University of British Columbia

CPSC 414 Computer Graphics

Scan Conversion

Week 6, Mon 6 Oct 2003

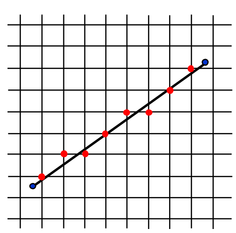
- recap: Bresenham line algorithm
- scan conversion: polygons

News

- Homework 1 out
  - due Wed 15 Oct at beginning of class
  - all late work must be in by Fri 17 Oct
    - solutions out then to help with midterm studying

Scan Conversion recap

- given vertices in DCS, fill in the pixels
  - start with lines

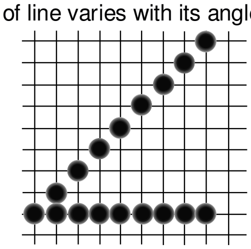


Lines: DDA -> Bresenham recap

- operate only on integers and avoid rounding
- decision variable: after drawing point (x,y) decide whether to draw
  - (x+1,y): case E ("east")
  - (x+1,y+1): case NE ("north-east")
- create discriminator, $d = d_1 - d_2$
  - If $d > 0$ y increases
  - If $d <= 0$ y stays the same
- http://www.cs.technion.ac.il/~cs234325/Homepage/Applets/applets/bresenham/GermanApplet.html

Scan Conversion of Lines

- discussion
  - Bresenham sets same pixels as DDA
  - intensity of line varies with its angle!



Scan Conversion of Lines

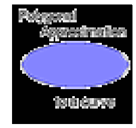
- discussion
  - Bresenham
    - good for hardware implementations (integer!)
  - DDA
    - may be faster for software (depends on system)!
    - floating point ops higher parallelized (pipelined)
      - e.g. RISC CPUs from MIPS, SUN
    - no if statements in inner loop
      - more efficient use of processor pipelining

Rasterizing Polygons

- In interactive graphics, polygons rule the world
- Two main reasons:
  - – Lowest common denominator for surfaces
    - Can represent any surface *with arbitrary accuracy*
    - Splines, mathematical functions, volumetric isosurfaces…
  - – Mathematical simplicity lends itself to simple, regular rendering algorithms
    - Like those we're about to discuss…
    - Such algorithms embed well in hardware

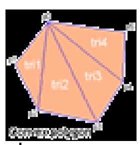
Rasterizing Polygons

- Triangle is the *minimal unit* of a polygon
  - – All polygons can be broken up into triangles
  - – Triangles are guaranteed to be:
    - Planar
    - Convex

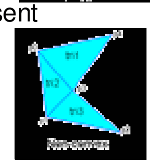


Triangularization

- Convex polygons easily triangulated

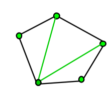


- Concave polygons present a challenge

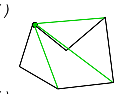


OpenGL Triangularization

- – simple convex polygons
  - break into triangles, trivial
  - `glBegin(GL_POLYGON) ... glEnd()`



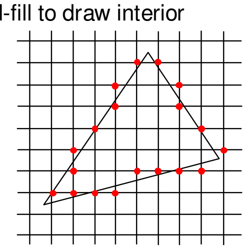
- – concave or non-simple polygons
  - break into triangles, more effort
  - `gluNewTess(), gluTessCallback(), ...`



Scan Conversion of Polygons

•Simple Algorithm:

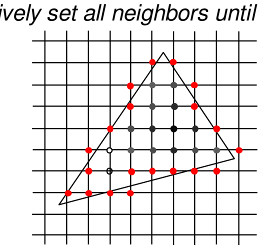
- Draw edges of polygon
- Use flood-fill to draw interior



Scan Conversion of Polygons

•Flood Fill

- Start with *seed point*
- *Recursively set all neighbors until boundary is hit*

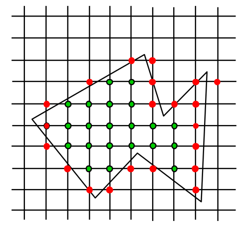


Scan Conversion of Polygons

•Discussion – Flood Fill
– Breadth-first traversal
– Temporary memory needed for storing boundary
– Pixels visited up to 4 times to check if already set
– Problems with other primitives with same color
  • Need per-pixel flag indicating if set already
  • Needs to be cleared for every polygon!

