

### Viewing and Projection

- Shear Transformation
- Camera Positioning
- Simple Parallel Projections
- Simple Perspective Projections [Angel, Ch. 4]

Jernej Barbic  
University of Southern California

### Reminder: Affine Transformations

- Given a point  $[x \ y \ z]$ , form homogeneous coordinates  $[x \ y \ z \ 1]$ .

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The transformed point is  $[x' \ y' \ z']$ .

### Transformation Matrices in OpenGL

- Transformation matrices in OpenGL are vectors of 16 values (column-major matrices)

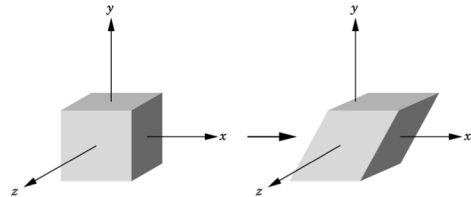
$m = \{m_1, m_2, \dots, m_{16}\}$  represents

$$\begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

- Some books transpose all matrices!

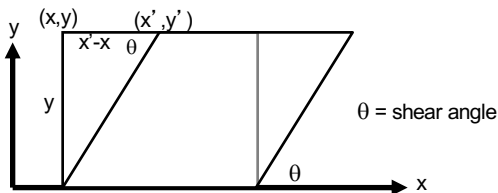
### Shear Transformations

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



### Specification via Shear Angle

- $\cot(\theta) = (x' - x) / y$
  - $x' = x + y \cot(\theta)$
  - $y' = y$
  - $z' = z$
- $$H_x(\theta) = \begin{bmatrix} 1 & \cot(\theta) & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



### Specification via Ratios

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

• Solve

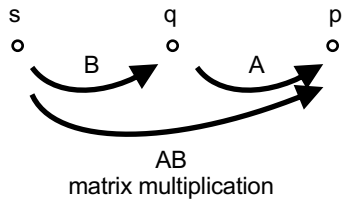
$$H_{x,z}(\alpha, \gamma) \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + \alpha y \\ y \\ z + \gamma y \\ 1 \end{bmatrix}$$

- Yields

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

## Composing Transformations

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



7

## 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!

8

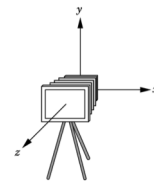
## Outline

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

9

## Transform Camera = Transform Scene

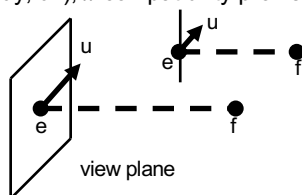
- Camera position is identified with a frame
- Either move and rotate the objects
- Or move and rotate the camera
- Initially, camera at origin, pointing in negative z-direction



10

## The Look-At Function

- Convenient way to position camera
- `OpenGLMatrix::LookAt(ex, ey, ez, fx, fy, fz, ux, uy, uz); // core profile`
- `gluLookAt(ex, ey, ez, fx, fy, fz, ux, uy, uz); // compatibility profile`
- $e$  = eye point
- $f$  = focus point
- $u$  = up vector



11

## OpenGL code (camera positioning)

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    OpenGLMatrix.SetMatrixMode(OpenGLMatrix::ModelView);
    OpenGLMatrix.LoadIdentity();
    OpenGLMatrix.LookAt(e_x, e_y, e_z, f_x, f_y, f_z, u_x, u_y, u_z);

