1995-11-14 Bo Nordin (bosse&cb.uu.se)

Lecture notes, computer graphics (F10) (Foley 3.6, 3.3, 3.7)

Polygon filling

What about overlaps? Priority!

Pattern filling (3.6)

I have showed how to fill the interior of a polygon with a value
Filling with a pattern is done by adding extra control to the function that writes a pixel
if transparent: write 1 in foreground, don't write 0's
if opaque 1 in foregrund, 0 in background colour

1/ Anchor the pattern at a vertex (the pattern moves with the primitive) - the point may be specified by the programmer or user or perhaps applied to a group of primitives.
No self-evident point on polygons, circles.

2/ The entire screen is considered tiled with the pattern and the primitive as an outline or filled area of transparent bits which lets the background show through. Using this technique the pattern does not stick to the primitive.

if (pattern[y % n][x % m]) WritePixel(x,y,value); (pattern[0][0] anchored at screen origin

Pattern filling may be done before scan conversion (characters, icons !!):
scan convert into a rectangular area, then copy from work area -> screen
rectangle write: double for loop 1 for current colour, 0 for nowrite/background
This means more work and is meaningful only for primitives that would otherwise be scan converted repeatedly.
why don't just write the rectangle: ok for bi-level, for n-level we must convert to colour!
example: how do we write a symbol with holes that are not solid but patterned.
the problem is that 0's within the pattern must be treated differently from 0's that are holes in the pattern. If we scan convert on the fly the problem does not arise because we never treat pixels outside the boundary of the object.

1/ create a solid bitmap from the outline of the icon (write mask/clipping region) done only once
2/ Any time a patterned copy is needed: write the solid bitmap in bgr clr/replace mode
3/ Create a patterned version of the bitmap copypixel (and) of the patterned and solid
4/ write the new pixmap.in the current colour

Circle/ellipse generators (3.3)

Circles and circular arcs are generated using short straight-line segments.

There are algorithms based on the equation.

Because of symmetry it's enough to calculate points for one octant,

$x^2 + y^2 = R^2$ -> $y = \text{sqrt}(R^2 - x^2)$ -- inefficient multiply, square root + gaps (or long stra lines)

$x = r \cos(\theta) (+ x_c)$ - BETTER, AVOIDS GAPS, STILL INEFFICIENT THOUGH
$y = r \cos(\theta) (+ y_c)$

1/ simple iterative method (θ is the angle producing (x_i,y_i), ϕ is the increment (from TJ)

$x_{i+1} = r \cos(\theta+\phi) = r \cos(\theta) \cos(\phi) - r \sin(\theta) \sin(\phi) = x_i \cos(\phi) - y_i \sin(\phi)$
$y_{i+1} = r \sin(\theta+\phi) = r \cos(\theta) \sin(\phi) + r \sin(\theta) \cos(\phi) = y_i \cos(\phi) + x_i \sin(\phi)$

Because ϕ is a contant "everything" may be tabulated. Avoids large gaps (or long lines)

A refinement is to use 8-way symmetry:
(x,y) (y,x), (y,-x), (x,-y), (-x,-y), (-y,-x), (-y,x) & (-x,y)

2/ parametric ellipse generation
$x = a \cos(\theta)$
$y = b \cos(\theta)$

more complex: ϕ is the angle between the a axis and the x axis:

$x = a \cos(\theta) \cos(\phi) - b \sin(\theta) \sin(\phi) + x_c$
$y = a \cos((\theta) \sin(\phi) + b \sin(\theta) \cos(\phi) + y_c$

$x = A \cos(\theta) - B \sin(\theta) + x_c$
$y = C \cos(\theta) + D \sin(\theta) + y_c$

Here we see that (x_c,y_c) does not enter into the calculations. It may be added last after everything's been calculated (in device coordinates even)

