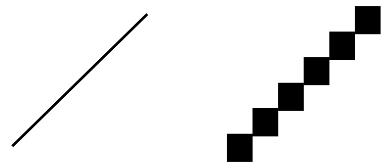
2D Primitives I

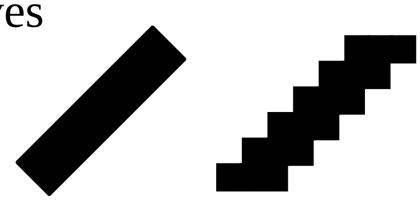
Point-plotting (Scan Conversion)

- Lines
- Circles
- Ellipses

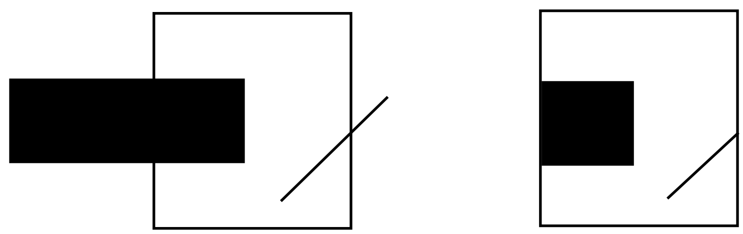


Scan Conversion of Filled Primitives

- Rectangles
- Polygons



Clipping



In graphics must approximate the ideal mathematical continuous primitive with a discrete version

Many concepts are easy in continuous space - Difficult in discrete space

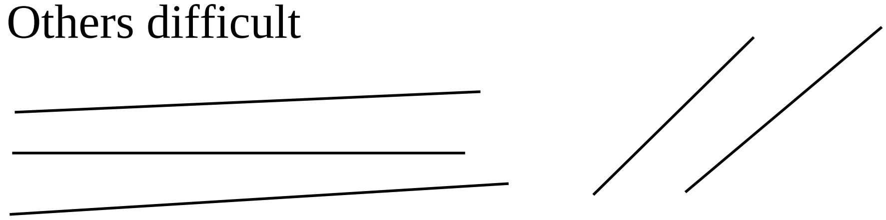
Example: Lines

Line Drawing Algorithms

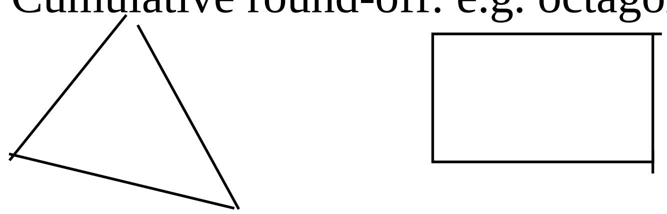
Lines used a lot - want to get them right

Criteria for Line Drawing Algorithms:

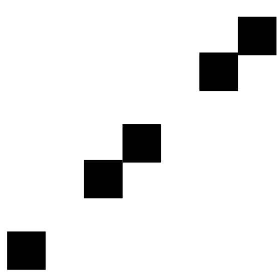
1) Lines should appear straight - no jaggies
   - Discretization problem
   - Horizontal, vertical and diagonals easy
   - Others difficult



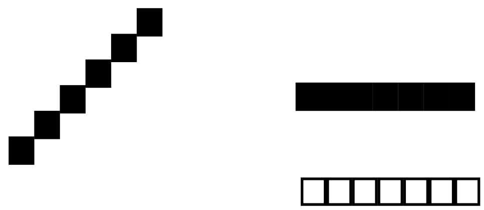
2) Lines should terminate accurately
   - Discretization
   - Cumulative round-off: e.g. octagon



3) Lines should have constant density
   - dots/line length
   - equal spacing of dots



4) Line density should be independent of line length or angle



5) Lines should be drawn rapidly

Efficient algorithms

Mathematical Preliminaries

How to represent a line with an equation?

Nonparametric:

Explicit

$y = f(x)$

example?

