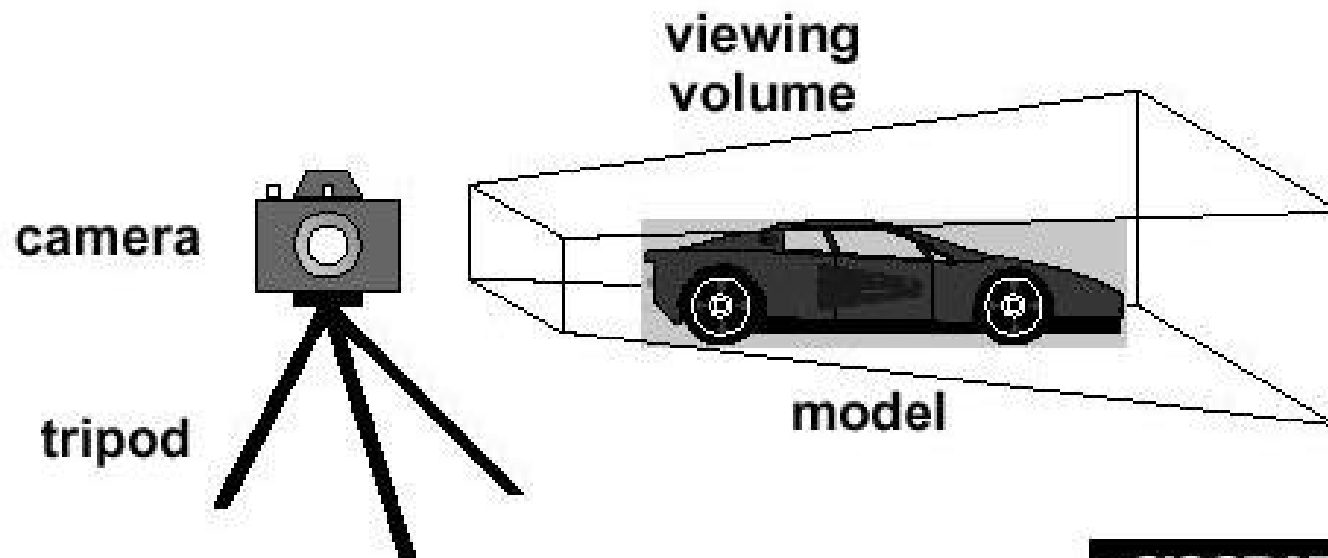


**CS 4731: Computer Graphics**  
**Lecture 13: 3D Viewing and Camera Control**

Emmanuel Agu

## 3D Viewing

- Similar to taking a photograph
- Control the “lens” of the camera
- Project the object from 3D world to 2D screen



# Viewing Transformation

- Recall, setting up the Camera:
  - `gluLookAt (Ex, Ey, Ez, cx, cy, cz, Up_x, Up_y, Up_z)`
  - The view up vector is usually  $(0,1,0)$
  - Remember to set the OpenGL matrix mode to `GL_MODELVIEW` first
- Modelview matrix:
  - combination of modeling matrix  $M$  and Camera transforms  $V$
- `gluLookAt` fills  $V$  part of modelview matrix
- What does `gluLookAt` do with parameters (*eye, LookAt, up vector*) you provide?

