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
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

- Scene data
  - 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

Scene data

Modeling and viewing transformation

Shading

Projection

Rasterization, visibility

Image

- Transform object to camera coordinates
- Specified by \( \text{GL}_\text{MODELVIEW} \) matrix in OpenGL
- User computes \( \text{GL}_\text{MODELVIEW} \) matrix as discussed

\[
p_{\text{camera}} = C^{-1} M p_{\text{object}}
\]

MODELVIEW matrix
Rendering Pipeline

Scene data

Modeling and viewing transformation

Shading

Projection

Rasterization, visibility

Image

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

Scene data

Modeling and viewing transformation

Shading

Projection

Rasterization, visibility

Image

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

Scene data

Modeling and viewing transformation

Shading

Projection

Rasterization, visibility

Image

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

Scene data

Modeling and viewing transformation

Shading

Projection

Rasterization, visibility

Image

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
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

\[
\frac{y'}{d} = \frac{y_1}{z_1}
\]

\[
y' = \frac{y_1d}{z_1}
\]

\[
x' = \frac{x_1d}{z_1}
\]

\[
z' = d
\]
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

\[
x' = \frac{x_1d}{z_1}
\]

\[
y' = \frac{y_1d}{z_1}
\]

\[
z' = d
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/d & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
= 
\begin{bmatrix}
x \\
y \\
z \\
d
\end{bmatrix}
\]

Projection matrix Homogeneous division
Perspective Projection

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

**Orthographic view volume**
- Camera coordinates

**World coordinates**
Perspective View Volume

General view volume

- 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

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

- \( \tan\left(\frac{\text{FOV}}{2}\right) = \frac{\text{top}}{\text{near}} \)
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 \([-1..1] \times [-1..1] \times [-1..1]\) 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

\[
P_{\text{persp}}(\text{left}, \text{right}, \text{top}, \text{bottom}, \text{near}, \text{far}) =
\]
\[
\begin{bmatrix}
\frac{2\text{near}}{\text{right}-\text{left}} & 0 & \frac{\text{right}+\text{left}}{\text{right}-\text{left}} & 0 \\
0 & \frac{2\text{near}}{\text{top}-\text{bottom}} & \frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & 0 \\
0 & 0 & \frac{-(\text{far}+\text{near})}{\text{far}-\text{near}} & \frac{-2\text{far} \cdot \text{near}}{\text{far}-\text{near}} \\
0 & 0 & -1 & 0
\end{bmatrix}
\]
Perspective Projection Matrix

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

\[
P_{\text{persp}}(\text{FOV}, \text{aspect}, \text{near}, \text{far}) = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  \frac{\text{aspect} \cdot \tan(\text{FOV}/2)}{\text{near} - \text{far}} & 1 & 0 & \frac{2 \cdot \text{near} \cdot \text{far}}{\text{near} - \text{far}} \\
  0 & \frac{\tan(\text{FOV}/2)}{\text{near} - \text{far}} & 1 & \frac{\text{near} + \text{far}}{\text{near} - \text{far}} \\
  0 & 0 & 0 & -1 \\
\end{bmatrix}
\]
Orthographic Projection Matrix

\[
P_{\text{ortho}}(\text{right}, \text{left}, \text{top}, \text{bottom}, \text{near}, \text{far}) = \begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} + \text{left}}{\text{right} - \text{left}} \\
0 & \frac{2}{\text{top} - \text{bottom}} & 0 & -\frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} \\
0 & 0 & \frac{2}{\text{far} - \text{near}} & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
P_{\text{ortho}}(\text{width}, \text{height}, \text{near}, \text{far}) = \begin{bmatrix}
\frac{2}{\text{width}} & 0 & 0 & 0 \\
0 & \frac{2}{\text{height}} & 0 & 0 \\
0 & 0 & \frac{2}{\text{far} - \text{near}} & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Viewport Transformation

- After applying projection matrix, scene points are in *normalized viewing coordinates*
- Per definition range $[-1..1] \times [-1..1] \times [-1..1]$
- Normalized viewing coordinates can be mapped to image (=pixel=frame buffer) coordinates
- Range depends on window (view port) size: $[x_0\ldots x_1] \times [y_0\ldots y_1]$
- **Scale and translation required:**

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

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

\[ p' = DPC^{-1}M_{\text{Object space}} p \]

- **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:

\[ p' = DPC^{-1}M_p \]

- **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:

\[ p' = DPC^{-1}Mp \]

- **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:

\[
p' = DPC^{-1}Mbp
\]

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

Object space

World space

Camera space

Canonical view volume
The Complete Transform

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

\[
p' = \text{DPC}^{-1}\text{M}p
\]

- **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:
  \[ p' = DPC^{-1}Mp \]

Pixel coordinates: \[ \frac{x'}{w'} \quad \frac{y'}{w'} \]

- **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:

\[ p' = DPC^{-1}Mp \]

- **M**: Object-to-world matrix
- **C**: camera matrix
- **P**: projection matrix
- **D**: viewport matrix
The Complete Transform in OpenGL

- **GL_MODELVIEW, \( C^{-1}M \)**
  - 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)