CSE 167:
Introduction to Computer Graphics
Lecture #4: Projection Part 2

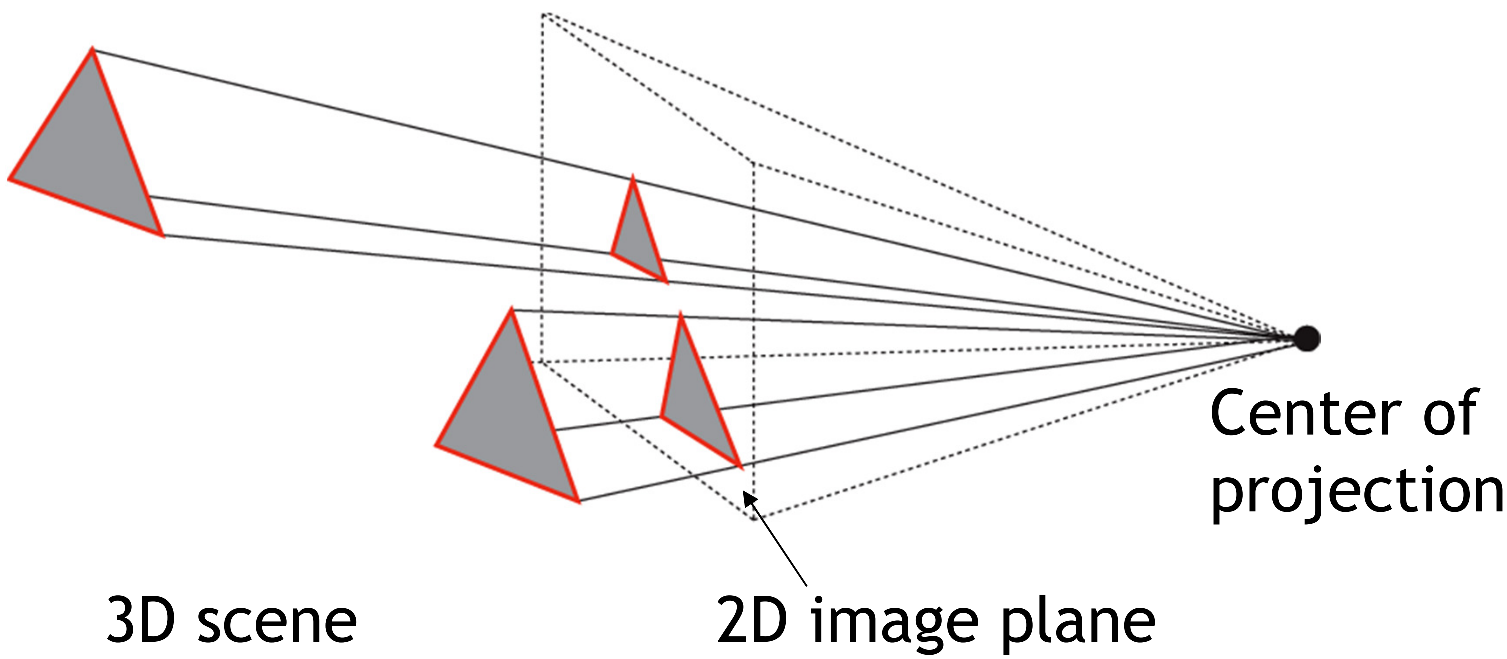
Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2014

Announcements

- Project 1 due Friday, 10/17 at 3:30pm
- Presentations start at 3:30pm in labs 260 and 270
- Weekly office hours:
  - Jurgen Schulze: Tue 3:30-4:30pm
  - Dylan McCarthy: Tue 5-9pm + Thu 11-1pm + Thu 8-10pm
  - Krishna Mullia: Tue 5-9pm + Thu 11-1pm + Thu 8-10pm
  - Phillip Ho: Tue 5-8pm
  - Max Takano: Wed 4-5:30pm + Thu 3:30-6pm
  - Rex West: Fri 9-11am + 1-2pm

Perspective Projection

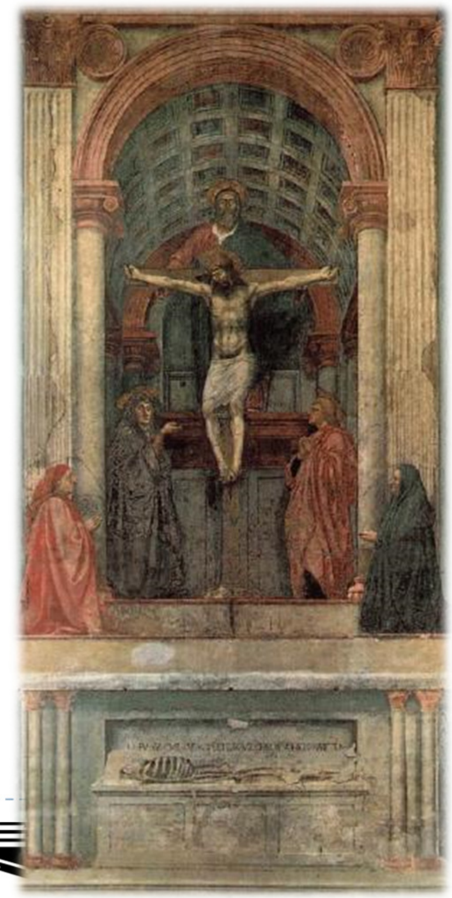
- Project along rays that converge in center of projection



UCSD

Perspective Projection

Parallel lines are no longer parallel, converge in one point



Earliest example:
La Trinitá (1427) by Masaccio

Video

- UCSD Professor Ravi Ramamoorthi on Perspective Projection
  - http://www.youtube.com/watch?v=VpNJbvZhNCQ

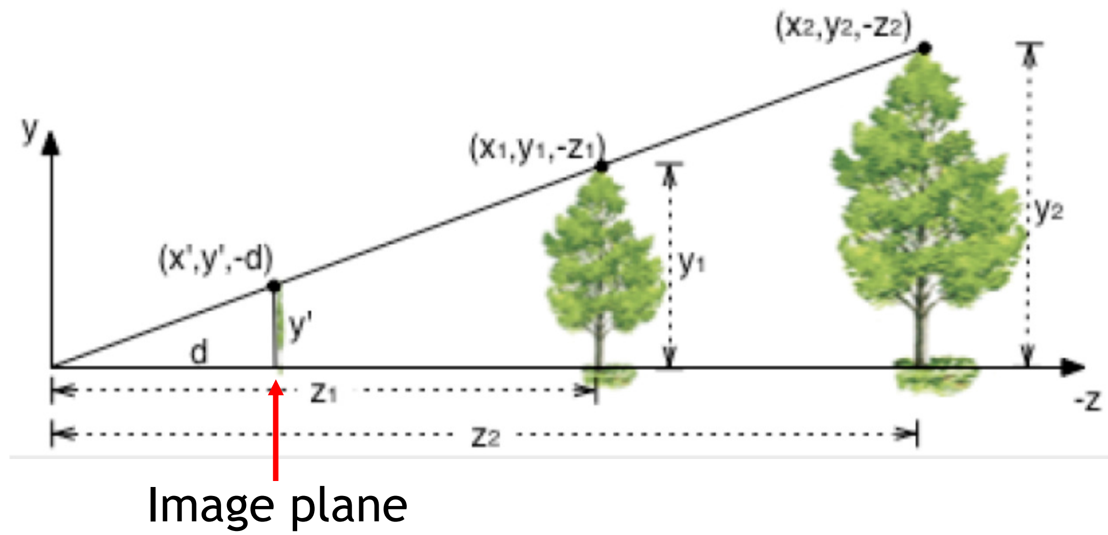
Perspective Projection

From law of ratios in similar triangles follows:

