + ez.vz

**e.n** = ex.nx + ey.ny + ez.nz

# 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**

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

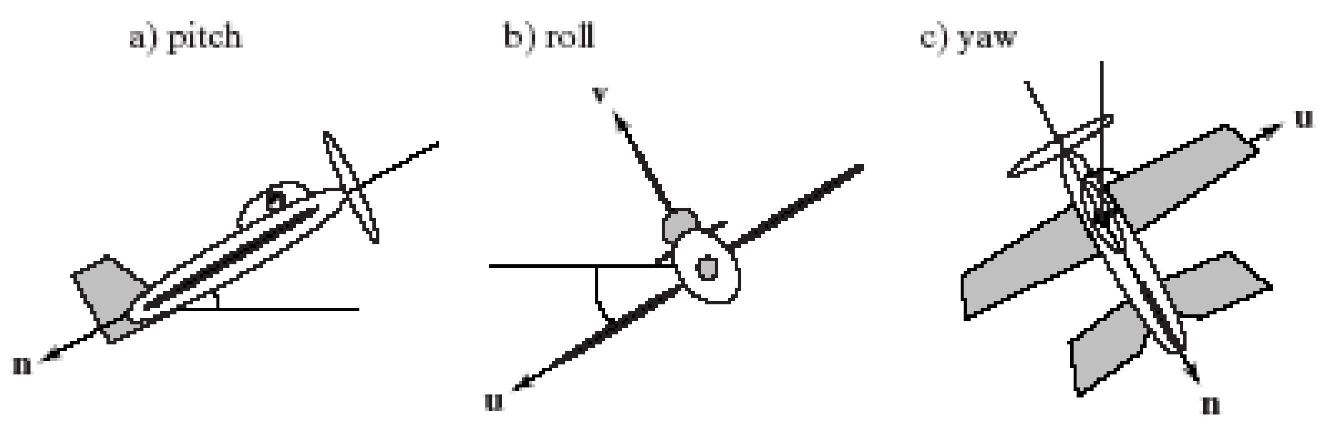
```
mat4 LookAt( const vec4& eye, const vec4& at, const vec4& up )
{
    vec4 n = normalize(eye - at);
    vec4 u = normalize(cross(up,n));
    vec4 v = normalize(cross(n,u));
    vec4 t = vec4(0.0, 0.0, 0.0, 1.0);
    mat4 c = mat4(u, v, n, t);
    return c * Translate( -eye );
}
```

# Other Camera Controls

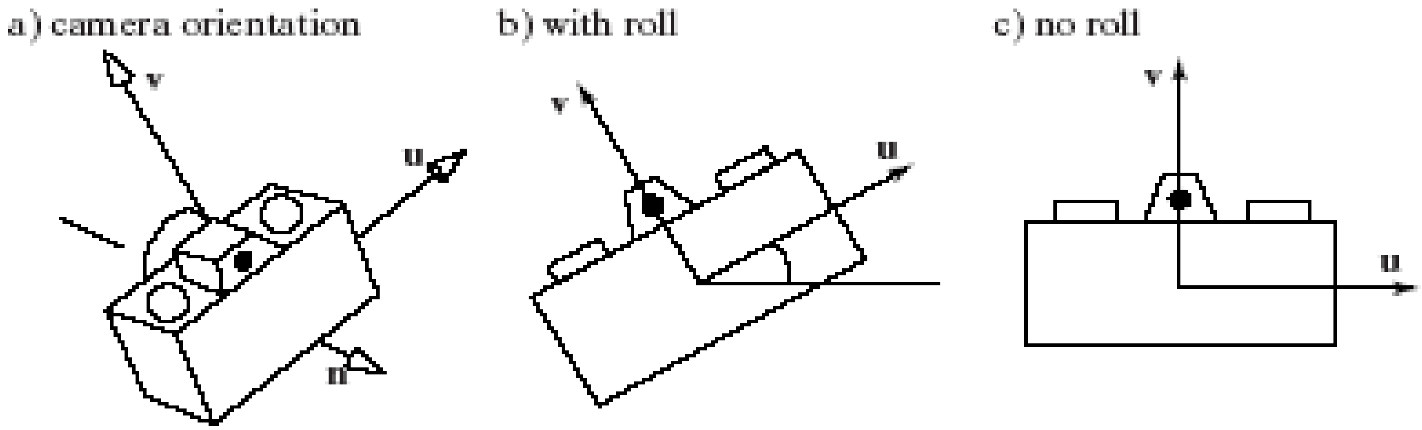
- The LookAt function is only for positioning camera
- Other ways to specify camera position/orientation/movement
  - Yaw, pitch, roll
  - Elevation, azimuth, twist
  - Direction angles

