CSE 167:
Introduction to Computer Graphics
Lecture #3: Projection

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2010

Announcements

- Remaining office hours in the lab before deadline:
  - Iman: Thu 3:30pm-7:30pm
  - Haili: Thu 3:30pm-4:30pm
- Project 1 due Friday October 1st, presentation in lab 260 from 2-5pm
  - Both executable and source code required for grading. We will ask questions about the code!
  - List your name on the whiteboard in the grading section once you get to the lab. Homework will be graded in this order.
  - We will also have a help section on the whiteboard. List your name there to get help. We will give priority to the grading list!
- Project 2 due Friday October 8th; presentation in lab 260 from 2-5pm
  - Introduction by Iman on Mon at 2pm in lab 260
- Don't save anything on the C: drive of the lab PCs! You will lose it when you log out.

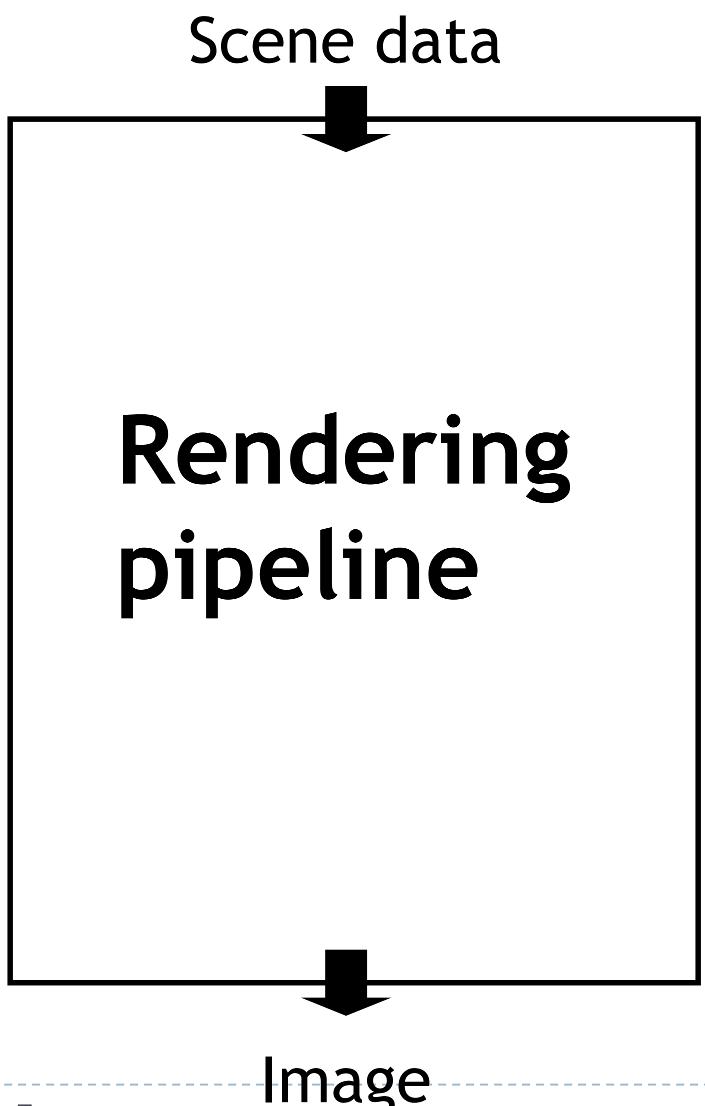
Objects in camera coordinates

- We have things lined up the way we like them on screen
  - x to the right
  - y up
  - $-z$ going into the screen
  - Objects to look at are in front of us, i.e. have negative z values
- But objects are still in 3D
- Problem: project them into 2D

Lecture Overview

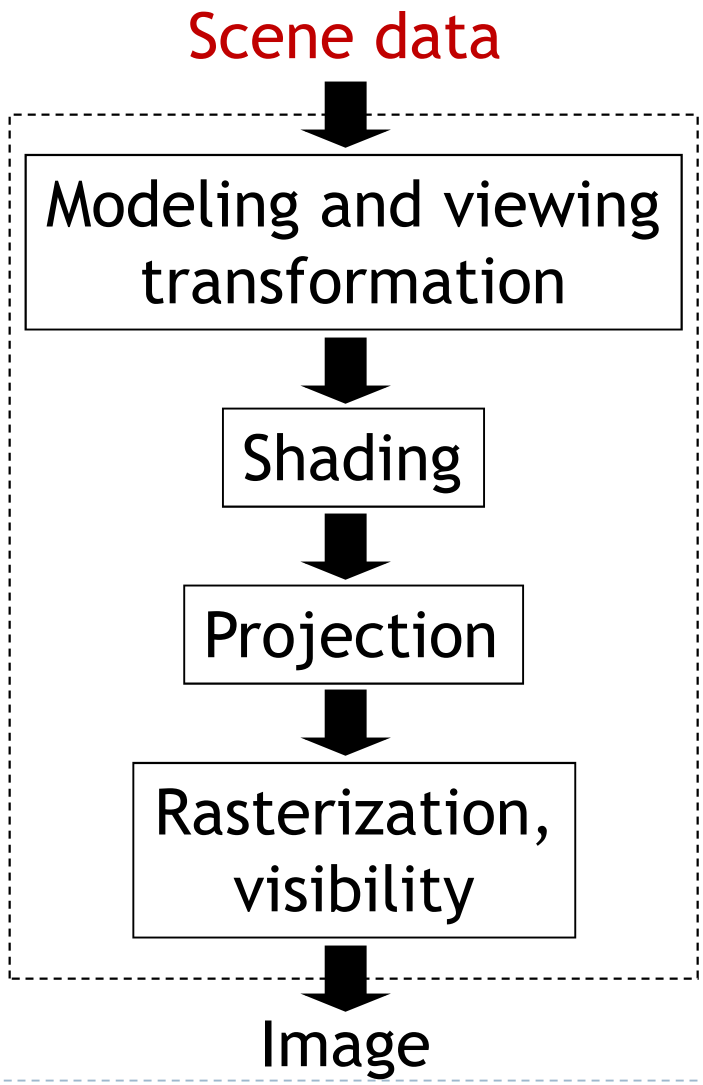
- Rendering Pipeline
- Projections
- View Volumes, Clipping

Rendering Pipeline

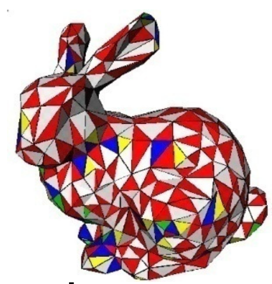


- Hardware and software which draws 3D scenes on the screen
- Consists of several stages
  - Simplified version here
- Most operations performed by specialized hardware (GPU)
- Access to hardware through low-level 3D API (OpenGL, DirectX)
- All scene data flows through the pipeline at least once for each frame

