Introduction to 3D viewing

- 3D is just like taking a photograph!

Viewing Transformation

- Position and orient your camera

Projection Transformation

- Control the “lens” of the camera
- Project the object from 3D world to 2D screen

Viewing Transformation (2)

- Important camera parameters to specify
  - Camera (eye) position (Ex,Ey,Ez) in world coordinate system
  - Center of interest (coi) (cx, cy, cz)
  - Orientation (which way is up?) View-up vector (Up_x, Up_y, Up_z)
Viewing Transformation (3)
- Transformation?
  - Form a camera (eye) coordinate frame
  - Transform objects from world to eye space

Viewing Transformation (4)
- Eye space?
  - Transform to eye space can simplify many downstream operations (such as projection) in the pipeline

Viewing Transformation (5)
- In OpenGL:
  - gluLookAt (Ex, Ey, Ez, cx, cy, cz, 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

Viewing Transformation (6)
void display()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0,0,1,0,0,0,0,0,1,0);
  display_all();  // your display routine
}
**Perspective Projection**

- Characterized by object foreshortening
  - Objects appear to be larger if they are closer to the camera
  - This is what happens in the real world
- Need:
  - Projection center
  - Projection plane
  - Projection: Connecting the object to the projection center

**Orthographic Projection**

- No foreshortening effect – distance from camera does not matter
- The projection center is at infinite
- Projection calculation – just drop z coordinates
Field of View

- Determine how much of the world is taken into the picture
- The larger is the field view, the smaller is the object projection size

Near and Far Clipping Planes

- Only objects between near and far planes are drawn
- Near plane + far plane + field of view = Viewing Frustum

Viewing Frustum

- 3D counterpart of 2D world clip window
- Objects outside the frustum are clipped

Projection Transformation

- In OpenGL:
  - Set the matrix mode to GL_PROJECTION
  - Perspective projection: use gluPerspective(fovy, aspect, near, far) or glFrustum(left, right, bottom, top, near, far)
  - Orthographic: glOrtho(left, right, bottom, top, near, far)
gluPerspective(fovy, aspect, near, far)

- Aspect ratio is used to calculate the window width

\[
\text{Aspect} = \frac{w}{h}
\]

Or You can use this function in place of gluPerspective()

glFrustum(left, right, bottom, top, near, far)

glOrtho(left, right, bottom, top, near, far)

Projection Transformation

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJETION);
    glLoadIdentity();
    gluPerspective(fovy, aspect, near, far);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    display_all(); // your display routine
}
3D viewing under the hood

Topics of Interest:
- Viewing transformation
- Projection transformation

Viewing Transformation
- Transform the object from world to eye space
  - Construct an eye space coordinate frame
  - Construct a matrix to perform the coordinate transformation
  - Flexible Camera Control
Eye Coordinate Frame

- Known: eye position, center of interest, view-up vector
- To find out: new origin and three basis vectors

Assumption: the direction of view is orthogonal to the view plane (the plane that objects will be projected onto)

center of interest (COI)

Eye Coordinate Frame (2)

- Origin: eye position (that was easy)
- Three basis vectors: one is the normal vector ($\mathbf{n}$) of the viewing plane, the other two are the ones ($\mathbf{u}$ and $\mathbf{v}$) that span the viewing plane

$\mathbf{n}$ is pointing away from the world because we use right hand coordinate system

$\mathbf{n} = \mathbf{eye} - \mathbf{COI}$

$\mathbf{n} = \mathbf{N} / |\mathbf{N}|$

(u,v,n should be orthogonal to each other)

Eye Coordinate Frame (3)

- How about $\mathbf{u}$ and $\mathbf{v}$?

We can get $\mathbf{u}$ first - $\mathbf{u}$ is a vector that is perpendicular to the plane spanned by $\mathbf{N}$ and view up vector ($\mathbf{V}_{\text{up}}$)

$\mathbf{U} = \mathbf{V}_{\text{up}} \times \mathbf{n}$

$\mathbf{u} = \mathbf{U} / |\mathbf{U}|$

Eye Coordinate Frame (4)

- How about $\mathbf{v}$?