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

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

By definition: $$z' = d$$



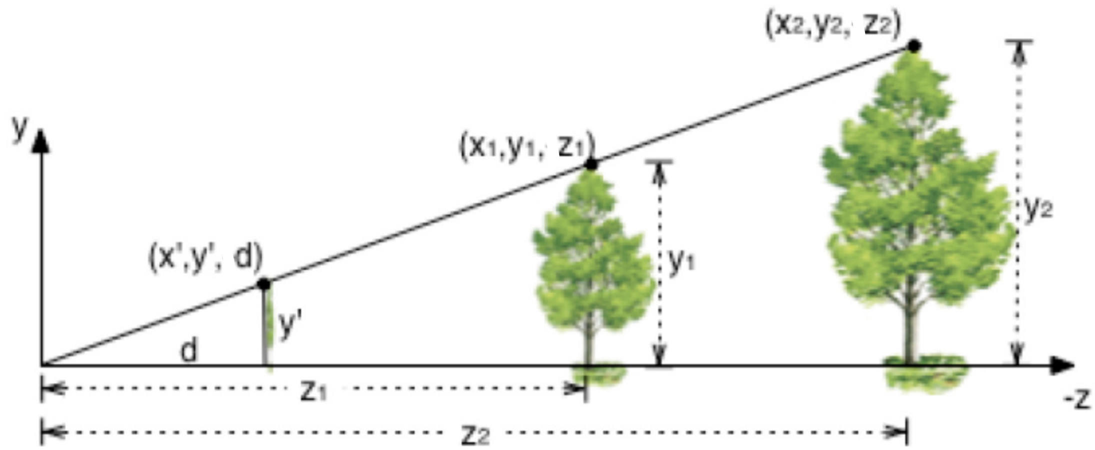
- We can express this using homogeneous coordinates and 4x4 matrices as follows

Perspective Projection

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

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

$$z' = d$$



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

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 P

- 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

Lecture Overview

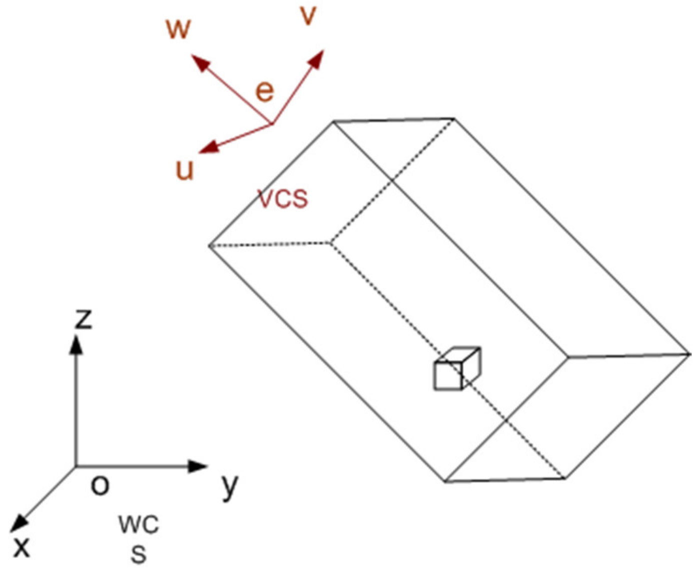
- View Volumes
- Vertex Transformation
- Rendering Pipeline
- Culling