Rendering Pipeline

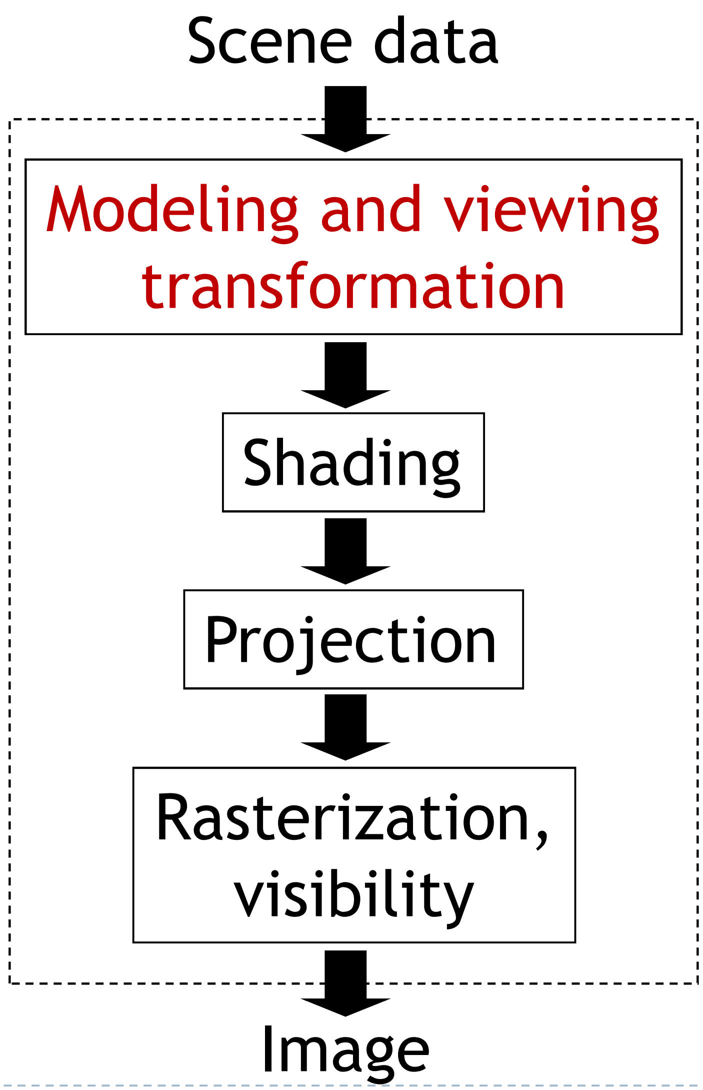


- Textures, lights, etc.
- Geometry
  - Vertices and how they are connected
  - Triangles, lines, points, triangle strips
  - Attributes such as color



- Specified in object coordinates
- Processed by the rendering pipeline one-by-one

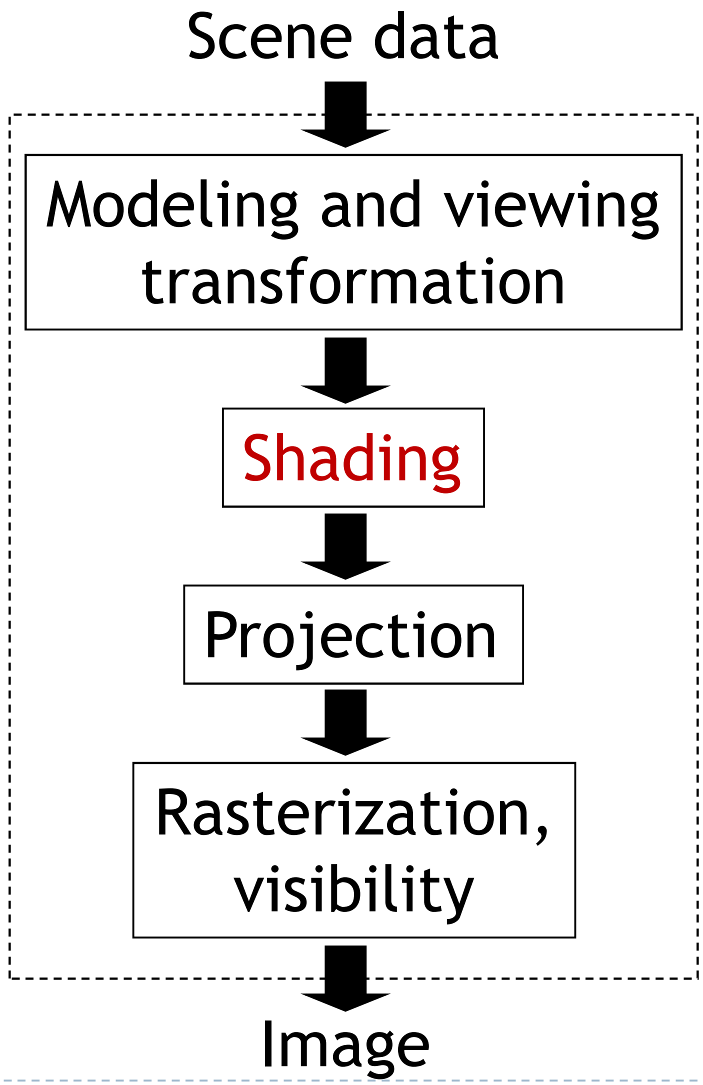
Rendering Pipeline



- Transform object to camera coordinates
- Specified by GL_MODELVIEW matrix in OpenGL
- User computes GL_MODELVIEW matrix as discussed

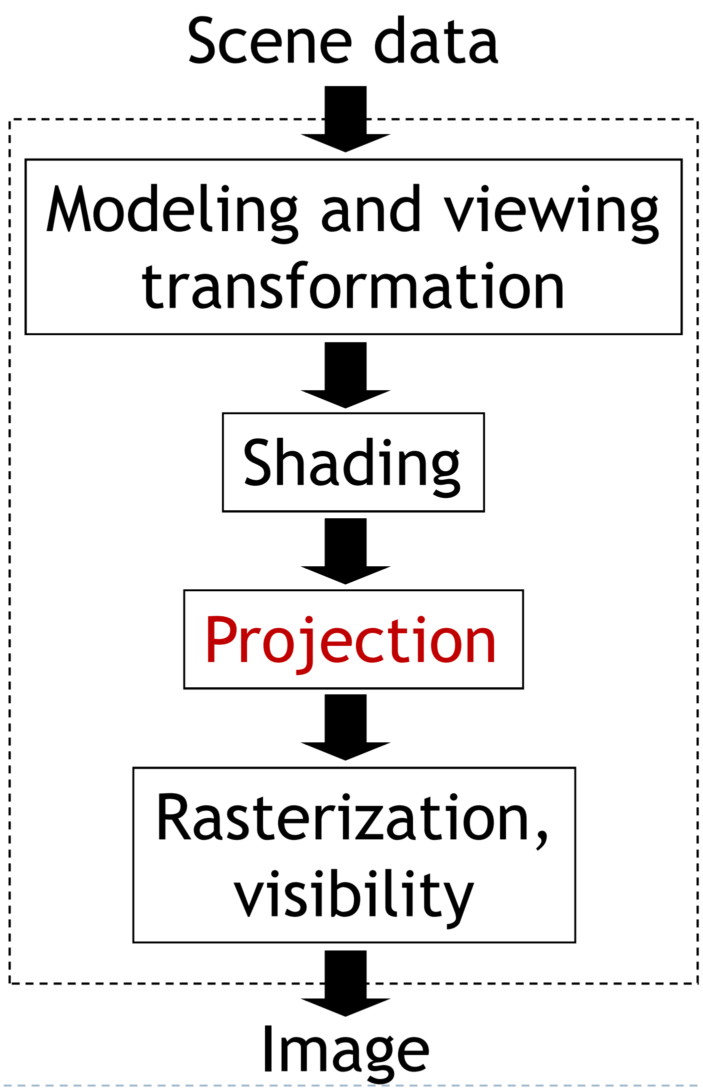
$$\mathbf{p}_{camera} = \underbrace{\mathbf{C}^{-1}\mathbf{M}}_{\text{MODELVIEW matrix}}\mathbf{p}_{object}$$