Knowing $\mathbf{n}$ and $\mathbf{u}$, getting $\mathbf{v}$ is easy

$\mathbf{v} = \mathbf{n} \times \mathbf{u}$

$\mathbf{v}$ is already normalized
**Eye Coordinate Frame (5)**

- Put it all together

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

Basis vectors:

\[
\begin{align*}
\mathbf{n} &= \frac{\text{eye} - \text{COI}}{|\text{eye} - \text{COI}|} \\
\mathbf{u} &= \frac{\mathbf{V}_{\text{up}} \times \mathbf{n}}{|\mathbf{V}_{\text{up}} \times \mathbf{n}|} \\
\mathbf{v} &= \mathbf{n} \times \mathbf{u}
\end{align*}
\]

**World to Eye Transformation**

- Transformation matrix \((M_{w2e})\)

\[
P' = M_{w2e} \times P
\]

1. Come up with the transformation sequence to move eye coordinate frame to the world
2. And then apply this sequence to the point \(P\) in a reverse order

**World to Eye Transformation (2)**

- Rotate the eye frame so that it will be "aligned" with the world frame
- Translate \((-ex, -ey, -ez)\)

Rotation:

\[
\begin{pmatrix}
ux & uy & uz & 0 \\
vx & vy & vz & 0 \\
xn & yn & zn & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

How to verify the rotation matrix?

Translation: \[1\ 0\ 0\ -ex\]

\[0\ 1\ 0\ -ey\]

\[0\ 0\ 1\ -ez\]

\[0\ 0\ 0\ 1\]
World to Eye Transformation (3)

- Head tilt: Rotate your head by $\delta$
- Just rotate the object about the eye space z axis - $\delta$
- $M_{w2e} = \begin{bmatrix} \cos(-\delta) & -\sin(-\delta) & 0 & 0 & ux & uy & ux & 0 & 1 & 0 & -ex \\ \sin(-\delta) & \cos(-\delta) & 0 & 0 & vx & vy & vz & 0 & 0 & 1 & -ey \\ 0 & 0 & 1 & 0 & nx & ny & nz & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Why -$\delta$?
When you rotate your head by $\delta$, it is like rotate the object by -$\delta$.

Projection Transformation

- Projection - map the object from 3D space to 2D screen
- OpenGL maps (projects) everything in the visible volume into a canonical view volume
  - Parallel Projection (2)
  - After transforming the object to the eye space, parallel projection is relative easy - we could just drop the Z
    - $X_p = x$
    - $Y_p = y$
    - $Z_p = -d$
  - We actually want to keep Z - why?
Parallel Projection (3)

- Transformation sequence:
  1. Translation (M1): \((-\text{near} = \text{zmax}, \ -\text{far} = \text{zmin})\)
      \(-\text{ymax+ymin}/2, \ -\text{ymax+ymin}/2, \ -\text{zmax+zmin}/2\)
  2. Scaling (M2):
      \(2/(\text{xmax-xmin}), \ 2/(\text{ymax-ymin}), \ 2/(\text{zmax-zmin})\)

\[
M_2 \times M_1 = \begin{bmatrix}
2/(\text{xmax-xmin}) & 0 & 0 & - (\text{xmax+xmin})/(\text{xmax-xmin}) \\
0 & 2/(\text{ymax-ymin}) & 0 & - (\text{ymax+ymin})/(\text{ymax-ymin}) \\
0 & 0 & 2/(\text{zmax-zmin}) & -(\text{zmax+zmin})/(\text{zmax-zmin}) \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Perspective Projection

- Side view:

\[
\begin{array}{c}
(0,0,0) \\
d \\
\text{Eye (projection center)}
\end{array}
\]

Based on similar triangle:

\[
\begin{align*}
y' &= y \times \frac{-z}{d} \\
y &= \frac{-z}{\text{distance} (z)}
\end{align*}
\]

