CSCI 420 Computer Graphics
Lecture 6

Viewing and Projection

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

Oded Stein
University of Southern California

Administrative update

- HW1 goes out today

- HW1 is due on Oct 7
  - Refer to course website for late policy, academic honesty policy, etc.

- Next week (Sep 23) we will discuss HW1 in detail in class, and you can start working on it in-class while I am here.
  - Will do this for all homework releases

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)

$m = \{m_1, m_2, ..., m_{16}\}$ 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!

Transpose

- I forgot to introduce transpose last time in our linear algebra review.

- Transpose is switching rows and columns of a matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}^T = \begin{pmatrix} a & g & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

- If you confuse row-major and col-major, you will be using the transpose of your matrix.

Translation

- $\mathbf{q} = \mathbf{p} + \mathbf{d}$ where $\mathbf{d} = [\alpha_x \;\; \alpha_y \;\; \alpha_z \;\; 0]^T$
- $\mathbf{p} = [x \;\; y \;\; z \;\; 1]^T$
- $\mathbf{q} = [x' \;\; y' \;\; z' \;\; 1]^T$
- Express in matrix form **q** = **T p** and solve for **T**

$$T = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

- $x' = \beta_x\ x$
- $y' = \beta_y\ y$
- $z' = \beta_z\ z$
- Express as **q** = **S p** and solve for **S**

$$S = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation in 3 Dimensions

- Orthogonal matrices:

$$RR^{\mathrm{T}} = R^{\mathrm{T}}R = I$$
$$\det(R) = 1$$

- As a 4x4 homogeneous matrix:

$$A = \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rigid maps

Rigid maps (+scale)

- A rigid map does not distort the shape of an object

$$||x - y|| = ||f(x) - f(y)|| \qquad \forall x, y$$

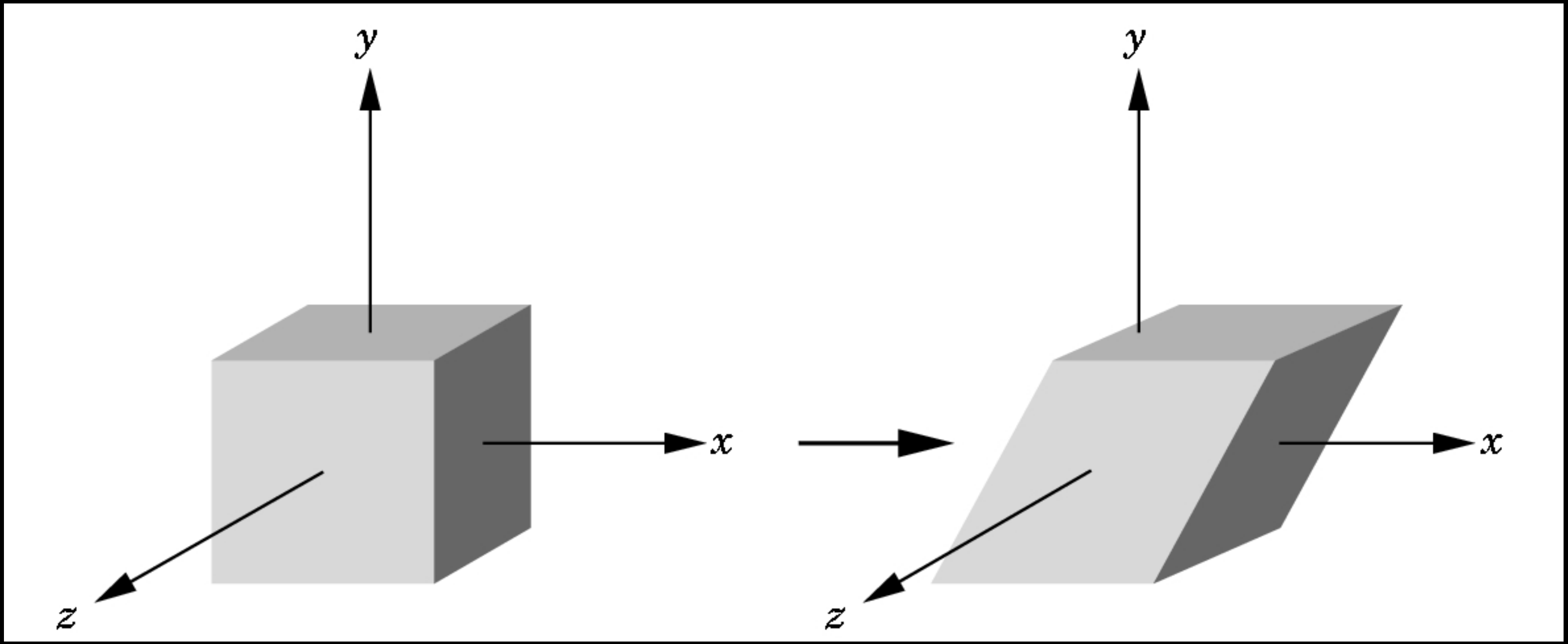
- \+ scale (we have been multiplying the whole scene by a constant to scale it into a volume)

- classical model-view matrices usually only rigidly transform and scale

- **non-rigid** maps can distort

Shear Transformations

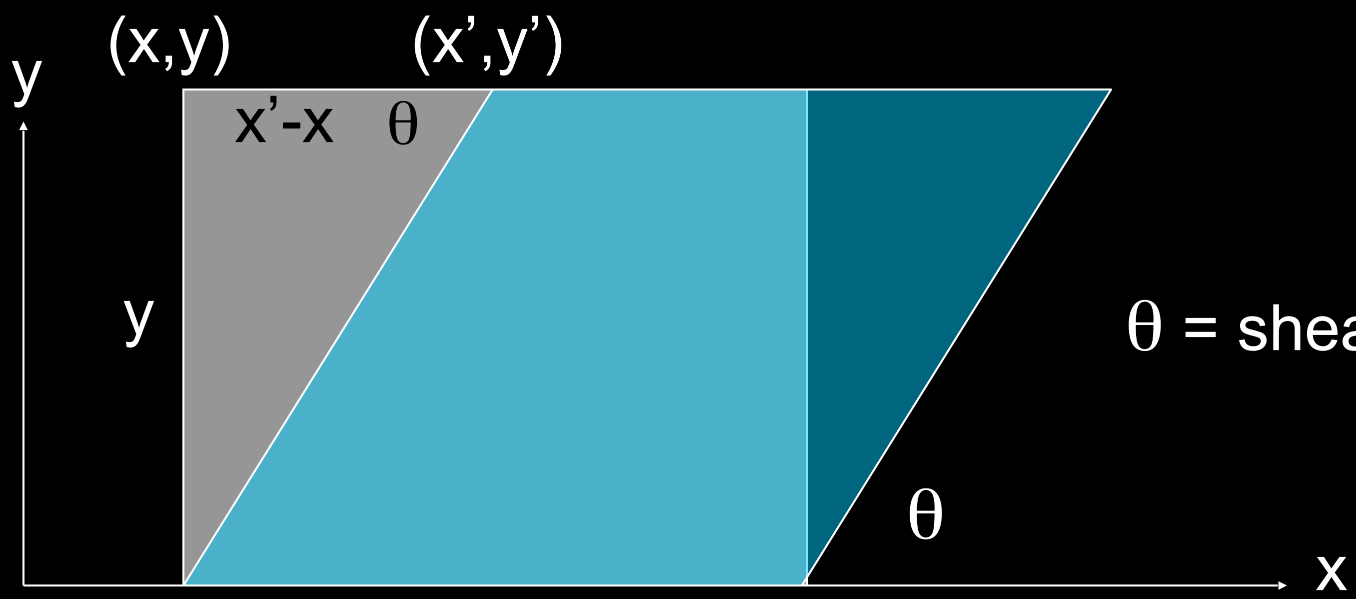
- x-shear scales x proportional to y
- Leaves y and z values fixed