Scan Conversion of Polgons

•Scanline Algorithm
– Scanline: a line of pixels in an image



Scan Conversion of Polygons

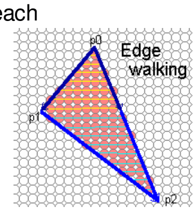
•Classes of Polygons
– Triangles
  • Simple algorithms based on scan-conversion of edges followed by filling of scanlines
– General polygons
  • More complicated scanline algorithms

Rasterizing Triangles

• Interactive graphics hardware commonly uses *edge walking* or *edge equation* techniques for rasterizing triangles

Edge Walking

• Basic idea:
  – Draw edges vertically
    • Interpolate colors down edges
  – Fill in horizontal spans for each scanline
    • At each scanline, interpolate edge colors across span

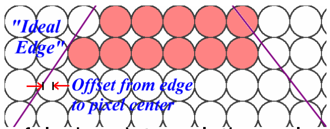


Edge Walking: Notes

• Order three triangle vertices in x and y
  – Find middle point in y dimension and compute if it is to the left or right of polygon. Also could be flat top or flat bottom triangle
• We know where left and right edges are.
  – Proceed from top scanline downwards
  – Fill each span
  – Until breakpoint or bottom vertex is reached
• Advantage: can be made very fast
• Disadvantages:
  – Lots of finicky special cases

Edge Walking: Disadvantages

- Fractional offsets:



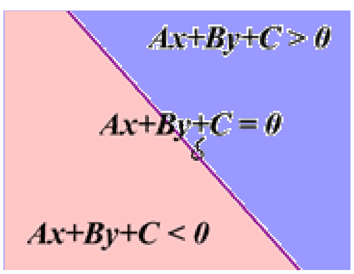
- Be careful when interpolating color values!
- Beware of gaps between adjacent edges
- Beware of duplicating shared edges

Edge Equations

- An edge equation is simply the equation of the line defining that edge
  - Q: *What is the implicit equation of a line?*
  - A: $Ax + By + C = 0$
  - Q: *Given a point (x,y), what does plugging x & y into this equation tell us?*
  - A: Whether the point is:
    - On the line: $Ax + By + C = 0$
    - "Above" the line: $Ax + By + C > 0$
    - "Below" the line: $Ax + By + C < 0$

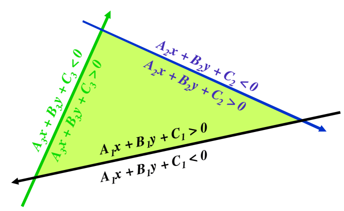
Edge Equations

- Edge equations thus define two *half-spaces*:



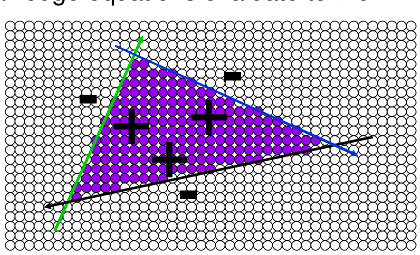
Edge Equations

- And a triangle can be defined as the intersection of three positive half-spaces:



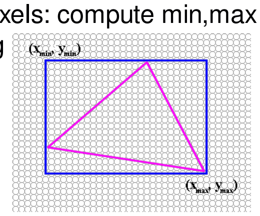
Edge Equations

- So…simply turn on those pixels for which all edge equations evaluate to > 0:



Using Edge Equations

- Which pixels: compute min,max bounding



- Edge equations: compute from vertices
- Orientation: ensure area is positive (*why?*)

Computing Edge Equations

- Want to calculate A, B, C for each edge from (x_1, y_1) and (x_2, y_2)
- Treat it as a linear system:

  $Ax_1 + By_1 + C = 0$

  $Ax_2 + By_2 + C = 0$
- Notice: two equations, three unknowns
- *What can we solve?*
- Goal: solve for A & B in terms of C

Computing Edge Equations

- Set up the linear system:

$$\begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = -C \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

- Multiply both sides by matrix inverse:

$$\begin{bmatrix} A \\ B \end{bmatrix} = \frac{-C}{x_0 y_1 - x_1 y_0} \begin{bmatrix} y_1 - y_0 \\ x_1 - x_0 \end{bmatrix}$$

- Let $C = x_0 y_1 - x_1 y_0$ for convenience
  - Then $A = y_0 - y_1$ and $B = x_0 - x_1$

Edge Equations

- So…we can find edge equation from two verts.
- Given three corners P_0, P_1, P_2 of a triangle, what are our three edges?
- *How do we make sure the half-spaces defined by the edge equations all share the same sign on the interior of the triangle?*
- A: Be consistent (Ex: $[P_0 P_1], [P_1 P_2], [P_2 P_0]$)
- *How do we make sure that sign is positive?*
- A: Test, and flip if needed ($A= -A, B= -B, C= -C$)

Edge Equations: Code

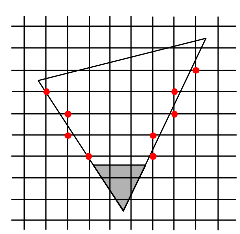
- Basic structure of code:
  - Setup: compute edge equations, bounding box
  - (Outer loop) For each scanline in bounding box...
  - (Inner loop) …check each pixel on scanline, evaluating edge equations and drawing the pixel if all three are positive

Edge Equations: Code

```
findBoundingBox(&xmin, &xmax, &ymin, &ymax);
setupEdges (&a0,&b0,&c0,&a1,&b1,&c1,&a2,&b2,&c2);

/* Optimize this: */
for (int y = yMin; y <= yMax; y++) {
      for (int x = xMin; x <= xMax; x++) {
            float e0 = a0*x + b0*y + c0;
            float e1 = a1*x + b1*y + c1;
            float e2 = a2*x + b2*y + c2;
            if (e0 > 0 && e1 > 0 && e2 > 0)
                        setPixel(x,y);
      }}
```


Scanline Algorithm

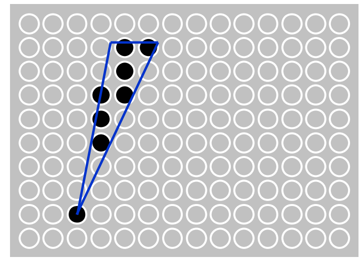


Triangle Rasterization Issues

- *Exactly which pixels should be lit?*
- A: Those pixels inside the triangle edges
- *What about pixels exactly on the edge?*
  - Draw them: order of triangles matters (it shouldn't)
  - Don't draw them: gaps possible between triangles
- We need a consistent (if arbitrary) rule
  - Example: draw pixels on left or top edge, but not on right or bottom edge

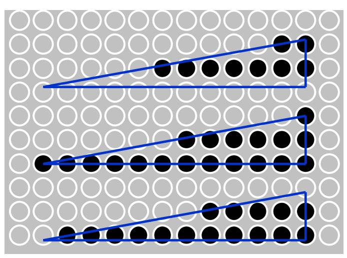
Triangle Rasterization Issues

- Sliver



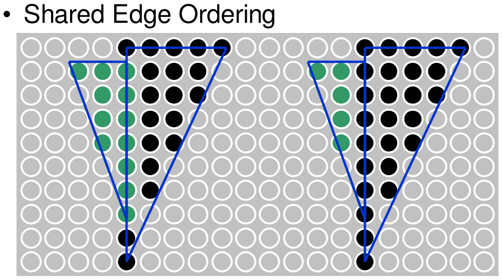
Triangle Rasterization Issues

- Moving Slivers



Triangle Rasterization Issues

- Shared Edge Ordering

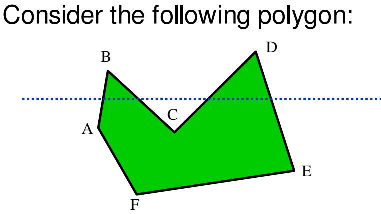


General Polygon Rasterization

- Now that we can rasterize triangles, what about general polygons?
- We'll take an edge-walking approach

