CS 536
Computer Graphics

Scanline Rendering
Week 8, Lecture 16

David Breen
Department of Computer Science
Drexel University

Additional slides from Hank Childs, U. or Oregon
CIS 441/541: Intro to Computer Graphics
Lecture 2: The Scanline Algorithm
What Are We Rendering?

- Models made up of polygons
- Usually triangles
- Lighting tricks make surfaces look non-faceted
- More on this later...
Reminder: ray-tracing vs rasterization

• Two basic ideas for rendering: rasterization and ray-tracing

• Ray-tracing: cast a ray for every pixel and see what geometry it intersects.
  – O(nPixels)
    • (actually, additional computational complexity for geometry searches)
  – Allows for beautiful rendering effects (reflections, etc)
  – Will discuss at the end of the quarter
Reminder: ray-tracing vs rasterization

• Two basic ideas for rendering: rasterization and ray-tracing

• Rasterization: examine every triangle and see what pixels it covers.
  – O(nTriangles)
    • (actually, additional computational complexity for painting in pixels)
  – GPUs do rasterization very quickly
What color should we choose for each of these four pixels?
What color should we choose for each of these four pixels?

Most dominant triangle
What color should we choose for each of these four pixels?
What color should we choose for each of these four pixels?
Problem: how to deposit triangle colors onto an image?

- Let’s take an example:
  - 12x12 image
  - Red triangle
    - Vertex 1: (2.5, 1.5)
    - Vertex 2: (2.5, 10.5)
    - Vertex 3: (10.5, 1.5)
    - Vertex coordinates are with respect to pixel locations
Our desired output

How do we make this output? Efficiently?
Don’t need to consider any pixels outside these lines.
Scanline algorithm: consider all rows that can possibly overlap

Don’t need to consider any Pixels outside these lines
Scanline algorithm: consider all rows that can possibly overlap.

We will extract a “scanline”, i.e. calculate the intersections for one row of pixels.
- Red triangle
  - Vertex 1: (2.5, 1.5)
  - Vertex 2: (2.5, 10.5)
  - Vertex 3: (10.5, 1.5)
– Red triangle
  • Vertex 1: (2.5, 1.5)
  • Vertex 2: (2.5, 10.5)
  • Vertex 3: (10.5, 1.5)
– Red triangle

• Vertex 1: (2.5, 1.5)
• Vertex 2: (2.5, 10.5)
• Vertex 3: (10.5, 1.5)

What are the end points?
– Red triangle
  • Vertex 1: (2.5, 1.5)
  • Vertex 2: (2.5, 10.5)
  • Vertex 3: (10.5, 1.5)

What are the end points?

Y = 5
(2.5, 5)
- Red triangle
  - Vertex 1: (2.5, 1.5)
  - Vertex 2: (2.5, 10.5)
  - Vertex 3: (10.5, 1.5)

What are the end points?

Y=5

(2.5, 5)
Red triangle
- Vertex 1: (2.5, 1.5)
- Vertex 2: (2.5, 10.5)
- Vertex 3: (10.5, 1.5)

Y = mx + b
- 10.5 = m * 2.5 + b
- 1.5 = m * 10.5 + b
- \[ \Rightarrow 9 = -8m \]
- \[ m = -1.125 \]
- \[ b = 13.3125 \]
- \[ 5 = -1.125 \times x + 13.3125 \]
- \[ x = 7.3888 \]

What are the end points?

Y = 5
Scanline algorithm: consider all rows that can possibly overlap

Don’t need to consider any Pixels outside these lines

2.5

7.3888

Y=5
Scanline algorithm: consider all rows that can possibly overlap

Color is deposited at (3,5), (4,5), (5,5), (6,5), (7,5)
Scanline algorithm

• Determine rows of pixels triangles can possibly intersect
  – Call them rowMin to rowMax
    • rowMin: ceiling of smallest Y value
    • rowMax: floor of biggest Y value
  • For r in [rowMin → rowMax] ; do
    – Find end points of r intersected with triangle
      • Call them leftEnd and rightEnd
    – For c in [ceiling(leftEnd) → floor(rightEnd) ] ; do
      • ImageColor(r, c) ← triangle color
Scanline algorithm

- Determine rows of pixels triangles can possibly intersect
- For $r$ in $[\text{rowMin} \rightarrow \text{rowMax}]$ ; do
  - Find end points of $r$ intersected with triangle
    - Call them leftEnd and rightEnd
  - For $c$ in $[\text{ceiling(leftEnd)} \rightarrow \text{floor(rightEnd)}]$ ; do
    - ImageColor($r$, $c$) $\leftarrow$ triangle color

For $r = 5$, leftEnd = 2.5, rightEnd = 7.3888

Y values from 1.5 to 10.5 mean rows 2 through 10

For $r = 5$, we call ImageColor with $(5,3), (5,4), (5,5), (5,6), (5,7)$
Arbitrary Triangles

• The description of the scanline algorithm in the preceding slides is general.
• But the implementation for these three triangles vary:
Supersampling: use the scanline algorithm a bunch of times to converge on the "average" picture.
Filling Primitives: Rectangles, Polygons & Circles

• Two part process
  – Which pixels to fill?
  – What values to fill them with?