Specification via Shear Angle

- $\cot(\theta) = (x' - x) / y$
- $x' = x + y \cot(\theta)$
- $y' = y$
- $z' = z$

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



θ = shear angle

Specification via Ratios

- For example, shear in both x and z direction
- Leave y fixed
- Slope α for x-shear, γ for z-shear
- Solve

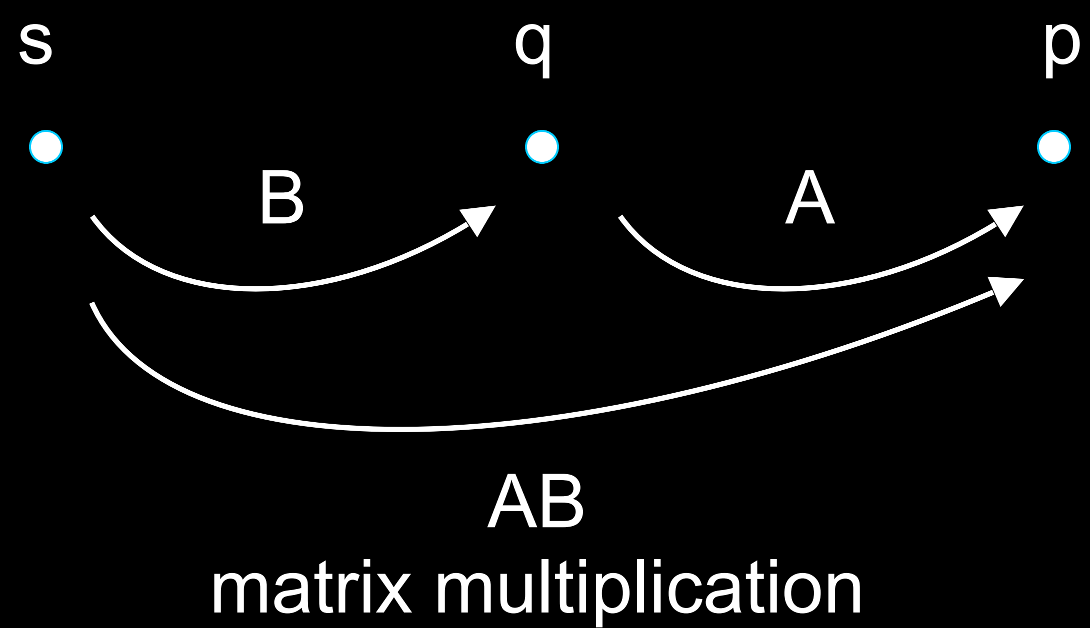
$$H_{x,z}(\alpha,\gamma)\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+\alpha y \\ y \\ z+\gamma y \\ 1 \end{bmatrix}$$

- 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 .



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?

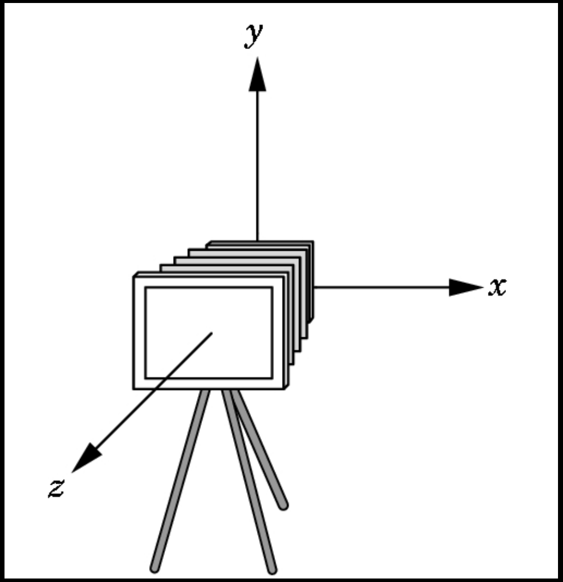
- Exercise!

Outline

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

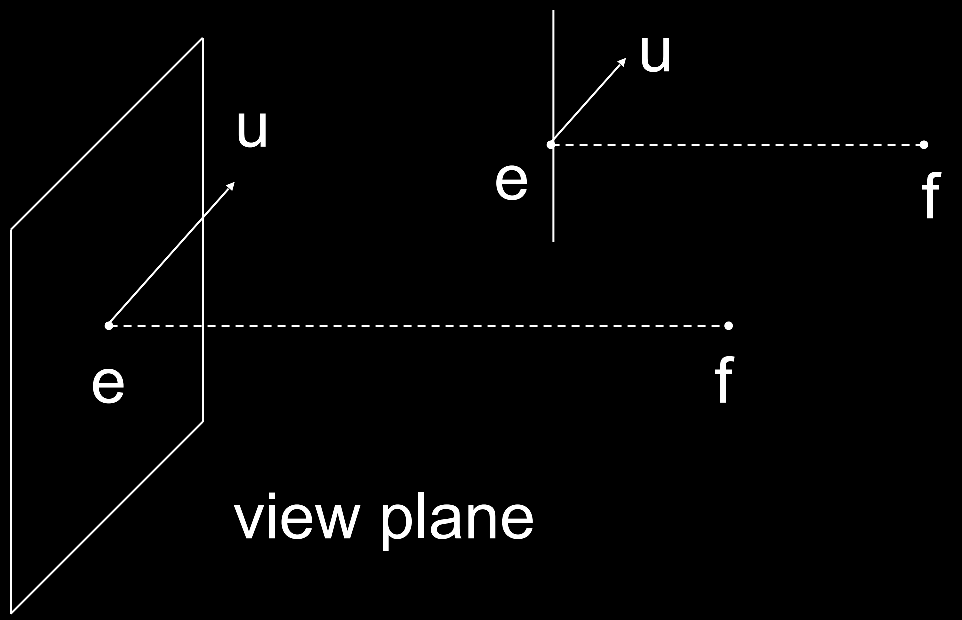
Transform Camera = Transform Scene

- 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
- There is an intuitive interface to specify camera position and rotation.



The Look-At Function

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



OpenGL code (camera positioning)

```
void display()
{
  glClear (GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
  openGLMatrix.SetMatrixMode(OpenGLMatrix::ModelView);
  openGLMatrix.LoadIdentity();
  openGLMatrix.LookAt(ex, ey, ez,  fx, fy, fz,  ux, uy, uz);
  openGLMatrix.Translate(x, y, z); //(if needed) apply transforms
  openGLMatrix.Rotate(angle, ax, ay, az);
  ...
  float m[16]; //column-major
  openGLMatrix.GetMatrix(m); //fill “m” with the matrix entries
  pipelineProgram->SetUniformVariableMatrix4fv(
      "modelViewMatrix", GL_FALSE, m);
  renderBunny();
  glutSwapBuffers();
}
```

