CSE167: Introduction to Computer Graphics

Lecture #5

Jürgen Schulze
University of California, San Diego
Fall 2008
Projects

- Project 1 due Friday October 10
- Project 2 due Friday October 17
- Robert to announce lab on Monday instead of Wednesday
Today

• Review

Drawing triangles
• Barycentric coordinates
• Culling, clipping
• Rasterization
• Visibility
Review

- Rendering pipeline
- Perspective projection
Rendering pipeline

Scene data

Modeling and viewing transformation

Shading

Projection

Rasterization, visibility

Image

- Simplified version
- Geometry (triangles) is processed one-by-one
- Most operations performed by specialized hardware (GPU)
  - Heavily parallel
- Access to hardware through low-level 3D API (OpenGL, DirectX)
Rendering pipeline

Scene data → Modeling and viewing transformation → Shading → Projection → Rasterization, visibility → Image

- Lectures 2 and 3
- Lectures 6-8
- Lecture 4 (last time)
- Lecture 5 (today)
Base code architecture

Primitives RE167
- Modeling and viewing transformation
- Shading
- Projection
- Rasterization, visibility
- Image

Dynamic link library (.dll)

BasicApp
- Application program
  - No OpenGL calls
  - Independent of rendering backend
  - Can easily change rendering backend (OpenGL, DirectX, software renderer)

Executable (.exe)
Rendering pipeline

- Alternatives?
Rendering pipeline

- Alternatives?

- Rendering pipeline
  - Object order: primitive by primitive

- Ray tracing (CSE168)
  - Image order: pixel by pixel
Perspective projection

- Given 3D points (vertices) in camera coordinates, determine corresponding image coordinates
- Things farther away seem smaller
- Simplified model of human eye, or camera lens (pinhole camera)
Perspective projection

The math: simplified case

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

\[ z' = d \]
**Perspective projection**

**The math: simplified case**

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

\[ z' = d \]

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

**Projection matrix** \hspace{1cm} \text{Homogeneous division}
View volumes

• Define 3D volume seen by camera

Perspective view volume
Camera coordinates

Orthographic view volume
Camera coordinates

World coordinates

World coordinates
• Defines 3D volume that is mapped to image
• Left, right, top, bottom boundaries
• Near, far clipping planes
  - Avoid numerical problems during rendering, like divide by zero
  - Avoid low precision for distant objects
Perspective view volume

Symmetric view volume

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

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

\[
\tan\left(\frac{\text{FOV}}{2}\right) = \frac{\text{top}}{\text{near}}
\]
Orthographic view volume

- Parametrized by 6 parameters:
  - Right, left, top, bottom, near, far
- If symmetrical:
  - Width, height, near, far
Projection matrix

Camera coordinates

Projection matrix

Canonical view volume

Clipping
Perspective projection matrix

\[ P_{\text{persp}}(\text{left}, \text{right}, \text{top}, \text{bottom}, \text{near}, \text{far}) = \]

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

Sign change!
Perspective projection matrix

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

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

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

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

- After applying projection matrix, image points are in *normalized view coordinates*
  - Per definition range $[-1..1] \times [-1..1]$
- Map points to image (i.e., pixel) coordinates
  - User defined range $[x_0 \ldots x_1] \times [y_0 \ldots y_1]$
- Scale and translation

$$D(x_0, x_1, y_0, y_1) = \begin{bmatrix}
  \frac{(x_1 - x_0)}{2} & 0 & 0 & \frac{(x_0 + x_1)}{2} \\
  0 & \frac{(y_1 - y_0)}{2} & 0 & \frac{(y_0 + y_1)}{2} \\
  0 & 0 & \frac{1}{2} & \frac{1}{2} \\
  0 & 0 & 0 & 1
\end{bmatrix}$$
The complete transform