Rendering Pipeline



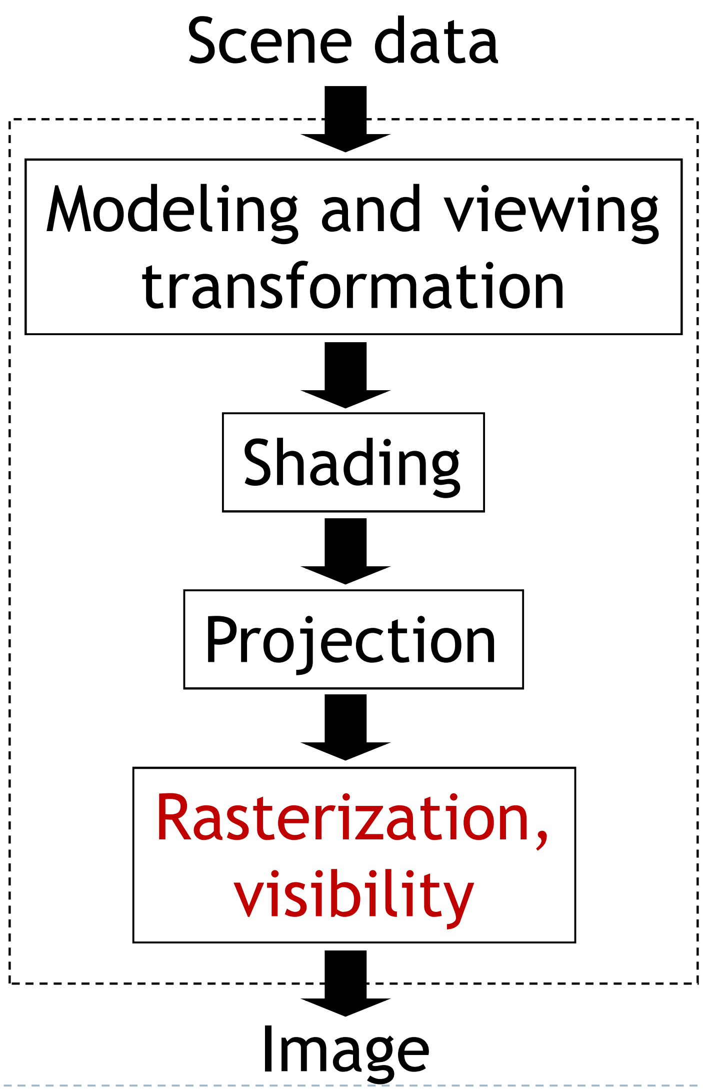
- Look up light sources
- Compute color for each vertex
- Covered later in the course

Rendering Pipeline

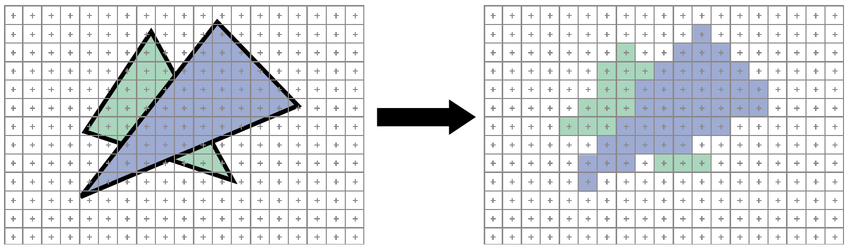


- Project 3D vertices to 2D image positions
- GL_PROJECTION matrix
- Covered in today's lecture

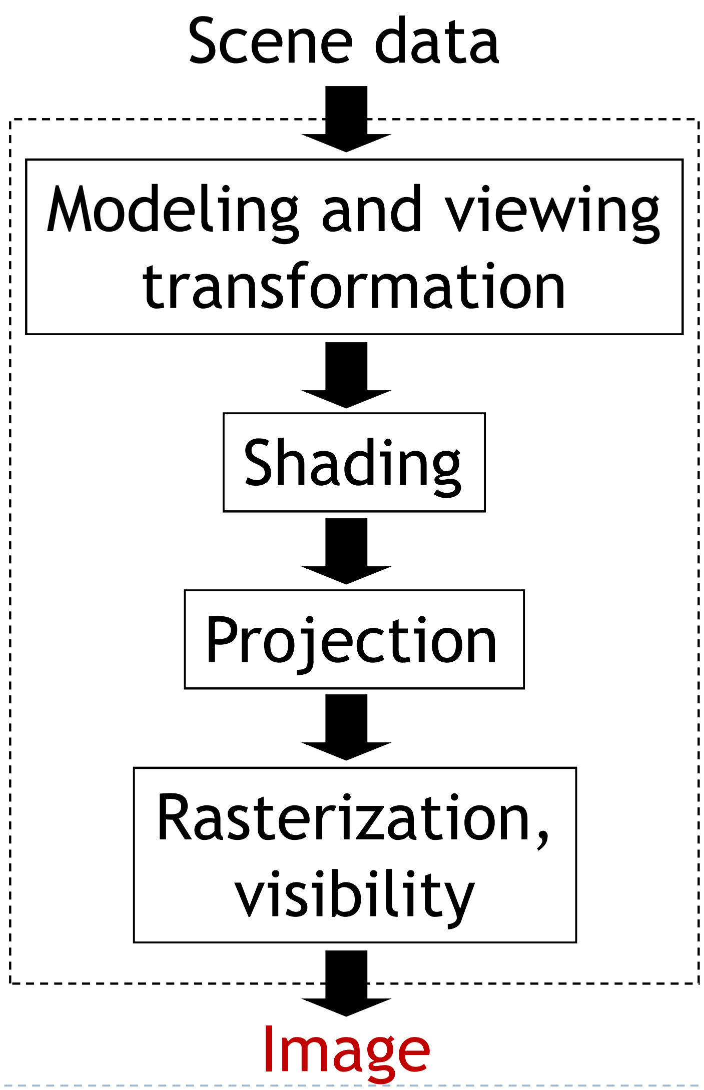
Rendering Pipeline



- Draw primitives (triangles, lines, etc.)
- Determine what is visible
- Covered in next lecture

