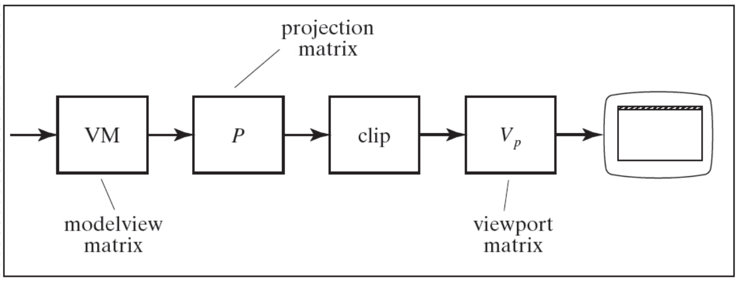
CS 543 - Computer Graphics: 3D Camera Control

by
Robert W. Lindeman
gogo@wpi.edu
(with help from Emmanuel Agu ;-)

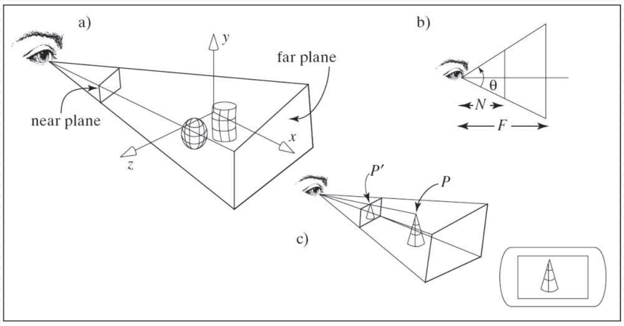
Modelview Matrix

- Recall the graphics pipeline
  - Modelview matrix is composed of the scene transformations, **M**, and the camera transformations, **V**
  - Here we will focus on **V**



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(` $e_x, e_y, e_z, c_x, c_y, c_z, Up_x, Up_y, Up_z$ `)`
  - 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, interest, up vector*) you provide?

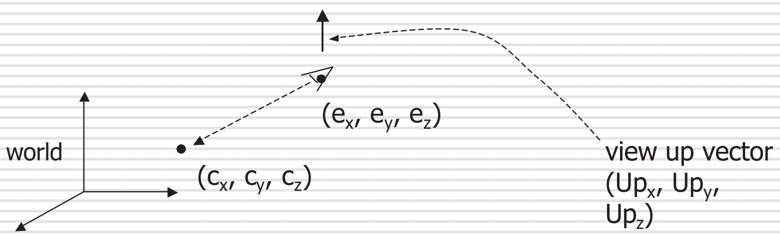
Viewing Transformation (cont.)

- OpenGL Code

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

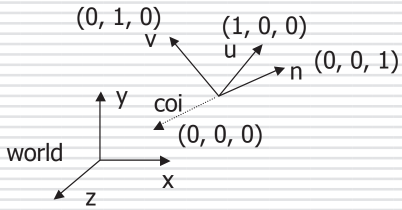

Viewing Transformation (cont.)

- Control the "lens" of the camera
- Important camera parameters to specify
  - Camera (eye) position (e_x, e_y, e_z) in world coordinate system
  - Center-of-interest point (c_x, c_y, c_z)
  - Orientation (which way is up?): Up vector (Up_x, Up_y, Up_z)



Viewing Transformation (cont.)

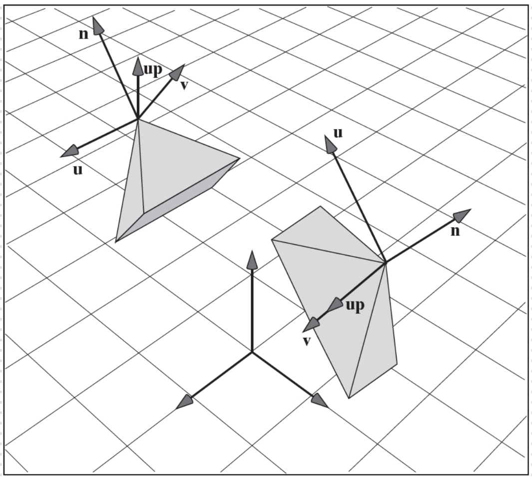
- Transformation?
  - Form a camera (eye) coordinate frame
  - Transform objects from world to eye space
- Eye space?
  - Transforming to eye space can simplify many downstream operations (such as projection) in the pipeline



Viewing Transformation (cont.)

- gluLookAt call transforms the object from world to eye space by
  - Constructing eye coordinate frame (u, v, n)
  - Composing matrix to perform coordinate transformation
  - Loading this matrix into the V part of modelview matrix
- Allows flexible camera control

Sample Cameras

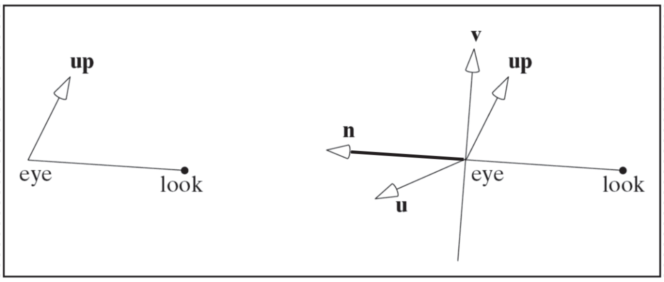


Computing LookAt

- How do we construct **u**, **v**, **n**?
- Known
  - eye position
  - Center of interest (look)
  - Up vector (just a hint)
- Need to find
  - New origin
  - Three basis vectors (axes)