Usually one tabulates sin/cos for even degrees. Because cos and sin are only shifts of each other only one table is needed:

```
void ellipseArc( int phi, int theta1, int theta2, double a, double b, int x, int y) {
 int x0, y0, x1, y1;
 double *sinTheta, *cosTheta;
 double A = a * sintab[phi+90];  /* i.e. cos */
 double B = b * sintab[phi];
 double C = a * sintab[phi];
 double D = b *sintab[phi+90]; /* i.e cos */
 int theta;;
 sinTheta = sintab + theta1;
 cosTheta = sinTheta + 90;
 x0 = A * *cosTheta - B * *sinTheta;
 y0 = C * *cosTheta + D * *sinTheta;
 for (theta = theta1+1; theta <= theta2; theta++) {
  sinTheta = sintab + theta;
  cosTheta = sinTheta + 90;                                    USE SYMMETRY!!!
  x1 = A * *cosTheta - B * *sinTheta;
  y1 = C * *cosTheta + D * *sinTheta;
  DrawLine( x0+x, y0+y, x1+x, y1+x);
  x0 = x1; y0 = y1;
 }
}
```

Problems when the ratio of the major to minor axis is large.

It is possible to use integer-only by scaling the cosines/sines (in hw/microcode)

But a problems with the methods so far is that it plots lines rather than dots (not suited to raster graphics).
So we neeed an incremental algorithm which finds the next pixel from the current one.
Bresenham (1977) developed an incremental circle generator (conceived for pen plotters it incremented all the way around the circle).

Midpoint circle algorithm
Consider only 45 degrees of a circle (2nd octant from x=0 to x = y = sqrt(2)
Strategy: select which of two pixels is closer to the circle by evaluating a function at the mid-point between the two pixels.

choose between E and SE (see fig. 3.12 page 81 - DRAW)
$F(x,y) = x^2 + y^2 - R^2$ 0 on the circle, > 0 outside the circle, and < 0 inside the circle.
If the midpoint between E and SE is outside -> SE closer. if inside -> E closer

We choose pixel on the basis of the decision variable d:
$d_{old} = F(x_P+1,y_P-1/2) = (x_P+1)^2 + (y_P-1/2)^2 - R^2$

if ($d_{old} < 0$) choose E and $d_{new} = F(x_P+2,y_P-1/2) = (x_P+2)^2 + (y_P-1/2)^2 - R^2 = d_{old} + (2x_P+3)$
$\Delta E = 2x_p + 3$
else choose SE and dnew = $F(x_P+2,y_P-3/2) = (x_P+2)^2 + (y_P-3/2)^2 - R^2 = d_{old} + (2x_P-2y_P+5)$
$\Delta SE = 2x_p - 2y_p + 5$

In the linear case, the increments were constants, but here they are functions of the position of the previous pixel.

Start condition: we start at (0,R), the next midpoint is (1, R-1/2) and
$F(1,R-1/2) = 1 + (R^2-R+1/4)-R^2 = 5/4-R$
But we don't want fractions
define h = d-1/4 and substitute h+1/4 for d in the code of Program 3.3
d < 0 becomes h < -1/4. But h starts out as an integer and is incremeneted by integers so we can use h < 0 instead
PGM 3.3, page 83

Ellipses
From the hand-out

$F(x,y) = b^2x^2 + a^2y^2-a^2b^2 = 0$
2a is the length of the major axis along the x axis and 2b is the length of the major axis along the y axis.
We will only handle standard ellipses centered at the origin (moving the origin is a simple translation but general ellipses is a great deal more complex).
Only one quadrant, the others are drawn by symmetry.

The first thing to do is determining the point at which the curve has a slope of -1 since this determines which two pixels we will choose from
Calculate the gradient: grad $F(x,y) = (dF/dx) i + (dF/dy) j = 2b^2xi + 2a^2yj$
when i becomes large switch reg: $a^2y <= b^2x$

when $a^2(y_P-1/2) <= b^2(x_P+1)$ switch from reg 1 to 2.
In region 1: F(x,y) is evaluated at $(x_P+1,y_P-1/2)$
$d_{old} = F(x_P+1,y_P-1/2) = b^2(x_P+1)^2 + a^2(y_P-1/2)^2 - a^2b^2$