General Polygon Rasterization

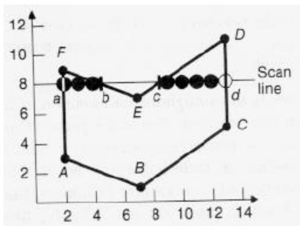
- Consider the following polygon:



- How do we know whether a given pixel on the scanline is inside or outside the polygon?

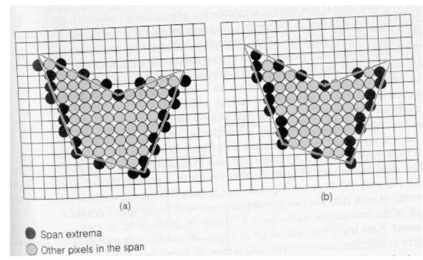
Polygon Rasterization

- Inside-Outside Points



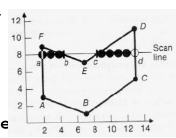
Polygon Rasterization

- Inside-Outside Points



General Polygon Rasterization

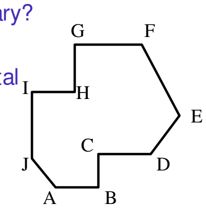
- Basic idea: use a *parity test*



```
for each scanline
    edgeCnt = 0;
    for each pixel on scanline (l to r)
        if (oldpixel->newpixel crosses edge)
            edgeCnt ++;
        // draw the pixel if edgeCnt odd
        if (edgeCnt % 2)
            setPixel(pixel);
```


General Polygon Rasterization

- Count your vertices carefully
  - If exactly on pixel boundary?
  - Shared vertices?
  - Vertices defining horizontal edge?
    - Consider A-B versus I-H

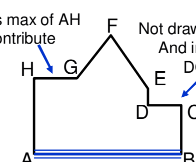


Polygon Rasterization Edge Cases

For scanline, determine all intersections of polygon edges with scanline
Sort edge intersections in least to greatest order
Use parity count to determine when pixels are drawn
Horizontal lines do not contribute to parity count
Y_{min} endpoints do contribute to parity count
Y_{max} endpoints do not contribute to parity count

Not drawn because H is max of AH
And HG does not contribute

Not drawn because D is min of ED
And increments counter to 2.
DC doesn't contribute



Bottom edge drawn because A is min of AH. AB does not contribute

Faster Polygon Rasterization

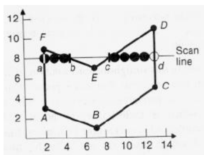
- *How can we optimize the code?*

```
for each scanline
    edgeCnt = 0;
    for each pixel on scanline (l to r)
        if (oldpixel->newpixel crosses edge)
            edgeCnt ++;
        // draw the pixel if edgeCnt odd
        if (edgeCnt % 2)
            setPixel(pixel);
```

- Big cost: testing pixels against each edge
- Solution: *active edge table (AET)*

Active Edge Table

- Idea:
  - Edges intersecting a given scanline are likely to intersect the next scanline
  - The order of edge intersections doesn't change much from scanline to scanline



Active Edge Table

- Algorithm: scanline from bottom to top…
  - Sort all edges by their minimum y coord
  - Starting at bottom, add edges with Y_{min}= 0 to AET
  - For each scanline:
    - Sort edges in AET by x intersection
    - Walk from left to right, setting pixels by parity rule
    - Increment scanline
    - Retire edges with $Y_{max} < Y$
    - Add edges with $Y_{min} < Y$
    - Recalculate edge intersections
  - Stop when $Y > Y_{max}$ for last edges