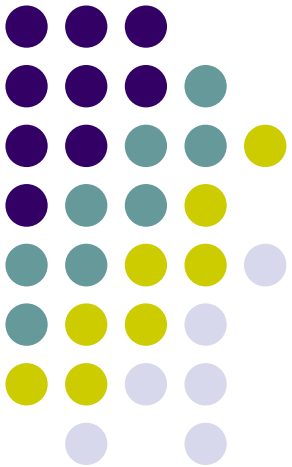
Computer Graphics
CS 543 – Lecture 6 (Part 1)
Setting Camera & Camera Controls

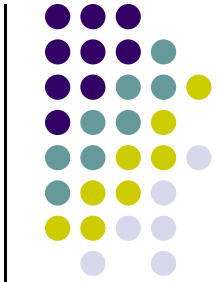
Prof Emmanuel Agu

Computer Science Dept.

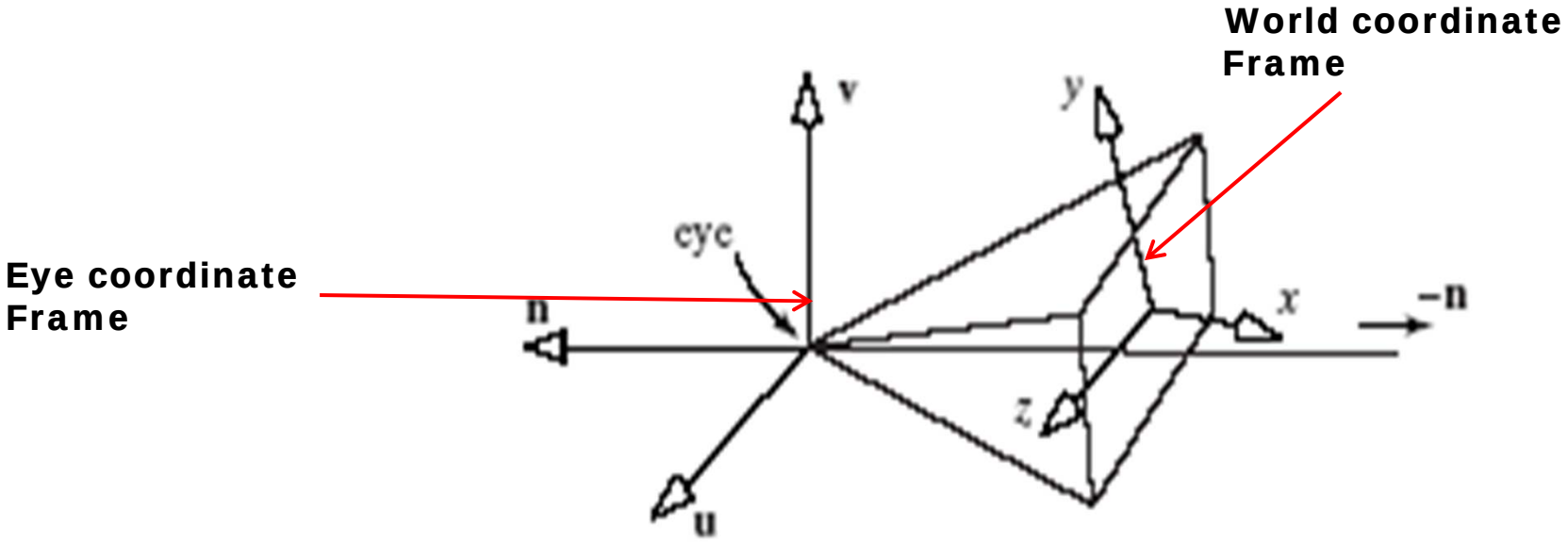
Worcester Polytechnic Institute (WPI)



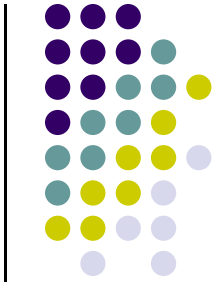
Camera with Arbitrary Orientation and Position



