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
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

  ![Eye coordinate frame diagram](image)

- 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
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
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    display_all(); // your display routine
}
Demo

Click on the arguments and move the mouse to modify values.
Projection Transformation

- Important things to control
  - Perspective or Orthographic
  - Field of view and image aspect ratio
  - Near and far clipping planes
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
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

```
gluPerspective(fovy, aspect, near, far)
```

Aspect ratio is calculated as 

\[
\text{Aspect} = \frac{w}{h}
\]
glFrustum(left, right, bottom, top, near, far)

- Or You can use this function in place of gluPerspective()
glOrtho(left, right, bottom, top, near, far)

- For orthographic projection
void display()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(fove, aspect, near, far);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0,0,1,0,0,0,0,1,0);
  display_all(); // your display routine
}
Click on the arguments and move the mouse to modify values.
3D viewing under the hood
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)
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

$n$ is pointing away from the world because we use right hand coordinate system

\[ \mathbf{N} = \text{eye} - \text{COI} \]
\[ \mathbf{n} = \mathbf{N} / |\mathbf{N}| \]

Remember $\mathbf{u}, \mathbf{v}, \mathbf{n}$ should be all unit vectors.

$\mathbf{n}$ should be orthogonal to each other.
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)

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

\[
u = U / |U|
\]
Eye Coordinate Frame (4)

- How about $v$?

Knowing $n$ and $u$, getting $v$ is easy.

\[
v = n \times u
\]

$v$ is already normalized.
Eye Coordinate Frame (5)

- Put it all together

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

Basis vectors:

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

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

Rotation:
\[
\begin{pmatrix}
ux & uy & uz & 0 \\
vx & vy & vz & 0 \\
x & n & n & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

How to verify the rotation matrix?

Translation:
\[
\begin{pmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
**World to Eye Transformation (2)**

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

\[
M_{w2e} = \begin{bmatrix}
ux & uy & ux & 0 \\
vx & vy & vz & 0 \\
x & ny & nx & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
World to Eye Transformation (3)

- **Head tilt**: Rotate your head by \(\delta\)
- **Just rotate the object about the eye space z axis** - \(\delta\)

\[
M_{w2e} = \begin{bmatrix}
\cos(-\delta) & -\sin(-\delta) & 0 & 0 \\
\sin(-\delta) & \cos(-\delta) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
ux & uy & ux & 0 \\
vx & vy & vz & 0 \\
x & y & z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez
\end{bmatrix}
\]

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

Perspective: `gluPerspective()`

Parallel: `glOrtho()`
Parallel Projection

After transforming the object to the eye space, parallel projection is relatively easy – we could just drop the Z

\[
\begin{align*}
X_p &= x \\
Y_p &= y \\
Z_p &= -d
\end{align*}
\]

- We actually want to keep Z - why?
Parallel Projection (2)

- OpenGL maps (projects) everything in the visible volume into a **canonical view volume**

```markdown
glOrtho(xmin, xmax, ymin, ymax, near, far)
```

Canonical View Volume

- Canonical View Volume
  - (xmin, ymin, -near)
  - (xmax, ymax, -far)
  - (-1, -1, 1)
  - (1, 1, -1)
Parallel Projection (3)

Transformation sequence:

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

2. Scaling (M2):

   \[2/(\text{xmax}-\text{xmin}), 2/(\text{ymax}-\text{ymin}), 2/(\text{zmax}-\text{zmin})\]

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

- Side view:

Based on similar triangle:

\[
\frac{y}{y'} = \frac{-z}{d}
\]

\[
Y' = y \times \frac{d}{-z}
\]
Perspective Projection (2)

- Same for x. So we have:

\[ x' = \frac{x \times d}{-z} \]
\[ y' = \frac{y \times d}{-z} \]
\[ z' = -d \]

- Put in a matrix form:

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

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

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

How to solve a and b?
### Perspective Projection (4)

- **Solve a and b**

\[
\begin{align*}
\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & (1/-d) & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\
(0,0,1)^T &= M \times (0,0,-\text{near})^T \\
(0,0,-1)^T &= M \times (0,0,-\text{far})^T
\end{align*}
\]

- Verify this!

- **a = -(far+\text{near})/(far-\text{near})**

- **b = \((-2 \times \text{far} \times \text{near}) / (\text{far-\text{near}})\)**
Perspective Projection (5)

- 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:

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

\text{glFrustum}(x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, N, F) \quad N = \text{near plane, } 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)

Canonical View Volume:
- Vertex A: (-1, -1, 1)
- Vertex B: (1, 1, -1)
Flexible Camera Control

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

\[ \phi, \theta, \delta \]
Flexible Camera Control

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

\[
\begin{align*}
\phi &= 0 \\
\theta &= 0
\end{align*}
\]

\[
x = R \cos(\phi) \cos(\theta)
\]

\[
y = R \sin(\phi)
\]

\[
z = R \cos(\phi) \cos(90 - \theta)
\]
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 \( \text{COI} = \text{Eye} + (x, y, z) \) before you can call `gluLookAt()`.