Shear Transformations

- x-shear scales x proportional to y
- Leaves y and z values fixed

Specification via Shear Angle

- \( \cot(\theta) = \frac{x' - x}{y} \)
- \( x' = x + y \cot(\theta) \)
- \( y' = y \)
- \( z' = z \)

\[
(x', y', z') = H_x(\theta)(x, y, z)
\]

\[
H_x(\theta) = \begin{bmatrix}
1 & \cot(\theta) & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Specification via Ratios

- For example, shear in both x and z direction
- Leave y fixed
- Slope \( \alpha \) for x-shear, \( \gamma \) for z-shear

\[
H_{x,z}(\alpha, \gamma) = \begin{bmatrix}
1 & \alpha & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & \gamma & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Yields
Composing Transformations

- Let \( p = A q \), and \( q = B s \).
- Then \( p = (A B) s \).

\[
\begin{array}{c}
\text{B} \\
\text{s} \\
\text{q} \\
\text{p}
\end{array}
\]

**AB matrix multiplication**

Composing Transformations

- Fact: Every affine transformation is a composition of rotations, scalings, and translations
- So, how do we compose these to form an x-shear?
  - Exercise!

Outline

- Shear Transformation
- Camera Positioning
- Simple Parallel Projections
- Simple Perspective Projections

Transform Camera = Transform Scene

- Camera position is identified with a frame
- Either move and rotate the objects
- Or move and rotate the camera
- Initially, camera at origin, pointing in negative z-direction

The Look-At Function

- Convenient way to position camera
- \text{gluLookAt}(ex, ey, ez, fx, fy, fz, ux, uy, uz);
  - \( e = \text{eye point} \)
  - \( f = \text{focus point} \)
  - \( u = \text{up vector} \)

OpenGL code

```c
void display()
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (ex, ey, ez, fx, fy, fz, ux, uy, uz);
    glTranslatef(x, y, z);
    renderBunny();
    glutSwapBuffers();
}
```
Implementing the Look-At Function

Plan:

1. Transform world frame to camera frame
   - Compose a rotation $R$ with translation $T$
   - $W = T \ R$

2. Invert $W$ to obtain viewing transformation $V$
   - $V = W^{-1} = (T \ R)^{-1} = R^{-1} \ T^{-1}$
   - Derive $R$, then $T$, then $R^{-1} \ T^{-1}$

World Frame to Camera Frame I

• Camera points in negative z direction
• $n = (f - e) / \|f - e\|$ is unit normal to view plane
• Therefore, $R$ maps $[0 \ 0 \ -1]^T$ to $[n_x \ n_y \ n_z]^T$

World Frame to Camera Frame II

• $R$ maps $[0,1,0]^T$ to projection of $u$ onto view plane
• This projection $v$ equals:
  - $\alpha = (u \cdot n) / \|n\| = u \cdot n$
  - $v_0 = u - \alpha \ n$
  - $v = v_0 / \|v_0\|$

World Frame to Camera Frame III

• Set $w$ to be orthogonal to $n$ and $v$
• $w = n \times v$
• $(w, v, -n)$ is right-handed

World Frame to Camera Frame IV

• Translation of origin to $e = [e_x \ e_y \ e_z]^T$
• Rotation must map:
  - $(1,0,0)$ to $w$
  - $(0,1,0)$ to $v$
  - $(0,0,-1)$ to $n$

Summary of Rotation

• `gluLookAt(e_x, e_y, e_z, f_x, f_y, f_z, u_x, u_y, u_z);`
• $n = (f - e) / \|f - e\|$
• $v = (u - (u \cdot n) n) / \|u - (u \cdot n) n\|$
• $w = n \times v$
• Rotation must map:
  - $(1,0,0)$ to $w$
  - $(0,1,0)$ to $v$
  - $(0,0,-1)$ to $n$

$$T = \begin{bmatrix}
 1 & 0 & 0 & e_x \\
 0 & 1 & 0 & e_y \\
 0 & 0 & 1 & e_z \\
 0 & 0 & 0 & 1
\end{bmatrix}$$
Camera Frame to Rendering Frame

• \( V = W^{-1} = (T R)^{-1} = R^{-1} T^{-1} \)
• \( R \) is rotation, so \( R^{-1} = R^T \)

\[
R^{-1} = \begin{bmatrix}
w_x & w_y & w_z & 0 \\
w_y & v_y & v_z & 0 \\
-n_x & -n_y & -n_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

• \( T \) is translation, so \( T^{-1} \) negates displacement

\[
T^{-1} = \begin{bmatrix}
1 & 0 & 0 & -e_x \\
0 & 1 & 0 & -e_y \\
0 & 0 & 1 & -e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Putting it Together