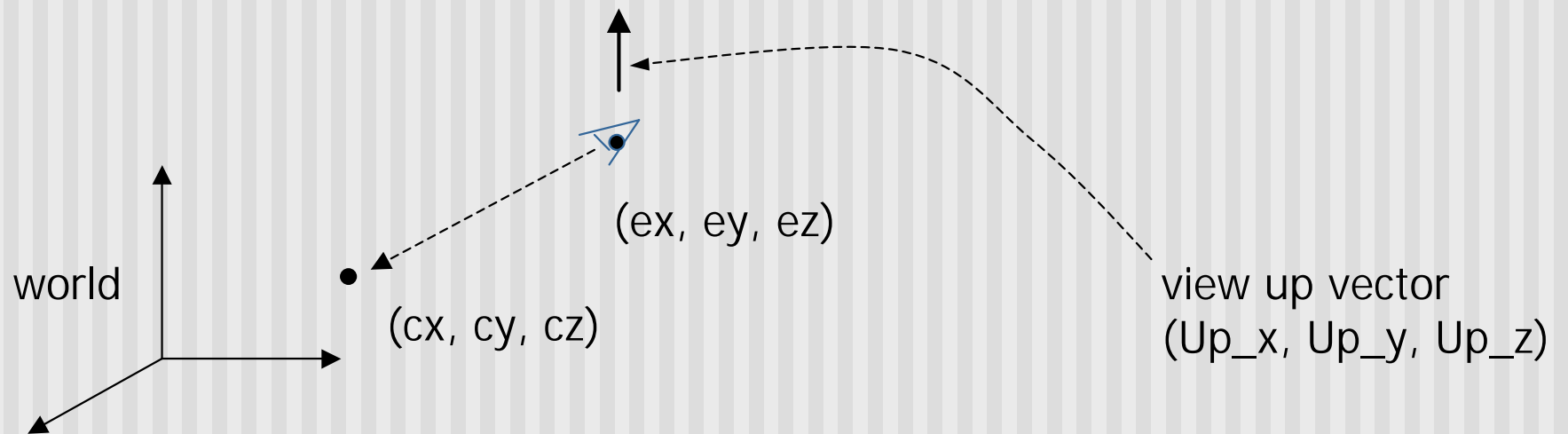
# Viewing Transformation

- OpenGL Code:

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    display_all();    // your display routine
}
```

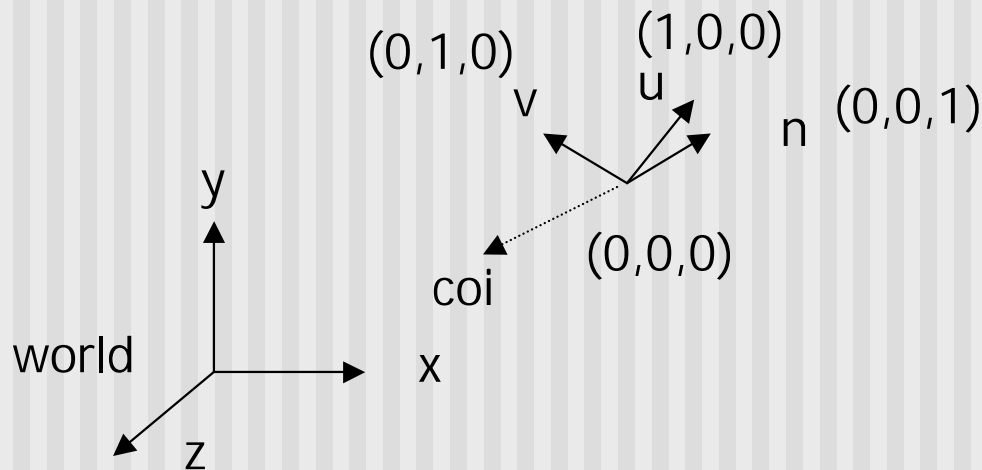
# Viewing Transformation

- Control the “lens” of the camera
- Important camera parameters to specify
  - Camera (eye) position  $(Ex, Ey, Ez)$  in world coordinate system
  - lookAt point  $(cx, cy, cz)$
  - Orientation (which way is up?): Up vector  $(Up_x, Up_y, Up_z)$



# 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

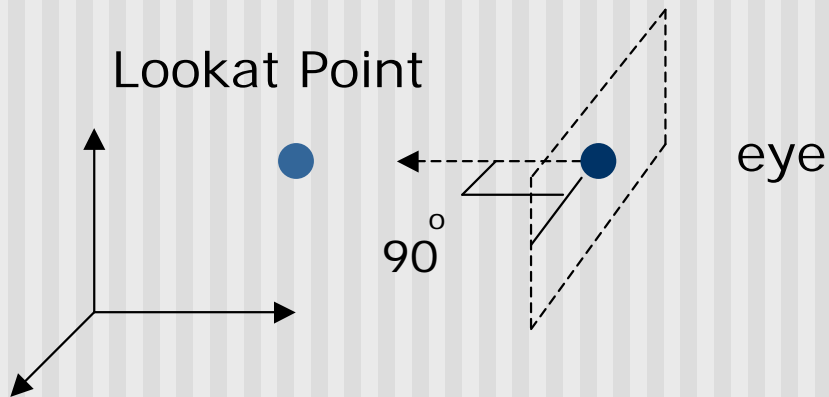


# Viewing Transformation

- gluLookAt call transforms the object from world to eye space by:
  - Constructing eye coordinate frame (u, v, n)
  - Composes matrix to perform coordinate transformation
  - Loads this matrix into the V part of modelview matrix
  - Allows flexible Camera Control

# Eye Coordinate Frame

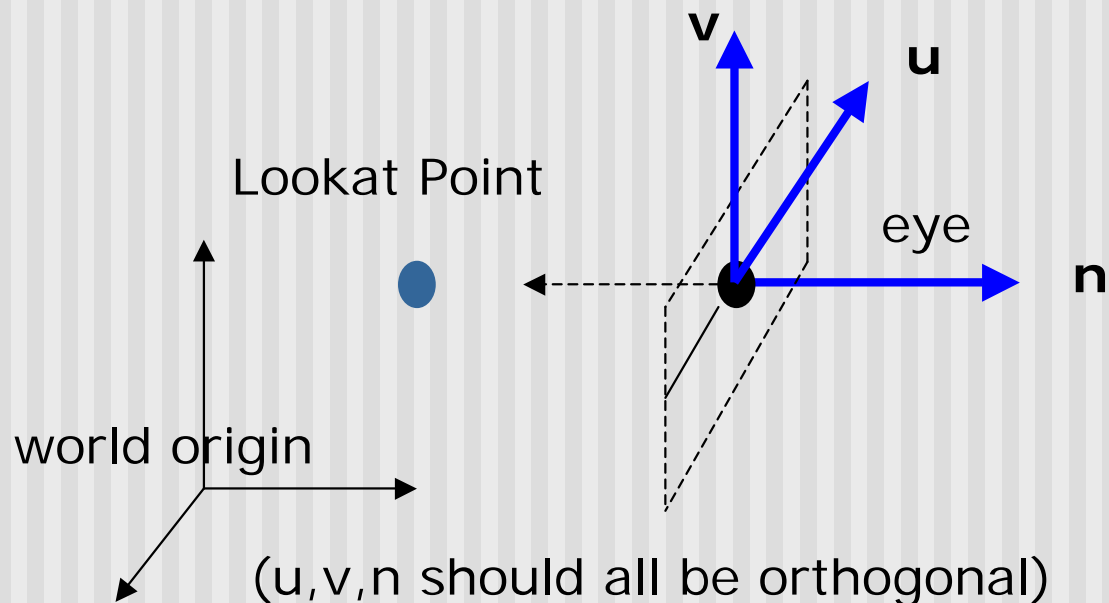
