CSE167
Introduction to
Computer Graphics

Matthias Zwicker
University of California, San Diego
Spring 2007

Project 1
- Due tomorrow
- Present during lab session

Today
- Review
- Rendering pipeline
- Projections
- View volumes, clipping
- Viewport transformation

Review
- Change of coordinates
- Object, world, camera coordinates
- Camera matrix

Change of coordinates

Coordinates of $xyzw$ frame w.r.t. $uvwq$ frame

$x = \begin{bmatrix} x_x \\ x_y \\ x_z \\ 0 \end{bmatrix} 
\quad y = \begin{bmatrix} y_x \\ y_y \\ y_z \\ 0 \end{bmatrix} 
\quad z = \begin{bmatrix} z_x \\ z_y \\ z_z \\ 0 \end{bmatrix} 
\quad o = \begin{bmatrix} o_x \\ o_y \\ o_z \\ 1 \end{bmatrix}$
### Change of coordinates

Given coordinates of basis \( x, y, z, o \) with respect to new frame \( u, v, w, q \)

\[
\begin{align*}
\mathbf{x} &= \begin{bmatrix} x_0^* \ y_0^* \ z_0^* \ o_0^* \end{bmatrix} \\
\mathbf{y} &= \begin{bmatrix} x_0^* \ y_0^* \ z_0^* \ o_0^* \end{bmatrix} \\
\mathbf{z} &= \begin{bmatrix} x_0^* \ y_0^* \ z_0^* \ o_0^* \end{bmatrix} \\
\mathbf{o} &= \begin{bmatrix} x_0^* \ y_0^* \ z_0^* \ o_0^* \end{bmatrix}
\end{align*}
\]

- Coordinates of point \( \mathbf{p}_{uvwq} \) w.r.t. new frame

\[
\mathbf{p}_{uvwq} = \begin{bmatrix} x \ y \ z \ o \end{bmatrix} = \begin{bmatrix} x_0 \ y_0 \ z_0 \ o_0 \end{bmatrix}
\]

- Columns of transformation matrix are new coordinates \((uvw)\) of old basis vectors \((xyz)\)

### Object, world, camera coords.

- **Object-to-world transformation matrix** \( \mathbf{M} \)
  - Columns contain basis vectors of object space in world coordinates

- **Camera-to-world transformation matrix** \( \mathbf{C} \)
  - Columns contain basis vectors of camera space in world coordinates

- **Object-to-camera**
  \[
  \mathbf{p}_{\text{camera}} = \mathbf{C}^{-1} \mathbf{M} \mathbf{p}_{\text{object}}
  \]
Camera matrix

- **z-axis**
  \[ z_c = \frac{e - d}{\|e - d\|} \]
- **x-axis**
  \[ x_c = \frac{up \times z_c}{\|up \times z_c\|} \]
- **y-axis**
  \[ y_c = z_c \times x_c \]
- **Camera matrix**
  \[
  C = \begin{bmatrix}
  x_c & y_c & z_c & e \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]

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
- Today: how to project them into 2D

Questions?

Today

- Review
- **Rendering pipeline**
- Projections
- View volumes, clipping
- Viewport transformation

Rendering pipeline

- Hardware & software that draws 3D scenes on the screen
- Consists of several stages
  - Simplified version here
- Most operations performed by specialized hardware (GPU)
- Access to hardware through 3D API (DirectX, OpenGL)
- State machine

Rendering pipeline

- 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

- Primitives

  Rendering pipeline

  - Modeling and viewing transformation
  - Shading
  - Projection
  - Rasterization, visibility

  Image

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

  $\mathbf{P}_{\text{camera}} = \mathbf{C} \cdot \mathbf{M}_{\text{object}}$ [MODELVIEW matrix]

  Questions?

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

  Rendering pipeline

  - Primitives

  Rendering pipeline

  - Modeling and viewing transformation
  - Shading
  - Projection
  - Rasterization, visibility

  Image

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

  Rendering pipeline

  - Primitives

  Rendering pipeline

  - Modeling and viewing transformation
  - Shading
  - Projection
  - Rasterization, visibility

  Image

  - Pixel colors
Today

• Review
• Rendering pipeline
• Projections
• View volumes, clipping
• Viewport transformation

Projections

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

Orthographic projection

• Simply ignore z-coordinate
• Use camera space xy coordinates as image coordinates

Orthographic projection

• Project points to x-y plane along parallel lines
• Graphical illustrations, architecture

Perspective projection

• Most common for computer graphics
• Simplified model of human eye, or camera lens (pinhole camera)
• Things farther away seem smaller
• Discovery/formalization attributed to Filippo Brunelleschi in the early 1400’s

Perspective projection

• Project along rays that converge in center of projection

Perspective projection

Parallel lines no longer parallel, converge at one point

Earliest example
La Trinità (1427) by Masaccio
Perspective projection
The math: simplified case
\[ y' = \frac{y'd}{z_1} \]
\[ z' = d \]

