1996-11-18 Bo Nordin (bosse&cb.uu.se)

Lecture notes, computer graphics (F09) (Foley 3.2, 3.4, 3.5)

Raster vs. vector

Today, once more we're looking from the implementor's point of view (inside)
Most systems are raster sytems. Fundamental difference: out description & hw description (page 66, figure 3.2)
Many primitives are described in terms of vectors:
Scan conversion.

Sometimes in hw: least job for the package (justs sends attributes and primitives)
We'll look at the *output pipeline* in sw. Alg. apply in sw/hw/microcode.

clipping (was treated before)
1/ *scissoring*. (check each pixel) (may be faster than more elaborate techniques)
also: general clip regions.
2/ computing analytical intersections (we've seen that before)
3/ generate everything and copy only visible portions (used for text)

Problems:
Fast (used every time an image is changed/displayed) - incremental methods!
Aliasing (if n-bits-per-pixel)
Area fill
(Hidden lines)

Scan conversion of lines

The problem is to draw a line between two points in a discrete system (DRAW)

Vertical, horizontal, 45-degree lines are not a problem.

We start with 1-pixel-thick lines on a bi-level device (and assume integer coordinates)

Also: a pixel is a cicular dot centered at that pixel's position.

Specification: - straight line, no aliasing
- start and terminate correctly, make polygon segments connect (clipping!)
- constant brightness (constant # of dots per unit length)
- fast (this is done all the time) integers!
- symmetrical (identical regardless of direction, or if drawn backward)

draw a line from (x_1,y_1) to (x_2,y_2):

$$y = k * x + l = (y_2-y_1)/(x_2-x_1) * x + y_1 - (y_2-y_1)/(x_2-x_1)/ * x_1$$
$$= (\Delta y/\Delta x) * x + y_1 - (\Delta y/\Delta x) * x_1$$

Simple brute-force method:
calculate y for all x between the endpoints of the vector
if $|k| > 1$ calc x = f(y): as dense a line as possible
special-case horizontal, vertical lines
one mult & one add per step (plus one floor).

But: $y_{i+1} = (\Delta y/\Delta x) * x_{i+1} + y_1 - (\Delta y/\Delta x) * x_1 = (\Delta y/\Delta x)(x_i+\Delta x) + B = y_i + (\Delta y/\Delta x)\,\Delta x$

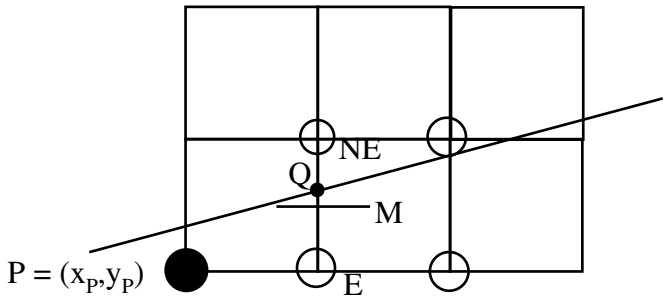
Basic incremental algorithm (DDA - Digital Differential Analyzer)

eliminate the mult by noting that $y_{i+1} = kx_{i+1} + l = k(x_i + \Delta x) + l = y_i + k\,\Delta x$
if (Dx == 1) then $y_{i+1} = y_i + k$

unit change in x -> change y by k
start from (x_0,y_0) (Alg. pages 72-73)

Problems: rounding takes time, y and m must be floating.
Bresenham's algorithm (1965) removes the floating-point and the rounding problems.
But we choose a slightly different approach:
The Midpoint Line Algorithm (one octant only) (slope between 0 and 1)

(pixels drawn at intersections)



We check on which side of the real line the midpoint M lies.

implicit function: $F(x,y) = ax + by + c = 0$ ($dy = y_1 - y_0$, $dx = x_1 - x_0$)
$y = (dy/dx) * x + l$
Therefore: $F(x,y) = dy * x - dx * y + l * dx = 0$ (multiply by dx)
$a = dy$, $b = -dx$, $c = l*dx$
$F(x,y) == 0$ on the line, > 0 below the line, and < 0 above the line
To apply the algorithm compute $F(M) = d = F(x_P+1, y_P+1/2)$ (decision var.)
$d = a(x_P+1) + b(y_P+1/2) + c$. $d <= 0$ (M above) -> E, $d > 0$ (M below): NE

What about next grid line?
E -> $d_{new} = F(x_P+2, y_P+1/2) = a(x_P+2) + b(y_P+1/2) + c$, $d_{new} = d_{old} + a$
NE -> $d_{new} = F(x_P+2, y_P+3/2) = a(x_P+2) + b(y_P+3/2) + c$, $d_{new} = d_{old} + a + b$

if E add dy, if NE add dy - dx

to start: $F(x_0+1, y_0+1/2) = a(x_0+1) + b(y_0 + 1/2) + c = ax_0 + by_0 + c + a + b/2 =$
$F(x_0,y_0) + a + b/2 = a + b/2 = dy - dx/2$

Redefine F by multiplying it by 2: $F(x,y) = 2(ax+by+c)$ (doesn't change sign)
Program pg. 75-76.