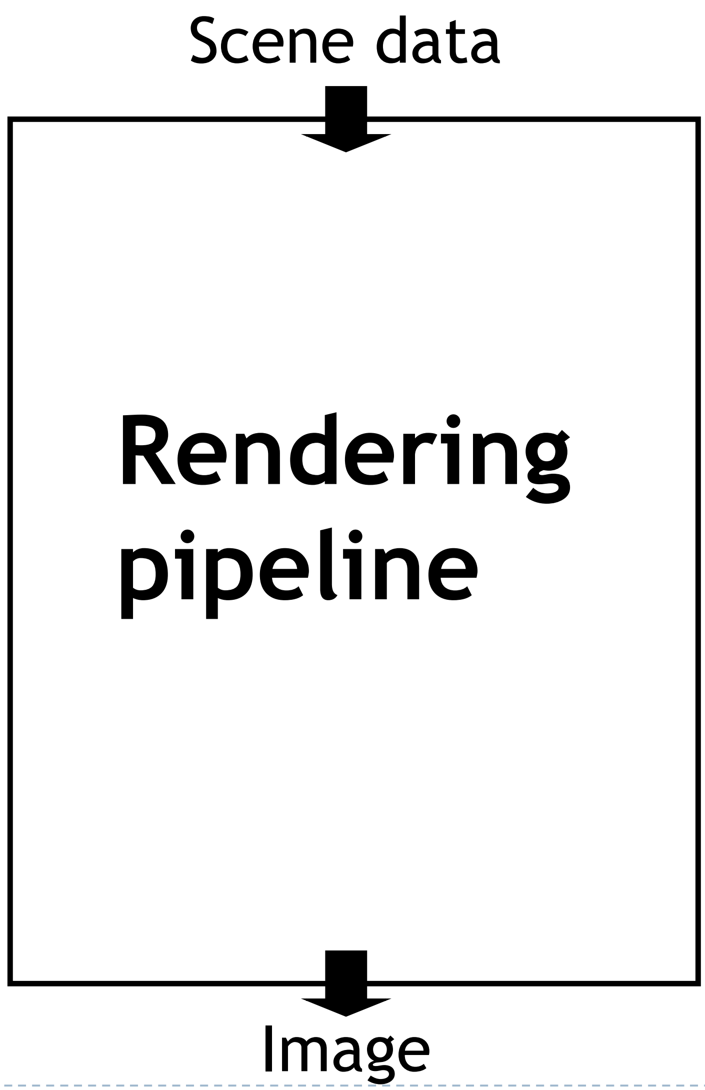


Rendering Pipeline



- Pixel colors

Rendering Engine



- Additional software layer encapsulating low-level API
- Higher level functionality than OpenGL
- Platform independent
- Layered software architecture common in industry
  - Game engines http://en.wikipedia.org/wiki/Game_engine

Lecture Overview

- Rendering Pipeline
- Projections
- View Volumes, Clipping

Projections

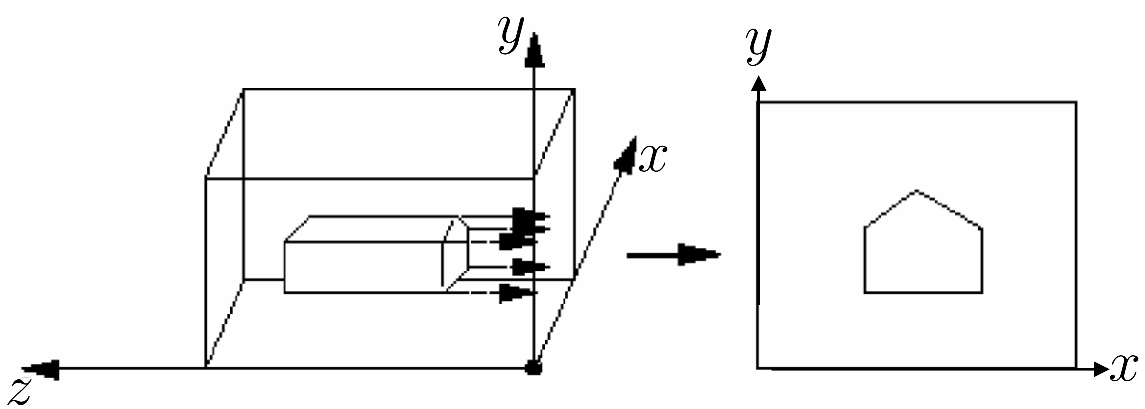
- Given 3D points (vertices) in camera coordinates, determine corresponding image coordinates

Orthographic Projection

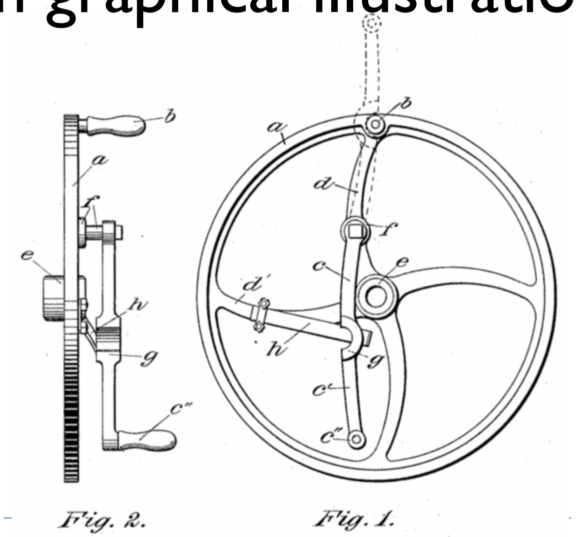
- a.k.a. Parallel Projection
- Done by ignoring z-coordinate
- Use camera space xy coordinates as image coordinates