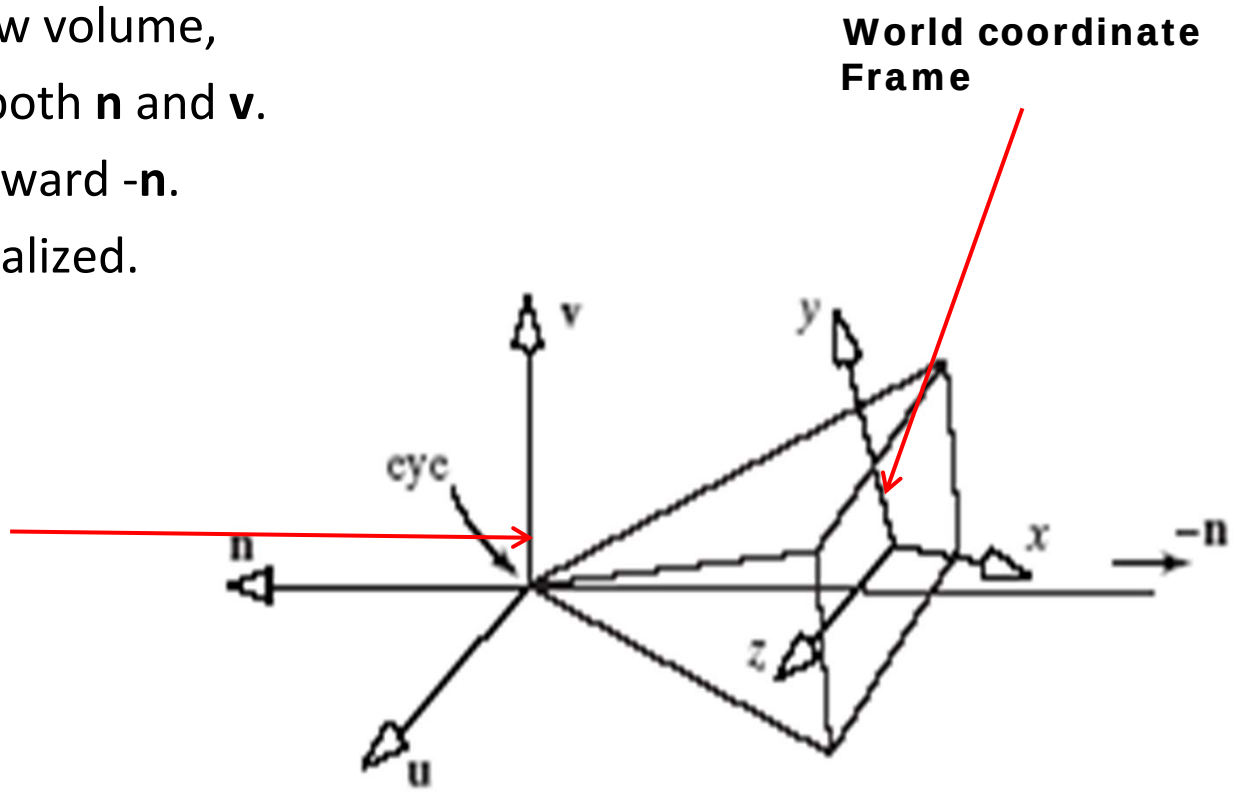
- Programmer defines eye, lookAt and Up
- **Goal:**
  - Form new axes at camera
  - Transform objects from world to eye camera frame



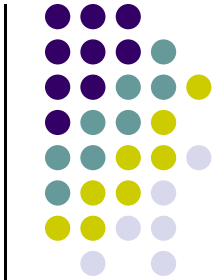
Camera with Arbitrary Orientation and Position



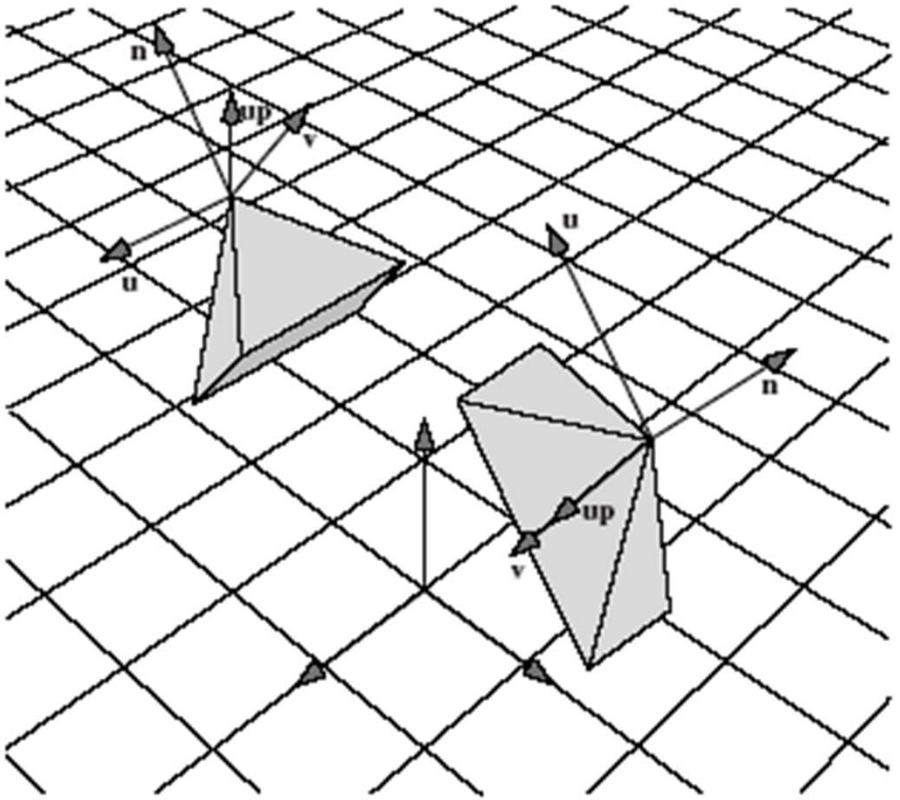
- Define new axes at eye
  - **v** points vertically upward,
  - **n** away from the view volume,
  - **u** at right angles to both **n** and **v**.
  - The camera looks toward -**n**.
  - All vectors are normalized.