    OpenGLMatrix.Translate(x, y, z); // transform the object
    ...
    float m[16]; // column-major
    OpenGLMatrix.GetMatrix(m); // fill "m" with the matrix entries
    glUniformMatrix4fv(h_modelViewMatrix, 1, GL_FALSE, m);
    renderBunny();
    glutSwapBuffers();
}
```

12

## Implementing the Look-At Function

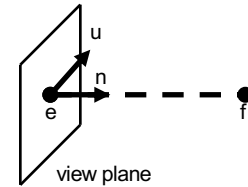
Plan:

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  $R^{-1} T^{-1}$

13

## World Frame to Camera Frame I

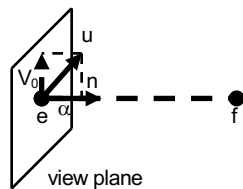
- Camera points in negative z direction
- $n = (f - e) / |f - e|$  is unit normal to view plane
- Therefore, R maps  $[0 \ 0 \ -1]^T$  to  $[n_x \ n_y \ n_z]^T$



14

## World Frame to Camera Frame II

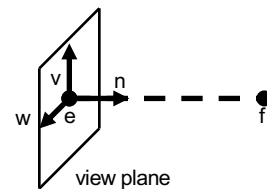
- R maps  $[0, 1, 0]^T$  to projection of u onto view plane
- This projection v equals:
  - $\alpha = (u \cdot n) / |n| = u \cdot n$
  - $v_0 = u - \alpha n$
  - $v = v_0 / |v_0|$



15

## 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



16

## Summary of Rotation

- `gluLookAt(ex, ey, ez, fx, fy, fz, ux, uy, uz);`
- $n = (f - e) / |f - e|$
- $v = (u - (u \cdot n) n) / |u - (u \cdot n) n|$
- $w = n \times v$
- Rotation must map:
 

- (1,0,0) to w	]
- (0,1,0) to v	
- (0,0,-1) to n	

$$\begin{bmatrix} w_x & v_x & -n_x & 0 \\ w_y & v_y & -n_y & 0 \\ w_z & v_z & -n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

17

## World Frame to Camera Frame IV

- Translation of origin to  $e = [e_x \ e_y \ e_z \ 1]^T$

$$T = \begin{bmatrix} 1 & 0 & 0 & e_x \\ 0 & 1 & 0 & e_y \\ 0 & 0 & 1 & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

18

## World Frame to Camera Frame

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

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

- T is translation, so  $T^{-1}$  negates displacement

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

19

## Putting it Together

- Calculate  $V = R^{-1} T^{-1}$

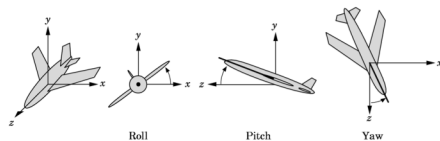
$$V = \begin{bmatrix} w_x & w_y & w_z & -w_x e_x - w_y e_y - w_z e_z \\ v_x & v_y & v_z & -v_x e_x - v_y e_y - v_z e_z \\ -n_x & -n_y & -n_z & n_x e_x + n_y e_y + n_z e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

20

## Other Viewing Functions

- Roll (about z), pitch (about x), yaw (about y)



- Assignment 2 poses a related problem

21

## Outline

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

22

## Projection Matrices

- Recall geometric pipeline

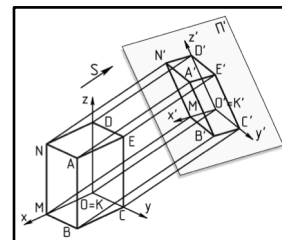


- Projection takes 3D to 2D
- Projections are not invertible
- Projections are described by a 4x4 matrix
- Homogenous coordinates crucial
- Parallel and perspective projections

23

## Parallel Projection

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

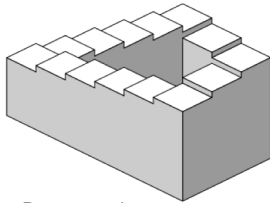


source: Wikipedia

24

## Parallel Projection

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



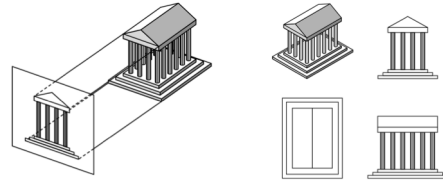
Penrose stairs

source: Wikipedia

25

## Orthographic Projection

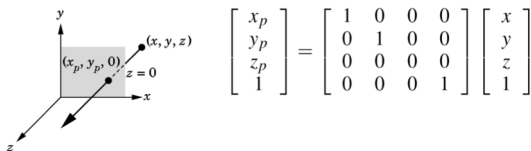
- A special kind of parallel projection: projectors perpendicular to projection plane
- Simple, but not realistic
- Used in blueprints (multiview projections)



26

## Orthographic Projection Matrix

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



27

## Perspective

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



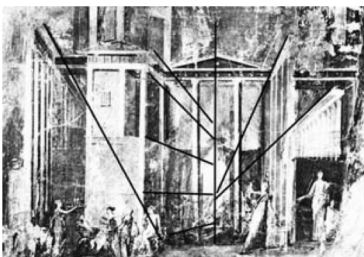
Lascaux, France

source: Wikipedia

28

## Discovery of Perspective

- Foundation in geometry (Euclid)



Mural from Pompeii, Italy

29

## Middle Ages

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



Ottonian manuscript, ca. 1000

30

## Renaissance

- Rediscovery, systematic study of perspective

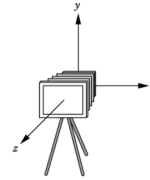


Filippo Brunelleschi  
Florence, 1415

31

## Projection (Viewing) in OpenGL

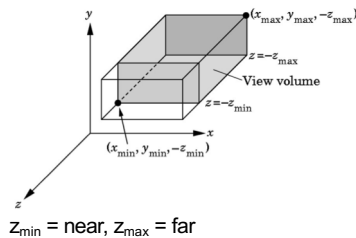
- Remember: camera is pointing in the negative z direction



32

## Orthographic Viewing in OpenGL (compatibility profile)

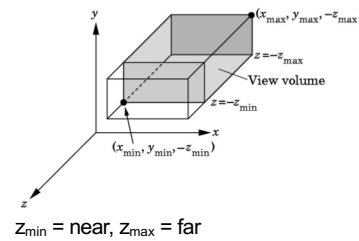
- `glOrtho(xmin, xmax, ymin, ymax, near, far)`



33

## Orthographic Viewing in OpenGL (core profile)

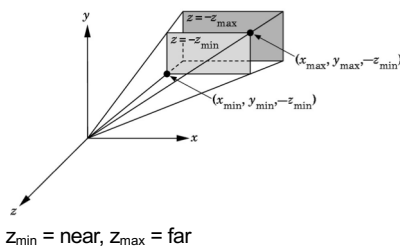
- `OpenGLMatrix::Ortho(xmin, xmax, ymin, ymax, near, far)`



34

## Perspective Viewing in OpenGL

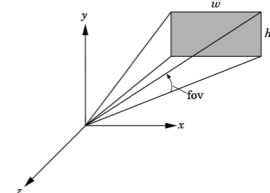
- Two interfaces: `glFrustum` and `gluPerspective`
- `{OpenGLMatrix::, gl}Frustum(xmin, xmax, ymin, ymax, near, far);`



35

## Field of View Interface

- `{OpenGLMatrix::, glu}Perspective(fovy, aspectRatio, near, far);`
- near and far as before
- `aspectRatio = w / h`
- Fovy specifies field of view as height (y) angle



36

## OpenGL code (reshape)