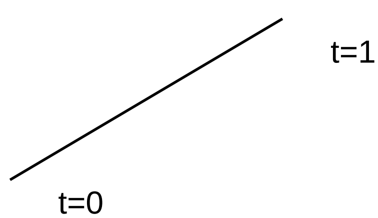
Implicit

$f(x,y) = 0$

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

Parametric:

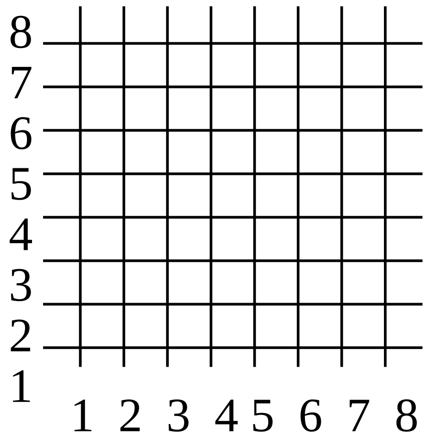
$x = f(t)$
$y = g(t)$



Scan Converting Lines

Drawing lines by identifying each point

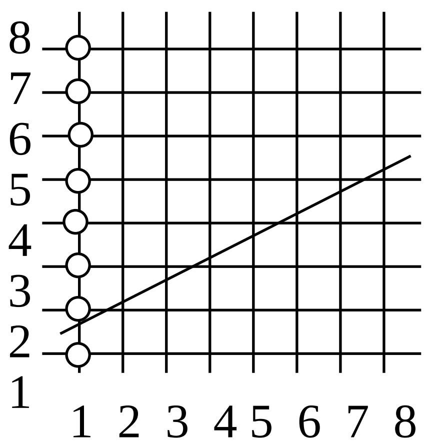
Rastor (discrete) Space



Different possibilities:
- Pixels lie at intersections
- Pixels lie in centers

Different locations for origin

Different size or shape pixels
- square in Timex/Sinclare
- ~round on CRT
- overlap versus none



Horizontal, vertical, diagonal

Oblique

Line Algorithm:

Line from (x1, y1) to (x2,y2)

```
dx = x2-x1
dy = y2-y1
m = dy/dx
y = y1
for x = x1 to x2
        {DrawPixel(x,Round(y)
        y = y + m}
```

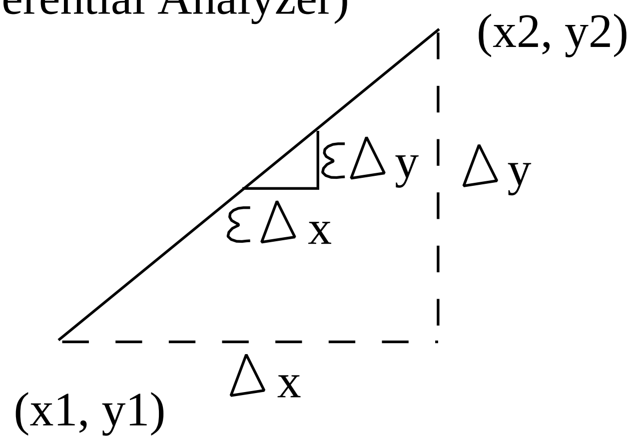
Problems:
- slow
- floating point math
- almost vertical lines get dotty

How solve last one?
- axis of greatest motion

DDA Algorithm: (Digital Differential Analyzer)

$$x_{n+1} = x_n + \varepsilon \Delta x$$

$$y_{n+1} = y_n + \varepsilon \Delta y$$

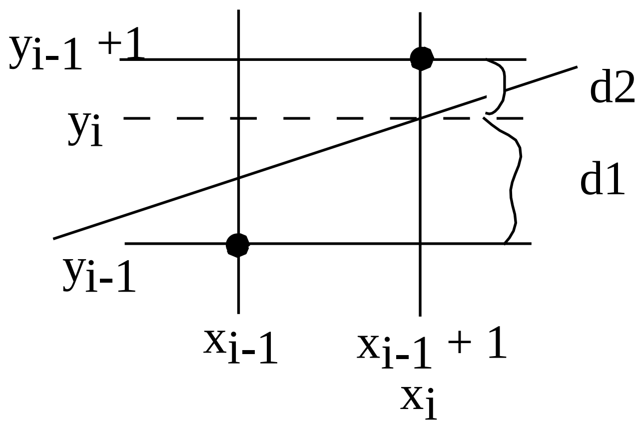


update x and y by their differentials
(epsilon is some small positive constant)