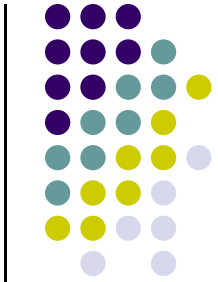
LookAt and Camera Coordinate System



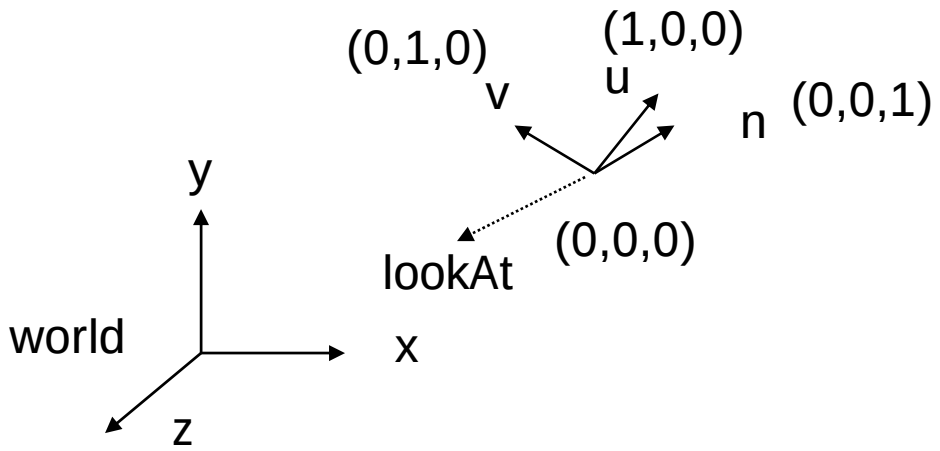
- Effect of LookAt
- Programmer changes **eye**, **lookAt** point
- **u,v,n** changes



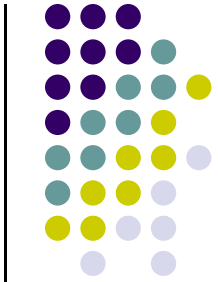
Viewing Transformation



- Transformation?
  - Form a camera (eye) coordinate frame
  - Transform objects from world to eye space
- Eye space?
  - Transform to eye space can simplify many downstream operations (such as projection) in the pipeline

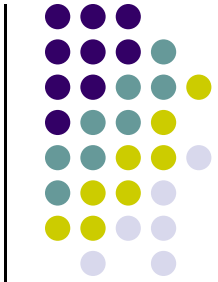


Viewing Transformation

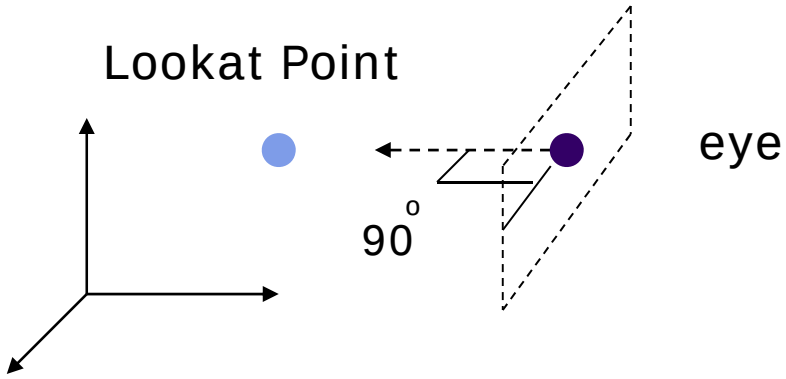


- OpenGL previously had gluLookAt
- We implement similar LookAt function
- LookAt call transforms the object from world to eye space by:
  - Constructing eye coordinate frame (u, v, n)
  - Composes matrix for coordinate transformation
  - Allows flexible Camera Control