Orthographic Projection

- Project points to x-y plane along parallel lines



- Used in graphical illustrations, architecture, 3D modeling

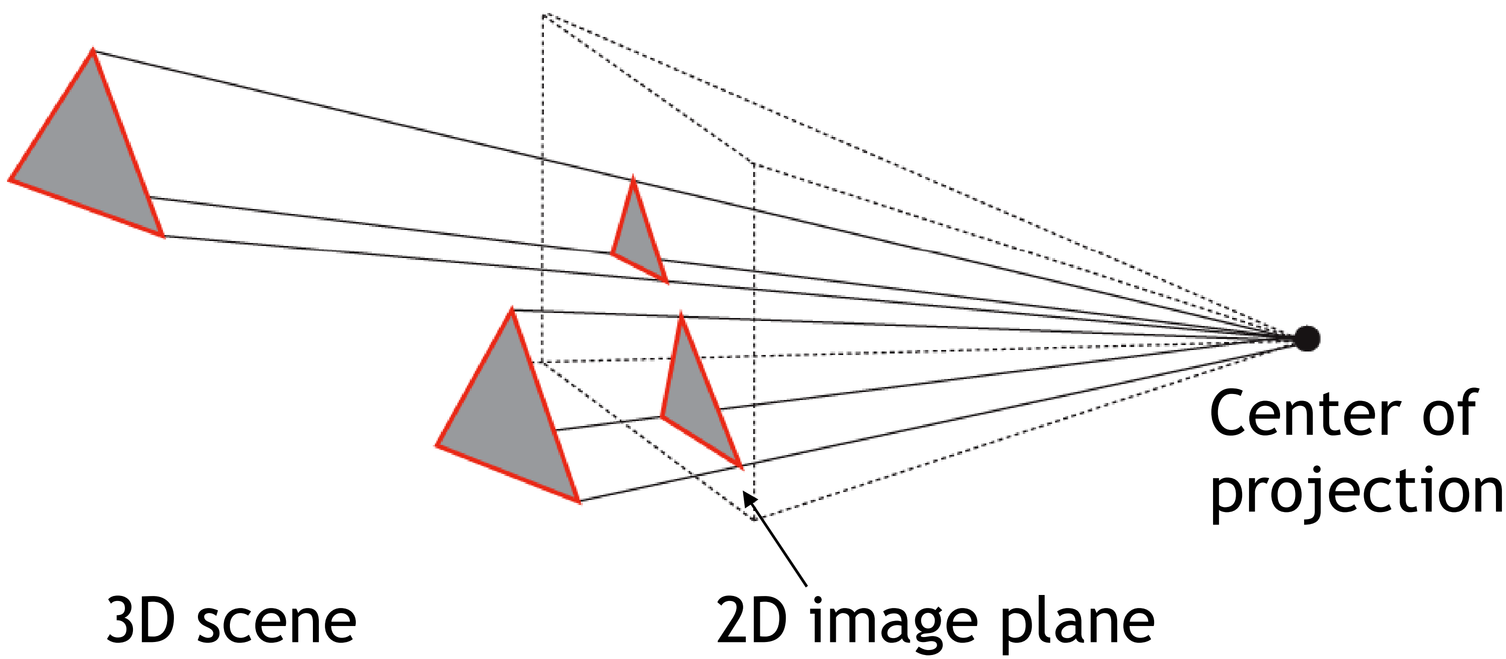


Perspective Projection

- Most common for computer graphics
- Simplified model of human eye, or camera lens (*pinhole camera*)
- Things farther away appear to be smaller
- Discovery attributed to Filippo Brunelleschi (Italian architect) in the early 1400's