```
void reshape(int x, int y)
{
    glViewport(0, 0, x, y);

    openGLMatrix.SetMatrixMode(OpenGLMatrix::Projection);
    openGLMatrix.LoadIdentity();
    openGLMatrix.Perspective(60.0, 1.0 * x / y, 0.01, 10.0);
    openGLMatrix.SetMatrixMode(OpenGLMatrix::ModelView);
}

```

37

## OpenGL code

```
void displayFunc()
{
    ...

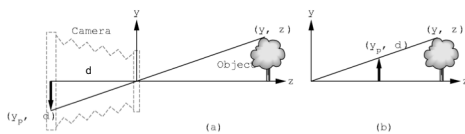
    openGLMatrix.SetMatrixMode(OpenGLMatrix::Projection);
    float p[16]; // column-major
    openGLMatrix.GetMatrix(p);
    const GLboolean isRowMajor = false;
    glUniformMatrix4fv(h_projectionMatrix, 1, isRowMajor, p);
    ...

    renderBunny();
    glutSwapBuffers();
}

```

38

## Perspective Viewing Mathematically



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

39

## Exploiting the 4<sup>th</sup> Dimension

- Perspective projection is not affine:

$$M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix} \text{ has no solution for } M$$

- Idea: exploit homogeneous coordinates

$$p = w \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \text{ for arbitrary } w \neq 0$$

40

## Perspective Projection Matrix

- Use multiple of point

$$(z/d) \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix}$$

- Solve

$$M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} \text{ with } M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

41

## 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:  
 $X / W, Y / W, Z / W$

**Output:**  $(X / W, Y / W, Z / W)$   
(last coordinate will be  $d$ )

42

## Perspective Division

- Normalize  $[x \ y \ z \ w]^T$  to  $[(x/w) \ (y/w) \ (z/w) \ 1]^T$
- Perform perspective division after projection



- Projection in OpenGL is more complex (includes clipping)