Eye Coordinate Frame



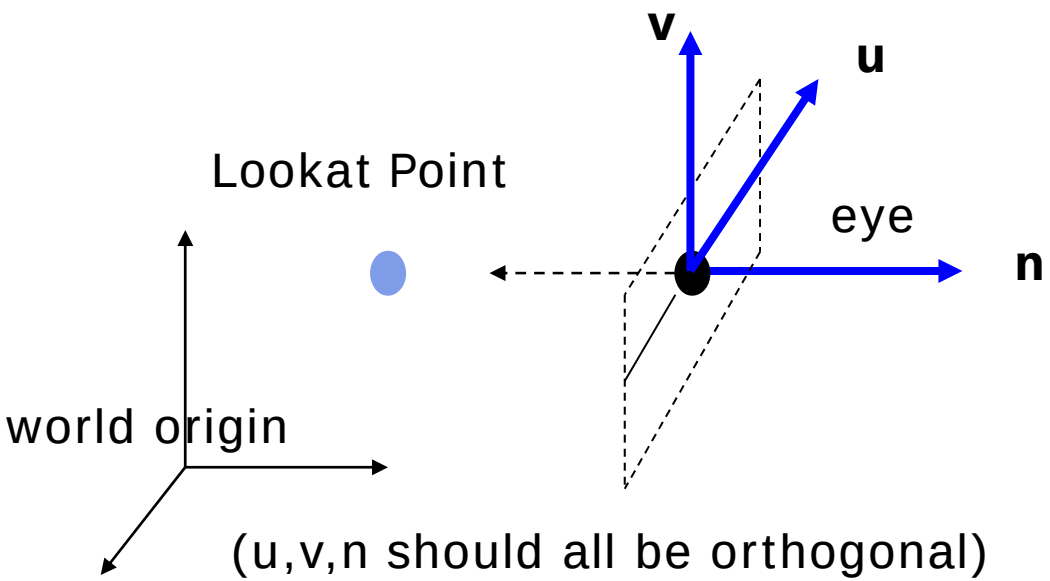
- Constructing u,v,n?
- **Known:** eye position, LookAt Point, up vector
- To find out: new origin and three basis (u,v,n) vectors



Assumption: direction of view is orthogonal to view plane (plane that objects will be projected onto)

Eye Coordinate Frame

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



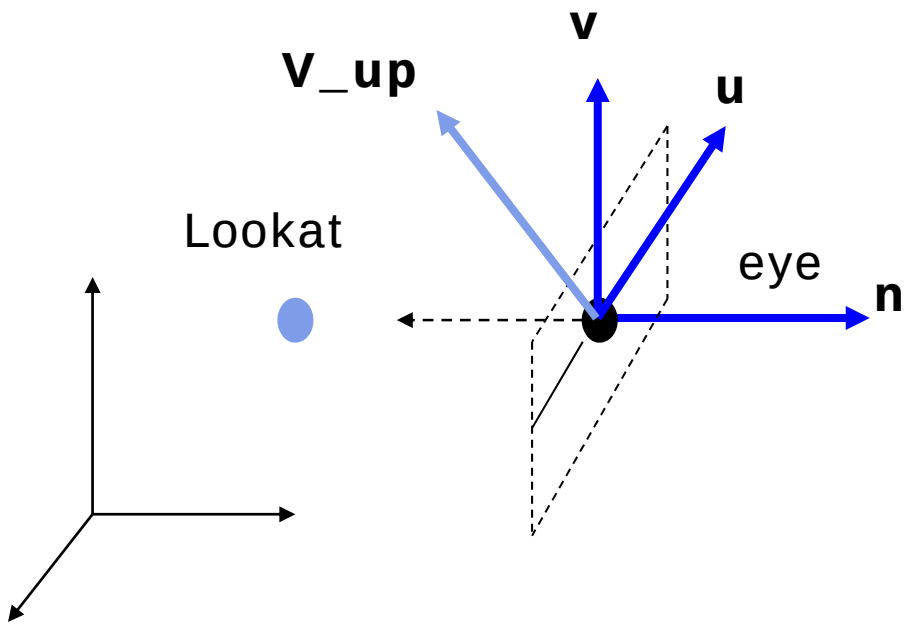
n is pointing away from the world because we use left hand coordinate system

N = eye – Lookat Point
n = N / | N |

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

Eye Coordinate Frame

- How about u and v?



