
CS770/870 Fall 2008

Scan Conversion

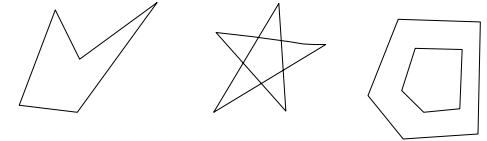
04:19

CS770/870 Fall 2008 Bergeron/Plumlee

1

What polygons can be filled?

- Arbitrary polygons
 - Non-convex
 - self-intersecting
 - can have holes
- What is the interior of the polygon?
 - *odd-parity* rule: a point is inside the polygon if a semi-infinite line from the point crosses a boundary of the polygon an odd number of times



04:19

CS770/870 Fall 2008 Bergeron/Plumlee

2

What is a “pixel”?

- Is it the *area* representing the displayed light?
- Is it an infinitesimal *point* representing the center of that area?
- Unfortunately, there is no rigorous definition; sometimes both definitions are convenient.
- For scan conversion, **we consider a pixel to be an area.**
 - So, the statement "is a pixel inside the polygon?" is ambiguous.
 - We must ask "is the *center* of the pixel inside the polygon?"

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

3

Pixel Filling Criteria

- Need an unambiguous rule for whether to fill a pixel for a given polygon
 - should result in a good approximation to the polygon
 - should guarantee unambiguous decisions for adjoining non-overlapping polygons
 - a given pixel should be filled for only one of those polygons
- Filling rule:
 - Fill a pixel if its *center* is *inside* the polygon or if its *center* is *on a left* or *bottom* edge
- Pixel addressing
 - The pixel address (y,x) where y and x are integers designates the pixel's center

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

4

Filling one Polygon

miny = minimum y-value of all vertices of polygon
 maxy = maximum y-value of all vertices of polygon

for $y = \text{miny}$ to maxy
 find intersections of scanline y with all polygon edges
 sort intersections by increasing x
 fill spans defined by pairs of successive intersections

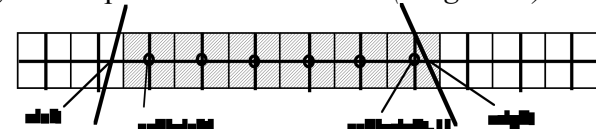
04:19

CS770/870 Fall 2008 Bergeron/Plumlee

5

Filling a Span

- A span is the portion of a scan line defined by 2 successive x-intersection values, $xLeft$ and $xRight$
- To obey the “fill rule”
 - the left most pixel to be filled is $\text{ceil}(xLeft)$
 - the right most pixel to be filled is $\text{ceil}(xright - 1)$



- Note: $\text{trunc}(xright)$ doesn't work since it would fill if the edge intersects the center of the pixel, which is against the rule!
- If the left edge intersects the pixel center, $\text{ceil}(xLeft)$ will be at the center and that pixel will be filled -- as it should be.

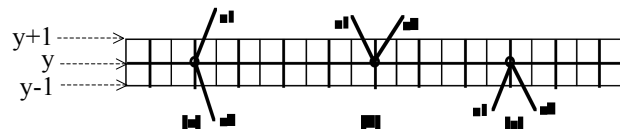
04:19

CS770/870 Fall 2008 Bergeron/Plumlee

6

Vertices at Pixel Centers

- What about vertices that fall on pixel centers?
- There are 3 cases as shown below



- a) can only call this an intersection with one of the 2 edges:
 - it is an intersection if the vertex is at a minimum y for its edge and not if it is a maximum y for its edge; i.e. it's an intersection for $e1$
 - *firsty* for an edge is $\text{ceil}(ymin)$; *lasty* for an edge is $\text{ceil}(ymax - 1)$
- b) min of both edges; there are intersections for both edges, so the pixel will be filled
- c) max of both edges, so there is no intersection with either

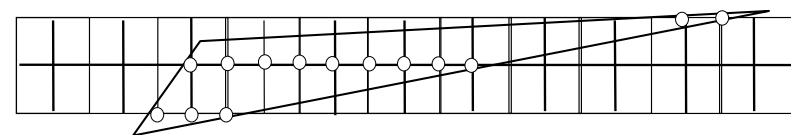
04:19

CS770/870 Fall 2008 Bergeron/Plumlee

7

Miscellaneous Issues

- Horizontal edges?
 - Ignore them! Each vertex of a horizontal edge is also a vertex of another edge
 - the decision to fill the pixels along the horizontal edge will be based on the “other” 2 edges and will correctly fill “bottom” edges, but not “top” edges.
- Slivers can result in pixels filled that are disconnected from others in the polygon



