Computer Graphics (CS 543)  
Lecture 6 (Part 2): Viewing & Camera Control  

Prof Emmanuel Agu  

Computer Science Dept.  
Worcester Polytechnic Institute (WPI)
Objectives

- Introduce viewing functions
- Look at enhanced camera controls
3D Viewing?

- Scene objects **inside** view volume show up on screen
- Objects outside view column **clipped**!

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

- **Foreshortening?** Near objects bigger
- Perpective projection has **foreshortening**
- Orthogonal projection: no foreshortening
The World Frames

- Objectsscene initially defined in **world frame**
- Transformations (translate, scale, rotate) applied to objects in **world frame**
Camera Frame

- More natural to refer to object positions relative to eye
- After we define camera (eye) position, then represent objects in **camera frame** (origin at eye position)
- Objects positions in world frame to positions in camera frame using **model-view matrix**
The OpenGL Camera

- Initially object and camera frames the same
- Camera located at origin and points in negative z direction
- Default view volume is cube (orthogonal) with sides of length 2, at origin

View volume (only objects inside are seen)

clipped out

Projection plane

z=0
Moving the Camera Frame

- If we want to move objects some distance from camera (e.g. 5m from camera), we can either
  1. Move camera backwards -5m (in +z direction)
  2. Move objects forwards +5m (in -z direction)

- Both approaches yield same result

- 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)**
Moving Camera back from Origin

frames after translation by $-d$

$d > 0$

default frames

(a)  

(b)
Moving the Camera

- We can move camera to any position by a 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

- The GLU library contained function `gluLookAt` to form required modelview matrix for camera positioning
- `gluLookAt` deprecated!!
- Homegrown mat4 method LookAt() in mat.h
  - Can concatenate with modeling transformations

```cpp
void display() {
    ........

    mat4 mv = LookAt(vec4 eye, vec4 at, vec4 up);
    ........
}
```
LookAt

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

```c
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.
```

Click on the arguments and move the mouse to modify values.
Camera with Arbitrary Orientation and Position

- Programmer defines eye, lookAt and Up
- **LookAt method:**
  - Form new axes \((u, v, n)\) 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.

World coordinate Frame

Eye coordinate Frame
LookAt: Effect of Changing Eye Position or LookAt Point

- Programmer sets \texttt{LookAt(eye, at, up)}
- If \texttt{eye, lookAt} point changes => \texttt{u,v,n} changes
Viewing Transformation

- Viewing Transformation?
  - Form a camera \((u,v,n)\) coordinate frame
  - Transform objects from world to eye space (Composes matrix for coordinate transformation)
- So, first, let’s form camera \((u,v,n)\) frame
Eye Coordinate Frame

• Constructing u,v,n?
• Lookat function parameters: \texttt{LookAt(eye, at, up)}
• \textbf{Known}: eye position, LookAt Point, up vector
• Derive: new origin and three basis (u,v,n) vectors

\begin{tikzpicture}
  \node (eye) at (0,0) [violet] {	extbullet};
  \node (lookat) at (-2,2) [blue] {	extbullet};
  \draw[thick,->] (eye) -- (lookat); \node at (-1,1) {$90^\circ$};
  \draw[dashed,->] (eye) -- (0,0,1); \node at (0.5,0.5,1) {eye};
  \node at (-2,0) {Lookat Point};
\end{tikzpicture}

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

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

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

Remember \( \mathbf{u}, \mathbf{v}, \mathbf{n} \) should be all unit vectors
Eye Coordinate Frame

How about u and v?

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

\[
U = V_{\text{up}} \times n
\]

\[
u = U / |U|
\]
Eye Coordinate Frame

- How about \( v \)?

Knowing \( n \) and \( u \), getting \( v \) is easy

\[
v = n \times u
\]

\( v \) is already normalized
Eye Coordinate Frame

- Put it all together

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

Basis vectors:

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

- Next, use $u$, $v$, $n$ to compose LookAt matrix
- Transformation matrix ($M_{w2e}$)?
  
  \[ P' = M_{w2e} \times P \]

1. Come up with transformation sequence that lines up 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:

\[
\begin{bmatrix}
ux & uy & uz & 0 \\
vx & vy & vz & 0 \\
nx & ny & nz & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Translation:

\[
\begin{bmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
World to Eye Transformation

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

\[
\begin{bmatrix}
ux & uy & 0 & 0 \\
vx & vy & vz & 0 \\
x & y & n & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Note: \( e.u = ex.ux + ey.uy + ez.uz \)
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 way of 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:
  - **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

- Create a **camera** class

```cpp
class Camera

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

- User can specify pitch, roll, yaw to change camera. E.g

```cpp
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

- General approach
  - Camera class maintains current \((u,v,n)\) and eye position

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

- User inputs desired roll, pitch, yaw angle or slide
- Calculate modified vector \((u,v,n)\) or new eye position after applying roll, pitch, slide, or yaw
- Compose and load modified modelview matrix (CTM)
Load Matrix into CTM

```c++
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
}
```

- 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 (delU, delV, delN)
- Note: function below combines all slides into one

```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( );
}
```

E.g moving camera by \( D \) along its u axis
= \( \text{eye} + Du \)
Example: Camera Roll

\[
\begin{align*}
v' &= \cos(\alpha)u + \sin(\alpha)v \\
v' &= -\sin(\alpha)u + \cos(\alpha)v
\end{align*}
\]

```cpp
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