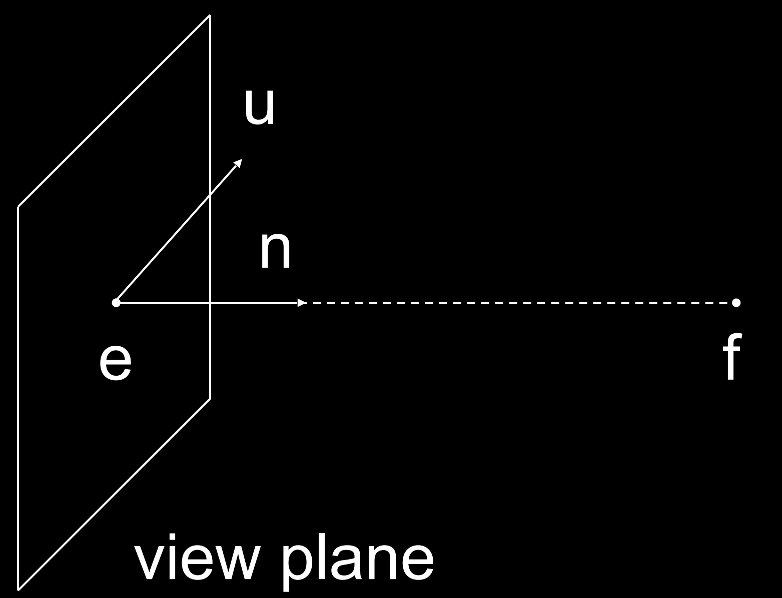
Implementing the Look-At Function

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

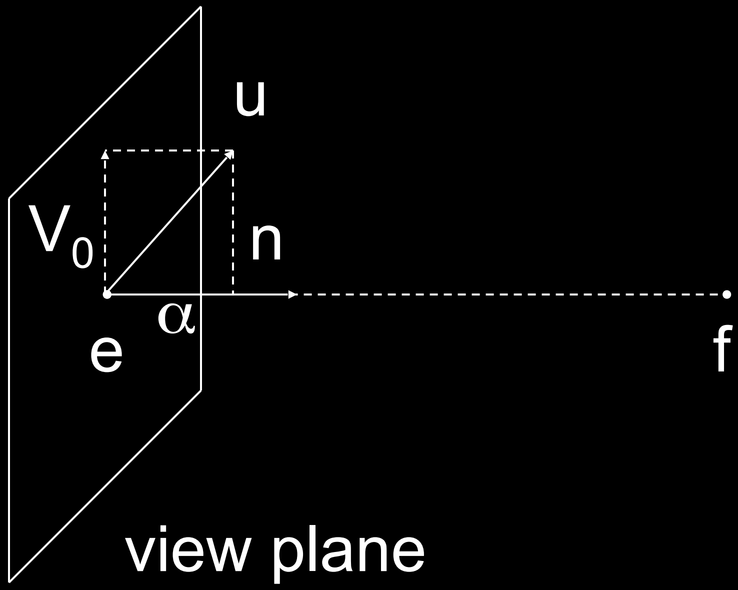
World Frame to Camera Frame I

- 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 $[n_x \ \ n_y \ \ n_z]^T$



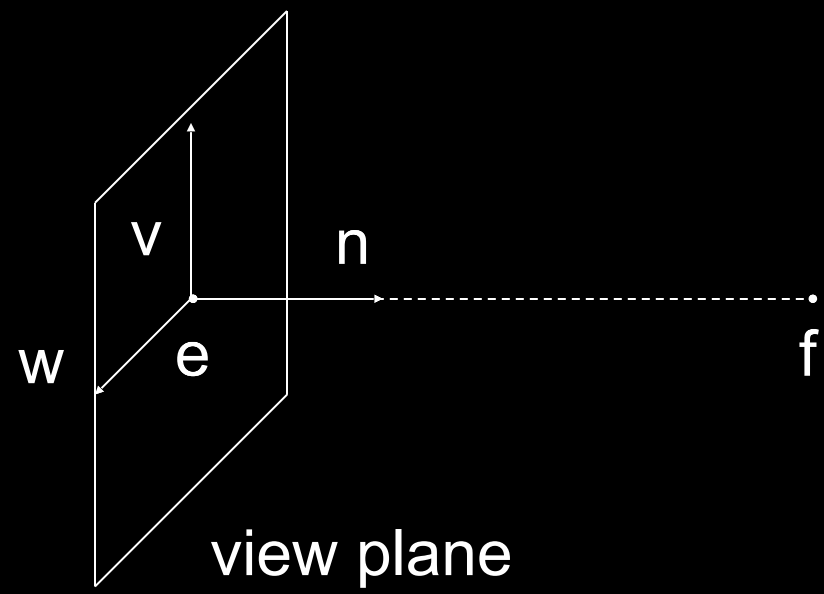
World Frame to Camera Frame II

- 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 x v
- (w, v, -n) is right-handed



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

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

World Frame to Camera Frame IV

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

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

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

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

World Frame to Camera Frame

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

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

- Wait, why is $R^{-1} = R^T$?

Putting it Together

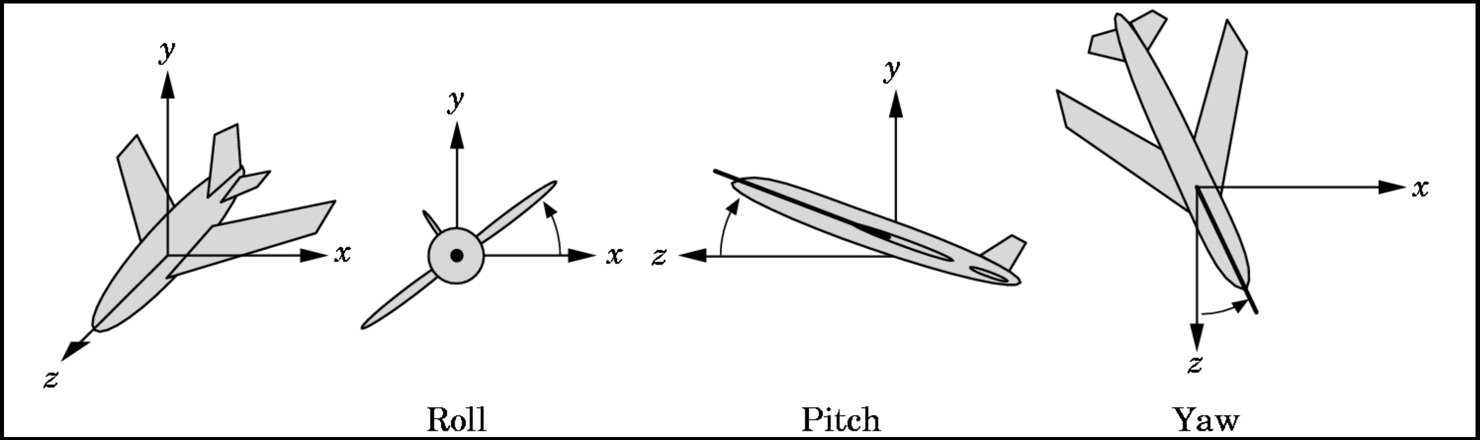
- Calculate $V = R^{-1} T^{-1}$

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

- (But really, use built-in matrix multiplication)
- This is different from book [Angel, Ch. 5.3.2]
- There, u, v, n are right-handed (here: u, v, -n)

Other Viewing Functions

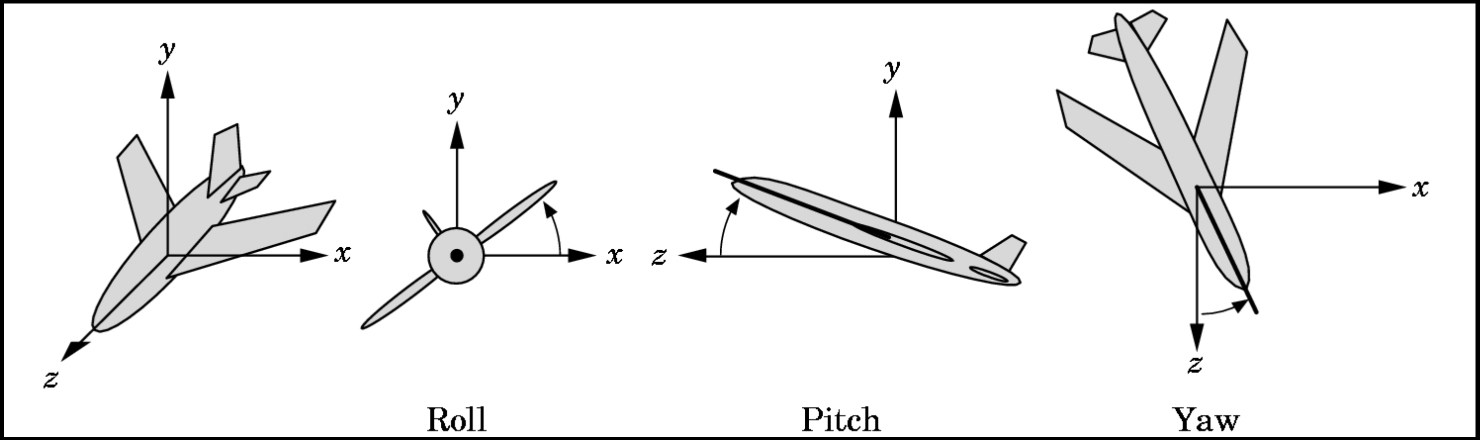
- Roll (about z), pitch (about x), yaw (about y)