View Volumes

- View volume = 3D volume seen by camera

Orthographic view volume

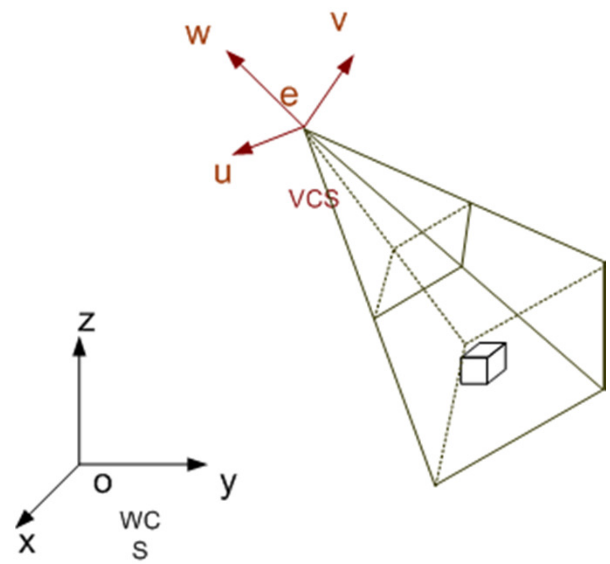
Camera coordinates



World coordinates

Perspective view volume

Camera coordinates



World coordinates

Projection Matrix

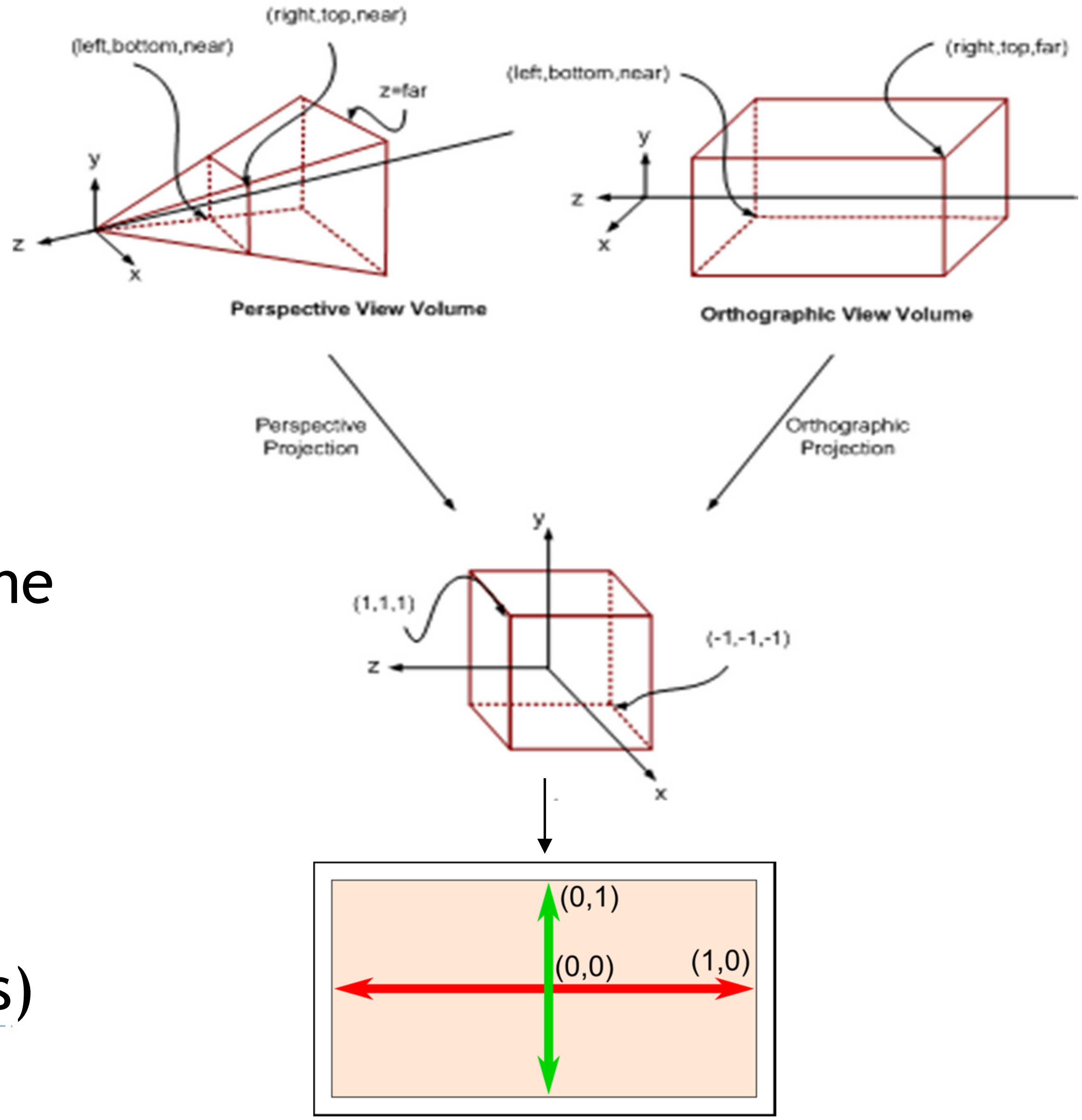
Camera coordinates

Projection matrix

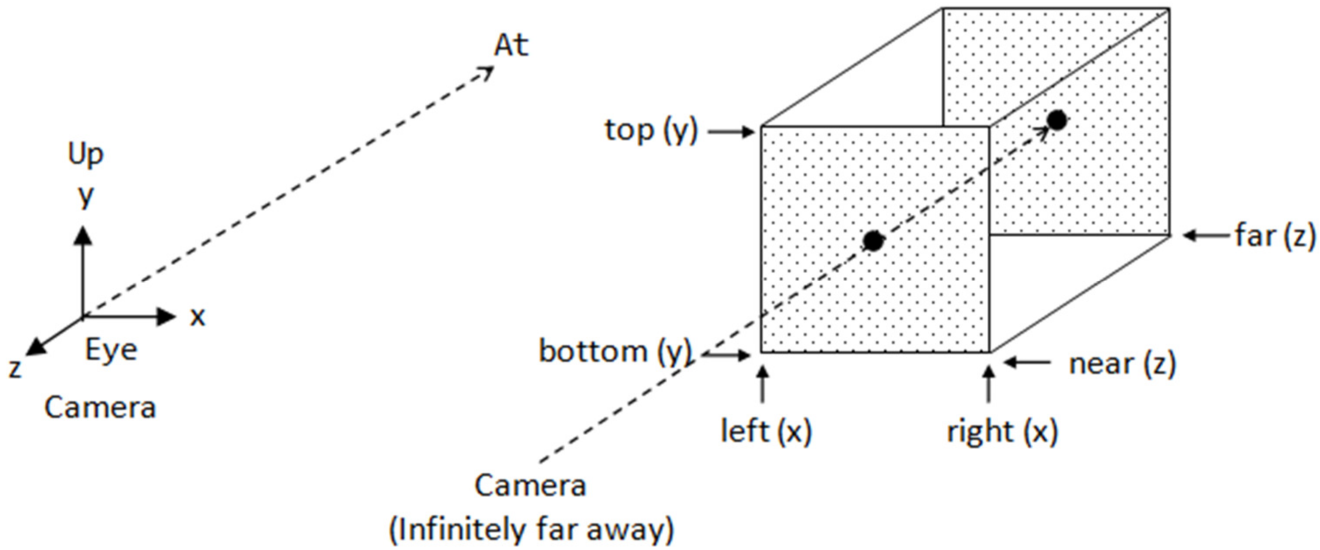
Canonical view volume

Viewport transformation

Image space
(pixel coordinates)



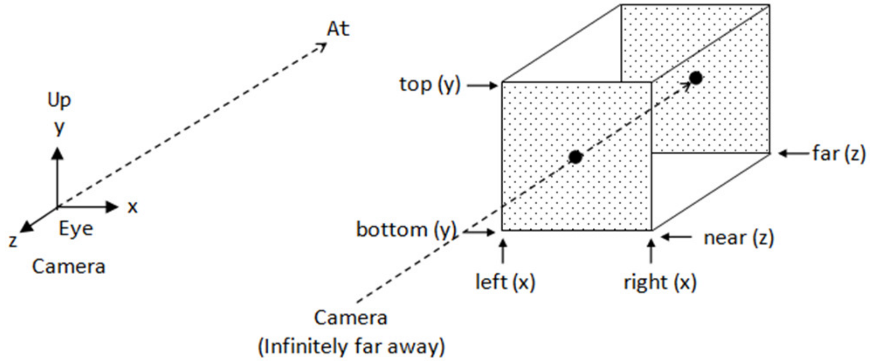
Orthographic View Volume



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

UCSD

Orthographic Projection Matrix



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

In OpenGL:
glOrtho(left, right, bottom, top, near, far)

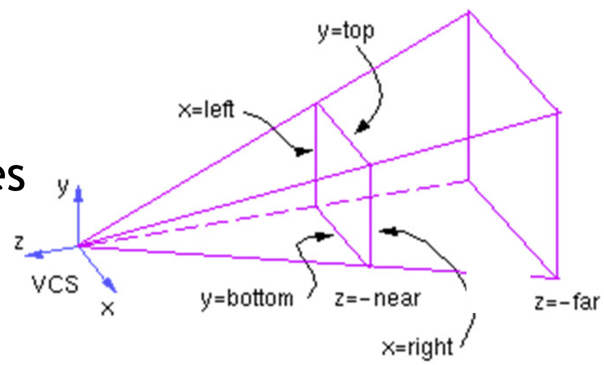
$$\mathbf{P}_{ortho}(width, height, near, far) = \begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

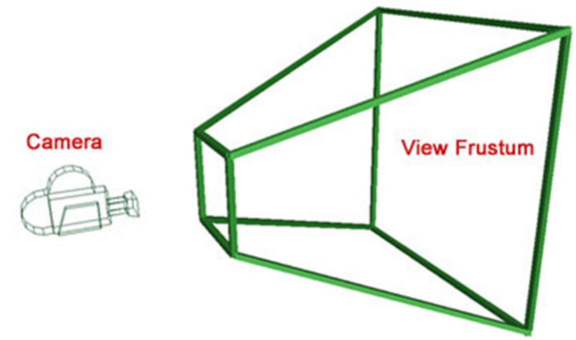
No equivalent in OpenGL

Perspective View Volume

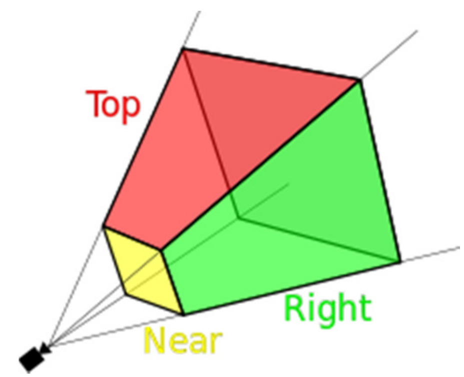
General view volume

Camera coordinates





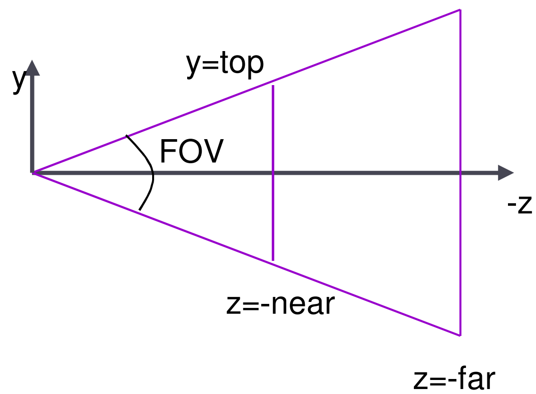
- Defined by 6 parameters, in camera coordinates
  - Left, right, top, bottom boundaries
  - Near, far clipping planes
- Clipping planes to avoid numerical problems
  - Divide by zero
  - Low precision for distant objects
- Usually symmetric, i.e., left=-right, top=-bottom



UCSD

Perspective View Volume

Symmetrical view volume



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

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

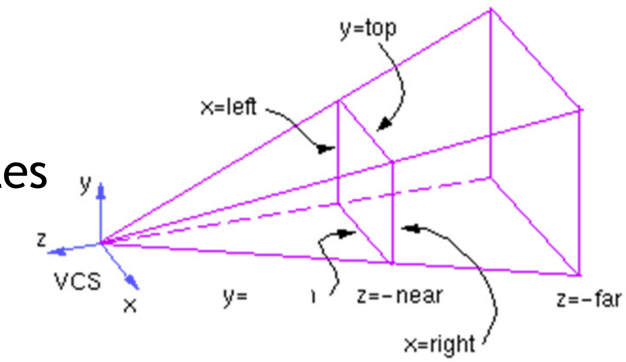
$$\tan(FOV/2)=\frac{top}{near}$$

UCSD

Perspective Projection Matrix

- General view frustum with 6 parameters

Camera coordinates



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

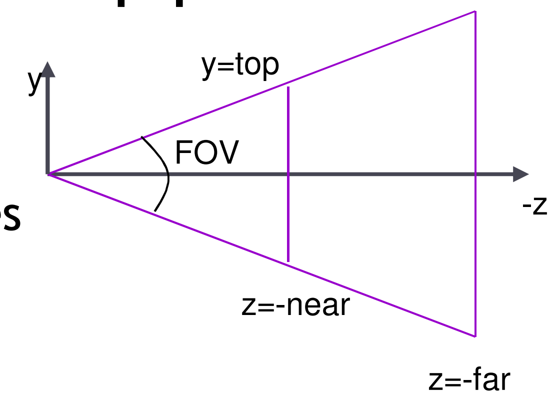
$$\begin{bmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

In OpenGL:
glFrustum(left, right, bottom, top, near, far)

Perspective Projection Matrix

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

Camera coordinates



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

In OpenGL:
gluPerspective(fov, aspect, near, far)

