Scan Converting Lines

A line is usually specified by its endpoints. In a raster graphics system, we must impose the line on a 2D raster grid. Therefore, we need to know which pixels to turn on.

This is called scan conversion for lines (we convert from vector format to raster scan format).

Basic Incremental Algorithm

$y_i = m x_i + B$, $m = \frac{\Delta y}{\Delta x}$ ← slope of line

Brute-force: Plug in x_i and compute y_i

Better:
$$y_{i+1} = m x_{i+1} + B$$
$$= m(x_i + \Delta x) + B$$
$$= y_i + m \Delta x$$

If $\Delta x = 1$, then $y_{i+1} = y_i + m$

This makes x and y defined in terms of their previous values

→ incremental algorithm

```
/* incremental line drawing for lines with slope
                                  between -1 and +1 */
line(x0, y0, x1, y1)
int x0, y0, x1, y1;
{
      int x;
      double m, y;

      m = (double) (y1-y0) / (x1-x0);
      y = y0;
      for (x = x0; x < x1; x++) {
            writePixel (x, (int)(y+.5));
            y += m;
      }
}
```

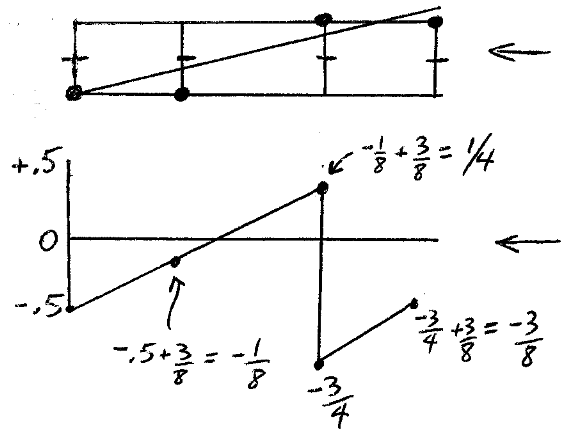
This algorithm is known as a digital differential analyzer (DDA) algorithm. It draws lines by simultaneously incrementing x and y by small steps proportional to first derivative of x and y ($dx = 1$, $dy = m$)

Drawbacks of line(): rounding y to int is time-consuming, floating point arithmetic is used, and holes when $|m| > 1$

Bresenham's Algorithm

Bresenham's alg. uses only integer arithmetic, avoids multiplications, and is an incremental algorithm.

$m = \frac{3}{8}$



← We check if line passes above or below 1/2

← Init error to -.5 (relative to 0 for $\frac{1}{2}$ pt)

Add slope m to error + check sign

```
x = x0;
y = y0;
Δx = x1 - x0;
Δy = y1 - y0;
m = Δy/Δx;
e = m - .5;
for (x = x0; x < x1; x++){
    writePixel (x, y);
    if (e >= 0) {
        y++;
        e--;
    }
    e += m;
}
```

Integer Bresenham's Algorithm

The algorithm presented above required floating point arithmetic and division to calculate m and e.
Speed is increased by using integer arith. and eliminating division.
Before, we had $e = e + \frac{\Delta y}{\Delta x}$ and $e = \frac{\Delta y}{\Delta x} - \frac{1}{2}$ initially.

$$e = \frac{\Delta y}{\Delta x} - \frac{1}{2}$$

$$2e = \frac{2\Delta y}{\Delta x} - 1$$

$$\bar{e} = 2e\Delta x = 2\Delta y - \Delta x \quad \text{(initially)}$$

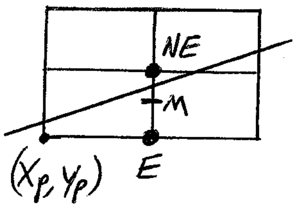
1) $e = e + \frac{\Delta y}{\Delta x} \xrightarrow[\text{by } 2\Delta x]{\text{mult}} \bar{e} = \bar{e} + 2\Delta y$

2) $e = e - 1 \longrightarrow \bar{e} = \bar{e} - 2\Delta x$

Now, rewrite previous code for integer version:

```
X=X0;
Y=Y0;
ΔX=X1-X0;
ΔY=Y1-Y0;
ē = 2ΔY-ΔX;
for(x=X0; x<x1; x++){
        writePixel (x,y);
        if (ē>=0) {
                y++;
                ē -= 2Δx;
        }
        ē +=2Δy;
}
```

Midpoint Line Algorithm: Another Formulation for the Bresenham Algorithm



In the midpoint alg., we observe on which side of midpoint M does the line lie. If M lies above the line, we choose E ; else NE.
Error is always $\le 1/2$.