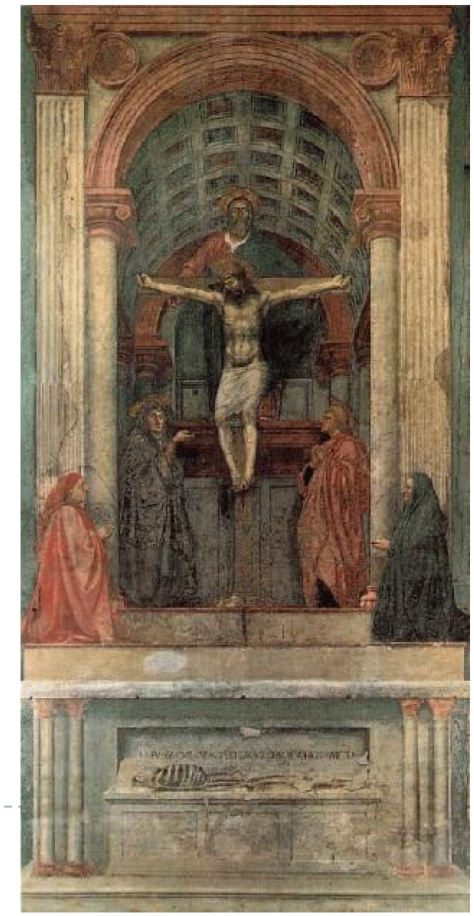
Perspective Projection

- Project along rays that converge in center of projection



Perspective Projection

Parallel lines are
no longer parallel,
converge in one point



Earliest example:
La Trinitá (1427) by Masaccio

Perspective Projection

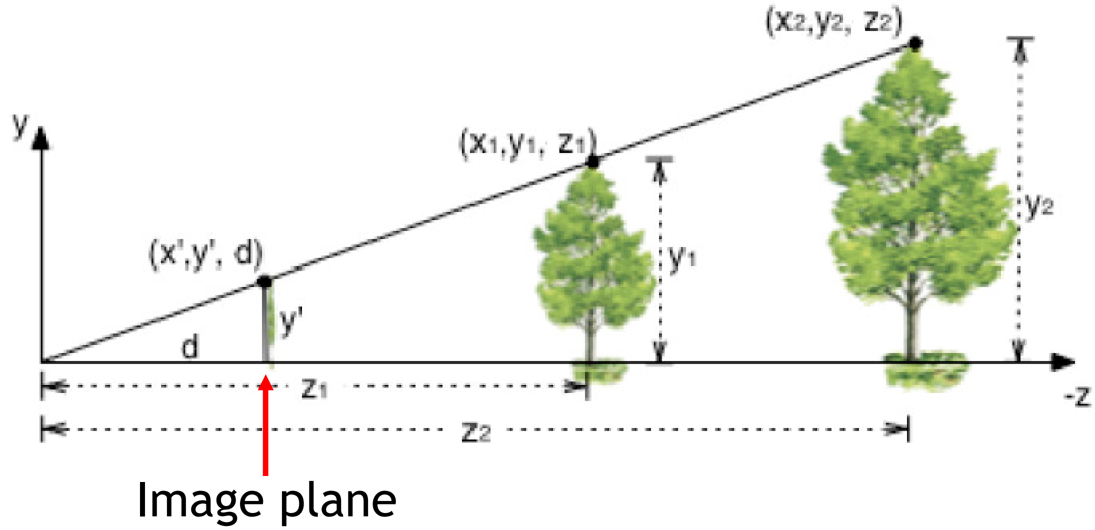
The math: simplified case

$$\frac{y'}{d} = \frac{y_1}{z_1}$$

$$y' = \frac{y_1 d}{z_1}$$

$$x' = \frac{x_1 d}{z_1}$$

$$z' = d$$



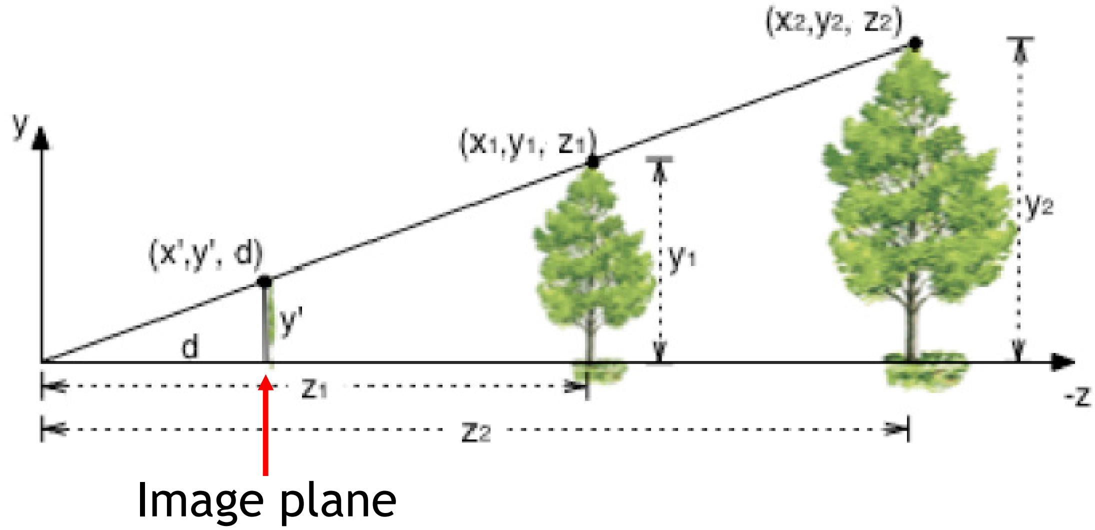
Perspective Projection

The math: simplified case

$$x' = \frac{x_1 d}{z_1}$$

$$y' = \frac{y_1 d}{z_1}$$

$$z' = d$$

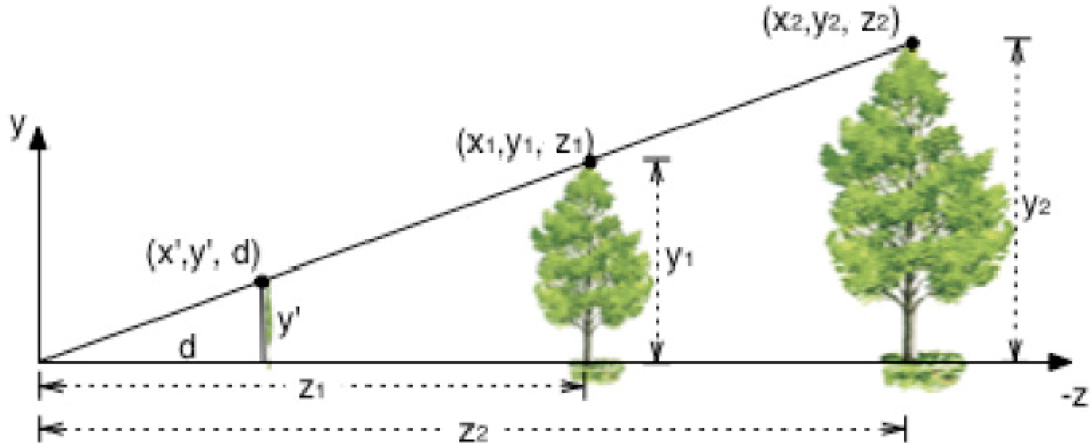


- We can express this using homogeneous coordinates and 4x4 matrices

Perspective Projection

The math: simplified case

$$x' = \frac{x_1 d}{z_1}$$



$$y' = \frac{y_1 d}{z_1}$$

$$z' = d$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \Rightarrow \begin{bmatrix} xd/z \\ yd/z \\ d \\ 1 \end{bmatrix}$$

Projection matrix Homogeneous division

Perspective Projection

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} xd/z \\ yd/z \\ d \\ 1 \end{bmatrix}$$

Projection matrix Homogeneous division

- Using projection matrix, homogeneous division seems more complicated than just multiplying all coordinates by d/z, so why do it?
- It will allow us to:
  - handle different types of projections in a unified way
  - define arbitrary view volumes
- Divide by *w* (perspective division, homogeneous division) after performing projection transform
  - Graphics hardware does this automatically

Photorealistic Rendering

- More than just perspective projection
- Some effects are too complex for hardware rendering
- For example: lens effects

Focus, depth of field

Fish-eye lens

Photorealistic Rendering

Chromatic Aberration

Motion Blur

Lecture Overview

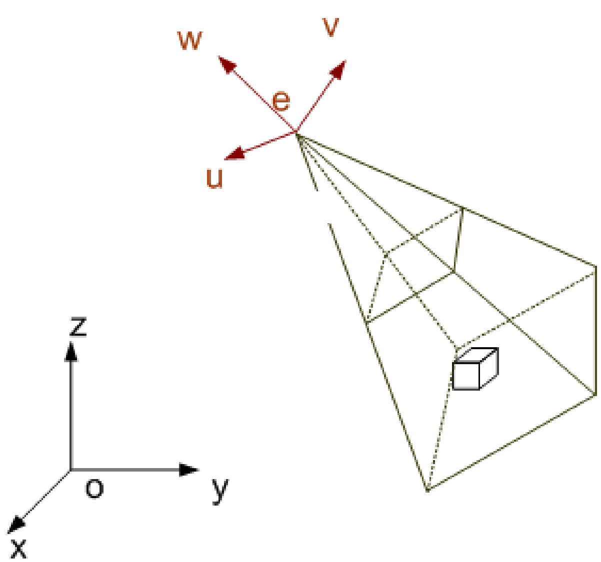
- Rendering Pipeline
- Projections
- View Volumes, Clipping

View Volumes

- Define 3D volume seen by camera

Perspective view volume

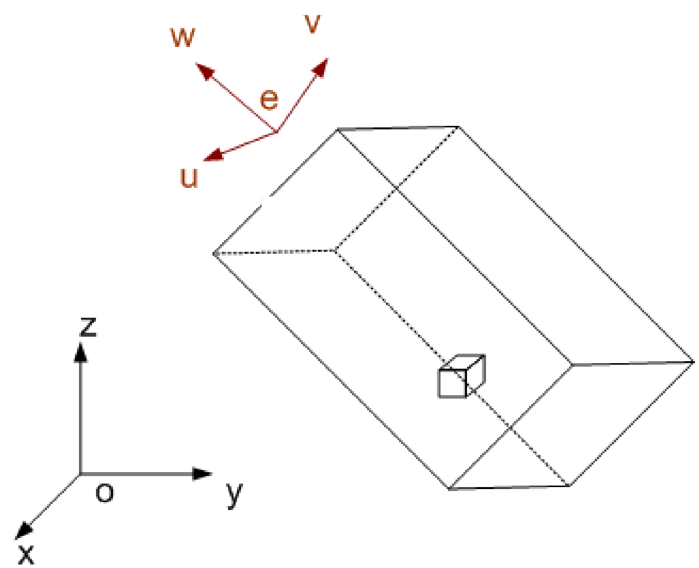
Camera coordinates



World coordinates

Orthographic view volume

Camera coordinates

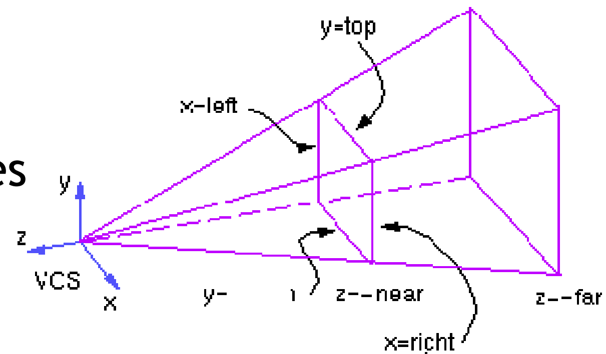


World coordinates

Perspective View Volume

General view volume

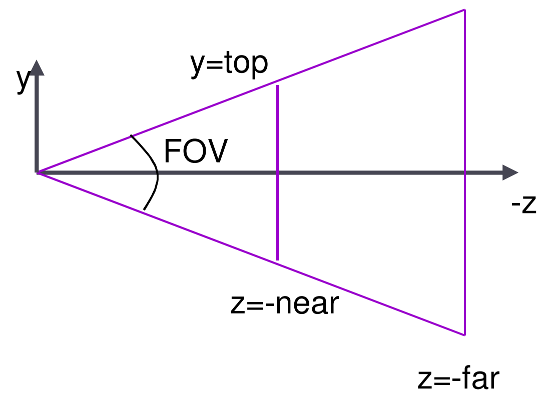
Camera coordinates



- Defined by 6 parameters, in camera coordinates
  - Left, right, top, bottom boundaries
  - Near, far clipping planes
- Clipping planes to avoid numerical problems
  - Divide by zero
  - Low precision for distant objects
- Usually symmetric, i.e., left=-right, top=-bottom