Canonical View Volume

- Goal: create projection matrix so that
  - User defined view volume is transformed into canonical view volume: cube [-1,1]x[-1,1]x[-1,1]
  - Multiplying corner vertices of view volume by projection matrix and performing homogeneous divide yields corners of canonical view volume
- Perspective and orthographic projection are treated the same way
- Canonical view volume is last stage in which coordinates are in 3D
  - Next step is projection to 2D frame buffer

Viewport Transformation

- After applying projection matrix, scene points are in *normalized viewing coordinates*
  - Per definition within range [-1..1] x [-1..1] x [-1..1]
- Next is projection from 3D to 2D (not reversible)
- 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:

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

Lecture Overview

- View Volumes
- Vertex Transformation
- Rendering Pipeline
- Culling

Complete Vertex Transformation

- 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

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

Complete Vertex Transformation

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

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

Object space (p)

World space (Mp)

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

Complete Vertex Transformation

- 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

Complete Vertex Transformation

- 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

Complete Vertex Transformation

- Mapping a 3D point in object coordinates to pixel coordinates: $\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$
  - $\mathbf{p}$: Object space
  - $\mathbf{M}\mathbf{p}$: World space
  - $\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$: Camera space
  - $\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$: Canonical view volume
  - $\mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$: Image space
  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

Complete Vertex Transformation

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

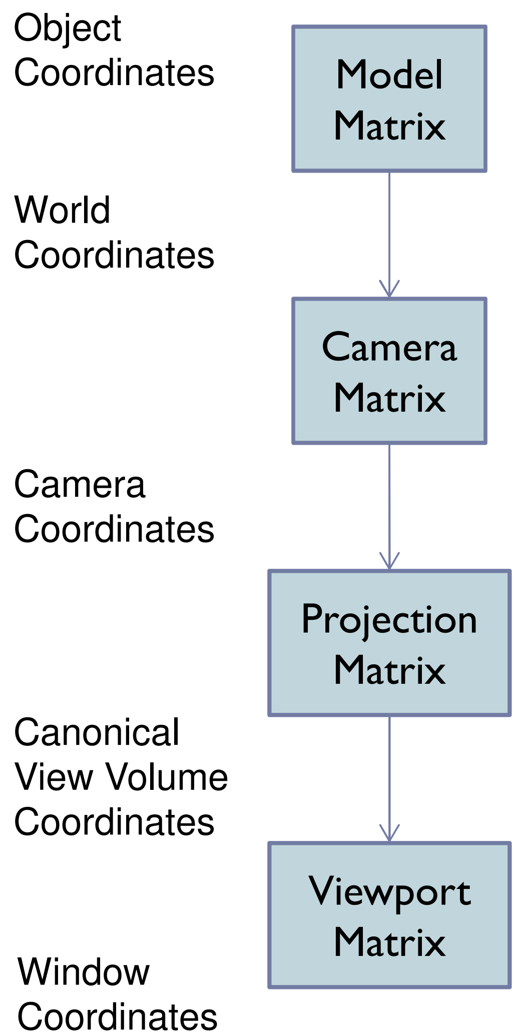
$$\mathbf{p}' = \mathbf{DPC}^{-1}\mathbf{Mp}$$

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

Pixel coordinates: x'/w', y'/w'

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

The Complete Vertex Transformation



Complete Vertex Transformation in OpenGL

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

OpenGL GL_MODELVIEW matrix

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

OpenGL GL_PROJECTION matrix

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

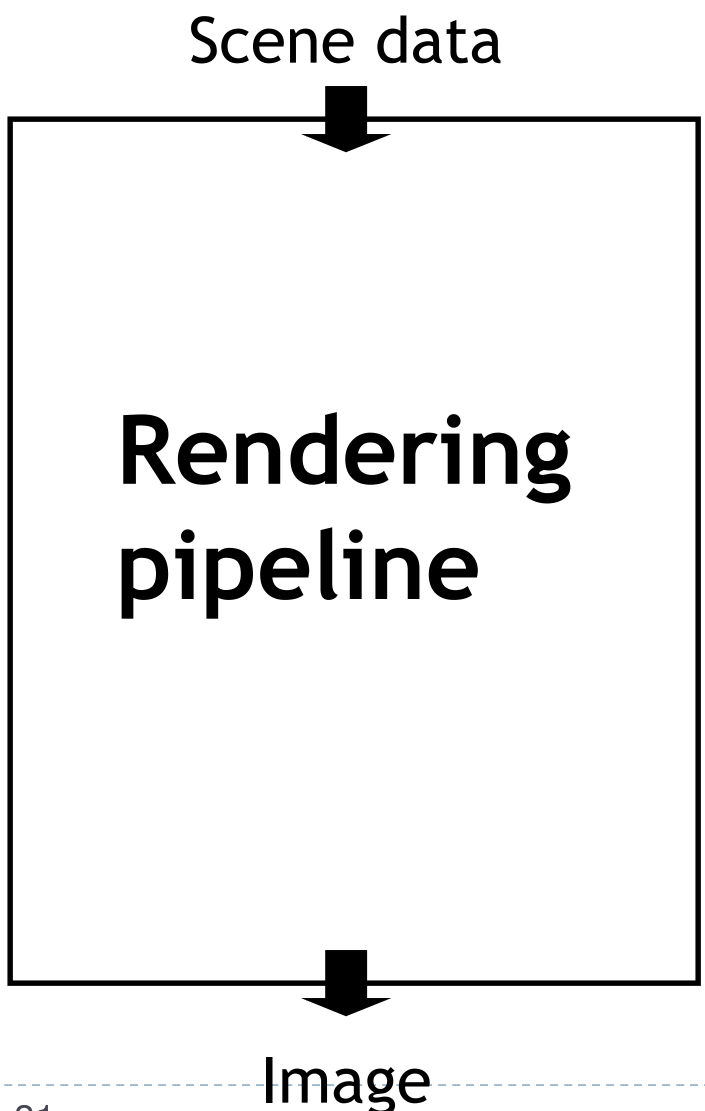
Complete Vertex Transformation in OpenGL

- GL_MODELVIEW, $\mathbf{C^{-1}M}$
  - Defined by the programmer.
  - Think of the ModelView matrix as where you stand with the camera and the direction you point it.
- GL_PROJECTION, **P**
  - Utility routines to set it by specifying view volume: glFrustum(), gluPerspective(), glOrtho()
  - Think of the projection matrix as describing the attributes of your camera, such as field of view, focal length, etc.
- Viewport, **D**
  - Specify implicitly via glViewport()
  - No direct access with equivalent to GL_MODELVIEW or GL_PROJECTION

Lecture Overview

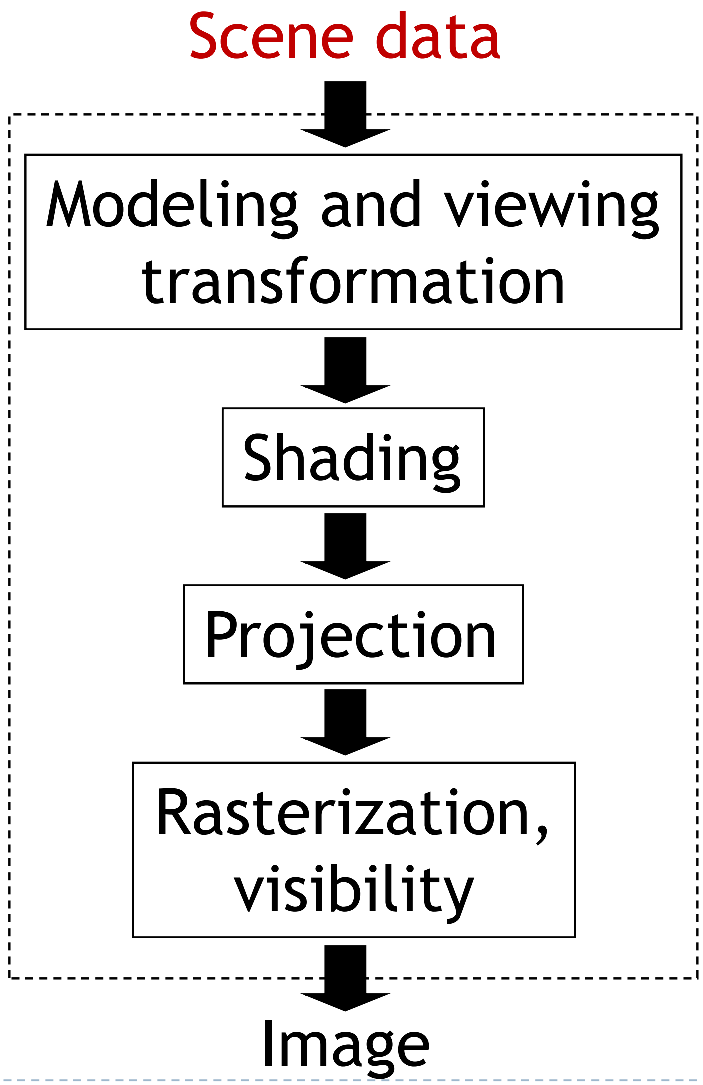
- View Volumes
- Vertex Transformation
- Rendering Pipeline
- Culling