- Constructing  $u, v, n$ ?
- Known: eye position, LookAt Point, up vector
- To find out: new origin and three basis vectors



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 ( $\mathbf{n}$ ) of the viewing plane,
  - other two ( $\mathbf{u}$  and  $\mathbf{v}$ ) span the viewing plane



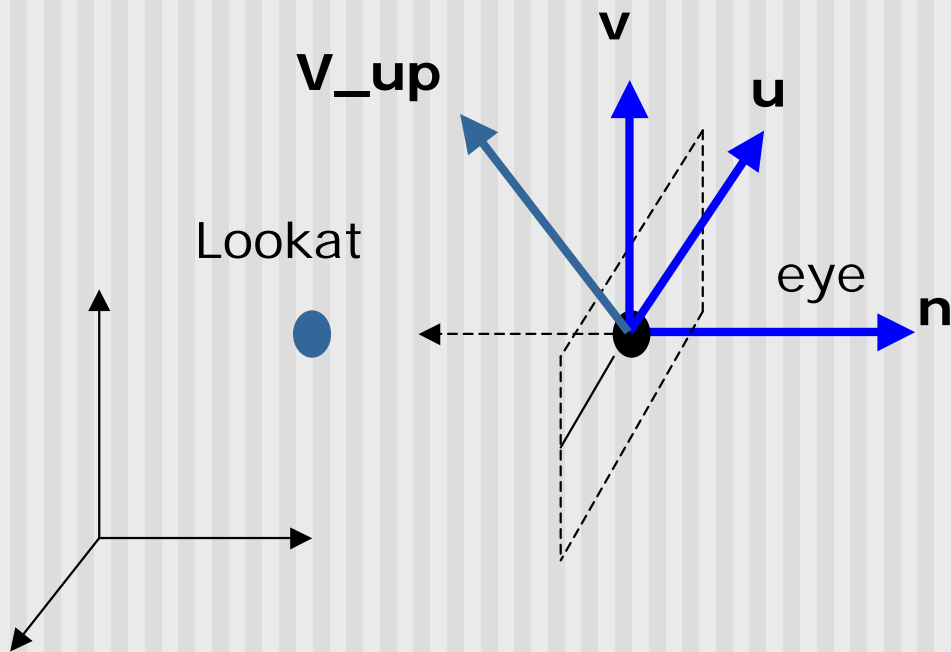
$\mathbf{n}$  is pointing away from the world because we use left hand coordinate system

$$\mathbf{N} = \text{eye} - \text{Lookat Point}$$
$$\mathbf{n} = \mathbf{N} / |\mathbf{N}|$$

Remember  $\mathbf{u}, \mathbf{v}, \mathbf{n}$  should be all unit vectors

# Eye Coordinate Frame

- How about u and v?



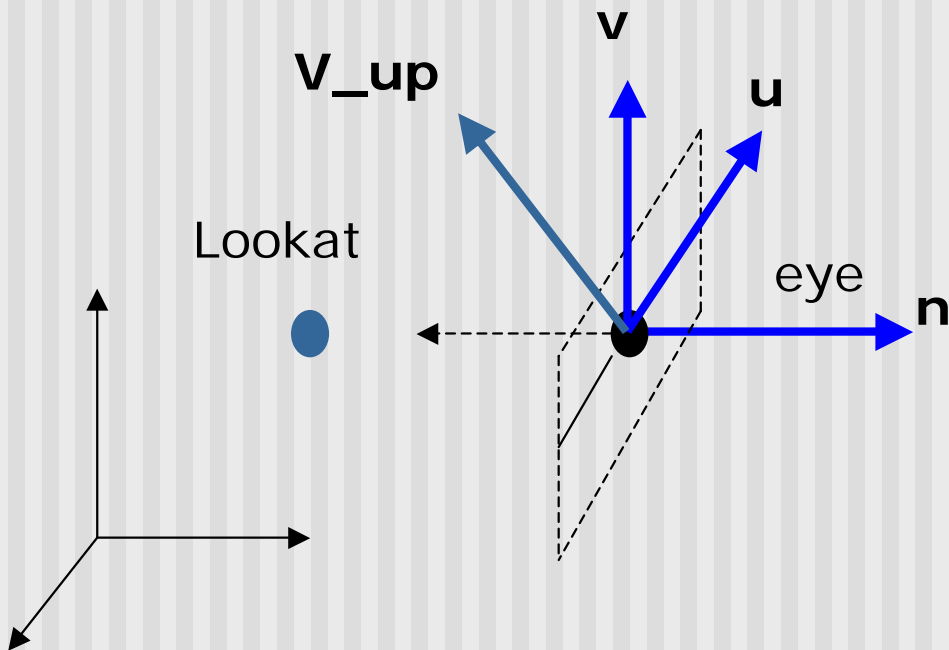
- We can get  $\mathbf{u}$  first -
  - $\mathbf{u}$  is a vector that is perp to the plane spanned by  $\mathbf{N}$  and view up vector ( $\mathbf{V\_up}$ )