• Calculate \( V = R^{-1} T^{-1} \)

\[
V = \begin{bmatrix}
w_x & w_y & w_z & -w_x e_x - w_y e_y - w_z e_z \\
w_y & v_y & v_z & -v_x e_x - v_y e_y - v_z e_z \\
-n_x & -n_y & -n_z & n_x e_x + n_y e_y + n_z e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

• This is different from book [Angel, Ch. 5.3.2]
• There, \( u, v, n \) are right-handed (here: \( u, v, -n \))

Other Viewing Functions

• Roll (about \( z \)), pitch (about \( x \)), yaw (about \( y \))

• Assignment 2 poses a related problem

Outline

• Shear Transformation
• Camera Positioning
• Simple Parallel Projections
• Simple Perspective Projections

Projection Matrices

• Recall geometric pipeline

Vertices → Transforms → Clipper → Projector → Texture → Pixels

• Projection takes 3D to 2D
• Projections are not invertible
• Projections also described by 4x4 matrix
• Homogenous coordinates crucial
• Parallel and perspective projections

Parallel Projection

• Project 3D object to 2D via parallel lines
• The lines are not necessarily orthogonal to projection plane

Parallel Projection

- Problem: objects far away do not appear smaller
- Can lead to "impossible objects":

![Penrose stairs](source: Wikipedia)

Orthographic Projection

- A special kind of parallel projection: projectors perpendicular to projection plane
- Simple, but not realistic
- Used in blueprints (multiview projections)

Orthographic Projection Matrix

- Project onto $z = 0$
- $x_p = x$, $y_p = y$, $z_p = 0$
- In homogenous coordinates

\[
\begin{bmatrix}
  x_p \\
  y_p \\
  z_p \\
  1
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

Perspective

- Perspective characterized by foreshortening
- More distant objects appear smaller
- Parallel lines appear to converge
- Rudimentary perspective in cave drawings:

![Lascaux, France](source: Wikipedia)

Discovery of Perspective

- Foundation in geometry (Euclid)

![Mural from Pompei, Italy](source: Wikipedia)

Middle Ages

- Art in the service of religion
- Perspective abandoned or forgotten

![Ottonian manuscript, ca. 1000](source: Wikipedia)
Renaissance

- Rediscovery, systematic study of perspective
  Filippo Brunelleschi
  Florence, 1415

Projection (Viewing) in OpenGL

- Remember: camera is pointing in the negative z direction

Orthographic Viewing in OpenGL

- `glOrtho(xmin, xmax, ymin, ymax, near, far)`
  
  \[ z_{\min} = \text{near}, \ z_{\max} = \text{far} \]

Perspective Viewing in OpenGL

- Two interfaces: `glFrustum` and `gluPerspective`
- `glFrustum(xmin, xmax, ymin, ymax, near, far);`
  
  \[ z_{\min} = \text{near}, \ z_{\max} = \text{far} \]

Field of View Interface

- `gluPerspective(fov, aspectRatio, near, far);`
- near and far as before
- aspectRatio = w / h
- Fovy specifies field of view as height (y) angle

OpenGL code

```c
void reshape(int x, int y)
{
    glViewport(0, 0, x, y);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0 * x / y, 0.01, 10.0);
}
```
Perspective Viewing Mathematically

- $d =$ focal length
- $y/z = y_p/d$ so $y_p = y/(z/d) = y/d \times z$
- Note that $y_p$ is non-linear in the depth $z$!

Exploiting the 4\textsuperscript{th} Dimension

- Perspective projection is not affine:
  \[ M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ \frac{z}{d} \\ 1 \end{bmatrix} \]
  has no solution for $M$

- Idea: exploit homogeneous coordinates
  \[ p = w \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \]
  for arbitrary $w \neq 0$

Perspective Projection Matrix

- Use multiple of point
  \[
  \begin{pmatrix} z/d \\ \frac{y}{d} \\ \frac{z}{d} \\ 1 \end{pmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}
  \]

- Solve
  \[
  M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}
  \]

Projection Algorithm

Input: 3D point $(x,y,z)$ to project

1. Form $[x \; y \; z \; 1]^T$
2. Multiply $M$ with $[x \; y \; z \; 1]^T$; obtaining $[X \; Y \; Z \; W]^T$
3. Perform perspective division:
   \[ X \; Y \; Z \; W \]

Output: $(X \; W, \; Y \; W, \; Z \; W)$
(last coordinate will be $d$)

Perspective Division

- Normalize $[x \; y \; z \; w]^T$ to $[(x/w) \; (y/w) \; (z/w) \; 1]^T$

- Perform perspective division after projection

  \[ \text{Model-view} \rightarrow \text{Projection} \rightarrow \text{Perspective division} \]

- Projection in OpenGL is more complex
  (includes clipping)