Rendering Pipeline

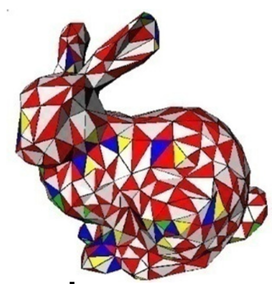


- Hardware and software which draws 3D scenes on the screen
- Consists of several stages
  - Simplified version here
- Most operations performed by specialized hardware (GPU)
- Access to hardware through low-level 3D API (OpenGL, DirectX)
- All scene data flows through the pipeline at least once for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

Rendering Pipeline



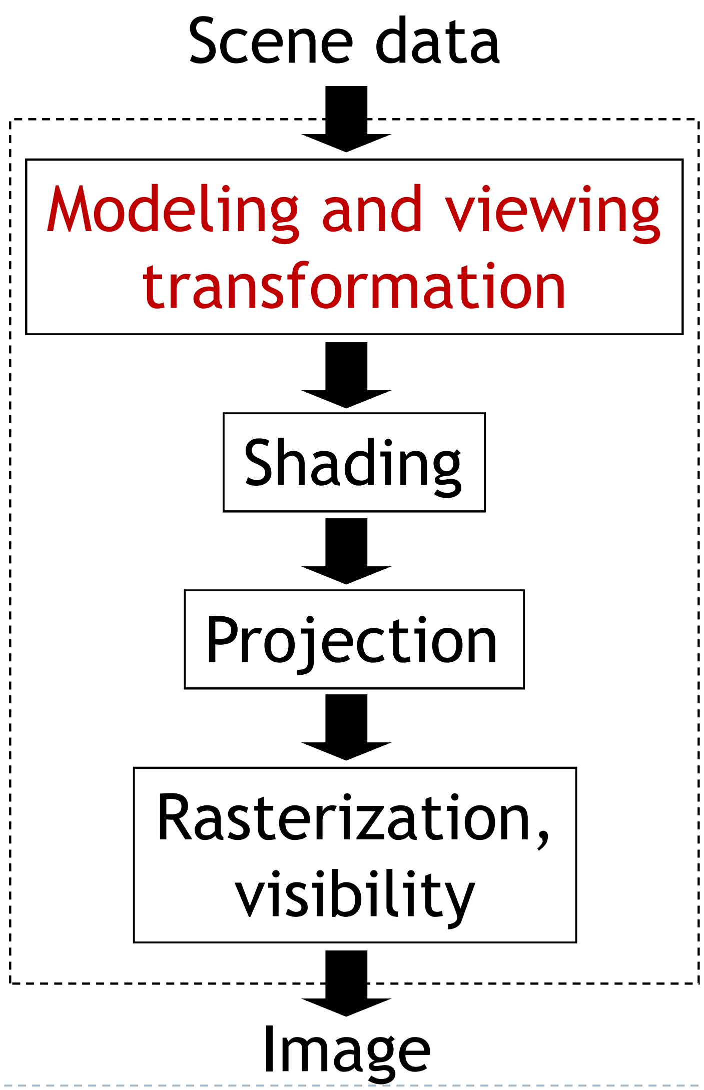
- Textures, lights, etc.
- Geometry
  - Vertices and how they are connected
  - Triangles, lines, points, triangle strips
  - Attributes such as color



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

UCSD

Rendering Pipeline

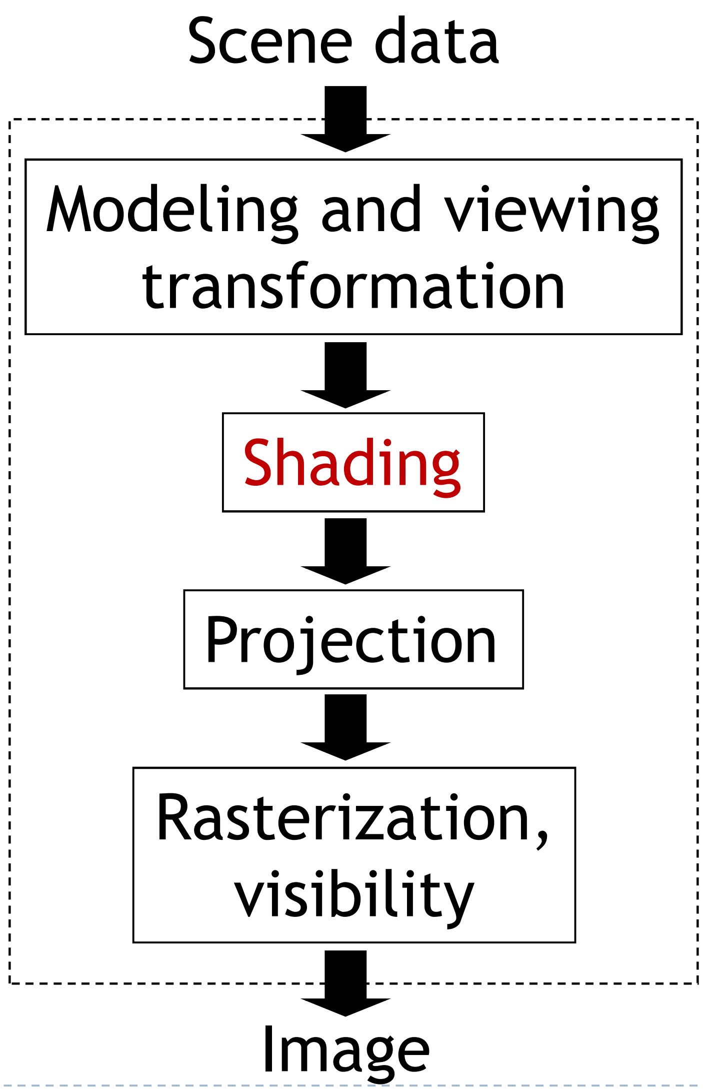


- Transform object to camera coordinates
- Specified by GL_MODELVIEW matrix in OpenGL
- User computes GL_MODELVIEW matrix as discussed