Perspective Projection (2)

- Same for x. So we have:
  \[x' = x \times \frac{d}{-z}\]
  \[y' = y \times \frac{d}{-z}\]
  \[z' = -d\]

- Put in a matrix form:

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

- OpenGL assume \(d = 1\), i.e. the image plane is at \(z = -1\)

Perspective Projection (3)

- We are not done yet. We want to somewhat keep the z information so that we can perform depth comparison

- Use pseudo depth - OpenGL maps the near plane to 1, and far plane to -1

- Need to modify the projection matrix: solve a and b

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & a & b \\
0 & 0 & (1/d) & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

How to solve a and b?

\[
Z = 1 \quad z = -1
\]
> **Perspective Projection (4)**

- Solve a and b

\[
\begin{pmatrix}
    x' \\
    y' \\
    z' \\
    w'
\end{pmatrix} =
\begin{pmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & a & b \\
    0 & 0 & (1/-d) & 0
\end{pmatrix}
\begin{pmatrix}
    x \\
    y \\
    z \\
    w
\end{pmatrix}
\]

\((0,0,1) = M \times (0,0,-\text{near})^T\)
\((0,0,-1) = M \times (0,0,-\text{far})^T\)

- \(a = -(\text{far} + \text{near})/ (\text{far} - \text{near})\)
- \(b = (\text{-2} \times \text{far} \times \text{near}) / (\text{far} - \text{near})\)

- Verify this!

- Not done yet. OpenGL also normalizes the x and y ranges of the viewing frustum to \([-1, 1]\) (translate and scale)

- And takes care the case that eye is not at the center of the view volume (shear)

---

**Perspective Projection (6)**

- **Final Projection Matrix:**

\[
\begin{pmatrix}
    x' \\
    y' \\
    z' \\
    w'
\end{pmatrix} =
\begin{pmatrix}
    2N/((\text{xmax} - \text{xmin}) \times \text{ymax} - \text{ymin}) & 0 & (\text{xmax} + \text{xmin})/((\text{xmax} - \text{xmin}) \times \text{ymin} - \text{ymin}) & 0 \\
    0 & 2N/((\text{ymin} + \text{ymin}) \times \text{ymax} - \text{ymin}) & (\text{ymax} + \text{ymin})/((\text{ymax} - \text{ymin}) \times \text{ymin} - \text{ymin}) & 0 \\
    0 & 0 & -(\text{F} + \text{N})/(\text{F} - \text{N}) & -2\text{F} \times \text{N}/(\text{F} - \text{N}) \\
    0 & 0 & -1 & 0
\end{pmatrix}
\begin{pmatrix}
    x \\
    y \\
    z \\
    w
\end{pmatrix}
\]

\text{glFrustum}(\text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}, \text{N}, \text{F}) \quad \text{N} = \text{near plane}, \text{F} = \text{far plane}

---

**Perspective Projection (7)**

- After perspective projection, the viewing frustum is also projected into a canonical view volume (like in parallel projection)

\((-1, -1, 1) \rightarrow (1, 1, -1)\)

\((1, 1, -1) \rightarrow (1, 1, 1)\)

**Canonical View Volume**
Flexible Camera Control

- Instead of provide COI, it is possible to just give camera orientation
- Just like control a airplane's orientation

\[ \begin{align*}
\phi &= 0 \\
\theta &= 0 \\
x &= R \cos(\phi) \cos(\theta) \\
y &= R \sin(\phi) \\
z &= R \cos(\phi) \cos(90 - \theta)
\end{align*} \]

Flexible Camera Control

- How to compute the viewing vector \((x,y,z)\) from pitch\((\phi)\) and yaw\((\theta)\)?

Flexible Camera Control

- gluLookAt() does not let you to control pitch and yaw
- you need to compute/maintain the vector by yourself
- And then calculate COI = Eye + \((x,y,z)\) before you can call gluLookAt().