Problems:
- still slow
- still floating point math

Advantages:
- mathematically well defined
- no spotty lines

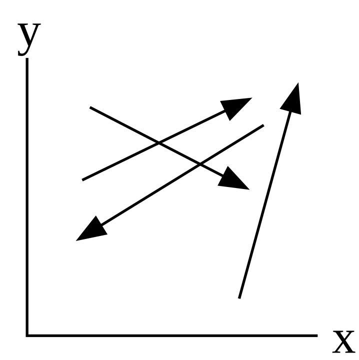
Bresenhams Algorithm:
- one of the best for lines (doesn't generalize)
- only integer math
- simple but weird algorithm
- based on the error
  - keeps track of how far a pixel is from the "true line" and corrects when it gets too far



Ideal Continuous Line: (x_a, y_a) to (x_b, y_b)

$y = m (x - x_a) + y_a$
where m is the slope

Assume:
$x_a < x_b$
$0 < m < 1$



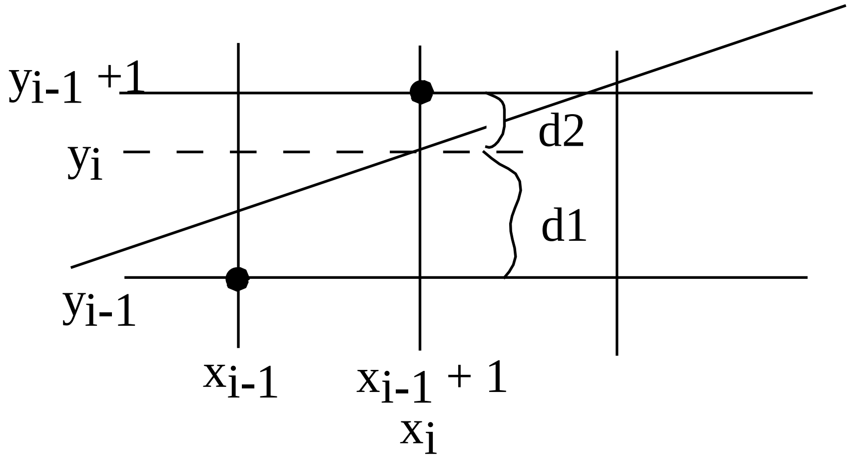
what set of possible lines?

how many sets of possible lines?

For this set, which will be axis of greatest movement?

true point is at y
choose y_{i-1} or $y_{i-1} + 1$?

look at difference (error)
is d1 or d2 smaller?
choose point with smallest difference



Look at (d1 - d2)

if > 0, choose $y_{i-1} + 1$

if < 0, choose y_{i-1}

$$d1 - d2 = (y - y_{i-1}) - (y_{i-1}+1 - y)$$

$$= 2\,y - 2(y_{i-1}) - 1 \qquad \text{(regroup)}$$

$$= 2m\,(x_i - x_a) + 2(y_a - y_{i-1}) - 1 \qquad \text{(substitute in for y)}$$

(multiply by Δx)

$$e_i = \Delta x(d1 - d2) = 2\Delta y\,(x_i - x_a) + 2\Delta x(y_a - y_{i-1}) - \Delta x$$

This will be the decision variable

Calculate e_i incrementally:

$$e_{i+1} = e_i + 2\Delta y\,\underbrace{(x_{i+1} - x_i)}_{=1} - 2\Delta x\underbrace{(y_i - y_{i-1})}_{=0 \text{ or } 1}$$

if y was incremented:

$$e_{i+1} = e_i + 2(\Delta y - \Delta x)$$

otherwise:

$$e_{i+1} = e_i + 2\Delta y$$

Now very simple to compute!