# Flexible Camera Control

- Sometimes, we want camera to move
- Like controlling an airplane’s orientation
- Adopt aviation terms:
  - **Pitch:** nose up-down
  - **Roll:** roll body of plane
  - **Yaw:** move nose side to side

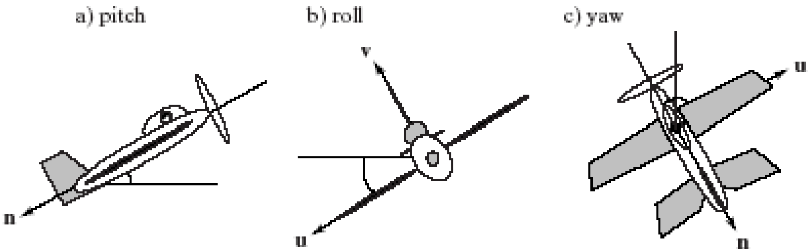


# Yaw, Pitch and Roll Applied to Camera



# Flexible Camera Control

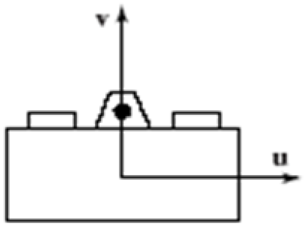
- Create a **camera** class




```
class Camera
   private:
        Point3 eye;
        Vector3 u, v, n;…. etc
```

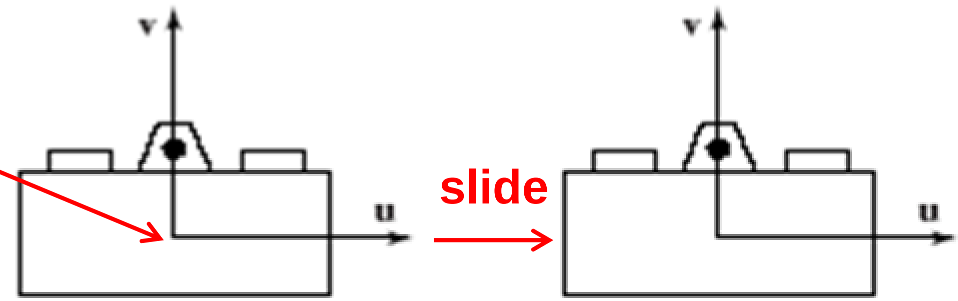
- Camera methods (functions) to specify slide, pitch, roll, yaw wrt u,v,n. E.g

```
cam.slide(1, 0, 2); // slide camera backward 2 and  right 1
cam.roll(30);   // roll camera 30 degrees
cam.yaw(40);   // yaw camera 40 degrees
cam.pitch(20);  // pitch camera 20 degrees
```



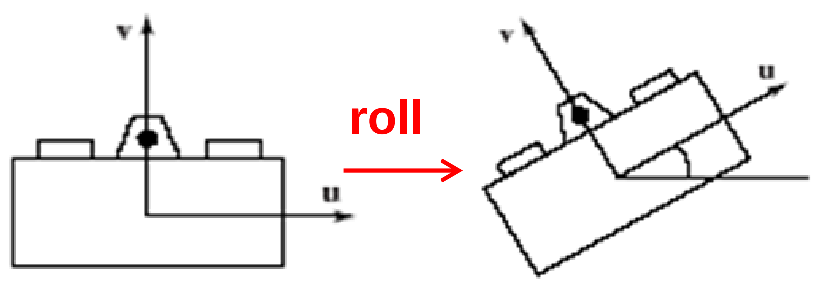
# Recall: Final LookAt Matrix

- Slide along u, v or n
- Changes eye position
- Changes these components




$$\left|\begin{array}{ccc|c} ux & uy & uz & -\mathbf{e} \,.\, \mathbf{u} \\ vx & vy & vz & -\mathbf{e} \,.\, \mathbf{v} \\ nx & ny & nz & -\mathbf{e} \,.\, \mathbf{n} \\ 0 & 0 & 0 & 1 \end{array}\right|$$