- Assignment 2 poses a related problem

Other Viewing Functions

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



- Important to know if you want to implement certain types of user control

Other Viewing Functions

- Trackball
  - very flexible

https://commons.wikimedia.org/wiki/File:Trackball-Kensington-ExpertMouse5.jpg

- Turntable
  - more modern
  - harder to get lost

https://commons.wikimedia.org/wiki/File:Thorens_TD124_mkii_+_SME_3012_(9509758745)_(cropped).jpg

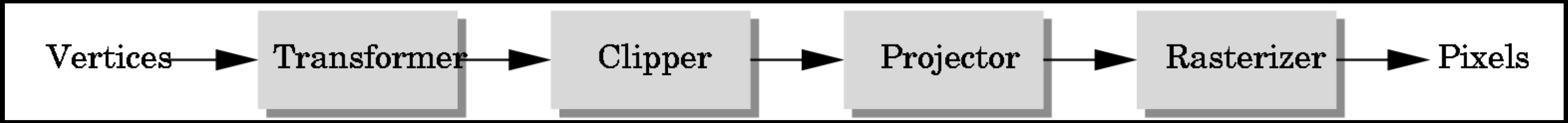
- (Blender demo)

Outline

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

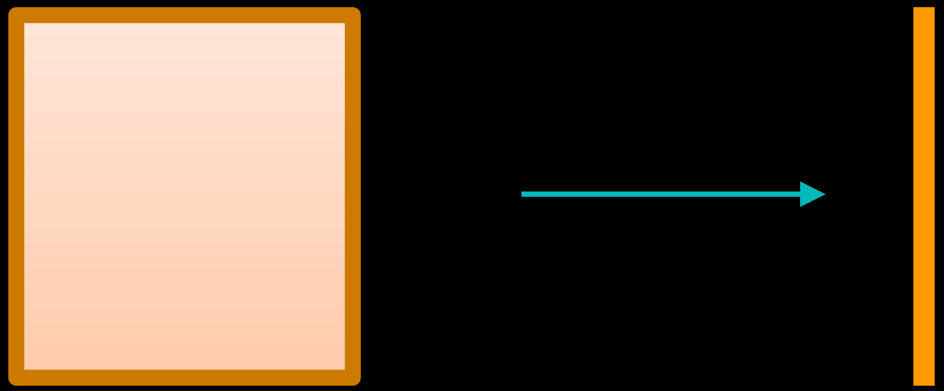
Projection Matrices

- Recall geometric pipeline

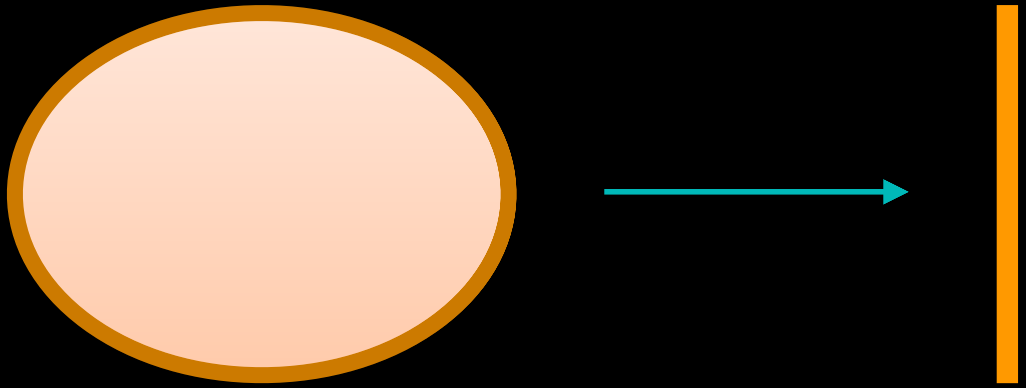


- Projection takes 3D to 2D
- Projections are not invertible
- Projections are described by a 4x4 matrix
- Homogenous coordinates crucial
- Parallel and perspective projections

Projection Matrices

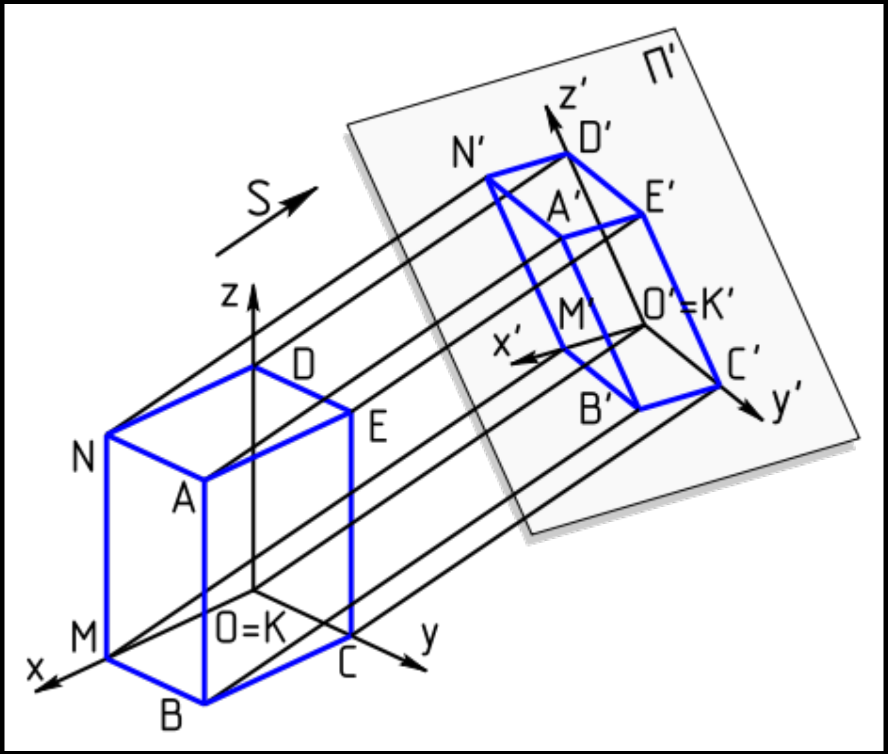


- Projections are not invertible



Parallel Projection

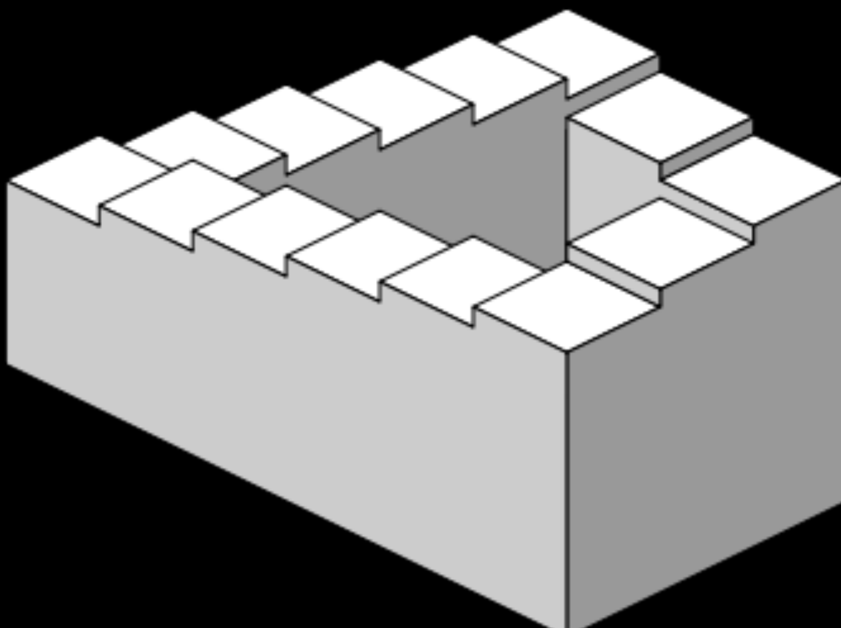
- Project 3D object to 2D via parallel lines
- The lines are not necessarily orthogonal to projection plane