For hackers and microcoders:
Further refinement (Wu and Rokne)
look two pixels ahead
if multilevel display one can draw both patterns 2 and 3 instead of discriminting (anti-aliasing for free!)

You can take advantage of the symmetry about the midpoint and scan convert the line from both endpoints simultaneously (and that leads us to the next issue...)

Endpoint order

A line from P_0 to P_1 must contain the same pixels as a line from P_1 to P_0.
Either chose SW when d = 0 (not W) or one can switch endpoints (*anchor* at a start point)
But this does not work for dashed or dotted lines

Starting at the edge of a clip rectangle

Must start ok when lines are clipped:
the intersection - with a vertical line converting from left to right) is
(x_{min}, Round(kx_{min}+l)) (round down if exactly halfway),. Given this initial point
we then initialize d at the midpoint between the E and NE positions in the next column.
This is important, we cannot start from (x_{min},Round(kx_{min}+l))

Horizontal intersections are more complicated: find the intersection of the line with the horizontal line y = y_{min} - 1/2. The first pixel is the one at (Round($x_{ymin-1/2}$), ymin)

The algorithm works even if endpoints are specified in floating-point (arithmetics in floating-point)

Varying the intensity as a function of the slope

Problems with 45 degree lines vs. horizontal, vertical: sqrt(2) longer but as many pixels.
On n-level displays, we can set the intensity to be a function of the slope. -> anti-aliasing

Scan-Line Filling of Polygons

Two different: 1/ which pixels? 2/ which colour(value)?
Rectangle: fill each span from x_{min} to x_{max}. spatial coherence - look only for pixels at which changes occur (special-case: span coherence) (later edge coherence)
Bilevel devices: write many pixels at a time

Note: if two rectangles share an edge some pixels will be written twice (this is a problem if we use xor or if the device is a film recorder, for instance)
Larger problem: which pixels belong to a primitive and which don't
interior pixels easy

Solution: pixels on the left and bottom edges belong, pixels on the right and top edges don't
works for arbitrary polygons, *but* the bottom-left vertex is drawn twice and right-most and top pixels won't be displayed.
works for unfilled as well
considered better than having strange colour in *xor* mode.

General polygon scan-line;
handles both concave and convex polygons plus self-intersecting or with holes
(talk about odd-even rule)

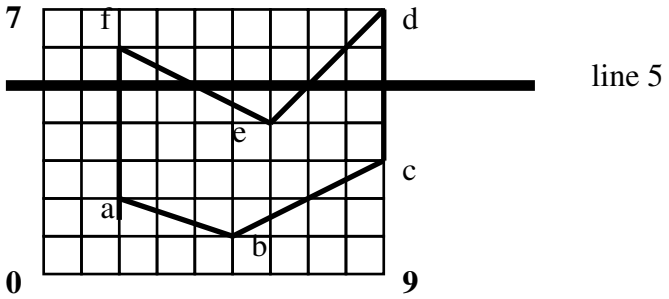
DRAW & DESCRIBE IN WORDS!! (incremental)

for each line (use bounding box):
find each span that lie betweem left and right edges

we could use the line scan-conversion algorithm on each edge and keep a table of span extrema for each scan line. But this will produce some extreme pixels outside the polygon (the line algorithm has no notion of in or out).

1. find intersection of the scan line with all edges of the polygon
2. sort the intersection by increasing x coordinate
3. fill in all pixels between pairs of intersections inside the polygon using the odd-parity rule: parity is initally even, and each intersection inverts the parity bit - draw when od

Example:



First the span-filling strategy:

1/ Given an intersection with an arbitrary, fractional x value, how do we determine which pixel on either side of that intersection is interior

Approaching a fractional intersection to the right and inside: round down x

outside: round up x to be inside

2/ What about intersections at integer pixel coordinates

If the leftmost in a span has integer x it's inside, if the rightmost has integer x it's outside

3/ What about shared vertices in 2/

We count the y_{min} vertex but not the y_{max} vertex. a) is the y_{min} for *fa* but the y_{max} for *ab*.