- 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 $\mathbf{D}$

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

Object space
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 $D$

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

Object space

World space
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 $D$

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

<table>
<thead>
<tr>
<th>Object space</th>
<th>World space</th>
<th>Camera space</th>
</tr>
</thead>
</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 $D$

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

Object space

World space

Camera space

Canonic view volume
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 $D$

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

- Object space
- World space
- Camera space
- Canonical view volume
- Image space
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 $D$

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

$$p' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$  
Pixel coordinates $x'/w'$, $y'/w'$
Questions?
Today

- Review

Drawing triangles

- Barycentric coordinates
- Culling, clipping
- Rasterization
- Visibility
Implicit 2D lines

• Given two 2D points \( a, b \)

• Define function \( f_{ab}(p) \) such that \( f_{ab}(p) = 0 \) if \( p \) lies on line defined by \( a, b \)
Implicit 2D lines

- Point $p$ lies on the line, if $p - a$ is perpendicular to normal of line

$$(a_y - b_y, b_x - a_x)$$

- Use dot product to determine if perpendicular

$$f_{ab}(p) = (a_y - b_y, b_x - a_x) \cdot (p_x - a_x, p_y - a_y)$$
Barycentric coordinates

- Coordinates for 2D plane defined by triangle vertices $a, b, c$

- Any point $p$ in the plane defined by $a, b, c$ is
  
  $$p = a + \beta (b - a) + \gamma (c - a)$$
  
  $$= (1 - \beta - \gamma) a + \beta b + \gamma c$$

- We define $\alpha = 1 - \beta - \gamma$
  
  $$\Rightarrow p = \alpha a + \beta b + \gamma c$$

- $\alpha, \beta, \gamma$ are called **barycentric** coordinates

- Works in 2D and in 3D

- If we imagine masses equal to $\alpha, \beta, \gamma$ attached to the vertices of the triangle, the center of mass (the barycenter) is then $p$. This is the origin of the term “barycentric” (introduced 1827 by Möbius)
Barycentric coordinates

\[ p = a + \beta(b - a) + \gamma(c - a) \]

- \( p \) is inside the triangle if \( 0 < \alpha, \beta, \gamma < 1 \)
Barycentric coordinates

- Problem: Given point $p$, find its barycentric coordinates
- Use equation for implicit lines

\[
\beta(p) = \frac{f_{ac}(p)}{f_{ac}(b)}
\]

\[
\gamma(p) = \frac{f_{ab}(p)}{f_{ab}(c)}
\]

\[
\alpha = 1 - \beta - \gamma
\]

\[
0 < \beta < 1
\]

- Division by zero if triangle is degenerate
Barycentric coordinates

• Points on triangle edges, e.g.,

\[ p(t) = a(1 - t) + bt \]

• Barycentric coordinates correspond to linear interpolation weights

\[ \alpha(p(t)) = 1 - t \]
\[ \beta(p(t)) = t \]

• Same for other edges

• Barycentric can be use to generalize linear interpolation to triangles
Barycentric interpolation

- Interpolate values across triangles, e.g., colors

\[ c(p) = \alpha(p)c_a + \beta(p)c_b + \gamma(p)c_c \]

- Linear interpolation on triangles
Questions?
Today

- Review

Drawing triangles

- Barycentric coordinates
- Culling, clipping
- Rasterization
- Visibility
Rendering pipeline

Primitives

Modeling and viewing transformation

Shading

Projection

Scan conversion, visibility

Image

Culling, clipping
- Discard geometry that should not be drawn
Culling

- Discard geometry that does not need to be drawn as early as possible

- Object-level frustum culling
  - Later in class

- Triangle culling
  - View frustum culling (clipping): outside view frustum
  - Backface culling: facing ‘away’ from the viewer
  - Degenerate culling: area=0
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: front side means vertices are ordered counterclockwise

Most renderers allow one- or two-sided triangles
- Two-sided triangles not backface culled
- Thin objects, non-closed objects
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 $n=0$
View frustum culling, clipping

