Clipping and Rasterization

Christian Miller
CS 354 - Fall 2011
Pipeline again

Model & View Transform — Vertex Shading — Projection — Clipping — Screen Mapping
Backface removal

- Ignore all triangles facing away from camera
- Determined by winding on screen (CCW normally)
Clipping

- Cut off excess geometry outside your screen
- Don’t waste time generating pixels that won’t be shown
Line clipping

- Simplest thing to clip is a line
- Several algorithms for doing this
Line clipping

- Pipeline version: clip against every plane in order
- Unnecessary work for lines that disappear
Line clipping

• Faster version: Cohen-Sutherland
  • Find out what octants the vertices lie in
  • That tells you what planes to clip against
  • Find intersections and replace vertices with clipped ones directly
  • Only one complicated case
• Many other algorithms: Liang-Barsky, etc.
Polygon clipping

- Sutherland-Hodgman clipping algorithm:
  - For each clip plane:
    - Traverse each line segment in polygon:
      - Test against the plane
      - Add a vertex if line crosses the plane
Polygon clipping
Non-convex polygons

- Extra zero-area sections are generated
- Should be separate polygons
• To fix this, tesselate into convex sub-polygons
• Stretch NDC cube to screen size
• We have final triangle geometry on screen now!
Finally can start generating fragments
Drawing a line

- Fill in closest pixel to line, but only one wide
DDA ALGORITHM

// line goes from x1, y1 to x2, y2 (integers)

float m = (float)(y2 - y1) / (float)(x2 - x1);
float y = (float)y1;

for (int x = x1; x <= x2; x++)
{
    y += m;
    plot(x, round(y));
}
Steep slopes

- Must always pick coordinates such that $0 \leq m \leq 1$
DDA NOTES

- DDA is super-easy to code and understand
- There’s a floating-point add and round per pixel
  - That adds up over lots of pixels...
Bresenham’s line drawing algorithm

• If $0 \leq m \leq 1$, then we only need to decide whether to stay on the same scanline, or move up a pixel

• Can multiply through by line length to avoid any floating point
Bresenham’s line drawing algorithm

// point goes from x1, y1 to x2, y2 (integers)

int dx = x2 - x1;
int dy = y2 - y1;
int D = 2 * dy - dx;
int y = y1;

plot(x1, y1);

for (int x = x1 + 1; x <= x2; x++)
{
   if (D > 0)
   {
      y += 1;
      plot(x, y);
      D += 2 * (dy - dx);
   }
   else
   {
      plot(x, y);
      D += 2 * dy;
   }
}
Bresenham notes

- Needs 8 cases, just like DDA does
- No floating-point stuff, but does have a branch
  - This is sometimes worse on modern processors
- Still the standard... used everywhere
Several ways of doing this

- Most common is the scanline algorithm
- Also tiled rendering methods (later)
Scanline algorithm

- For every scanline:
  - Figure out start and end points on the triangle
  - Shade each pixel in between
- Must be careful about edges
Interpolation

• We often have some attribute at the vertices that we want to interpolate over the surface of a triangle
  • Color, normals, texture coordinates, etc.
Interpolation problem

- Screen coordinates don’t represent actual distance between vertices anymore, thanks to perspective
- Interpolating as if they do leads to nonsense results
Perspective-correct interpolation

- Attributes don’t interpolate linearly in screen space, but $1/w$ and attribute$/w$ do
- Interpolate these, divide by $1/w$ after (per pixel)
Pixel shading

- Once we’ve generated fragments for each triangle, we shade them to determine what color they are
  - Almost the entire second half of the class will be devoted to this step
- We’ll move onto merging for now...
Depth buffer

- Store depth values in a buffer (just like color)
- When you try to add a fragment to the buffer, check to see if it’s in front of the current pixel first
Depth resolution

- Remember: Those Z values are converted to ints
- Resolution is nonlinear thanks to perspective
- Gets much worse as depth approaches the far plane
• If you render two coplanar triangles, numerical inaccuracy causes unequal Z values

• Can lead to noise that changes rapidly as you move
Blending

- Transparency is handled with alpha values
  - \([0, 1]\) where 0 = invisible, 1 = opaque
- Apply a blending function to merge new fragment into color buffer
**Blending functions**

- Determine new color and alpha values, given those of current pixel and new fragment
- Think layer opacity options from Photoshop
- Most common: alpha / 1 - alpha
  
  \[
  \begin{align*}
  \text{out}_A &= \text{src}_A + \text{dst}_A (1 - \text{src}_A) \\
  \text{out}_\text{RGB} &= (\text{src}_\text{RGB} \text{src}_A + \text{dst}_\text{RGB} \text{dst}_A (1 - \text{src}_A)) \div \text{out}_A \\
  \text{out}_A &= 0 \Rightarrow \text{out}_\text{RGB} &= 0
  \end{align*}
  \]

- Look up glBlendFunc for OpenGL blending info
Blending and the depth buffer

- Depth buffer doesn’t handle transparency for you
  - Can’t blend behind pixels that are already there
- Render opaque object first
- Then sort transparent objects back to front, and render them in that order
The worst case

- Sometimes, there’s no way to sort objects by depth
- You have to split one of them somewhere
Jaggies

- Rasterizing onto a pixel grid causes aliasing
- We want to smooth out the result somehow
ANTI-ALIASING

- Jaggies are caused by insufficient resolution
- The best way to fix this is to increase resolution
- The next best way is to take more samples per pixel and average them (AA)
Anti-aliasing

- Lots of cool theory behind AA
- Tons of ways to do it, some are very efficient
  - Most common: full-scene AA (FSAA)
- We’ll maybe cover this later
FIGURES COURTESY...

• Real-Time Rendering, 3rd ed. [RTR]
  - Tomas Akenine-Moller, Eric Haines, Naty Hoffman

• Mathematics for 3D Game Programming and Computer Graphics, 3rd ed. [M3D]
  - Eric Lengyel

  - Edward Angel, Dave Shreiner

• Wikipedia [WP]

• songho.ca [SH]

• David Breen’s CS 430 course notes, Drexel University [DB]