CSE167
Introduction to Computer Graphics
Matthias Zwicker
University of California, San Diego
Spring 2008

Projects
- Project 2 due Friday April 18
- Lab session this Thursday TBA

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

Review
- Rendering pipeline
  - Perspective projection

Rendering pipeline

- 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 (DirectX, OpenGL)

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**

- **RE167**
- **BasicApp**
- **Dynamic link library (.dll)**
- **Executable (.exe)**

**Rendering pipeline**

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

**Perspective projection**

- **The math: simplified case**
  - \[ y' = \frac{y_1 d}{z_1} \]
  - \[ z' = d \]

**Perspective projection**

- **The math: simplified case**
  - \[ y' = \frac{y_1 d}{z_1} \]
  - \[ z' = d \]

**Projection matrix**

- \[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/d & 1
\end{bmatrix}
\]

- **Homogeneous division**
**View volumes**

- Define 3D volume seen by camera

**Perspective view volume**
- Camera coordinates
  
**Orthographic view volume**
- Camera coordinates

**Perspective view volume**

- Defines 3D volume that is mapped to image
- Left, right, top, bottom boundaries
- Near, far clipping planes
  - Avoid numerical problems during rendering
  - Divide by zero
  - Low precision for distant objects

**Orthographic view volume**

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

**Projection matrix**

- Camera coordinates
- Projection matrix
- Canonic view volume
- Clipping

**Perspective projection matrix**

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

Sign change!
**Perspective projection matrix**

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

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

**Orthographic projection matrix**

- 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 \([x_0...x_1] \times [y_0...y_1]\)
- Scale and translation

\[
D(x,y,x',y') =
\begin{bmatrix}
(x' - x) & 0 & 0 & (x' + x) \\
(y' - y) & 0 & 0 & (y' + y) \\
0 & 1/2 & 0 & 1/2 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

**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 \([x_0...x_1] \times [y_0...y_1]\)
- Scale and translation

\[
D(x,y,x',y') =
\begin{bmatrix}
(x' - x) & 0 & 0 & (x' + x) \\
(y' - y) & 0 & 0 & (y' + y) \\
0 & 1/2 & 0 & 1/2 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

**The complete transform**

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

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

**Object space**

**World space**

**Camera space**
Today

- Review

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

Questions?

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 \(a, b\)

\[
\begin{align*}
\text{if } f_{ab}(p) = 0 & \quad \text{on line defined by } a, b \\
f_{ab}(p) \neq 0 & \quad \text{otherwise}
\end{align*}
\]
**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
  \[p = a + \beta(b - a) + \gamma(c - a)\]
  \[= (1 - \beta - \gamma)a + \beta b + \gamma c\]
- where $\alpha = 1 - \beta - \gamma$
- $\alpha, \beta, \gamma$ are called barycentric coordinates
- Works in 2D and in 3D

**Barycentric coordinates**
- Problem: Given point $p$, find its barycentric coordinates
- Use equation for implicit lines
  \[\beta(p) = \frac{f_{bc}(p)}{f_{bc}(b)}\]
  \[\gamma(p) = \frac{f_{ab}(p)}{f_{ab}(c)}\]
- $\alpha = 1 - \beta - \gamma \quad 0 < \beta < 1$
- Division by zero if triangle is degenerate

**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**

**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 the 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)
  \[ n = (p_1 - p_0) \times (p_2 - p_0) \]
• Third component of \( n \) 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 off-screen
• 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

Rendering pipeline
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?
- Rasterization is „hard-coded“, cannot be modified per 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

### Many different algorithms

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

### 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

- 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

Rasterization

- So far, we compute barycentric coordinates of many useless pixels
- Improvement?

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
### Rasterization

**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

```c
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