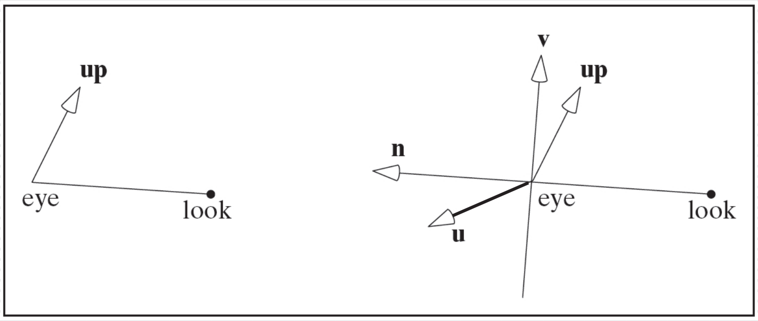
Eye Coordinate Frame

- Origin = eye position (that was easy!)
- Three basis vectors
  - Should be orthogonal and normalized
  - **n** = (eye - look) / |eye - look|



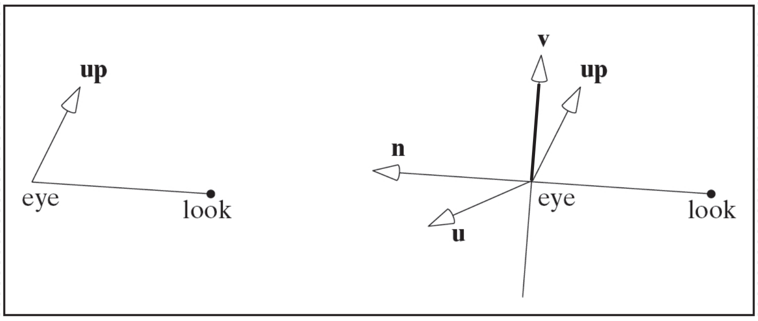
Eye Coordinate Frame (cont.)

- How about **u** and **v**?
  - **u** = (**Up** x **n**) / |**Up** x **n**|
  - How come this works?



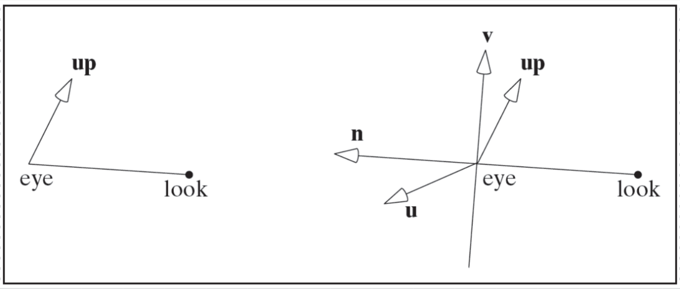
Eye Coordinate Frame (cont.)

- How about **v**?
  - **v** = **n** x **u**
  - Why is this already normalized?



Putting It All Together

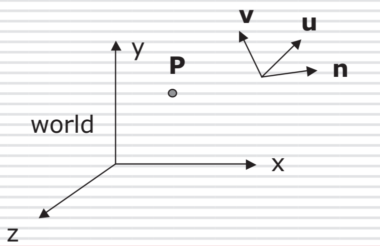
- Eye space
  - Origin = (eye_x, eye_y, eye_z)
  - **n** = (eye - look) / |eye - look|
  - **u** = (**Up** x **n**) / |**Up** x **n**|
  - **v** = **n** x **u**



World to Eye Transformation

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

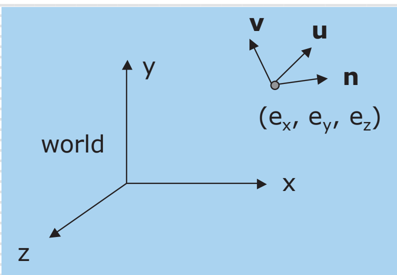
  $P' = M_{w2e} \times P$



1. Come up with the transformation sequence to move the eye coordinate frame to the world coordinate frame
2. Apply this sequence to the point P in reverse order

World to Eye Transformation (cont.)

- Rotate the eye frame to "align" it with the world frame
- Translate ($-e_x$, $-e_y$, $-e_z$)



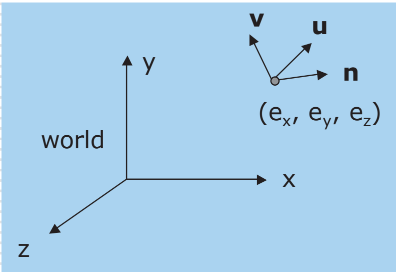
Rotation:
$$\begin{vmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Translation:
$$\begin{vmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

World to Eye Transformation (cont.)

- Transformation order
  - Apply the transformation to the object in *reverse* order - translate first, and then rotate

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



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

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

Flexible Camera Control

- May create a **camera** class

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

- Let user specify roll, pitch, yaw to change camera
- Example

```
cam.slide( -1, 0, -2 ); // move 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 (cont.)

- `gluLookAt( )` **does not** let you control roll, pitch & yaw
- Main idea behind flexible camera control
  - User supplies θ, ϕ 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`

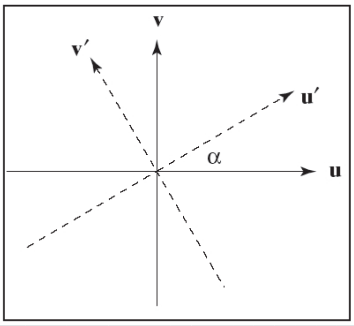
Camera Slide

- User changes eye by delU, delV or delN
- 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( );
}
```


Camera Roll

```
void Camera::roll( float angle )  {
  // roll the camera through angle degrees
  float cs = cos( M_PI/180 * angle );
  float sn = sin( M_PI/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( );
}
```



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

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

References

- Hill, Chapter 7