To do this, we use the implicit form for line:

$$F(x,y) = ax + by + c = 0$$

$$y = \frac{dy}{dx} x + B$$

m

$$0 = \frac{dy}{dx} x - y + B$$

$$0 = \underbrace{dy}_{a}\, x \underbrace{- dx}_{b}\, y + \underbrace{B dx}_{c}$$

Note: $F(x,y) = 0$ is on line
> 0 for pts below line
< 0 for pts above line

To apply the midpoint criterion, compute $d = F(m) = F(x_p+1, y_p+\frac{1}{2})$ and test sign.
If $d > 0$, select NE. If $d \leq 0$, select E.
Time saving $\longrightarrow$ d can be computed incrementally:

If E was chosen, then

$$d_{new} = F(x_p+2, y_p+\tfrac{1}{2})$$
$$= a(x_p+2) + b(y_p+\tfrac{1}{2}) + c$$

but $d_{old} = a(x_p+1) + b(y_p+\frac{1}{2}) + c$ $\leftarrow$ This is initially

$$ax_0 + by_0 + c + a + \frac{b}{2}$$
$$= \underbrace{F(x_0,y_0)}_{0 \text{ because } (x_0,y_0) \text{ is on line}} + a + \frac{b}{2}$$

$$\therefore d_{strt} = a + \frac{b}{2}$$
$$= dy - \frac{dx}{2}$$

$$\therefore \Delta d_E = d_{new} - d_{old} = a = dy$$

If NE was chosen, then

$$d_{new} = F(x_p+2, y_p+\tfrac{3}{2})$$
$$= a(x_p+2) + b(y_p+\tfrac{3}{2}) + c$$
$$\therefore \Delta d_{NE} = d_{new} - d_{old} = a+b = dy - dx$$

```
midpointLine(x0, y0, x1, y1)      ← Note: this is
int  x0, y0, x1, y1;                equivalent to the
{                                   Bresenham algorithm
    int  d, dx, dy, incE, incNE, x, y;

    dx = x1 - x0;
    dy = y1 - y0;
    d = 2*dy - dx;        ← To avoid division in
    incE = 2*dy;            dstrt = dy - dx/2 , we
    incNE = 2*(dy - dx);    mult both sides by 2
                            (does not affect sign of d)
    x = x0;
    y = y0;
    while (x < x1) {
        writePixel(x, y);
        if (d <= 0) {          /* choose E */
            d += incE;
            x++;
        } else {
            d += incNE;
            x++;
            y++;
        }
    }
}
```

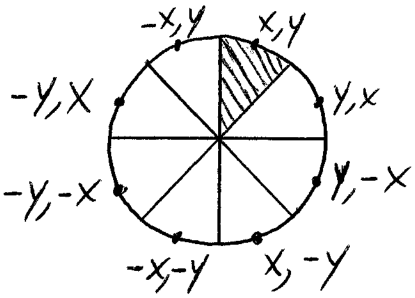
Scan Converting Circles

$x^2 + y^2 = R^2$

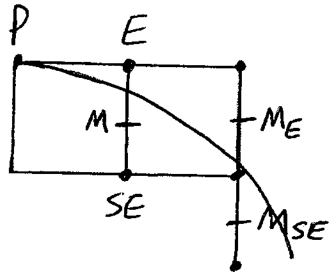
$y = \pm\sqrt{R^2 - x^2} \longrightarrow$ inefficient because of mult and square root. Also, circle will have large gaps for x close to R because slope $\to \infty$

$\left.\begin{array}{l} x = R\cos\theta \\ y = R\sin\theta \end{array}\right\} \longrightarrow$ also inefficient but avoids large gaps

For efficiency, exploit symmetry: compute one octant and copy to others.



Midpoint Circle Algorithm



As with the midpoint line algorithm, the strategy is to select which of 2 pixels is closer to the circle by evaluating a function at the midpoint between the 2 pixels.

$P \longrightarrow E$ or SE

Let $F(x,y) = x^2 + y^2 - R^2$

$$F(x,y) \begin{cases} = 0 & \text{on circle} \\ > 0 & \text{outside} \\ < 0 & \text{inside} \end{cases}$$

$d_{old} = F(x_p+1, y_p - \frac{1}{2})$

value of F at midpt

$= (x_p+1)^2 + (y_p - \frac{1}{2})^2 - R^2$

If $d_{old} < 0$, select E because midpt is inside circle

$d_{new} = F(x_p+2, y_p - \frac{1}{2})$

$= (x_p + 2)^2 + (y_p - \frac{1}{2})^2 - R^2$

$= d_{old} + (2x_p + 3)$

$\therefore \Delta d_E = 2x_p + 3$

If $d_{old} \geq 0$, select SE

$d_{new} = F(x_p+2, y_p - \frac{3}{2}) = (x_p+2)^2 + (y_p - \frac{3}{2})^2 - R^2$

$= d_{old} + (2x_p - 2y_p + 5)$

$\therefore \Delta d_{SE} = 2x_p - 2y_p + 5$

In the linear case, Δd_E and Δd_{NE} were constants (dy and $dy - dx$, respectively)

In this quadratic case, however, Δd_E and Δd_{SE} are linear fcts of x_p, y_p

Circle alg. is same as line:
1) choose pixel based on sign of d
2) update d with appropriate Δd

The only difference for circle: in updating d, we evaluate a linear function of (x_p, y_p)

Initially: the starting pixel lies at $(0, R)$
The next midpoint is $(1, R-\frac{1}{2})$.

$$F(1, R-\tfrac{1}{2}) = 1 + (R^2 - R + \tfrac{1}{4}) - R^2 = \underbrace{\tfrac{5}{4} - R}_{\text{initial } d}$$

```
midpointCircle (r)
int r;
{     int x, y, d;

      x=0;
      y=r;
      d=1-r;   ← equivalent to 5/4-r for int version
      for(; y>x; x++){
          circlePoints (x,y);    /* copy to other octants*/
          if (d<0) d+=(2*x+3);
          else { d+= 2*(x-y)+5;
                 y--;
          }
      }
}
```