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

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

# Eye Coordinate Frame

- How about v?



Knowing **n** and **u**, getting **v** is easy

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

**v** is already normalized

## Eye Coordinate Frame

- Put it all together

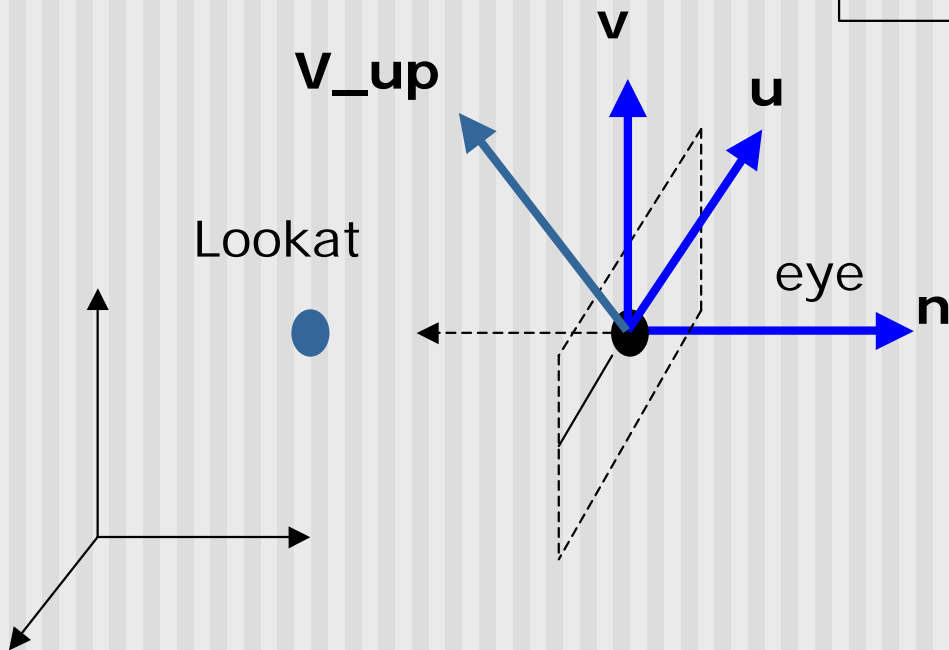
Eye space **origin**: **(Eye.x , Eye.y, Eye.z)**

Basis vectors:

$$\mathbf{n} = (\text{eye} - \text{Lookat}) / |\text{eye} - \text{Lookat}|$$

$$\mathbf{u} = (\mathbf{V\_up} \times \mathbf{n}) / |\mathbf{V\_up} \times \mathbf{n}|$$

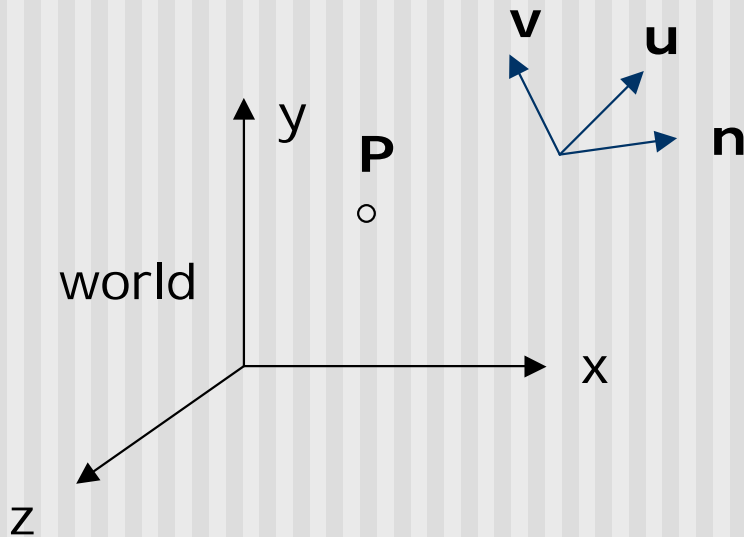
$$\mathbf{v} = \mathbf{n} \times \mathbf{u}$$



# World to Eye Transformation

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

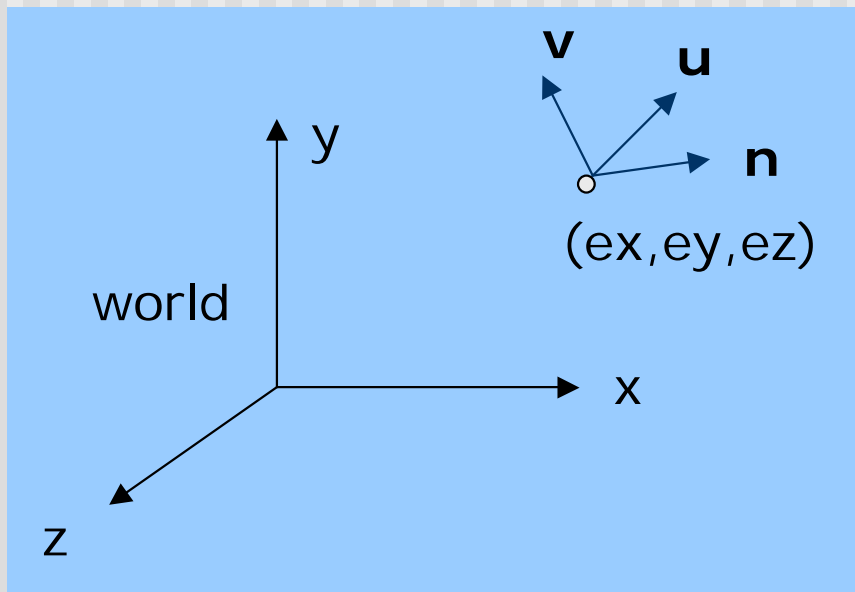
$$P' = M_{w2e} \times P$$