Perspective View Volume

Symmetrical view volume

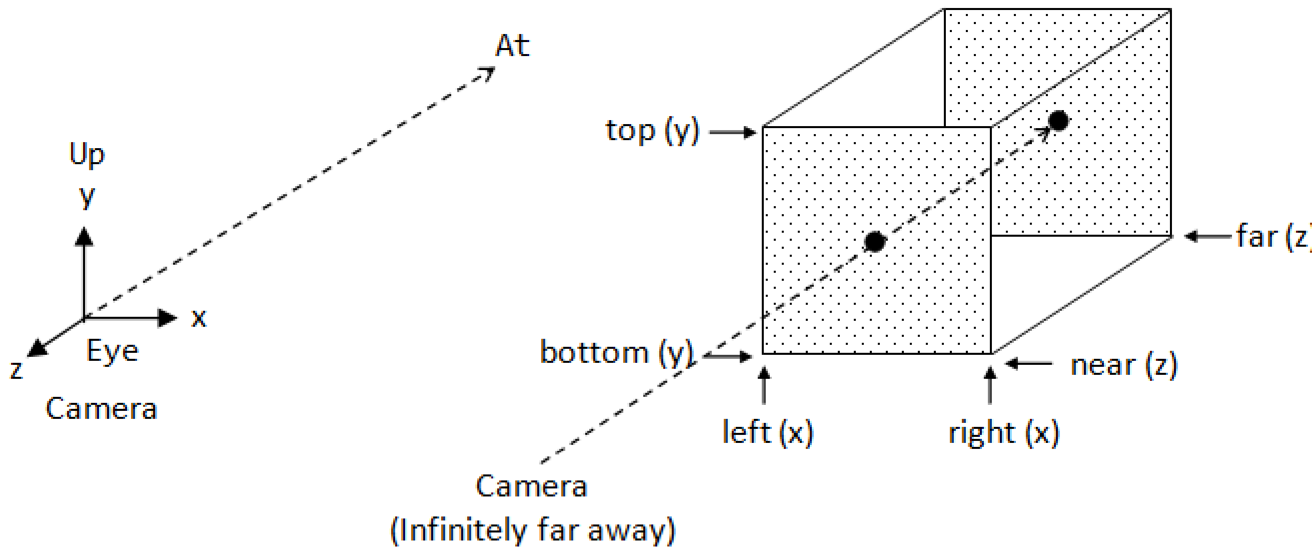


- Only 4 parameters
  - Vertical field of view (FOV)
  - Image aspect ratio (width/height)
  - Near, far clipping planes

$$\text{aspect ratio}=\frac{right-left}{top-bottom}=\frac{right}{top}$$

$$\tan(FOV/2)=\frac{top}{near}$$

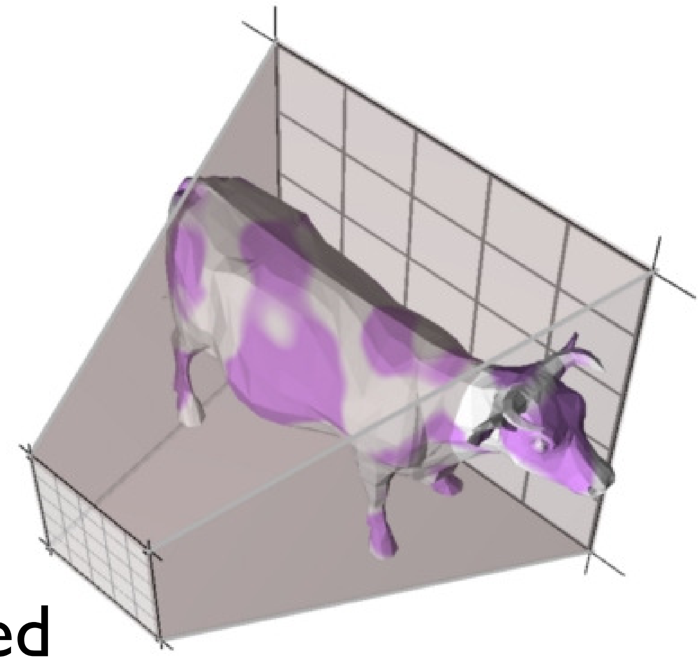
Orthographic View Volume



- Parameterized by 6 parameters
  - Right, left, top, bottom, near, far
- Or if symmetrical:
  - Width, height, near, far

Clipping

- Need to identify objects outside view volume
  - Avoid division by zero
  - Efficiency: don't draw objects outside view volume (view frustum culling)
- Performed in hardware
- Hardware always clips to the *canonical view volume:* cube [-1..1]x[-1..1]x[-1..1] centered at origin
- Need to transform **desired** view frustum to **canonical** view frustum



Canonical View Volume

- Projection matrix is set such that
  - User defined view volume is transformed into canonical view volume, i.e., cube $[-1,1] \times [-1,1] \times [-1,1]$
  - Multiplying vertices of view volume by projection matrix and performing homogeneous divide yields canonical view volume
- Perspective and orthographic projection are treated exactly the same way
- Canonical view volume is last stage in which coordinates are in 3D
- Next step is projection to 2D frame buffer