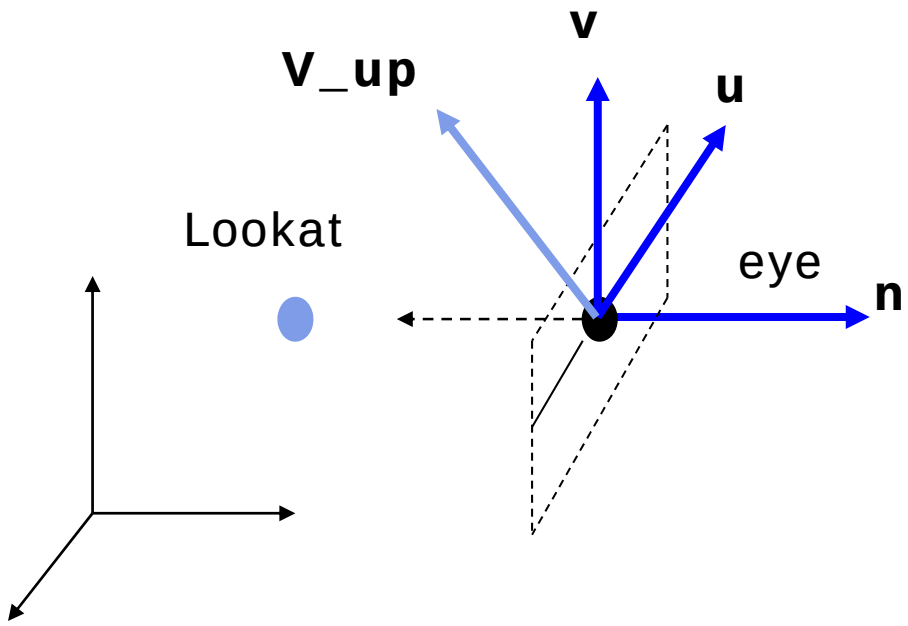
•We can get u first -
  •u is a vector that is perp to the plane spanned by N and view up vector (V_up)

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

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

Eye Coordinate Frame

- How about v?

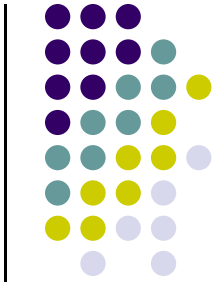


Knowing n and u, getting v is easy

v = n x u

v is already normalized

Eye Coordinate Frame

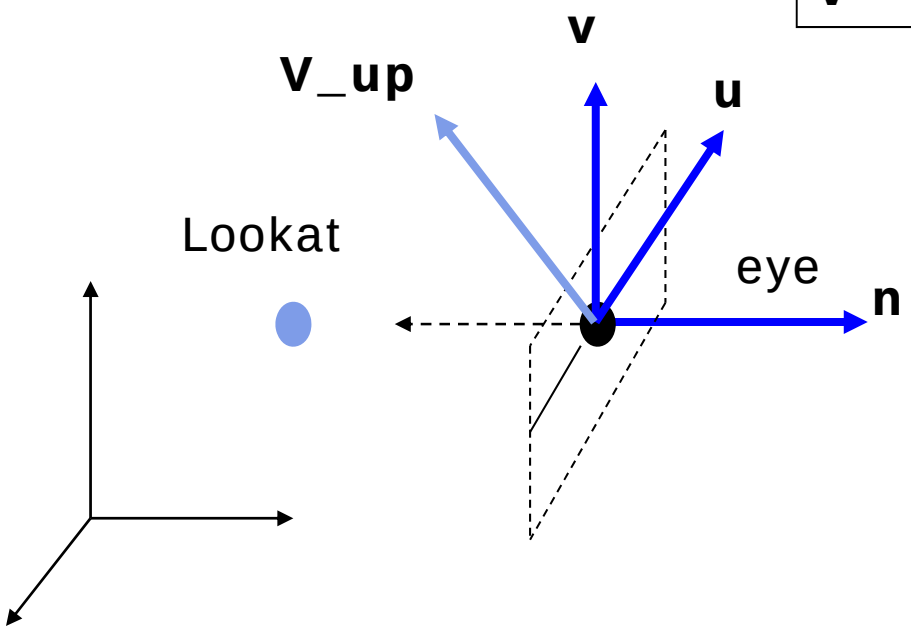


- Put it all together

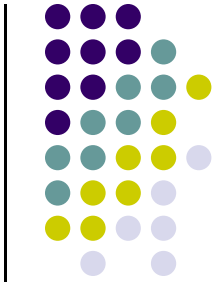
Eye space **origin: (Eye.x , Eye.y,Eye.z)**

Basis vectors:

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

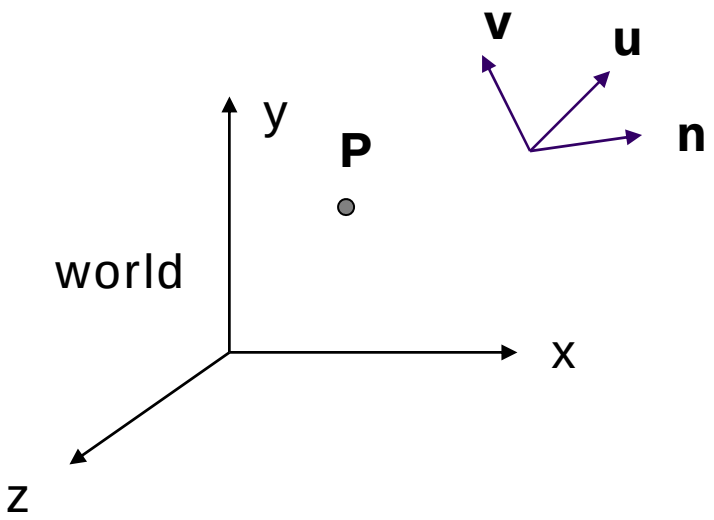


World to Eye Transformation



- Next, use u, v, n to compose V part of modelview
- Transformation matrix (M_{w2e}) ?

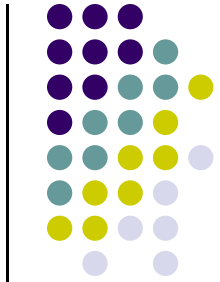
  $P' = M_{w2e} \times P$



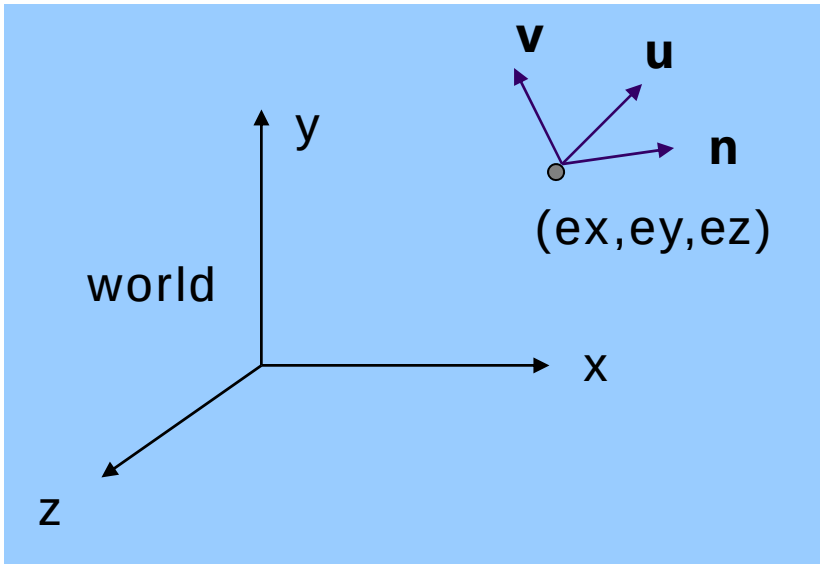
1. Come up with the transformation sequence to move eye coordinate frame to the world

2. And then apply this sequence to the point P in a reverse order

World to Eye Transformation



- Rotate the eye frame to “align” it with the world frame
- Translate (-ex, -ey, -ez)



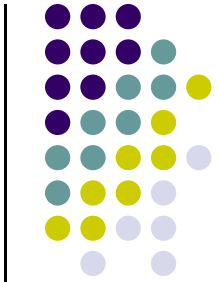
Rotation:

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

Translation:

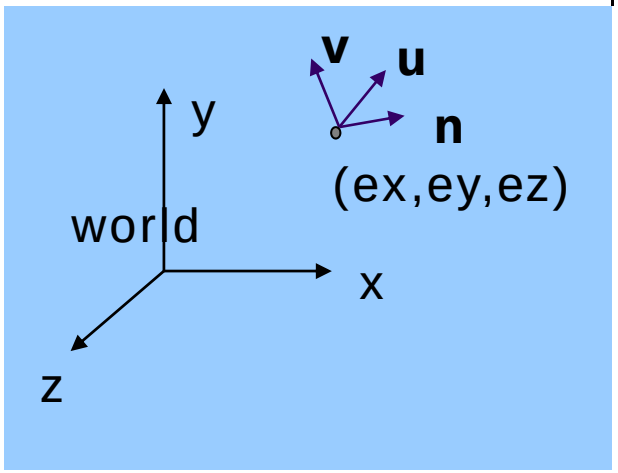
$$\begin{vmatrix} 1 & 0 & 0 & -ex \\ 0 & 1 & 0 & -ey \\ 0 & 0 & 1 & -ez \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

World to Eye Transformation



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

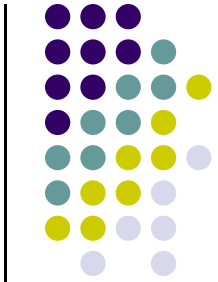
$$Mw2e = \begin{vmatrix} ux & uy & ux & 0 \\ vx & vy & vz & 0 \\ nx & ny & nz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 & -ex \\ 0 & 1 & 0 & -ey \\ 0 & 0 & 1 & -ez \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