source: Wikipedia

Parallel Projection

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

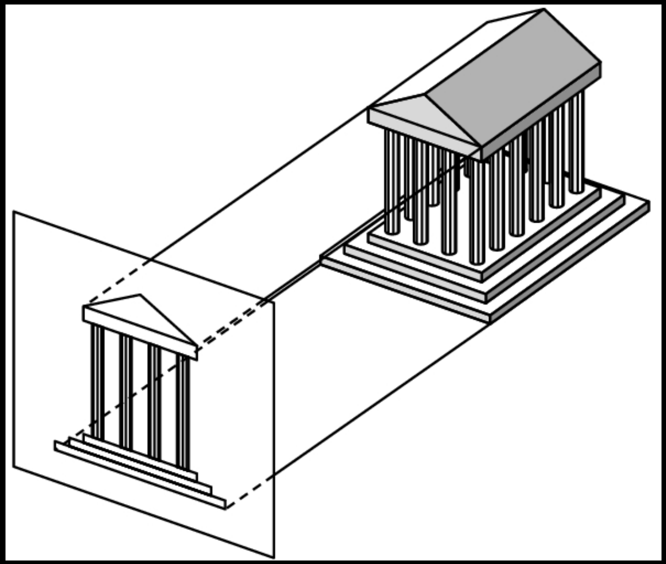


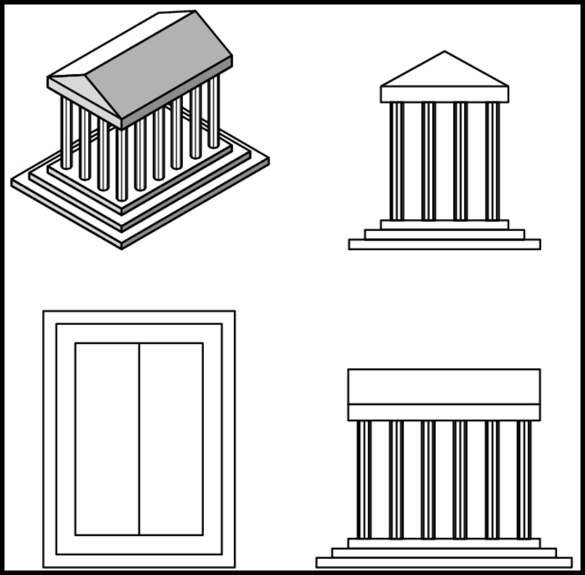
Penrose stairs

source: Wikipedia

Orthographic Projection

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



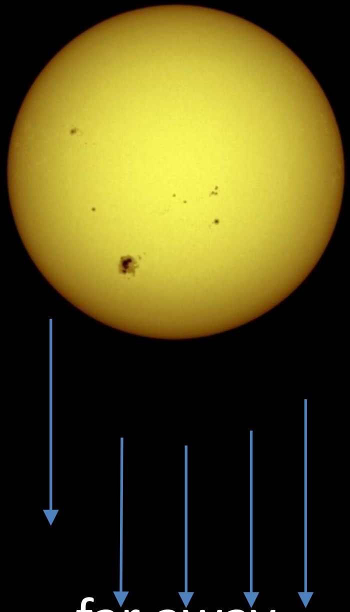


Orthographic Projection

just kill the z coordinate

$$(x, y, z) \mapsto (x, y)$$

flattening

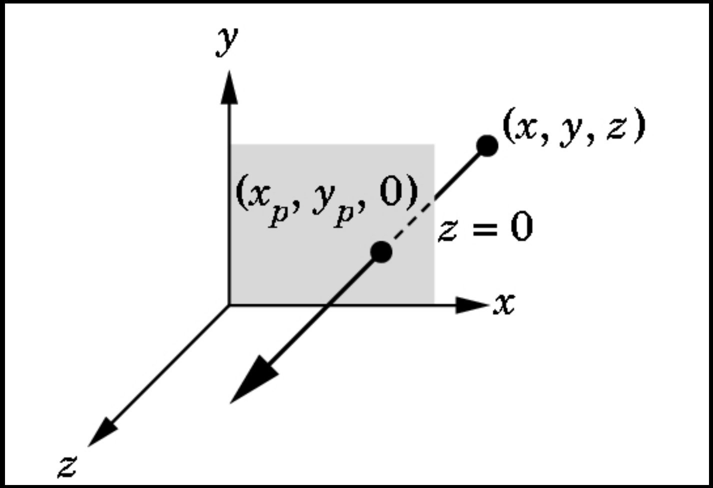


far away things

orthographic

Orthographic Projection Matrix

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



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

Orthographic Projection Matrix in Practice

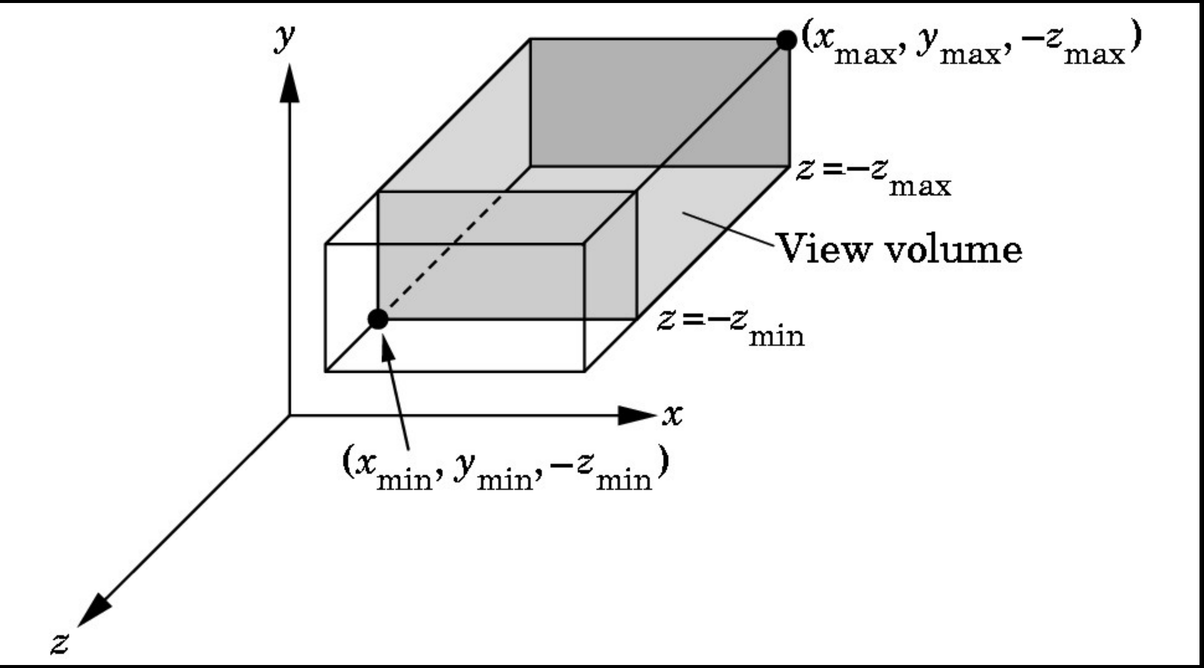
- Happens in the very last step of the OpenGL pipeline when actually drawing pixels.