$$\mathbf{p}_{camera} = \mathbf{C}^{-1}\mathbf{M}\mathbf{p}_{object}$$

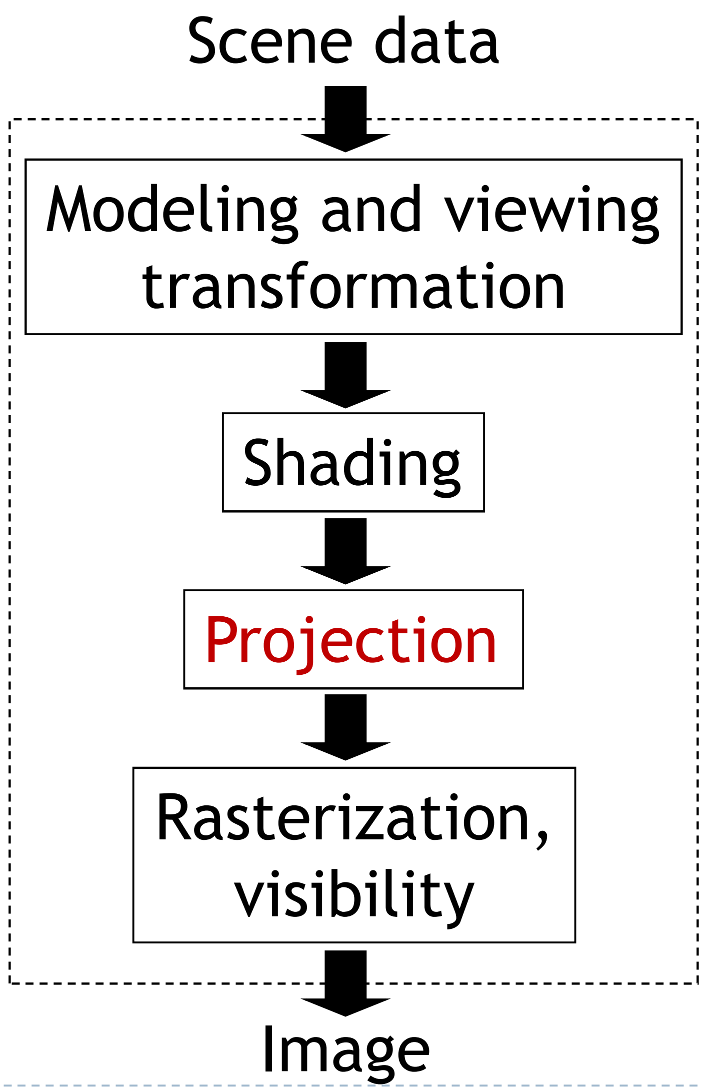
MODELVIEW matrix

Rendering Pipeline



- Look up light sources
- Compute color for each vertex

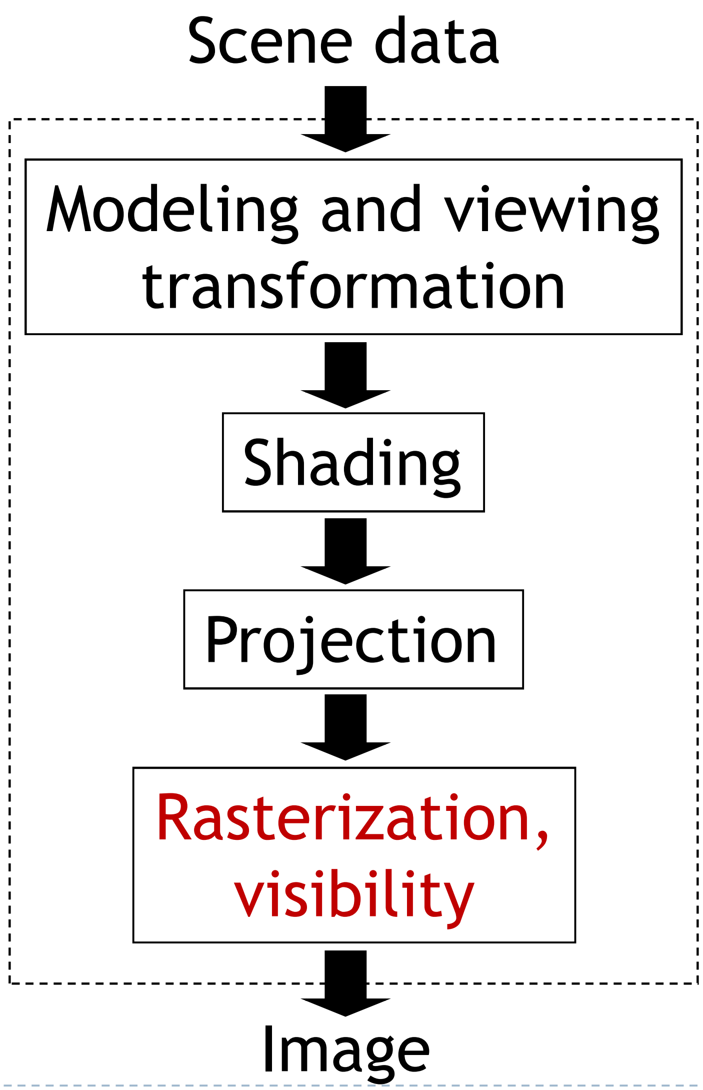
Rendering Pipeline



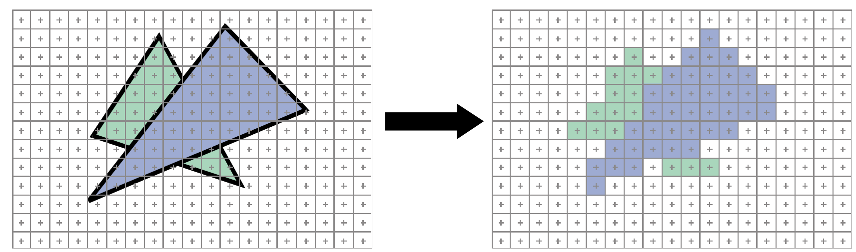
- Project 3D vertices to 2D image positions
- GL_PROJECTION matrix

UCSD

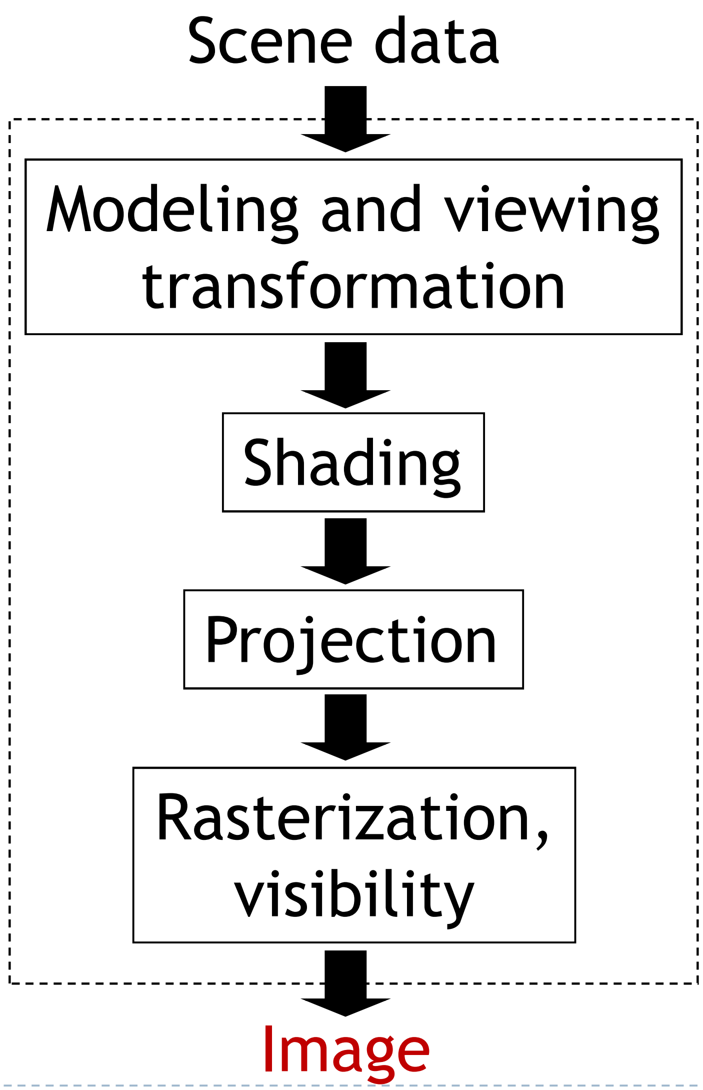
Rendering Pipeline



- Draw primitives (triangles, lines, etc.)
- Determine what is visible

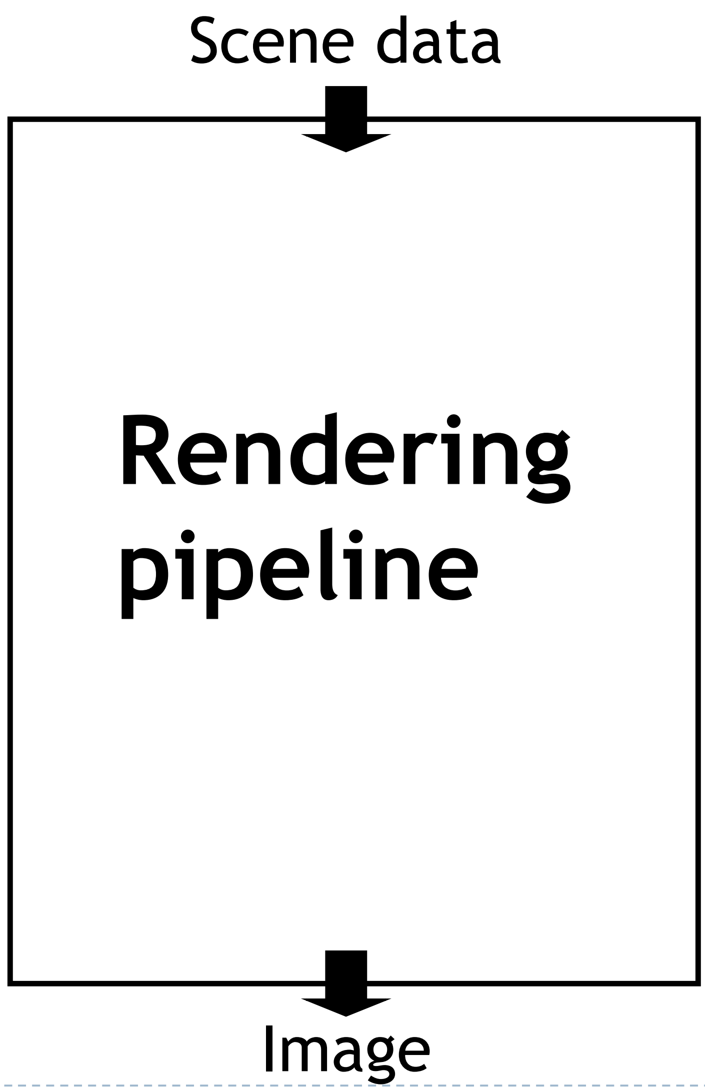


Rendering Pipeline



- Pixel colors

Rendering Engine



Rendering Engine:

- Additional software layer encapsulating low-level API
- Higher level functionality than OpenGL
- Platform independent
- Layered software architecture common in industry
  - Game engines
  - Graphics middleware

Lecture Overview

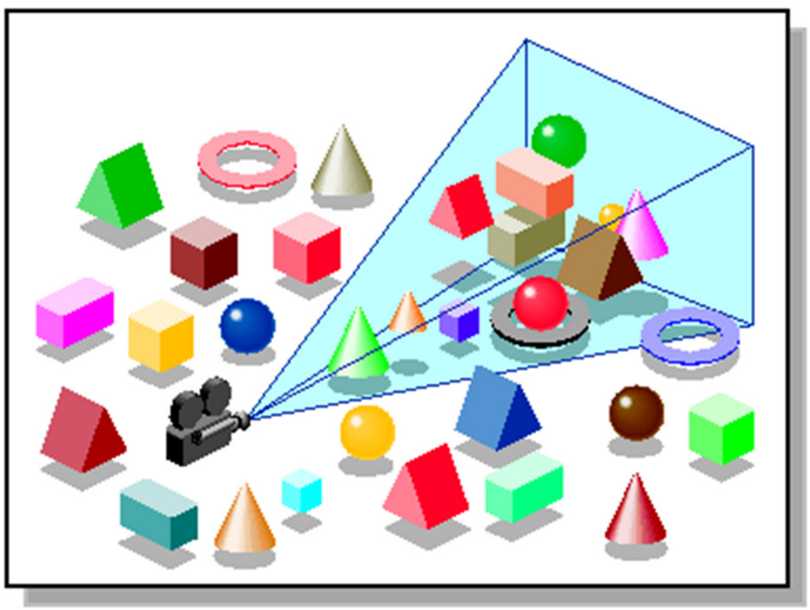
- View Volumes
- Vertex Transformation
- Rendering Pipeline
- Culling

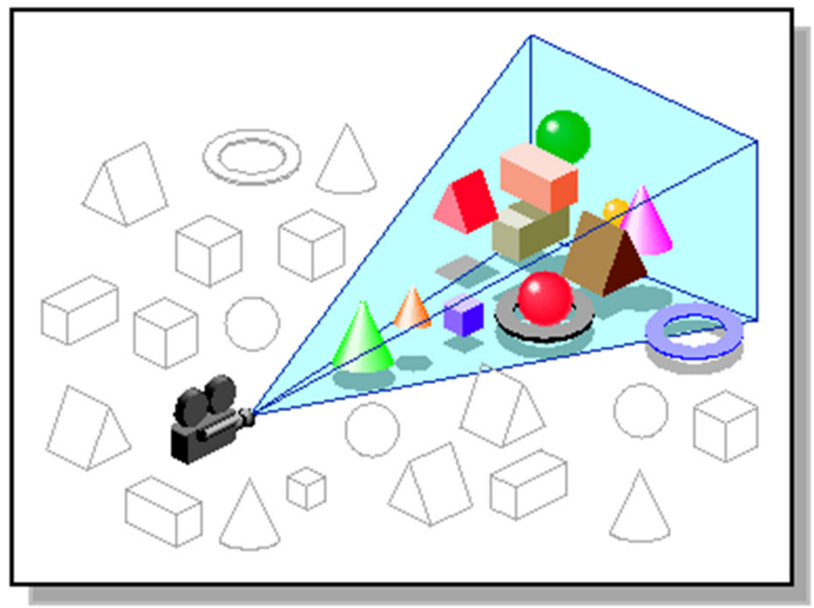
Culling

- Goal:
  Discard geometry that does not need to be drawn to speed up rendering
- Types of culling:
  - View frustum culling
  - Occlusion culling
  - Small object culling
  - Backface culling
  - Degenerate culling

View Frustum Culling

- Triangles outside of view frustum are off-screen
  - Done on canonical view volume





Images: SGI OpenGL Optimizer Programmer's Guide

Videos

- Rendering Optimizations - Frustum Culling
  - http://www.youtube.com/watch?v=kvVHp9wMAO8
- View Frustum Culling Demo
  - http://www.youtube.com/watch?v=bJrYTBGpwic

Bounding Box

- How to cull objects consisting of may polygons?
- Cull bounding box
  - Rectangular box, parallel to object space coordinate planes
  - Box is smallest box containing the entire object

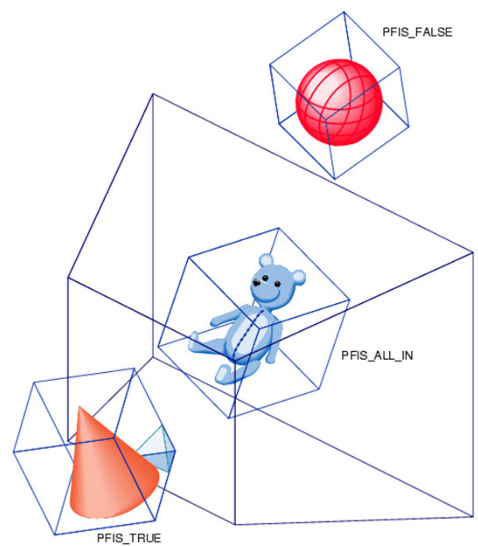
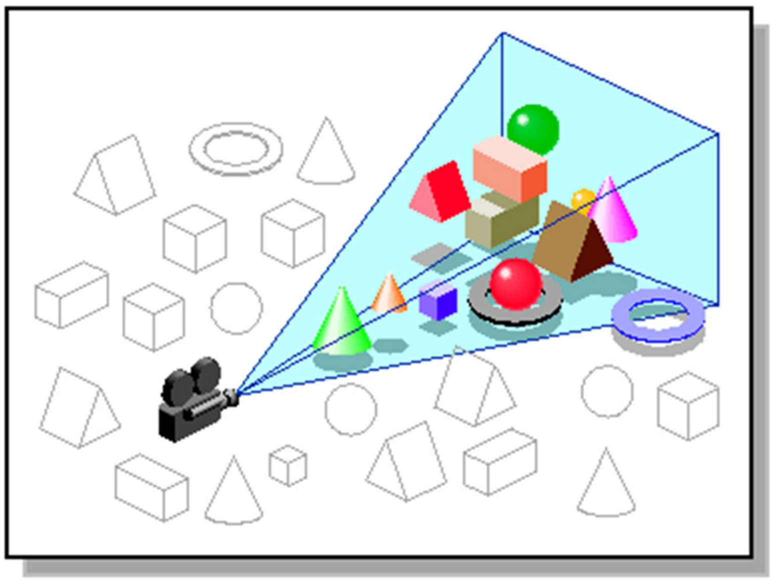
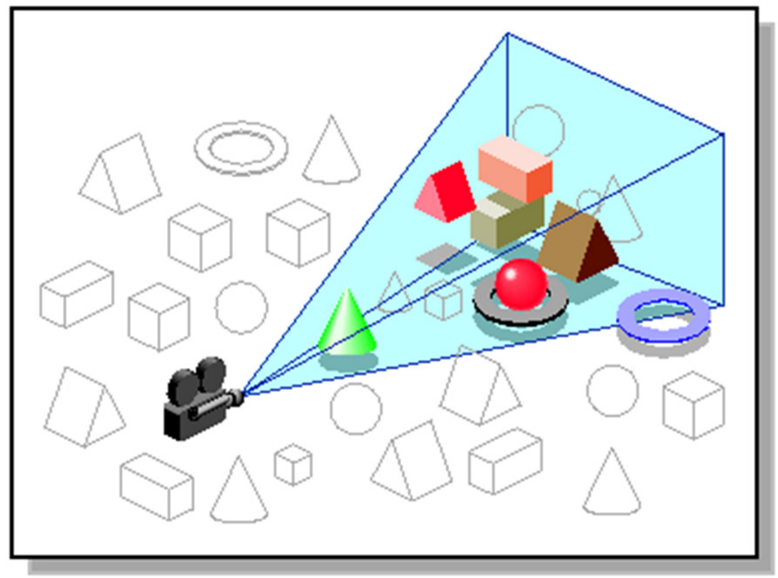