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

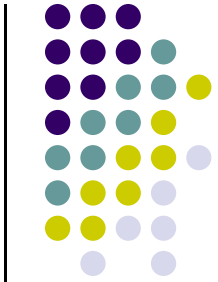
Note: $\mathbf{e.u} = ex.ux + ey.uy + ez.uz$

lookAt Implementation (from mat.h)



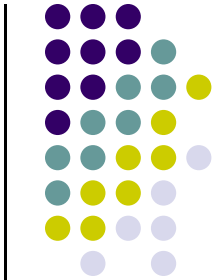
```
mat4 LookAt( const vec4& eye, const vec4& at, const
vec4& up )
{
    vec4 n = normalize(eye - at);
    vec4 u = normalize(cross(up,n));
    vec4 v = normalize(cross(n,u));
    vec4 t = vec4(0.0, 0.0, 0.0, 1.0);
    mat4 c = mat4(u, v, n, t);
    return c * Translate( -eye );
}
```

Other Camera Controls

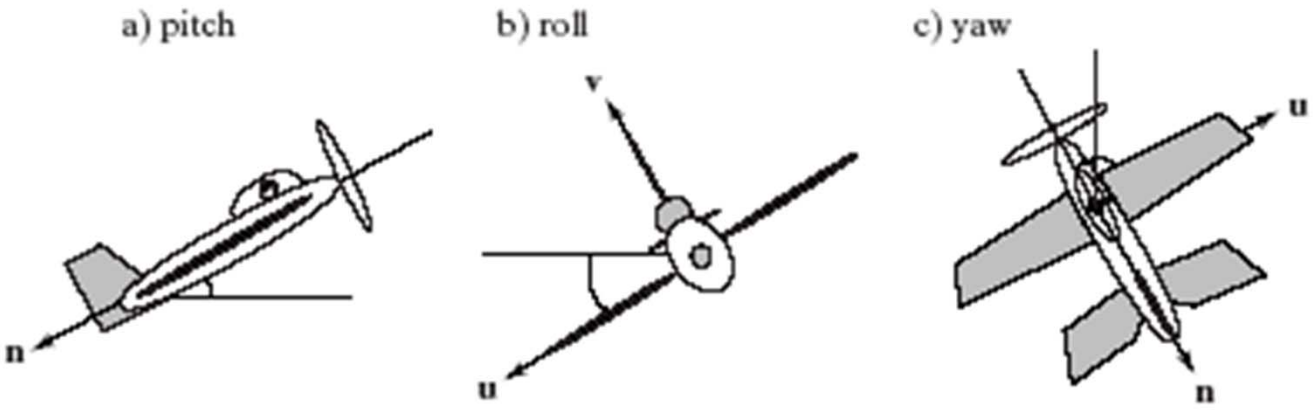


- The LookAt function is only one possible API for positioning the camera
- Other ways to specify camera position/movement
  - Yaw, pitch, roll
  - Elevation, azimuth, twist
  - Direction angles

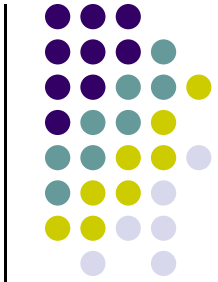
Flexible Camera Control



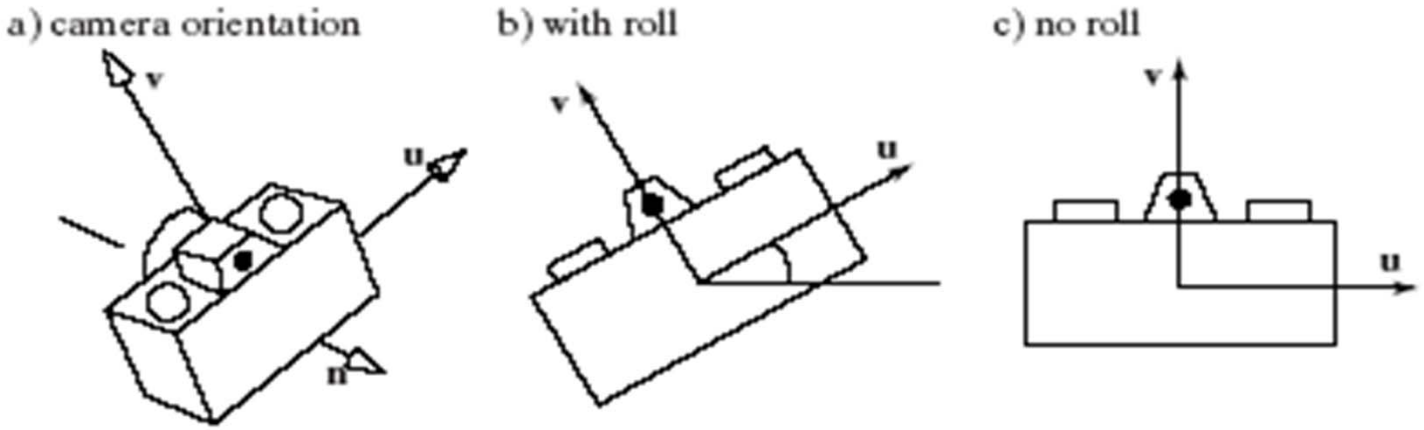
- Sometimes, we want camera to move
- Like controlling a airplane's orientation
- Adopt aviation terms for this
  - **Pitch:** nose up-down
  - **Roll:** roll body of plane
  - **Yaw:** move nose side to side