• Idea: Coherence
  – Spatial: pixels are the same from pixel-to-pixel and scan-line to scan line;
  – Span: all pixels on a span get the same value
  – Scan-line: consecutive scan lines are the same
  – Edge: pixels are the same along edges
Scan Filling Primitives: Rectangles

• Easy algorithm
  – Fill from $x_{\text{min}}$ to $x_{\text{max}}$
  – Fill from $y_{\text{min}}$ to $y_{\text{max}}$

• Issues
  – What if two adjacent rectangles share an edge?
  – Color the boundary pixels twice?
  – Rules:
    • Color only interior pixels
    • Color left and bottom edges
Scan Filling Primitives: Polygons

• Observe:
  – FA, DC intersections are integer
  – FE, ED intersections are not integer

• For each scan line, how to figure out which pixels are inside the polygon?
Scan Filling Polygons

- Idea #1: use midpoint algo on each edge, fill in between extrema points
- Note: many extrema pixels lie outside the polygon
- Why: midpoint algo has no sense of in/out

(a)

- Span extrema
- Other pixels in the span
Scan Filling Polygons

- **Idea #2**: draw pixels only strictly inside
  - Find intersections of scan line with edges
  - Sort intersections by increasing x coordinate
  - Fill pixels on inside based on a parity bit
    - $B_p$ initially even (off)
    - Invert at each intersect
    - Draw with odd, do not draw when even

- **Span extrema**
- **Other pixels in the span**
Scan Filling Polygons

• Issues with Idea #2:
  – If at a fractional x value, how to pick which pixels are in interior?
  – Intersections at integer vertex coordinates?
  – Shared vertices?
  – Vertices that define a horizontal edge?
How to handle vertices?

• Problem:
  – vertices are counted twice

• Solution:
  – If both neighboring vertices are on the same side of the scan line, don’t count it
  – If both neighboring vertices are on different sides of a scan line, count it once
  – Compare current y value with y value of neighboring vertices
How to handle horizontal edges?

- Idea: don’t count their vertices
- Apply open and closed status to vertices to other edges
  - $y_{\text{min}}$ vertex closed
  - $y_{\text{max}}$ vertex is open
- On AB, A is at $y_{\text{min}}$ for JA; AB does not contribute, $B_p$ is odd and draw AB
- Edge BC has $y_{\text{min}}$ at B, but AB does not contribute, $B_p$ becomes even and drawing stops
How to handle horizontal edges?

- Start drawing at IJ ($B_p$ becomes odd).
- C is $y_{max}$ (open) for BC. $B_p$ doesn’t change.
- Ignore CD. D is $y_{min}$ (closed) for DE. $B_p$ becomes even. Stop drawing.
- I is $y_{max}$ (open) for IJ. No drawing.
- Ignore IH. H is $y_{min}$ (closed) for GH. $B_p$ becomes odd. Draw to FE.
- Ignore GF. No drawing.
Polygon Filling Algorithm

- For each polygon
  - For each edge, mark each scan-line that the edge crosses by examining its $y_{\text{min}}$ and $y_{\text{max}}$:
    - If edge is horizontal, ignore it
    - If $y_{\text{max}}$ on scan-line, ignore it
    - If $y_{\text{min}} \leq y < y_{\text{max}}$ add edge to scan-line $y$'s edge list
  - For each scan-line between polygon’s $y_{\text{min}}$ and $y_{\text{max}}$
    - Calculate intersections with edges on list
    - Sort intersections in $x$
    - Perform parity-bit scan-line filling
    - Check for double intersection special case
  - Clear scan-lines’ edge list
Scan-Filling a Polygon
Scan Filling Curved Objects

- Hard in general case
- Easier for circles and ellipses.
- Use midpoint Alg to generate boundary points.
- Fill in horizontal pixel spans
- Use symmetry
Boundary-Fill Algorithm

• Start with some internal point \((x,y)\)
• Color it
• Check neighbors for filled or border color
• Color neighbors if OK
• Continue recursively
4 Connected Boundary-Fill Alg

Void BoundaryFill4( int x, int y, int fill, int bnd)
{
    If Color(x,y) != fill and Color(x,y) != bnd
    {
        SetColor(x,y) = fill;
        BoundaryFill4(x+1, y, fill, bnd);
        BoundaryFill4(x, y +1, fill, bnd);
        BoundaryFill4(x-1, y, fill, bnd);
        BoundaryFill4(x, y -1, fill, bnd);
    }
}
Boundary-Fill Algorithm

• Issues with recursive boundary-fill algorithm:
  – May make mistakes if parts of the space already filled with the Fill color
  – Requires very big stack size

• More efficient algorithms
  – First color contiguous span along one scan line
  – Only stack beginning positions of neighboring scan lines