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

- Not the projection matrix you use when you want to implement an orthographic viewer.
- Need to preserve some z-information for z-buffer, but put it in the correct range [z_{min}, z_{max}]
- Map to the boundaries of canonical viewing volume [-1,1] x [-1,1] x [-1,1]

Orthographic Viewing in OpenGL (compatibility profile)

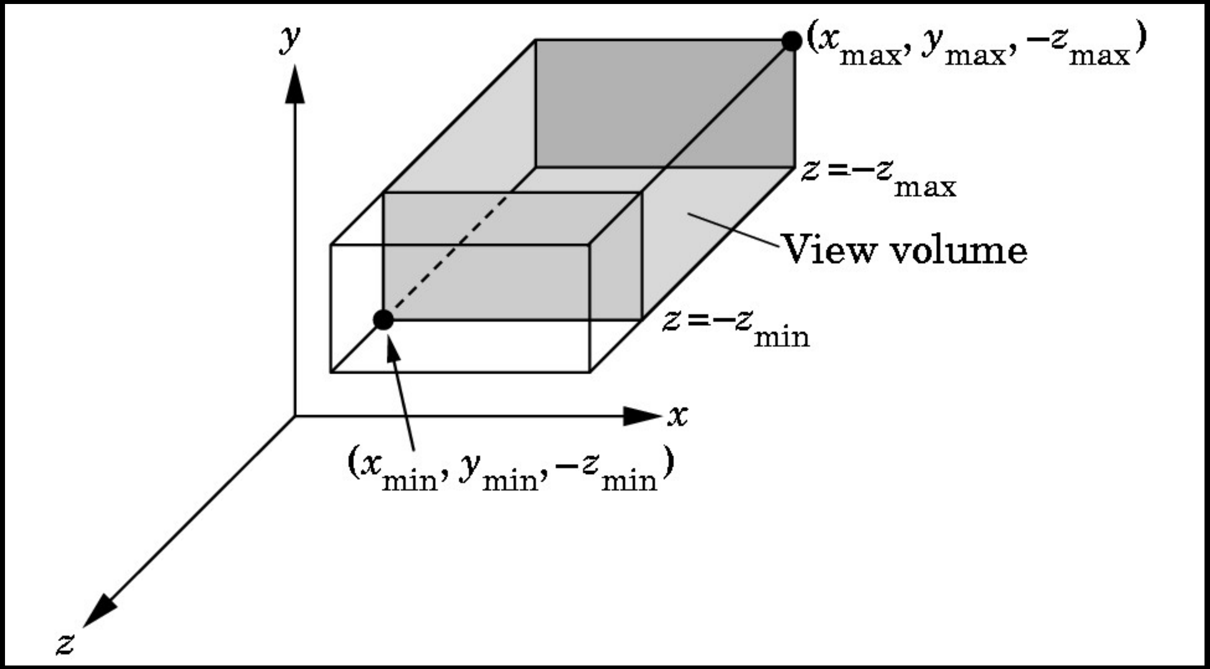
- `glOrtho(xmin, xmax, ymin, ymax, near, far)`



z_{min} = near, z_{max} = far

Orthographic Viewing in OpenGL (core profile)

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



z_{min} = near, z_{max} = far

Orthographic Viewing in OpenGL (core profile)

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

$$\begin{pmatrix} \frac{2}{x_{\max}-x_{\min}} & 0 & 0 & -\frac{x_{\max}+x_{\min}}{x_{\max}-x_{\min}} \\ 0 & \frac{2}{y_{\max}-y_{\min}} & 0 & -\frac{y_{\max}+y_{\min}}{y_{\max}-y_{\min}} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective

- Perspective characterized by foreshortening
- More distant objects appear smaller
- Parallel lines appear to converge

Intuitive understanding of perspective

- The two rails are always at the same distance in the real world.
- Projected onto the camera's sensor, they seem to get closer to each other the farther away they are from us.

https://commons.wikimedia.org/wiki/File:Ferrovia_PL_Saliscendi_Martina_Franca.jpg

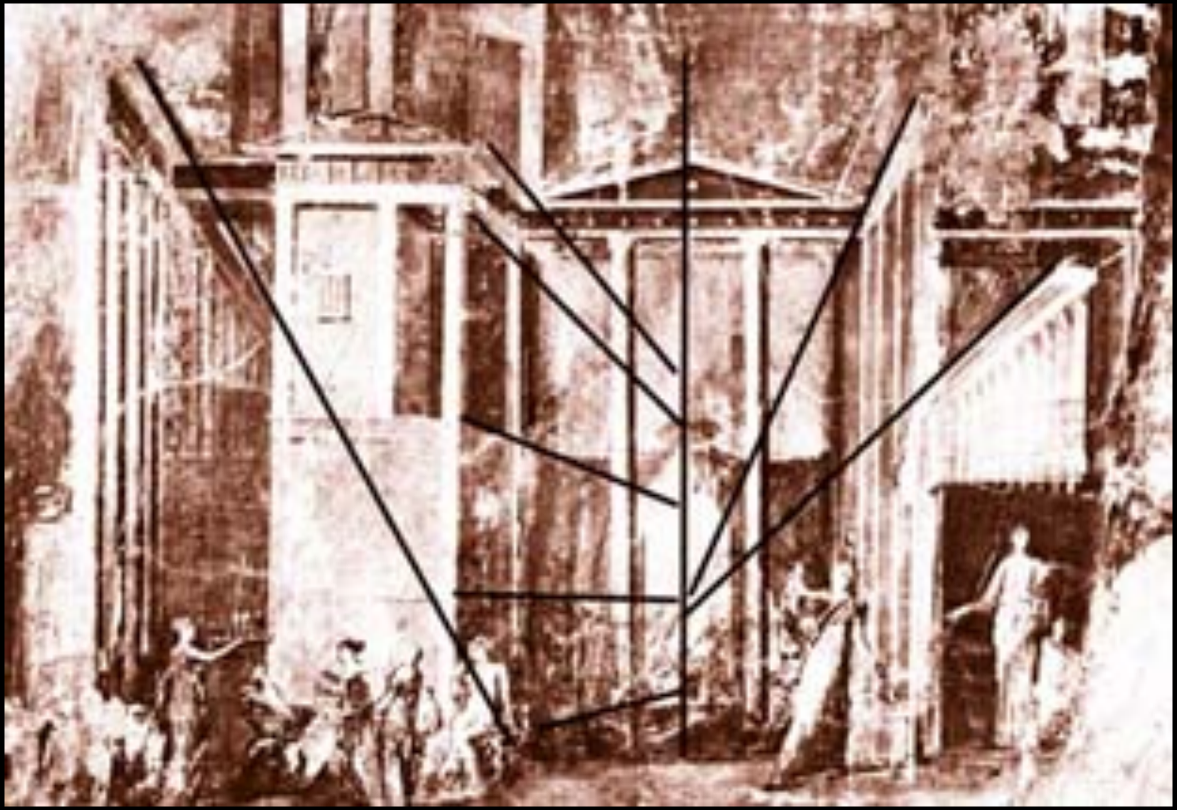
Perspective in Prehistoric Art

- Rudimentary perspective in cave drawings:

Lascaux, France
source: Wikipedia

Discovery of Perspective in Western Art

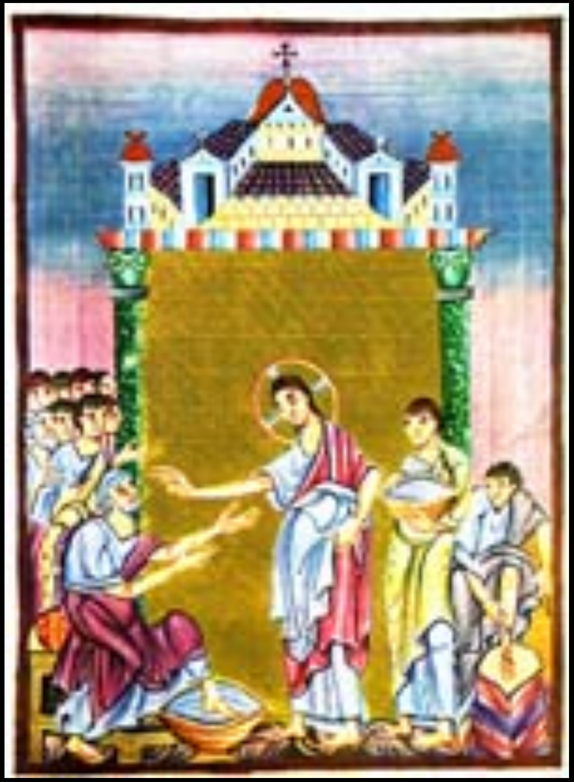
- Foundation in geometry (Euclid)



Mural from Pompeii, Italy

Middle Ages

- Different art styles become popular
- Perspective abandoned or forgotten



Ottonian manuscript, ca. 1000

Renaissance

- Rediscovery, systematic study of perspective

Filippo Brunelleschi
Florence, 1415

Japanese art

- Earlier ukiyoe: no perspective

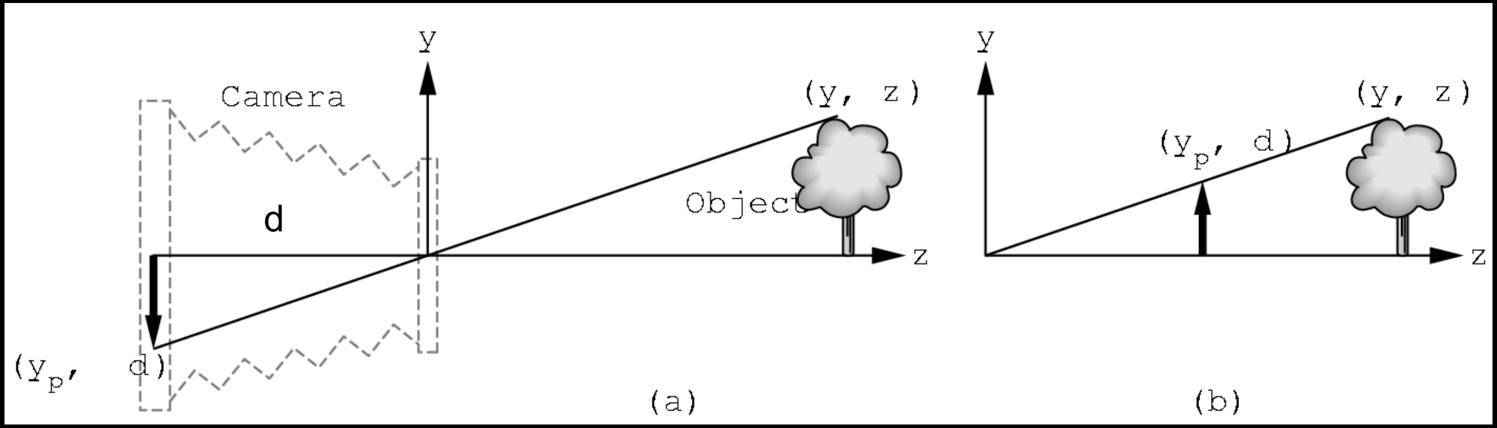
Kano Hideyori 16th century. "Maple Viewers"

Japanese art

- Later ukiyoe: western-style mathematical perspective

Okumura Masanobu 1745. "Taking the Evening Cool by Ryōgoku Bridge"

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!
  - (this will cause problems later)

Exploiting the 4th Dimension

- Perspective projection is not affine:

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

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

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

- Solve

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

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 / W, Y / W, Z / W

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

Perspective Division

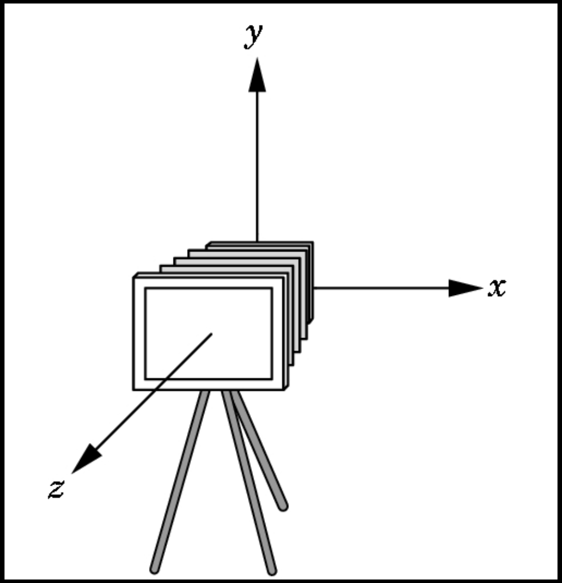
- Normalize $[x \;\; y \;\; z \;\; w]^T$ to $[(x/w) \;\; (y/w) \;\; (z/w) \;\; 1]^T$

- Perform perspective division after projection

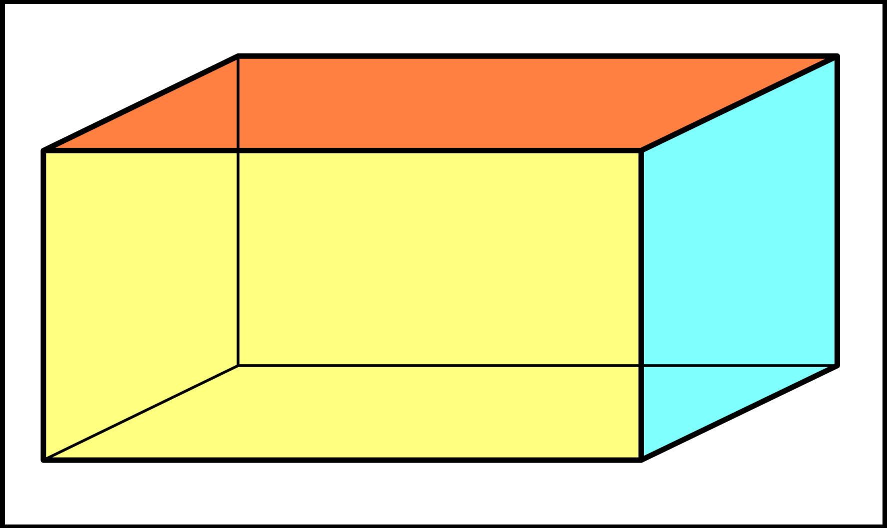
- Projection in OpenGL is more complex (includes clipping)

Projection (Viewing) in OpenGL

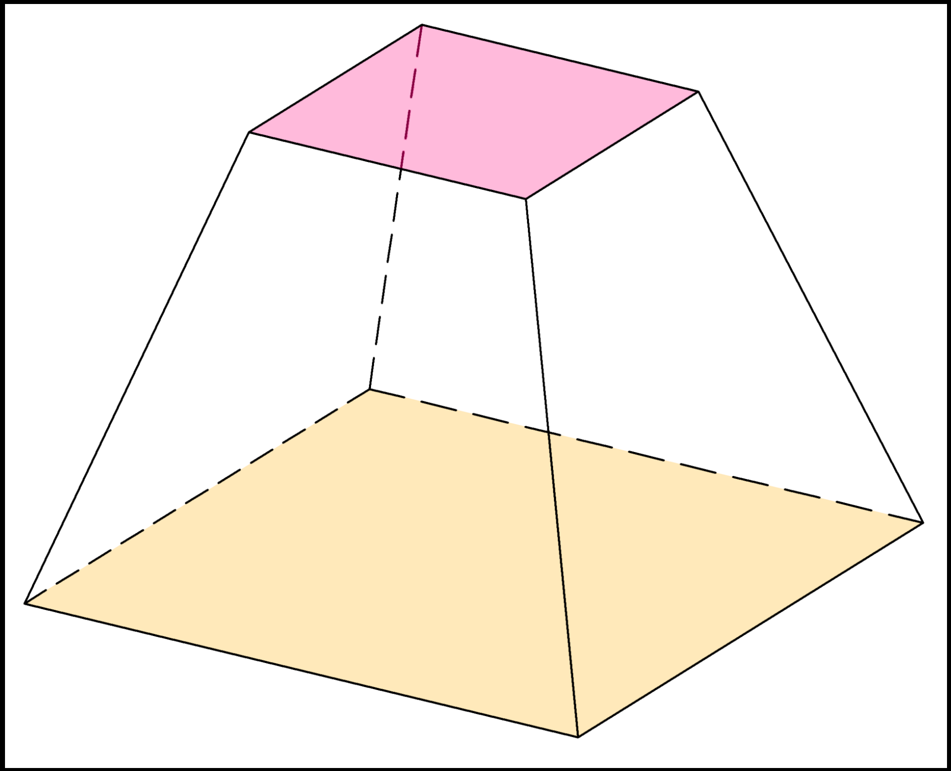
- Remember: camera is pointing in the negative z direction