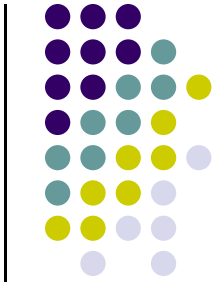
Yaw, Pitch and Roll Applied to Camera



- Similarly, yaw, pitch, roll with a camera



Flexible Camera Control



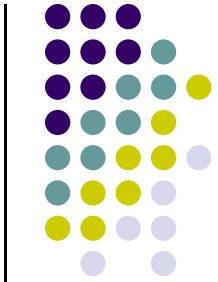
- May create a **camera** class

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

- Let user specify pitch, roll, yaw to change camera. E.g

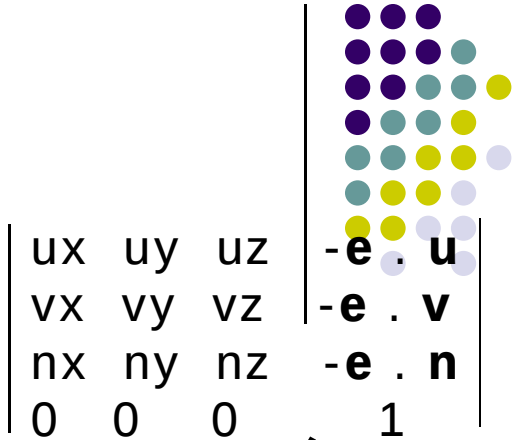
```
cam.slide(-1, 0, -2); // slide camera forward and  left
cam.roll(30);   // roll camera through 30 degrees
cam.yaw(40);   // yaw it through 40 degrees
cam.pitch(20);  // pitch it through 20 degrees
```

Implementing Flexible Camera Control



- Main idea behind flexible camera control
  - Camera class maintains current (u,v,n) and eye position
  - User inputs desired roll, pitch, yaw θ, ϕ angle or slide
  - Calculate modified vector (u, v, n) or new eye position **after** applying roll, pitch, slide, or yaw
  - Compose new modelview matrix yourself
  - Set CTM to modelview matrix

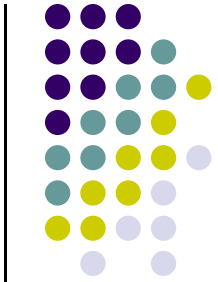
Load Matrix into CTM




```
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;
   Finally, load matrix m into CTM Matrix
}
```

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

Example: Camera Slide

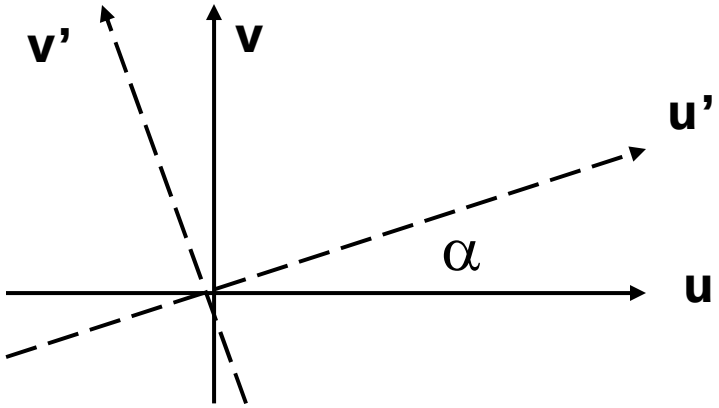


- User changes eye by delU, delV or delN
- eye = eye + changes
- Note: function below combines all slides into one

```
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( );
}
```

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

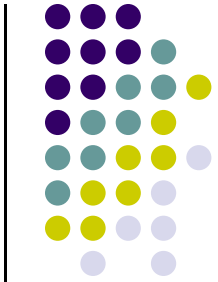
Example: Camera Roll



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

Reference: Sections 7.2, 7.3 of Hill and Kelley

```
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

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