- Triangles that intersect the faces of the view volume
  - Partly on screen, partly off
  - Do not rasterize the parts that are offscreen

- Traditional clipping
  - Split triangles that lie partly inside/outside viewing volume before homogeneous division
  - Avoid problems with division by zero

- Modern GPU implementations avoid clipping
Questions?
Today

- Review
- Drawing triangles
- Barycentric coordinates
- Culling, clipping
- Rasterization
- Visibility
Primitives

Modeling and viewing transformation

Shading

Projection

Rasterization, visibility

Image

- Scan conversion and rasterization are synonyms
- One of the main operations performed by GPU
- Draw triangles, lines, points (squares)
- Focus on triangles in this lecture
Rasterization
How many pixels can a modern graphics processor draw per second?
• How many pixels can a modern graphics processor draw per second?

• Rasterization is „hard-coded“, cannot be modified by the software

• NVidia Geforce 8800 GTX
  - Theoretical peak: up to 14 billion pixels per second
  - 600MHz clock frequency, 24 pixels per clock
  - Multiple of what the fastest CPU could do
Rasterization

- Many different algorithms
- Old style
  - Rasterize edges first
Rasterization

• Many different algorithms

• Old style
  - Rasterize edges first
  - Fill the spans (scan lines, scan conversion)
Rasterization

• Many different algorithms

• Old style
  
  - Rasterize edges first
  - Fill the spans (scan lines, scan conversion)
  - Requires clipping
  - Not preferred for hardware implementation today
Rasterization

• GPU rasterization today based on “homogeneous rasterization”

http://www.ece.unm.edu/course/ece595/docs/olano.pdf


• Does not require full clipping, does not perform homogeneous division at vertices

• Today in class
  - Simpler algorithm based on barycentric coordinates
  - Easy to implement
  - Technically, requires clipping
Rasterization

- Given vertices in pixel coordinates

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

- World space
- Camera space
- Clip space
- Image space

\[
p' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}
\]

Pixel coordinates: \[ x'/w' \quad y'/w' \]
Rasterization

• Simple algorithm

compute bbox
clip bbox to screen limits
for all pixels \([x,y]\) in bbox
    compute barycentric coordinates \alpha, \beta, \gamma
    if \(0<\alpha,\beta,\gamma<1\) // pixel in triangle
        image\[x,y\]=triangleColor

• Bounding box clipping trivial
• So far, we compute barycentric coordinates of many useless pixels

• Improvement?
Rasterization

Hierarchy

- If block of pixel is outside triangle, no need to test individual pixels
- Can have several levels, usually two-level
- Find right granularity, size of blocks, for best performance
Where is the center of a pixel?

- Depends on conventions
- With our viewport transformation from last lecture
  - 800 x 600 pixels ⇔ viewport coordinates are in [0...800]x[0...600]
  - Center of lower left pixel is 0.5, 0.5
  - Center of upper right pixel is 799.5, 599.5
**Shared edges**

- Each pixel needs to be rasterized exactly once
- Result image is independent of drawing order
- Rule: If pixel center exactly touches an edge or vertex
  - Fill pixel only if triangle extends to the right
Questions?
Visibility

- At each pixel, need to determine which triangle is visible
Painter’s algorithm

- Paint from back to front
- Every new pixel always paints over previous pixel
- Need to sort geometry according to depth
- May need to split triangles if they intersect

- Old style, before memory became cheap
Z-buffering

- Store z-value for each pixel
- Depth test
  - During rasterization, compare stored value to new value
  - Update pixel only if new value is smaller
    setpixel(int x, int y, color c, float z) if(z<zbuffer(x,y)) then
      zbuffer(x,y) = z
      color(x,y) = c

- z-buffer is dedicated memory reserved for GPU (graphics memory)
- Depth test is performed by GPU
Next time

- Perspective correct interpolation
- Color