3D Viewing?

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

1. Set camera position

2. Set view volume (3D region of interest)
Different View Volume Shapes

- Different view volume shape => different look
- **Foreshortening?** Near objects bigger
  - Perspective 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

Objects initially specified in world frame are more natural to view 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**

- **default frames**
- **Same RELATIVE distance after**
- **Same result/look**
- Translate objects -5 away from camera
- Translate camera +5 away from objects
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
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 \( C = TR \)

```cpp
// 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

```cpp
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)

Programmer defines:
• eye position
• LookAt point (at) and
• Up vector (Up direction usually (0,1,0))

But Why do we set Up direction?
Nate Robbins LookAt Demo

```
glTranslatef( 0.00 , 0.00 , 0.00 );
glRotatef( 0.0 , 0.00 , 1.00 , 0.00 );
glScalef( 1.00 , 1.00 , 1.00 );
glBegin( ... );
... 

Click on the arguments and move the mouse to modify values.
```

```
GLfloat pos[4] = { 1.50 , 1.00 , 1.00 , 0.00 };
gluLookAt( 0.00 , 0.00 , 2.00 , <- eye
          0.00 , 0.00 , 0.00 , <- center
          0.00 , 1.00 , 0.00 ); <- up

glLightfv(GL_LIGHT0, GL_POSITION, pos);

Click on the arguments and move the mouse to modify values.
```
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

- 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 \texttt{LookAt(eye, at, up)}
- If \texttt{eye, lookAt} point changes \(\Rightarrow\) \texttt{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 \( \mathbf{n} \) 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

\[
\mathbf{N} = \text{eye} - \text{Lookat Point}
\]

\[
\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|}
\]

Remember \( \mathbf{u}, \mathbf{v}, \mathbf{n} \) 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 perpendicular to the plane spanned by $N$ and view up vector ($V_{up}$)

$$U = V_{up} \times n$$

$$u = U / |U|$$
Eye Coordinate Frame

- How about v?

To derive v from n and u

\[ v = n \times u \]

v is already normalized
Eye Coordinate Frame

- Put it all together

Eye space origin: \((\text{Eye}.x, \text{Eye}.y, \text{Eye}.z)\)

Basis vectors:

\[
\begin{align*}
\mathbf{n} &= \frac{\text{eye} - \text{Lookat}}{|\text{eye} - \text{Lookat}|} \\
\mathbf{u} &= \frac{\text{V}_\text{up} \times \mathbf{n}}{|\text{V}_\text{up} \times \mathbf{n}|} \\
\mathbf{v} &= \mathbf{n} \times \mathbf{u}
\end{align*}
\]
Step 2: World to Eye Transformation

- Next, use $u$, $v$, and $n$ 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

\[
\begin{align*}
\text{Rotation:} & \quad \begin{bmatrix}
ux & uy & uz & 0 \\
vx & vy & vz & 0 \\
nx & ny & nz & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \\
\text{Translation:} & \quad \begin{bmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{bmatrix}
\end{align*}
\]
World to Eye Transformation

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

\[
M_{w2e} =
\]

\[
\begin{bmatrix}
ux & uy & ux & 0 \\
vx & vy & vz & 0 \\
xn & yn & zn & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

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** *(Eye.x, Eye.y, Eye.z)*

Basis vectors:

- \( \mathbf{n} = \frac{(\mathbf{eye} - \mathbf{Lookat})}{|\mathbf{eye} - \mathbf{Lookat}|} \)
- \( \mathbf{u} = \frac{(\mathbf{V}_{\text{up}} \times \mathbf{n})}{|\mathbf{V}_{\text{up}} \times \mathbf{n}|} \)
- \( \mathbf{v} = \mathbf{n} \times \mathbf{u} \)

**Mat4 LookAt:**

```cpp
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

a) camera orientation

b) with roll

c) no roll
Flexible Camera Control

- Create a **camera** class

```cpp
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

\[
\begin{bmatrix}
ux & uy & uz \\
vx & vy & vz \\
nx & ny & nz \\
0 & 0 & 0
\end{bmatrix}
- e \cdot u
- e \cdot v
- e \cdot n
1
\]

- 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 = $\text{eye} + Du$

```cpp
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

```cpp
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[2] = n.x; m[6] = n.y; m[10] = n.z; m[14] = -dot(eVec, n);  
    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
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