From Cuboid to Frustum



https://commons.wikimedia.org/wiki/
File:Cuboid_no_label.svg

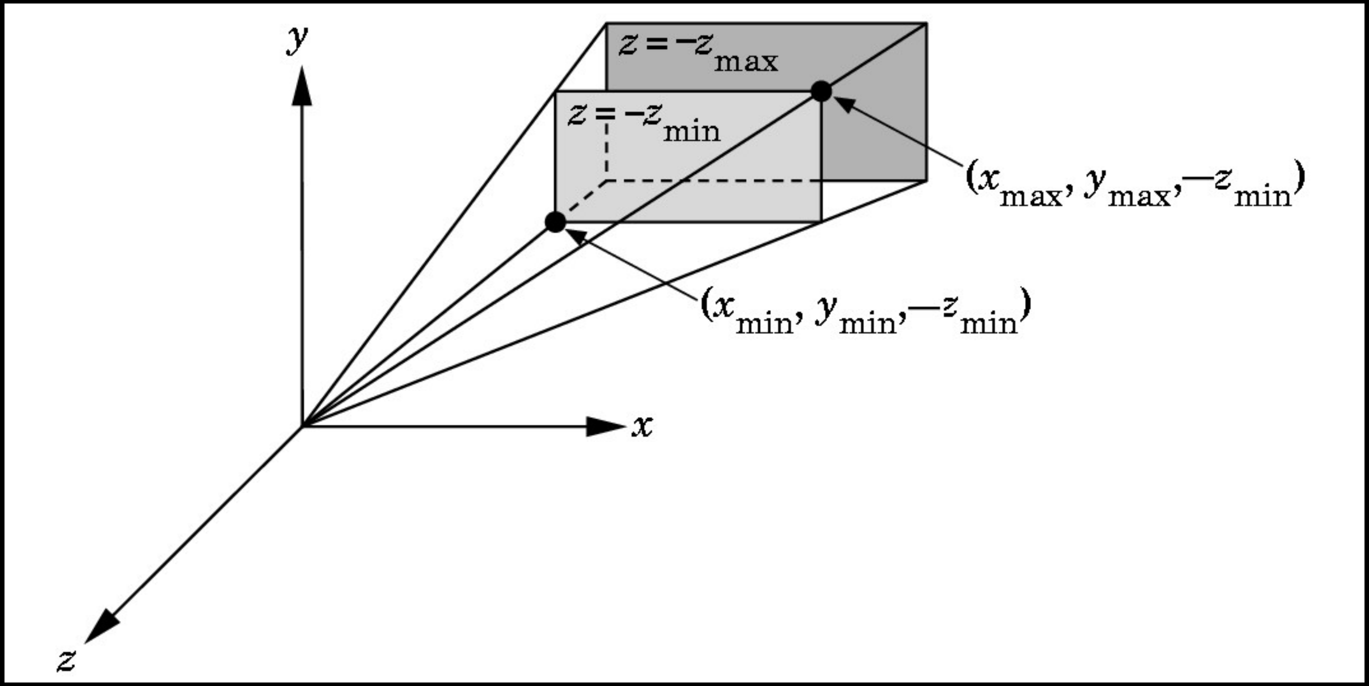


https://commons.wikimedia.org/wiki/
File:Square_frustum.png

Perspective Viewing in OpenGL

- Two interfaces: glFrustum and gluPerspective

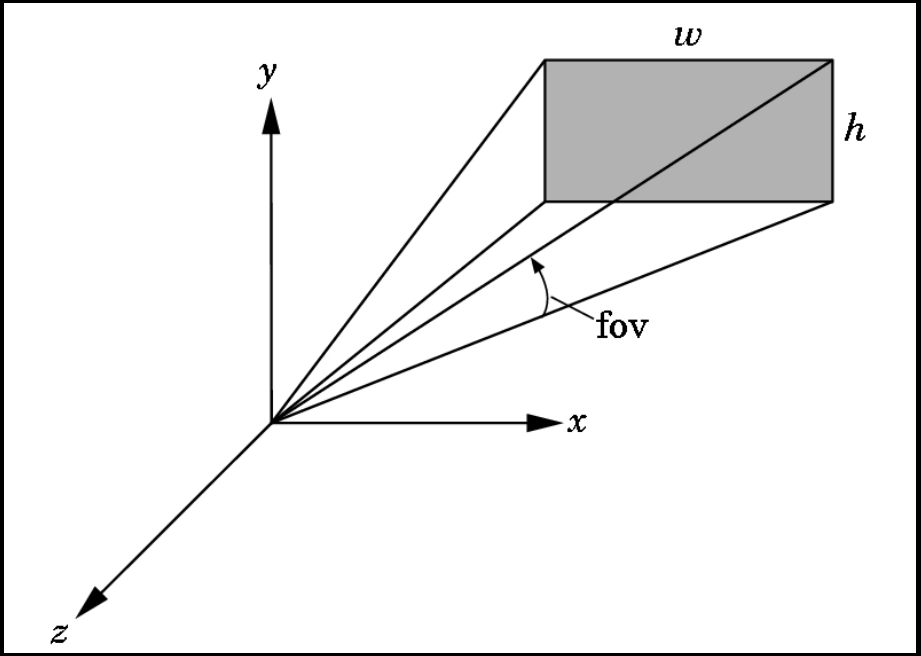
```
OpenGLMatrix::Frustum(xmin, xmax, ymin, ymax,
                      near, far);
```



z_{min} = near, z_{max} = far

Field of View Interface

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



Frustum Projection Matrix in Helper Code

- Transform view frustum to OpenGL's canonical cube

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

$$\begin{pmatrix} \frac{2n}{x_{\max} - x_{\min}} & 0 & \frac{x_{\max} + x_{\min}}{x_{\max} - x_{\min}} & 0 \\ 0 & \frac{2n}{y_{\max} - y_{\min}} & \frac{y_{\max} + y_{\min}}{y_{\max} - y_{\min}} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspective Projection Matrix in Helper Code

- Transform view frustum to OpenGL's canonical cube

```
OpenGLMatrix::Perspective(y, aspect, near, far)
```

$t = \tan y$ OR (fovy convention) $t = \tan \frac{y}{2}$

$$\begin{pmatrix} \frac{1}{at} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

OpenGL code (in reshape callback)

```
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);
}
```

OpenGL code

```
void displayFunc()
{
  …


openGLMatrix.SetMatrixMode(OpenGLMatrix::Projection);
  float p[16]; // column-major
  openGLMatrix.GetMatrix(p);
  const GLboolean isRowMajor = GL_FALSE;
  pipelineProgram->SetUniformVariableMatrix4fv(
      ”projectionMatrix", isRowMajor, p);
  …

  renderBunny();
  glutSwapBuffers();
}
```

Summary

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