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:

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

Translation:

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

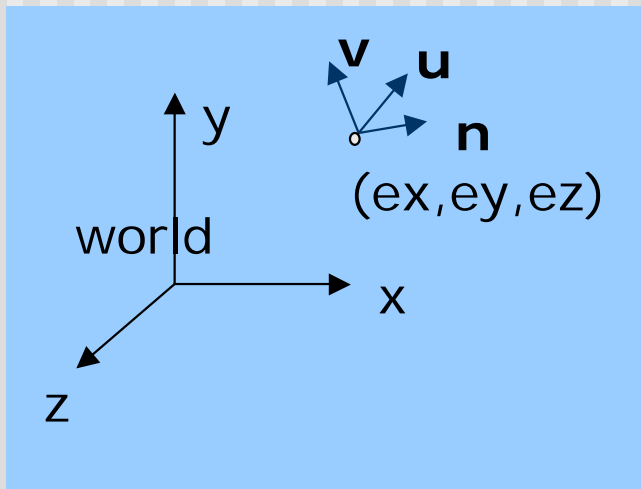
## World to Eye Transformation

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

$$M_{w2e} = \begin{vmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

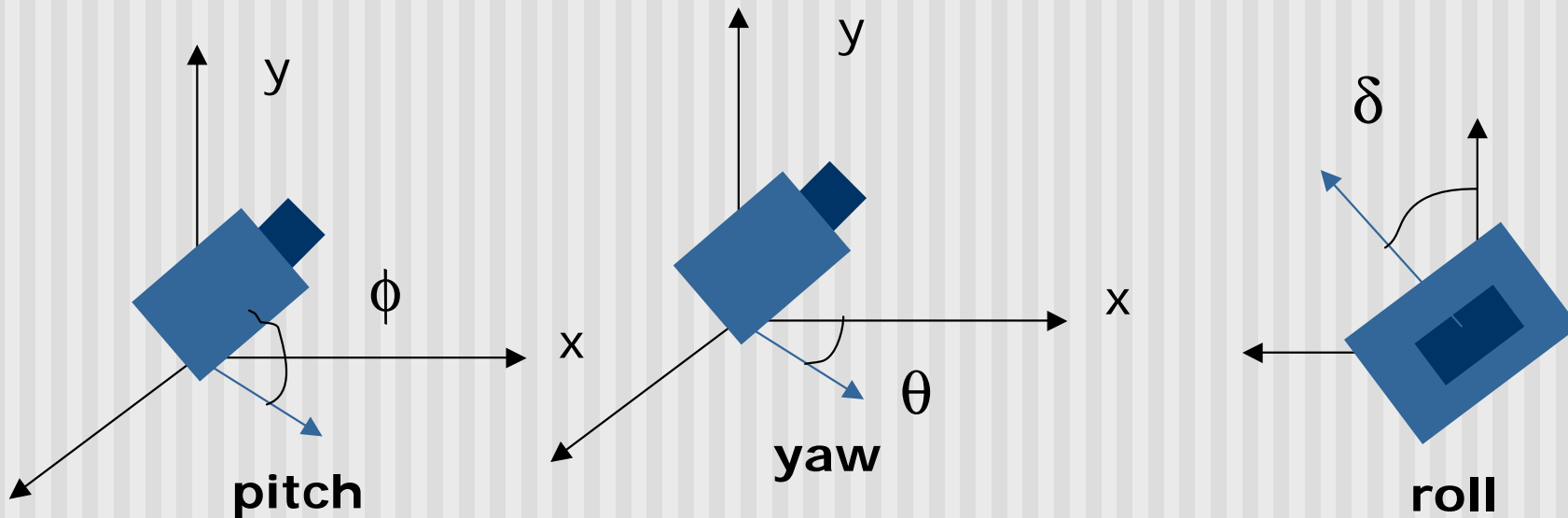
$$= \begin{vmatrix} u_x & u_y & u_z & -\mathbf{e} \cdot \mathbf{u} \\ v_x & v_y & v_z & -\mathbf{e} \cdot \mathbf{v} \\ n_x & n_y & n_z & -\mathbf{e} \cdot \mathbf{n} \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Note:  $\mathbf{e} \cdot \mathbf{u} = e_x \cdot u_x + e_y \cdot u_y + e_z \cdot u_z$



# Flexible Camera Control

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



## 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
- Example:

```
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
```

## Flexible Camera Control

- `gluLookAt()` **does not** let you control roll, pitch and yaw
- Main idea behind flexible camera control
  - User supplies  $\theta$ ,  $\phi$  or roll angle
  - Constantly maintain the vector  $(u, v, n)$  by yourself
  - Calculate new  $u'$ ,  $v'$ ,  $n'$  **after** roll, pitch, slide, or yaw
  - Compose new  $V$  part of modelview matrix yourself
  - Set modelview matrix directly yourself using **`glLoadMatrix`** call

## Loading Modelview Matrix directly

```
void Camera::setModelViewMatrix(void)
{ // load modelview matrix with existing camera values
  float m[16];
  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] = -eVec.dot(u);
  m[1] = v.x; m[5] = v.y; m[9] = v.z;   m[13] = -eVec.dot(v);
  m[2] = n.x; m[6] = n.y; m[10] = n.z;  m[14] = -eVec.dot(n);
  m[3] = 0;   m[7] = 0;   m[11] = 0;   m[15] = 1.0;
  glMatrixMode(GL_MODELVIEW);
  glLoadMatrixf(m); // load OpenGL's modelview matrix
}
```

Above setModelViewMatrix acts like gluLookAt

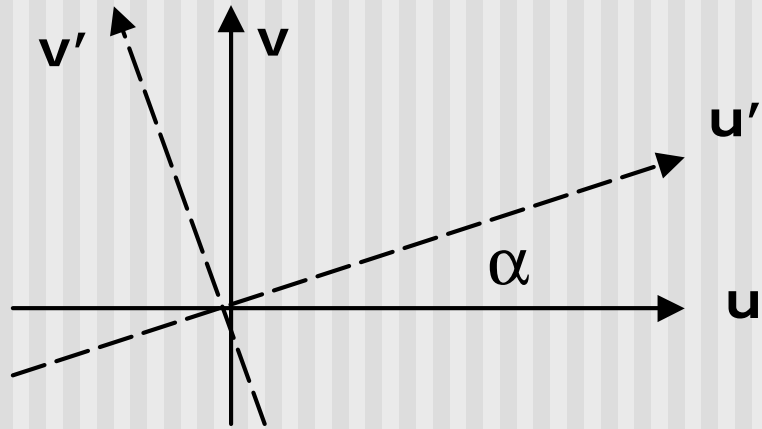
Slide changes eVec, roll, pitch, yaw, change u, v, n

## Camera Slide

- User changes eye by delU, delV or delN
- $\text{eye} = \text{eye} + \text{changes}$
- 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( );
}
```

