CS 543: Computer Graphics

Camera Control

Robert W. Lindeman
Associate Professor
Interactive Media & Game Development
Department of Computer Science
Worcester Polytechnic Institute
gogo@wpi.edu

(with lots of help from Prof. 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
  \[ \text{gl.lookAt( eye, at, up )} \]
  - The view up vector is usually \((0, 1, 0)\)

- Modelview matrix
  - Combination of modeling matrix \(M\) and Camera transforms \(V\)

- `lookAt()` returns \(V\) part of modelview matrix

- What does `lookAt()` do with parameters \((\text{eye, at, up})\) you provide?
Viewing Transformation (cont.)

- OpenGL Code

```c
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.)

- `lookAt()` 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 \( \mathbf{u}, \mathbf{v}, \mathbf{n} \)?

- 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
  - \( \mathbf{n} = \frac{\text{eye} - \text{look}}{|\text{eye} - \text{look}|} \)
Eye Coordinate Frame (cont.)

- How about $\mathbf{u}$ and $\mathbf{v}$?
  - $\mathbf{u} = \frac{\mathbf{Up} \times \mathbf{n}}{|\mathbf{Up} \times \mathbf{n}|}$
  - How come this works?
Eye Coordinate Frame (cont.)

- How about $\mathbf{v}$?
  - $\mathbf{v} = \mathbf{n} \times \mathbf{u}$
  - Why is this already normalized?
Putting It All Together

- **Eye space**
  - Origin = \((\text{eye}_x, \text{eye}_y, \text{eye}_z)\)
  - \(\mathbf{n} = (\text{eye} - \text{look}) / |\text{eye} - \text{look}|\)
  - \(\mathbf{u} = (\mathbf{Up} \times \mathbf{n}) / |\mathbf{Up} \times \mathbf{n}|\)
  - \(\mathbf{v} = \mathbf{n} \times \mathbf{u}\)
World to Eye Transformation

- Next, use \( u, v, n \) to compose \( V \) part of modelview 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 \((-e_x, -e_y, -e_z)\)
World to Eye Transformation (cont.)

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

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

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

Note: \( e \cdot u = e_x u_x + e_y u_y + e_z u_z \)
Flexible Camera Control

- May create a **Camera** class

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

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

- Example

```cpp
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.)

- `lookAt( )` does not let you control roll, pitch & 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
  - Get the new modelview matrix and pass it down to the shaders
Loading Modelview Matrix directly

Pseudo-code:

```c
getModelViewMatrix( ) {  
    // load modelview matrix with existing camera values
    mat4 m[16];
    vec3 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[2] = n.x; m[6] = n.y; m[10] = n.z; m[14] = -eVec.dot(n);
    return( m );
}

- 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

```c
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;
    return( getModelViewMatrix( ) );
}
```
Camera Roll

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 );
    return( getModelViewMatrix( ) );
}

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