- Pitch, yaw, roll rotates u, v or n
- Changes u, v or n
- E.g. roll rotates u,v




**Question: Pitch rotates which axes?**

# Implementing Flexible Camera Control

- Camera class: maintains current (u,v,n) and eye position

```
class Camera
private:
        Point3 eye;
        Vector3 u, v, n;…. etc
```

  - User inputs desired roll, pitch, yaw angle or slide
    1. **Roll, pitch, yaw:** calculate modified vector (u’, v’, n’)
    2. **Slide:** Calculate new eye position
    3. Update lookAt matrix, Load it into CTM

# Example: Camera Slide

- Recall: the axes are unit vectors
- User changes eye by delU, delV or delN
- eye = eye + changes (delU, delV, delN)
- Note: function below combines all slides into one

  E.g moving camera by ***D*** along its u axis = **eye + *D*u**

```
void camera::slide(float delU, float delV, float delN)
{
   eye.x += delU*u.x + delV*v.x + delN*n.x;
   eye.y += delU*u.y + delV*v.y + delN*n.y;
   eye.z += delU*u.z + delV*v.z + delN*n.z;
   setModelViewMatrix( );
}
```

Function to update new eye, u, v and n

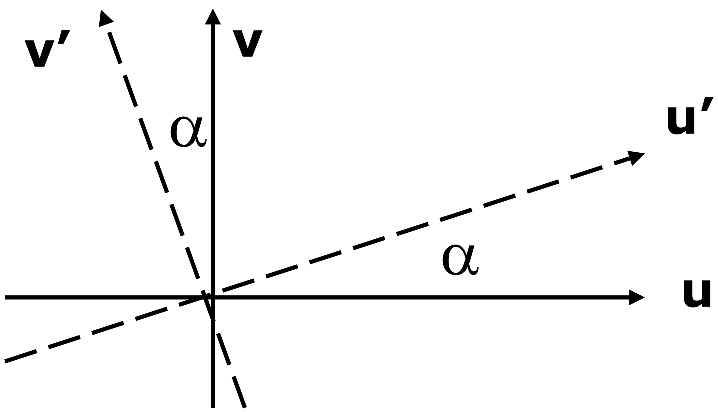
# Load Matrix into CTM

$$\left|\begin{array}{ccc|c} ux & uy & uz & -\mathbf{e} \cdot \mathbf{u} \\ vx & vy & vz & -\mathbf{e} \cdot \mathbf{v} \\ nx & ny & nz & -\mathbf{e} \cdot \mathbf{n} \\ 0 & 0 & 0 & 1 \end{array}\right|$$

```
void Camera::setModelViewMatrix(void)
{  // load modelview matrix with camera values
   mat4 m;
   Vector3 eVec(eye.x, eye.y, eye.z);// eye as vector
   m[0] = u.x; m[4] = u.y; m[8] = u.z;  m[12] = -dot(eVec,u);
   m[1] = v.x; m[5] = v.y; m[9] = v.z;  m[13] = -dot(eVec,v);
   m[2] = n.x; m[6] = n.y; m[10] = n.z; m[14] = -dot(eVec,n);
   m[3] = 0;   m[7] = 0;   m[11] = 0;   m[15] = 1.0;
   CTM = m; // Finally, load matrix m into CTM Matrix
}
```

- Slide changes **eVec**,
- roll, pitch, yaw, change **u, v, n**
- Call setModelViewMatrix after slide, roll, pitch or yaw

# Example: Camera Roll




$$\mathbf{u}' = \cos(\alpha)\mathbf{u} + \sin(\alpha)\mathbf{v}$$

$$\mathbf{v}' = -\sin(\alpha)\mathbf{u} + \cos(\alpha)\mathbf{v}$$

```
void Camera::roll(float angle)
{  // roll the camera through angle degrees
   float cs = cos(3.142/180 * angle);
   float sn = sin(3.142/180 * angle);
   Vector3 t = u; // remember old u
   u.set(cs*t.x - sn*v.x, cs*t.y - sn.v.y, cs*t.z - sn.v.z);
   v.set(sn*t.x + cs*v.x, sn*t.y + cs.v.y, sn*t.z + cs.v.z)
   setModelViewMatrix( );
}
```

# References

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