Image: SGI OpenGL Optimizer Programmer's Guide

Occlusion Culling

- Geometry hidden behind occluder cannot be seen
  - Many complex algorithms exist to identify occluded geometry





Images: SGI OpenGL Optimizer Programmer's Guide

Video

- Umbra 3 Occlusion Culling explained
  - http://www.youtube.com/watch?v=5h4QgDBwQhc

UCSD

Small Object Culling

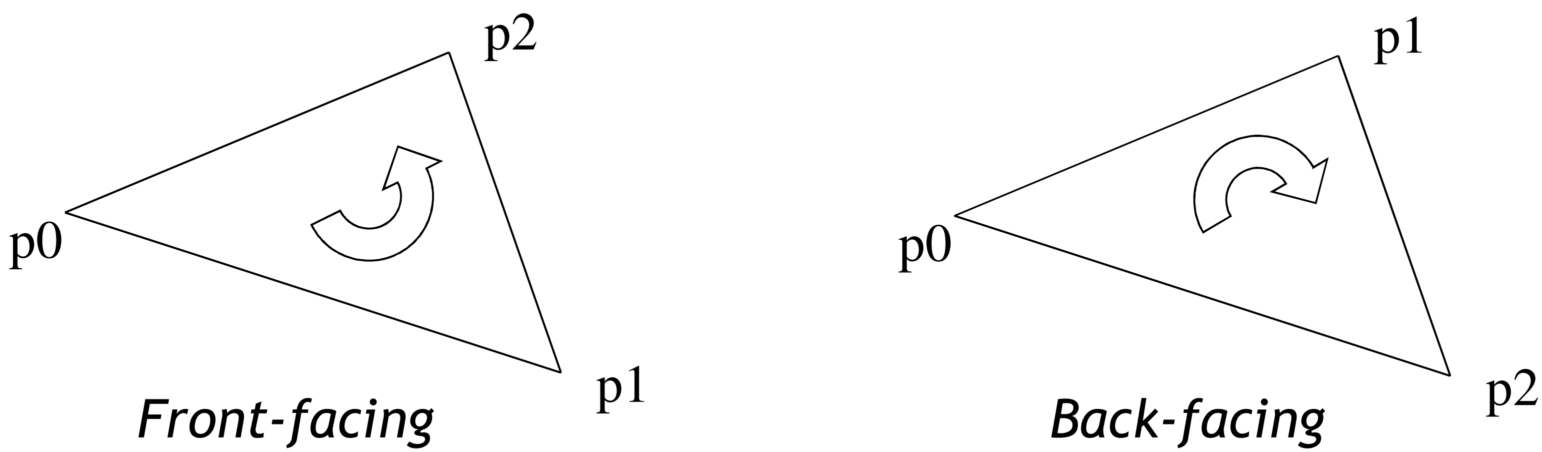
- Object projects to less than a specified size
  - Cull objects whose screen-space bounding box is less than a threshold number of pixels

Backface Culling

- Consider triangles as "one-sided", i.e., only visible from the "front"
- Closed objects
  - If the "back" of the triangle is facing the camera, it is not visible
  - Gain efficiency by not drawing it (culling)
  - Roughly 50% of triangles in a scene are back facing

Backface Culling

- Convention:
  Triangle is front facing if vertices are ordered counterclockwise



- OpenGL allows one- or two-sided triangles
  - One-sided triangles:
    glEnable(GL_CULL_FACE); glCullFace(GL_BACK)
  - Two-sided triangles (no backface culling):
    glDisable(GL_CULL_FACE)

Backface Culling

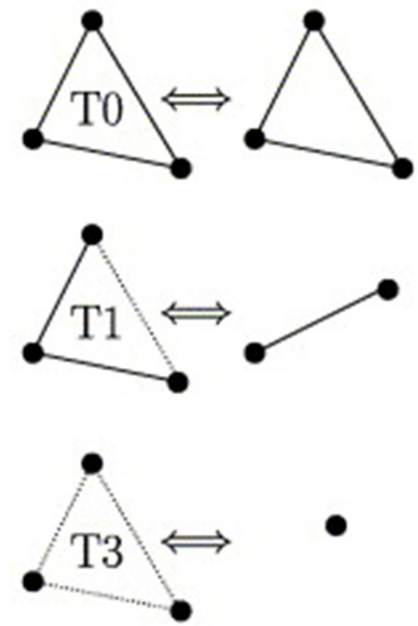
- Compute triangle normal after projection (homogeneous division)

$$\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)$$

- Third component of $\mathbf{n}$ negative: front-facing, otherwise back-facing
  - Remember: projection matrix is such that homogeneous division flips sign of third component

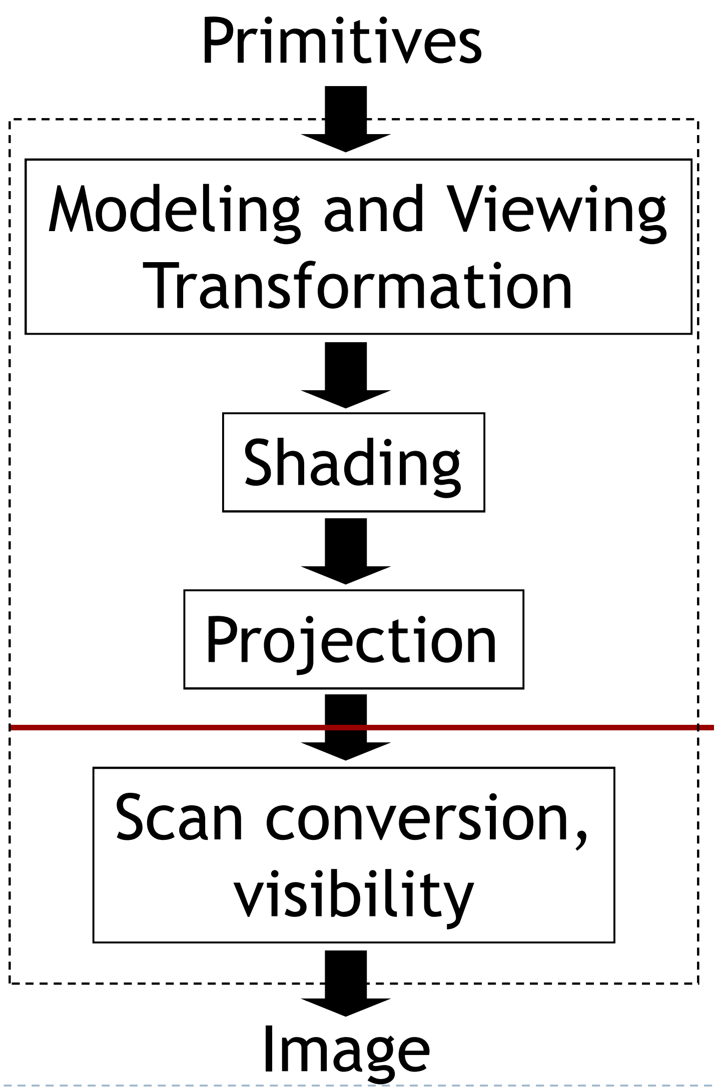
Degenerate Culling

- Degenerate triangle has no area
  - Vertices lie in a straight line
  - Vertices at the exact same place
  - Normal $\mathbf{n}$=0



Source: Computer Methods in Applied Mechanics and Engineering, Volume 194, Issues 48–49

Rendering Pipeline



Culling, Clipping

- Discard geometry that will not be visible

UCSD