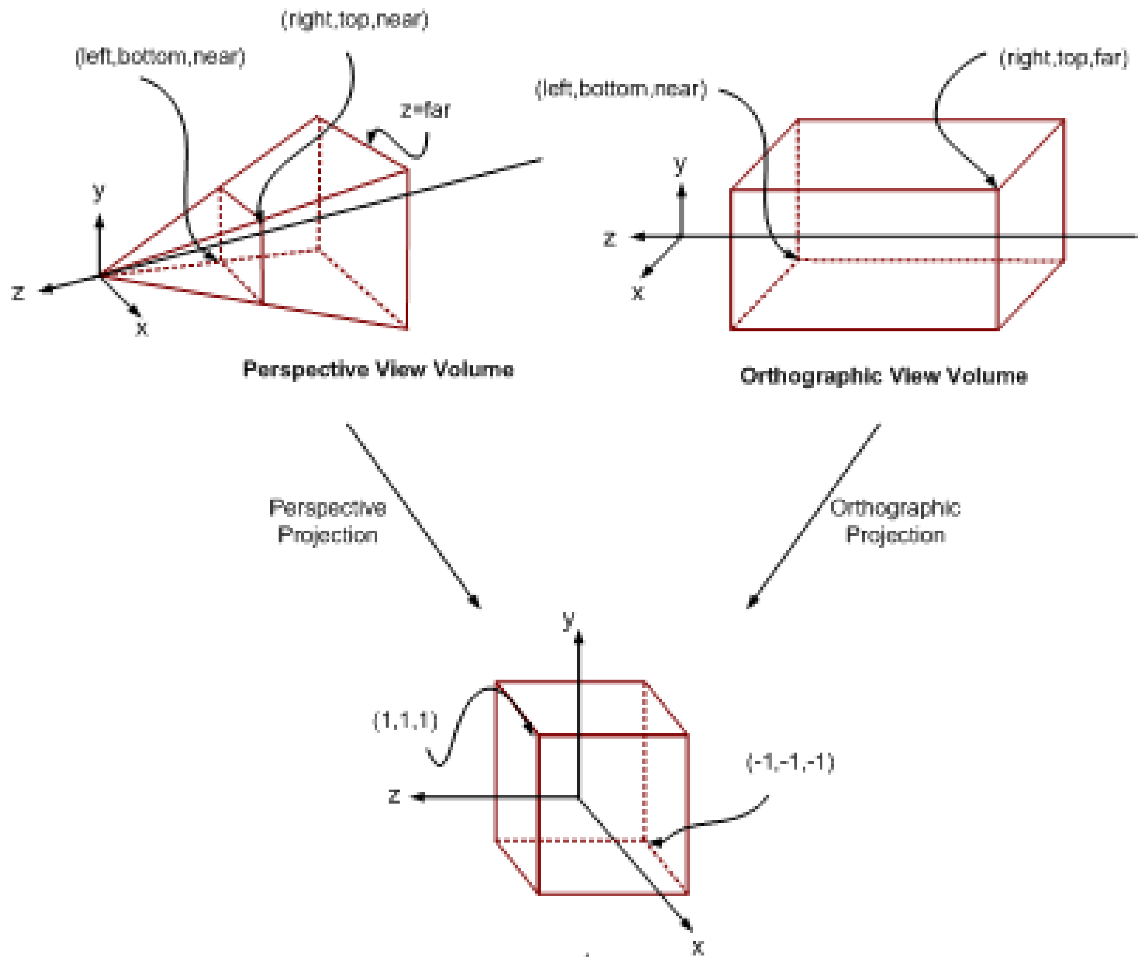
Projection Matrix

Camera coordinates

Projection matrix

Canonical view volume

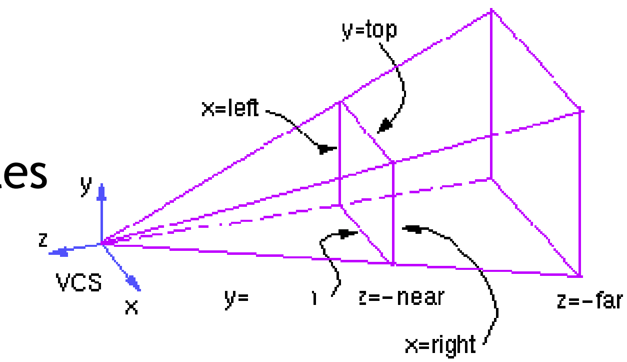
Clipping



Perspective Projection Matrix

- General view frustum with 6 parameters

Camera coordinates



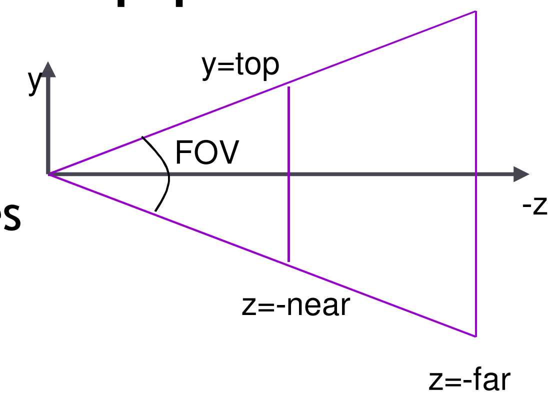
$$\mathbf{P}_{persp}(left, right, top, bottom, near, far) =$$

$$\begin{bmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2far\cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Perspective Projection Matrix

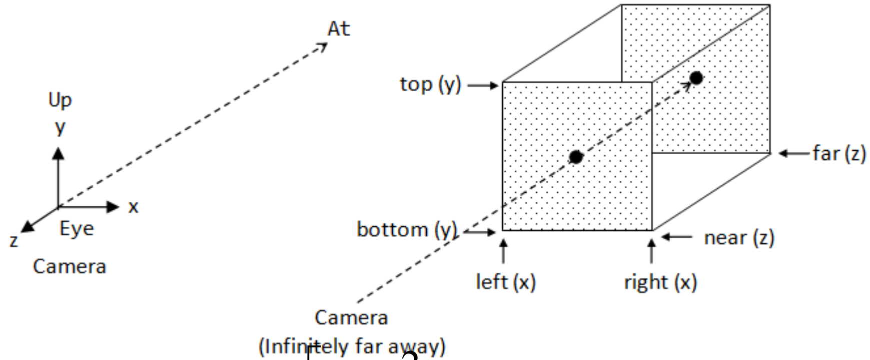
- Symmetrical view frustum with field of view, aspect ratio, near and far clip planes

Camera coordinates



$$\mathbf{P}_{persp}(FOV, aspect, near, far) = \begin{bmatrix} \frac{1}{aspect \cdot \tan(FOV/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(FOV/2)} & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2 \cdot near \cdot far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Orthographic Projection Matrix



$$\mathbf{P}_{ortho}(right, left, top, bottom, near, far) = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P}_{ortho}(width, height, near, far) = \begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewport Transformation

- After applying projection matrix, scene points are in *normalized viewing coordinates*
  - Per definition range [-1..1] x [-1..1] x [-1..1]
- Normalized viewing coordinates can be mapped to image (=pixel=frame buffer) coordinates
  - Range depends on window (view port) size: [x0…x1] x [y0…y1]
- Scale and translation required:

$$\mathbf{D}(x_0, x_1, y_0, y_1) = \begin{bmatrix} (x_1 - x_0)/2 & 0 & 0 & (x_0 + x_1)/2 \\ 0 & (y_1 - y_0)/2 & 0 & (y_0 + y_1)/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Complete Transform

- Mapping a 3D point in object coordinates to pixel coordinates:

$$\mathbf{p}' = \mathbf{DPC}^{-1}\mathbf{M}\underbrace{\mathbf{p}}_{\text{Object space}}$$

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

The Complete Transform

- Mapping a 3D point in object coordinates to pixel coordinates:

$$\mathbf{p}' = \mathbf{DPC}^{-1}\mathbf{M}\mathbf{p}$$

Object space

World space

- **M**: Object-to-world matrix
- **C**: camera matrix
- **P**: projection matrix
- **D**: viewport matrix

The Complete Transform

- Mapping a 3D point in object coordinates to pixel coordinates:

$$\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$$

Object space

World space

Camera space

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

The Complete Transform

- Mapping a 3D point in object coordinates to pixel coordinates:

$$\mathbf{p}' = \mathbf{D}\,\mathbf{P}\,\mathbf{C}^{-1}\,\mathbf{M}\,\mathbf{p}$$

Object space

World space

Camera space

Canonical view volume

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

The Complete Transform

- Mapping a 3D point in object coordinates to pixel coordinates: $\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$

  Object space

  World space

  Camera space

  Canonical view volume

  Image space

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

The Complete Transform

- Mapping a 3D point in object coordinates to pixel coordinates:

$$\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$$

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

Pixel coordinates: $\begin{matrix} x'/w' \\ y'/w' \end{matrix}$

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

The Complete Transform in OpenGL

- Mapping a 3D point in object coordinates to pixel coordinates:

OpenGL GL_MODELVIEW matrix

$$\mathbf{p}' = \mathbf{D}\underbrace{\mathbf{P}}_{\text{OpenGL GL\_PROJECTION matrix}}\overbrace{\mathbf{C}^{-1}\mathbf{M}}^{\text{OpenGL GL\_MODELVIEW matrix}}\mathbf{p}$$

OpenGL GL_PROJECTION matrix

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

The Complete Transform in OpenGL

- GL_MODELVIEW, $\mathbf{C}^{-1}\mathbf{M}$
  - Defined by programmer
- GL_PROJECTION, **P**
  - Utility routines to set it by specifying view volume: glFrustum(), glPerspective(), glOrtho()
  - Do not use utility functions in homework project 2
  - You will implement a software renderer in project 3, which will not use OpenGL
- Viewport, **D**
  - Specify implicitly via glViewport()
  - No direct access with equivalent to GL_MODELVIEW or GL_PROJECTION

Next Lecture

- Viewport Transformation
- Drawing (Rasterization)
- Visibility (Z-Buffering)