## Camera Roll



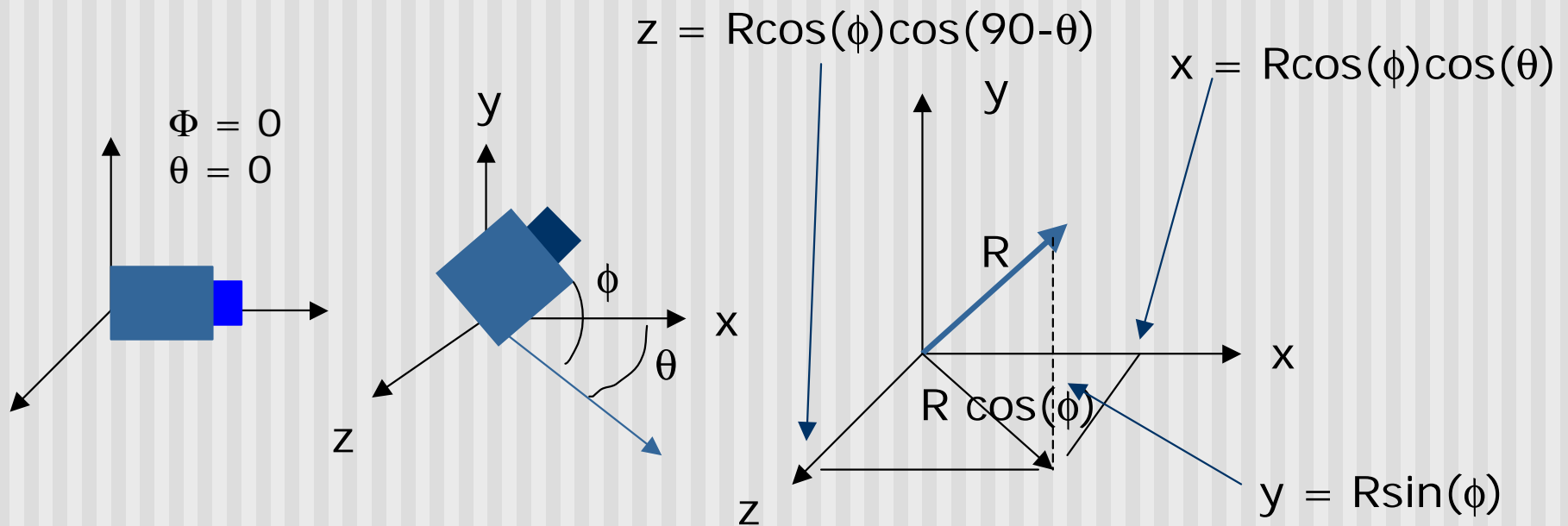
$$\mathbf{u}' = \cos(\mathbf{a})\mathbf{u} + \sin(\mathbf{a})\mathbf{v}$$

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

```
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( );
}
```

# Flexible Camera Control

- How to compute the viewing vector  $(x,y,z)$  from pitch( $\phi$ ) and yaw( $\theta$ ) ? Read sections 7.2, 7.3 of Hill



## References

- Hill, chapter 7