• Can express this using homogeneous coordinates, 4x4 matrices

Perspective projection
The math: simplified case
\[ y' = \frac{y'd}{z_1} \]
\[ z' = d \]

Homogeneous division

Perspective projection
The math: simplified case
\[ y' = \frac{y'd}{z_1} \]
\[ z' = d \]

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

• Using projection matrix, homogeneous division seems more complicated than just multiplying all coordinates by \( d/z \), so why do it?
• Will allow us to
  - handle different types of projections in a unified way
  - define arbitrary view volumes

Perspective projection
• Some deep math behind this, 3D projective space

In practice
• Use homogeneous matrices normally
• Modeling & viewing transformations use affine matrices
  - points keep \( w=1 \)
  - no need to divide by \( w \) when doing modeling operations or transforming into camera space
• Projection transform uses perspective matrices
  - \( w \) not always \( 1 \)
  - Divide by \( w \) (perspective division, homogeneous division) after performing projection transform
• Graphics hardware does this

Realistic image formation
• More than perspective projection
Realistic image formation

• More than perspective projection
• Lens effects

Focus, depth of field  
Fish-eye lens

Realistic image formation

• Often too complicated for hardware rendering pipeline

Today

• Review
• Rendering pipeline
• Projections
• View volumes, clipping
• Viewport transformation

View volumes

• Define 3D volume seen by camera

Perspective view volume

Orthographic view volume

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

Symmetric view volume

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

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

\[
\tan(\text{FOV}/2) = \frac{\text{top} - \text{near}}{\text{near}}
\]
**Orthographic view volume**

- Parametrized by 6 parameters
  - Right, left, top, bottom, near, far
- If symmetric
  - Width, height, near, far

**Clipping**

- Need to identify objects outside view volume
- Avoid division by zero
- Efficiency, don’t draw objects outside view volume
- Performed by hardware
- Hardware always clips to *canonic view volume*
- Cube [-1..1]x[-1..1]x[-1..1] centered at origin
- Need to transform desired view frustum to canonic view frustum

**Canonic view volume**

- Projection matrix is set such that
  - User defined view volume is transformed into canonic view volume, i.e., cube [-1..1]x[-1..1]x[-1..1]
  - Multiplying vertices of view volume by projection matrix and performing homogeneous divide yields canonic view volume
- Perspective and orthographic projection are treated exactly the same way

**Projection matrix**

- General view frustum
  
  \[
  P_{\text{proj}}(\text{left}, \text{right}, \text{top}, \text{bottom}, \text{near}, \text{far}) = \begin{bmatrix}
  \frac{2\text{near}}{\text{right}-\text{left}} & 0 & \frac{\text{right}+\text{left}}{\text{right}-\text{left}} & 0 \\
  0 & \frac{2\text{near}}{\text{top}-\text{bottom}} & \frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & 0 \\
  0 & 0 & \frac{\text{far}-\text{near}}{\text{far}-\text{near}} & 1 \\
  0 & 0 & 0 & 0
  \end{bmatrix}
  \]
Perspective projection matrix

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

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

Orthographic projection matrix

- After applying projection matrix, image points are in normalized view coordinates
  - Per definition range [-1..1] x [-1..1]
- Map points to image (i.e., pixel) coordinates
  - User defined range [x0...x1] x [y0...y1]
- Scale and translation

\[
D(\text{x}, \text{y}, \text{z}) = \begin{bmatrix}
(x - x_c)^2 & 0 & 0 & (x - x_c)^2 \\
0 & (y - y_c)^2 & 0 & (y - y_c)^2 \\
0 & 0 & \frac{1}{z^2} & \frac{1}{z^2} \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Questions?

Today

- Review
- Rendering pipeline
- Projections
- View volumes
- Viewport transformation

Viewport transformation

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix \( \mathbf{M} \), camera matrix \( \mathbf{C} \), projection matrix \( \mathbf{P} \), viewport matrix \( \mathbf{D} \)

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

Object space
OpenGL
• Object-to-world matrix $M$, camera matrix $C$
  projection matrix $P$, viewport matrix $D$

OpenGL GL_MODELVIEW matrix

$$p' = DPC^{-1}M p$$
OpenGL GL_PROJECTION matrix

OpenGL
• GL_MODELVIEW, $C^{-1}M$
  - Up to you to define
• GL_PROJECTION, $P$
  - Utility routines to set it by specifying view volume: glFrustum(), glPerspective(), glOrtho()
  - Do not use utility functions for project 2
  - You will implement a software renderer in project 3, which will not rely on any OpenGL
• Viewport, $D$
  - Specify implicitly via glViewport()
  - No direct access

Next time
• Drawing (rasterization)
• Visibility (z-buffering)