### Topics To Be Covered

- Viewing Transformations
- OpenGL programming
  - see further presentations on transformations and viewing
- Texture mapping
- Shading and illumination

---

### Object-Order Viewing - Overview

A view is specified by:
- eye position (Eye)
- view direction vector (u)
- screen center position (Cop)
- screen orientation (u, v)
- screen width W, height H

u, v, n are orthonormal vectors

After the viewing transform:
- the screen center is at the coordinate system origin
- the screen is aligned with the x, y-axis
- the viewing vector points down the negative z-axis
- the eye is on the positive z-axis

All objects are transformed by the viewing transform

---

### Step 1: Viewing Transform

- The sequence of transformations is:
  - translate the screen Center Of Projection (Cop) to the coordinate system origin ($T_{view}$)
  - rotate the translated screen such that the view direction vector $n$ points down the negative z-axis and the screen vectors $u$, $v$ are aligned with the x, y-axis ($R_{view}$)

- We get $M_{view} = R_{view} \cdot T_{view}$

- We transform all object (points, vertices) by $M_{view}$:

$$
\begin{bmatrix}
  x' \\
  y' \\
  z'
\end{bmatrix} =
\begin{bmatrix}
  u_x & u_y & u_z & 0 \\
  v_x & v_y & v_z & 0 \\
  n_x & n_y & n_z & 0
\end{bmatrix}
\begin{bmatrix}
  1 & 0 & 0 & -Cop_x \\
  0 & 1 & 0 & -Cop_y \\
  0 & 0 & 1 & -Cop_z
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
$$

- Now the objects are easy to project since the screen is in a convenient position
  - but first we have to account for perspective distortion...
Step 2: Perspective Projection

- A (view-transformed) vertex with coordinates \((x', y', z')\) projects onto the screen as follows:
  \[
  y_p = y' \cdot \frac{\text{eye}}{\text{eye} - z} \quad x_p = x' \cdot \frac{\text{eye}}{\text{eye} - z}
  \]
  - \(x_p\) and \(y_p\) can be used to determine the screen coordinates of the object point (i.e., where to plot the point on the screen)

Step 1 + Step 2 = World-To-Screen Transform

- The sequence of transformations is:
  - translate the screen Center Of Projection (COP) to the coordinate system origin \((T_{view})\)
  - rotate the translated screen such that the view direction vector \(n\) points down the negative \(z\)-axis and the screen vectors \(u, v\) are aligned with the \(x, y\)-axis \((R_{view})\)

  \[
  W_{\text{view}} = R_{\text{view}} \cdot T_{\text{view}}
  \]

- We transform all object (points, vertices) by \(W_{\text{view}}\):

  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z'
  \end{bmatrix}
  = 
  \begin{bmatrix}
  u_x & u_y & u_z & 0 \\
  v_x & v_y & v_z & 0 \\
  0 & 0 & 1 & 0
  \end{bmatrix}
  \begin{bmatrix}
  1 & 0 & 0 & -\text{Cop}_x \n \\
  0 & 1 & 0 & -\text{Cop}_y \n \\
  0 & 0 & 1 & -\text{Cop}_z
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z
  \end{bmatrix}
  
  \begin{bmatrix}
  1 \\
  0 \\
  0
  \end{bmatrix}
  \]

- Now the objects are easy to project since the screen is in a convenient position
  - but first we have to account for perspective distortion...

Step 3: Window Transform (1)

- Note: our camera screen is still described in world coordinates
- However, our display monitor is described on a pixel raster of size \((N_x, N_y)\)
- The transformation of (perspective) viewing coordinates into pixel coordinates is called window transform

  - Assume:
    - we want to display the rendered screen image in a window of size \((N_x, N_y)\) pixels
    - the width and height of the camera screen in world coordinates is \((W, H)\)
    - the center of the camera is at the center of the screen coordinate system

  - Then:
    - the valid range of object coordinates is \((-W/2, ..., +W/2, -H/2, ..., +H/2)\)
    - these have to be mapped into \((0, ..., N_x-1, 0, ..., N_y-1)\):

      \[
      x_s = \left( x_p + \frac{W}{2} \right) \cdot \frac{N_x - 1}{W} \\
      y_s = \left( y_p + \frac{H}{2} \right) \cdot \frac{N_y - 1}{H}
      \]

      \[
      \begin{array}{ccc}
      \text{camera} & \text{window} \\
      W & Ny & N_x
      \end{array}
      \]
Step 3: Window Transform (2)

- The window transform can be written as the matrix $M_{\text{window}}$:

$$
\begin{bmatrix}
\frac{x_3}{W} \\
\frac{y_3}{H} \\
1
\end{bmatrix} =
\begin{bmatrix}
\frac{Nx - 1}{2} & 0 & \frac{Nx - 1}{2} & 0 \\
0 & \frac{Ny - 1}{2} & 0 & \frac{Ny - 1}{2} \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x_3 \\
y_3 \\
1
\end{bmatrix}
$$

- After the perspective divide, all object points (vertices) are multiplied by $M_{\text{window}}$.
- Note: we could figure the window transform into $M_{\text{trans}}$:
  - in that case, there is only one matrix multiply per object point (vertex) with a subsequent perspective divide.
  - the OpenGL graphics pipeline does this.

Rendering With OpenGL (1)

- Define the viewing window:
  - glOrtho() for parallel projection
  - glFrustum() for perspective projection
- glMatrixMode(GL_MODELVIEW)
- Specify the viewpoint
  - glLookat() /* need to have GLUT */
- Model the scene
  - glTranslate(), glRotate(), glScale(), ...

Texture Mapping - Realistic Detail for Boring Polygons

Specify the light sources: glLight()  Enable the z-buffer: glEnable(GL_DEPTH_TEST)
Enable lighting: glEnable(GL_LIGHTING)
Enable light source $i$: glEnable(GL_LIGHT$i$) /* GL_LIGHT$i$ is the symbolic name of light $i$ */
Select shading model: glShadeModel() /* GL_FLAT or GL_SMOOTH */

For each object:
  
  We only need this for now
  /* duplicate the matrix on the stack if want to apply some extra transformations to the object */
  glBegin(GL_POLYGON);
  glColor3fv(c1); glVertex3fv(v1); glnormal3fv(n1); /* vertex 1 */
  glColor3fv(c2); glVertex3fv(v2); glnormal3fv(n2); /* vertex 2 */
  glColor3fv(c3); glVertex3fv(v3); glnormal3fv(n3); /* vertex 3 */
  glEnd();
  glPopMatrix() /* get rid of the object-specific transformations, pop back the saved matrix */

Leonard McMillan
Texture Mapping - Large Walls

Texture Mapping - Small Facets

Texture Mapping Large Walls - OpenGL Program

```c
#include <GL/gl.h>

void textureMappingLargeWalls()
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, textureName);
    glBegin(GL_QUAD);
        glTexCoord2f(0.0, 0.0);
        glVertex3f(v1);
        // vertex 1
        glTexCoord2f(1.0, 0.0);
        glVertex3f(v2);
        // vertex 2
        glTexCoord2f(1.0, 1.0);
        glVertex3f(v3);
        // vertex 3
        glTexCoord2f(0.0, 1.0);
        glVertex3f(v4);
        // vertex 4
    glEnd();
}
```

Texture Mapping Small Facets - OpenGL Program

```c
#include <GL/gl.h>

void textureMappingSmallFacets()
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, textureName);
    glBegin(GL_QUAD);
        glVertex3f(v[i][0]);
        glVertex3f(v[i][1]);
        glVertex3f(v[i][2]);
    glEnd();
}
```