04:19

CS770/870 Fall 2008 Bergeron/Plumlee

8

Efficiency

- Can take advantage of *edge coherence* to save time
 - most edges that intersect scanline y also intersect $y+1$
 - the intersection point on scanline $y+1$ can be calculated from the intersection on scanline y with a single addition:
 - $x_{y+1} = x_y + 1/m$ where m is the slope of the edge
- Can do all calculations with integers!

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

9

Floating Point Version

```
// Remember  $y = mx + b$  is line equation
float m = (y2-y1)/(x2-x1)
float dx = 1/m
int y // y scanline values are always integers
float x // x is the intersection with current y
x = x value of minimum y of the endpoints of the edge
for each y from minimum y of edge to max y of edge
  process scanline
  x += dx
```

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

10

Variation 1 - Less float

```
// split x into integral and fractional parts
float m = (y2-y1)/(x2-x1)
float dx = 1/m
int x, y initialize to minimum endpoint
float xf = 0; // fractional part
for each y from minimum y of edge to max y of edge
  process scanline
  if ( m > 1 )
    xf += dx
  if ( xf > 1 )
    x++
    xf--
else ...
```

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

11

More integer, more of the time

```
// We don't need the actual value of xf, only the condition its in
// Let's multiply xf and all the terms used by it by m
int dxm = 1 // dx = 1/m
int x, y initialize to minimum endpoint
int xfm = 0; // xfm can now be an int
for each y from minimum y of edge to max y of edge
  process scanline
  if ( y2 - y1 > x2 - x1 )
    xfm += dxm // xf += dx
  if ( xfm > m ) // if ( xf > 1 )
    x++
    xfm -= m // xf--
else ...
```

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

12

All integer, all the time

```
// Now, only m remains as a float; do the same thing
// multiply m and any place it is used by (x2-x1)
int x,y; /* current integral value of scanline/edge intersection */
int numerator = x2 - x1;  numerator of (x2 - x1)/(y2 - y1)
int denominator = y2 - y1;
inc = denominator;
for (y=ymin, x=xmin; y<ymax; y++)
  process scanline
  if ( y2 - y1 > x2 - x1 )
    inc += numerator;
    if ( inc > denominator ) // have we overflowed up to next x
      inc -= denominator;
      x++;
  else ...
```

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

13

Caveats

- Previous version assumed $m > 1$ and really only handled the left edge of the scan line
 - need case when $m < 1$
 - need to handle right edge as well
 - both are straightforward
- There are lots of other details and data structures needed
 - ET, *edge table*: one entry per y ; each entry is a list of edges that *start* at this scanline
 - AEL, *active edge list*: all edges that intersect the *current* y
 - An *edge* object includes endpoints, current x intersection value, various other convenient constants

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

14

Full Algorithm

Build ET:

```
for each edge (x0, y0, x1, y1) in polygon
  find  $ymin, ymax$ ,
   $yfirst = ceil(ymin)$ ;
   $xint = xvalue$  at  $yfirst$ ;
   $ylast = ceil(ymax-1)$ ;
   $incr =$  whatever needed depending on slope
  add edge to list  $ET[yfirst]$  in order based on  $xint$ 
```

AEL = empty;

```
for  $y =$  smallest  $yfirst$  of all edges to largest  $ylast$  of all edges
```

```
  add  $ET[y]$  to AEL
```

```
  sort AEL on  $xint$ 
```

```
  fill between pairs of entries on AEL
```

```
  for each edge on AEL
```

```
    if  $y == ylast$ 
```

```
      delete edge from AEL
```

```
    else
```

```
      update  $xint$  and  $incr$  values of edge
```

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

15

Triangle/Quad/Trapezoid

- If all polygons are triangles, trapezoids or convex quadrilaterals, algorithm can be greatly simplified
 - at most 2 edges are active at once
 - there are no lists and sorting is trivial
- Triangles are great since they are guaranteed to be planar
- Horizontal trapezoids are good since there are 1/2 as many as triangles.
- Need algorithms to convert from arbitrary polygons to these special ones.

04:19

CS770/870 Fall 2008 Bergeron/Plumlee

16