E -> $d_{new} = F(x_P+2,y_P-1/2) =b^2(x_P+2)^2 + a^2(y_P-1/2)^2 - a^2b^2 = d_{old} + b^2(2x_P+3)$
NE > $d_{new} = F(x_P+2,y_P-3/2) =b^2(x_P+2)^2 + a^2(y_P-3/2)^2 - a^2b^2 = d_{old} + b^2(2x_P+3) +a^2(-2y_P+2)$

In region 2: F(x,y) is evaluated at $(x_P+1/2,y_P-1)$

$d^{old} = F(x_P+1/2,y_P-1) = b^2(x_P+1/2) + a^2(y_P-1) - a^2b^2$

SE: $d_{new} = F(x_P+3/2,y_P-2) = b^2(x_P+3/2) + a2(y_P-2) - a^2b^2 = d_{old} + b^2(2x_P+2) + a^2(-2y_P+3)$

S: $d_{new} = F(x_P+1/2,y_P-2) = b^2(x_P+1/2) + a2(y_P-2) - a^2b^2 = d_{old} + a^2(-2y_P+3)$

Initialization:

$F(1,b-1/2) = b^2 + a^2(b-1/2)^2 - a^2b^2 = b^2 + a^2(-b+1/4)$

At every iteration in reg 1 we must test d, update the delta functions, and test for reg 2

when 2: $d = F(x+1/2,y-1) = b^2(x_p+1/2) + a^2(y_p-1) - a^2b^2$

Second-order differences for circle generation:

The D functions in the circle generation are linear functions and can thus be computed incrementally Evaluate the function directly at two adjacent points, calculate the difference which, for polynomials, is always a function of a lower degree, and apply the difference in each iteration.

$\Delta E_{old} = 2x_P+3$ & $\Delta SE_{old} = 2x_P-2y_P+5$

if we choose E in the current iteration: POINT OF EVAL: (x_p+1,y)

$\Delta Enew = 2(x_P+1) + 3$ and then $\Delta E_{new} - \Delta E_{old} = 2$

$\Delta SEnew = 2(x_P+1) - 2y_P + 5$ and then $\Delta SE_{new} - \Delta SE_{old} = 2$

if we choose SE in the current iteration: (x_p+1,y_p-1)

$\Delta Enew = 2(x_P+1) + 3$ and then $\Delta E_{new} - \Delta E_{old} = 2$

$\Delta SEnew = 2(x_P+1) - 2(y_P-1)+ 5$ and then $\Delta SE_{new} - \Delta SE_{old} = 4$

Starting:

$\Delta E: 2x_P+3 = 3$ (because $x_0 == 0$)

$\Delta SE: 2x_P - 2y_P + 5 = 5 - 2*radius$

Pgm page 85

And there are scanline methods, too.....

Thick primitives (3.7)

Trace the single-line scan-converted outline primitive

(Orientation of noncircular pens) Rectangular pen: upright or not.

Ends of thick lines (refer to X, cap_style), vertices of thick unfilled polygons (join_style)

Four basic methods (but only two are described in the book (pg 98)!!!):

1/ duplicate pixels in columns for lines with slopes between -1 and 1, rows otherwise

-: lines ends are always vertical/horizontal

gaps where pixel segments meet at an angle, misses pixels where there is a shift from hor to vert (ex. thinnes at arcs at the boundarires between octants)

thickness varies with the angle

what about even-numbered widths (generic problem)

works best for primitives that aren't very thick

2/ square pen whose centre or corner travels along the single-pixel outline

similar to pixel replication.

- thickness varies as a function of the angle (but opposite from 1/) . the square might

turn to follow the path but a cicular pen would be better
implement by copying the whole *footprint* for each pixel?? (pixels written twice)
an algorithm akin to the fill area algorithm would be better.

3/ Filling area between boundaries: construct inner and outer boundary at a distance of t/2 on either side of the ideal primitive trajectory. Or we can use the actual boundary as the outer (for areas) Simple for lines and circles (scan convert 2 circles)

4/ Approximation by thick polylines:
can use efficient alg. for:
line clipping
scan conversion
polygon clipping
polygon scan conversion