Both edges and spans are treated as intervals closed at min but open at max.

4/ What about when vertices define a horizontal edge in 2/

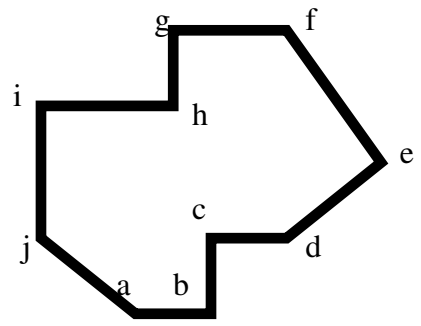
bottom edges are drawn, top edges are not (automatically if we don't count the vertices)

Apply rules to line 5 of the figure: fill in (2,5) and (2,6) (last to the left of the intersection) from the first to the right of the intersection (7,5) to (8,5)

For scan line 2 a counts once as y_{min} of *fa* but y_{max} of *ab*.

ab and *bc* both have their y_{min} at b (counted twice) (enter-draw-enter)

Horizontal edges: don't count their vertices (fig. 3.16, page 89)



Slivers: polygon with edges close together
missing pixels (aliasing, representing a continuous signal with a discrete approximation
if we had multiple bits per pixel we could do something

The intersection calculation must be done cleverly:
don't test ech polygon edge for intersection with each new scan line
Many edges intersected by line i is also intersected by line i+1: *edge coherence*
Use incremental techniques: $x_{i+1} = x_i + 1/m$ (m is the slope)
Consider lines with slopes > +1 (right edges and other slopes are handled similarly.
Vertical edges are special cases and horizontal edges are not at all
Draw at (x_{min},y_{min}). $m = (y_{max}-y_{min})/(x_{max}-x_{min})$
Thus, x has an integer and a fractional part.

Assume slope of 5/2 and $x_{min} = 3$

When the fraction is zero draw (x,y)
When nonzero round up - strictly inside
When fraction > 1, increment x and subtract denominator from the fractional part:

```
x = xmin;
num = xmax - xmin;
den = ymax - ymin
inc = den;
for (y=ymin, y < ymax; y++) {
  WritePixel(x,y,value);
  inc += num;
  if (inc > den ) {
    x += 1;
    inc -= den;
  }
}
```

ex: from (3,0) to (5,5) (DRAW and run thru' the algorithm if there's time)

Now we can show the algorithm:
for each line: AET - active edge table sorted by x intersection values
first create a global edge table (ET) containing all edges sorted by their smaller y coordinate
bucket sort with as many buckets as there are scan lines
within each bucket edges are kept in order of increaing x coordinate
Each entry contains the y_{max} coordinate, x_{min}, and the x increment, ie. 1/k (y_{min} implicit)

```
0
1     ab:  2, 5, -3/1        bc: 3, 5, 4/2
2     fa:  6, 2, 0
3     cd: 7, 9, 0
4    fe:   4, 6, -4/2      ed: 7, 6, 1
5
6
7
```

Algorithm

```
y = smallest non-zero y in the bucket list
AET  = empty;
while (AET or ET) {
  move edges from ET -> AET edges with y_min == y, sort on x
  filll in desired pixels on scan line by using pairs of x coord's from AET
  remove from AET entries for which y==y_max
  y++;
  for each nonvertical edge in the AET update x for new y
}
```

Triangles and trapezoids can be treated as special cases
(And any polygon can be transformed into a mesh of triangles....)

Seed Fill

Uses info in the frame buffer, *not* a functional description
1/ line convert each line and fill
or can be used to fill already created polygons.
2/ find interior pixel
a) scanline
b) guess and use Stokes' teorem (follow the countour $\Sigma d\theta = 2\pi$ if insde, 0 else

Leakage: we need a safe boundary:
4-neighbourhood vs. 8-neighbourhood

Simple recursive algorithm:

```
void seedFill( int x, int y) {
  if (getPixel(x,y) != value) {
    setPixel(x,y,value);
    seedFill(x+1,y);
    seedFill(x-1,y);
    seedFill(x,y+1);
    seedFill(x,y-1);
  }
}
```

Need a large stack. Use a combination

Flood fill

Any pixel value != seed value is considered the boundary.

seed fill.

There's boundary value & a fill value (equal on bi-level devices).