CSE167
Introduction to Computer Graphics

Matthias Zwicker
University of California, San Diego
Spring 2008

Project 1
• Due tomorrow
• Present during lab session
  - Starting 1:30p
  - EBU3B 240, look in different lab if you can’t find Neel
  - List your name on whiteboard no later than 2pm
In addition
  Zip-up your code and email to Neel

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

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

Change of coordinates

\[ p_{xyz} = p_{x} + p_{y} + p_{z} + o \]

Coordinates of \( p \) w.r.t. uvwq frame?
Change of coordinates

Given coordinates of basis \( x, y, z, o \) with respect to new frame \( u, v, w, q \):
\[
\begin{bmatrix}
x \\
y \\
z \\
o
\end{bmatrix}
= \begin{bmatrix}
x_0 \\
y_0 \\
z_0 \\
o_0
\end{bmatrix} + \begin{bmatrix}
x_1 \\
y_1 \\
z_1 \\
o_1
\end{bmatrix} 
\]

Coordinates of point \( p_{xyz} \) w.r.t. new frame:
\[
p_{xyz} = \begin{bmatrix}
x_0 \\
y_0 \\
z_0 \\
o_0
\end{bmatrix} + \begin{bmatrix}
x_1 \\
y_1 \\
z_1 \\
o_1
\end{bmatrix} \begin{bmatrix}
x \\
y \\
z \\
o
\end{bmatrix}
\]

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

Object, world, camera coords.

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

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

Object-to-camera
\[
P_{\text{camera}} = C^{-1}M_{\text{object}}
\]
Camera-to-world matrix

“Camera matrix”
- Construct from center of projection $e$, look at $d$, up-vector

```
<table>
<thead>
<tr>
<th>Camera coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_c</td>
</tr>
<tr>
<td>y_c</td>
</tr>
<tr>
<td>z_c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>World coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>
```

Camera matrix
- $z$-axis
  \[
  z_c = \frac{e - d}{\|e - d\|}
  \]
- $x$-axis
  \[
  x_c = \frac{\text{up} \times z_c}{\|\text{up} \times z_c\|}
  \]
- $y$-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

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

Questions?

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 low-level 3D API (DirectX, OpenGL)
- All scene data flows through the pipeline at least once for each frame
**Rendering engine**

- Additional software layer encapsulating low-level API
- More functionality
- Platform independent
- Layered software architecture common in industry

**Rendering pipeline**

1. Scene data
2. Modeling and viewing transformation
   - Transform object to camera coordinates
   - Specified by GL_MODELVIEW matrix in OpenGL
   - User computes GL_MODELVIEW matrix as discussed
   - \( \text{Pos} = \text{CM} \cdot \text{Pos}_{\text{Object}} \)
3. Shading
4. Projection
5. Rasterization, visibility
6. Image

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

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

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

- Project 3D vertices to 2D image positions
- GL_PROJECTION matrix
- This lecture
**Rendering pipeline**

- Scene data
- 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

**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

![Diagram of 3D scene projected onto a 2D image plane with rays converging at the center of projection.]

Parallel lines no longer parallel, converge at one point

Earliest example
La Trinità (1427) by Masaccio

---

**The math: simplified case**

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

\[ z' = d \]

- Can express this using homogeneous coordinates, 4x4 matrices

---

**The math: simplified case**

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

Projection matrix
Homogeneous division

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

- Some deep math behind this,
  - Projective transformations, 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

**Today**

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

**View volumes**

- Define 3D volume seen by camera
  - Perspective view volume
    - Camera coordinates
  - Orthographic view volume
    - Camera coordinates
  - World 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

**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

Camera coordinates

Projection matrix

Clipping

Perspective projection matrix

• General view frustum

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

Perspective projection matrix

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

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

Orthographic projection matrix

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

Questions?

Today

• Review
• Rendering pipeline
• Projections
• View volumes
• Viewport transformation
### Viewport transformation

- 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

\[
D(x_0, y_0, x_1, y_1) = \begin{bmatrix}
  \frac{x_1 - x_0}{2} & 0 & 0 & 0 \\
  0 & \frac{y_1 - y_0}{2} & 0 & 0 \\
  0 & 0 & \frac{1}{2} & 0 \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

### The complete transform

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

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

#### Object space

#### World space

#### Camera space

---

<table>
<thead>
<tr>
<th>Object space</th>
<th>World space</th>
<th>Camera space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-to-world matrix ( O ), camera matrix ( C ), projection matrix ( P ), viewport matrix ( D )</td>
<td>Mapping a 3D point in object coordinates to pixel coordinates</td>
<td>Canonic view volume</td>
</tr>
</tbody>
</table>
The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix $M$, camera matrix $C$
- Projection matrix $P$, viewport matrix $D$

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

Questions?

OpenGL

- Object-to-world matrix $M$, camera matrix $C$
- Projection matrix $P$, viewport matrix $D$

OpenGL GL_MODELVIEW matrix

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

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)