Bresenham's Initial Conditions

$i = 0$

$x_0 = ?$ $y_0 = ?$

$e_i = \Delta x(d1 - d2)$

$e_1 = \Delta x[(y-y_a) - ((y_a+1)-y)]$

$= ?$

Bresenham's Algorithm:

```
x = xa
y = ya
dx = xb -xa
dy = yb - ya
err = 2 dy - dx
for i = 1 to dx
    {
    drawpixel (x,y)
    if err > 0
        {
        y = y + 1
        err = err + 2dy - 2dx
        }
    else
        err = err + 2dy
    x = x + 1
    }
```

Why efficient?

How Generalize to other sets of lines?

was $x_a < x_b$ and $o < m < 1$

$x_a > x_b$?

$m > 1$?

$0 > m > -1$?
(dy = -dy and dec y)

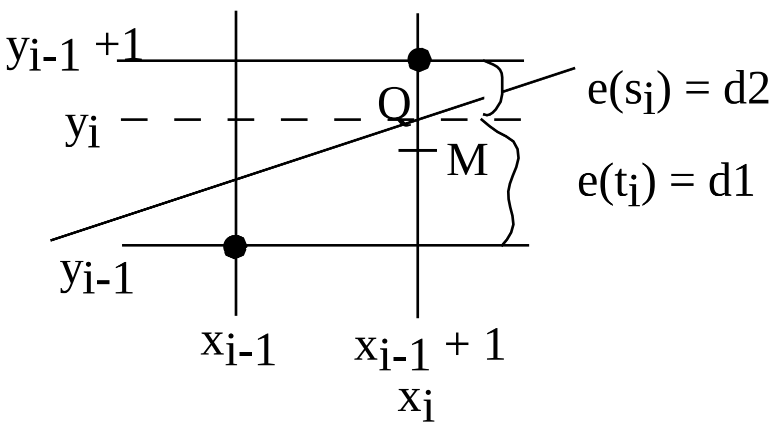
$m = 1$ or $m = 0$?

Midpoint Line Algorithm

Bresenham's cannot generalize to arbitrary conics

Thus use Midpoint Line Algorithm

For lines and circles, end up with identical algorithm



Bresenhams: look at sign of scaled difference in errors

Midpoint: look at which side of line midpoint falls on
(see derivation in the text)

It has been proven that Bresenhams gives an optimal fit for lines

It has been proven that Midpoint is equivalent to Bresenhams
for line

Scan Converting Circles

Circle equation: $x^2 + y^2 = R^2$

1) So try plotting

$y = +/- SQRT(R^2 - x^2)$

(see example)

Problem: gets spotty in places

Why?
(axis of greatest motion)

2) Try polar coordinates

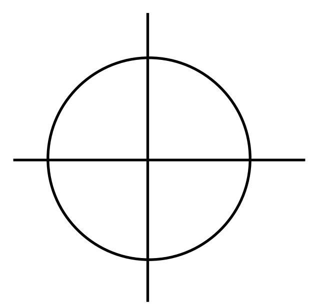
$x = R \cos(\theta)$

$y = R \sin(\theta)$

Problem: very slow

Why?

So we need a better technique - like for lines



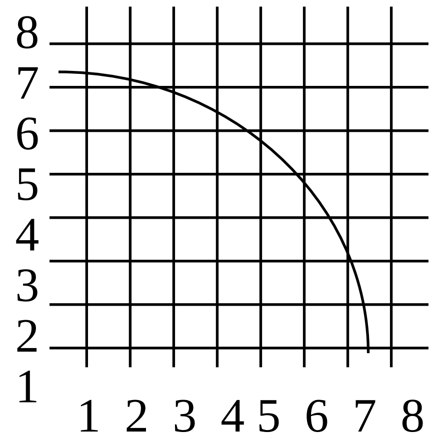
8 - Way Symmetry

assume circle is centered at origin
How much of circle do we have to compute?
(how many axes of symmetry)
If compute (x,y) for a point in the second octant

drawpixel(x,y) drawpixel(-x,y)
drawpixel(-y,x) drawpixel(-y,-x)
drawpixel(-x,-y) drawpixel(x,-y)
drawpixel(y,-x) drawpixel(y,x)

What if circle centered about pixel other than origin?
What if circle not centered about a pixel?

Mitchner's Circle Algorithm:



Based on Breshenham's ideas

Derive it for the second octant

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

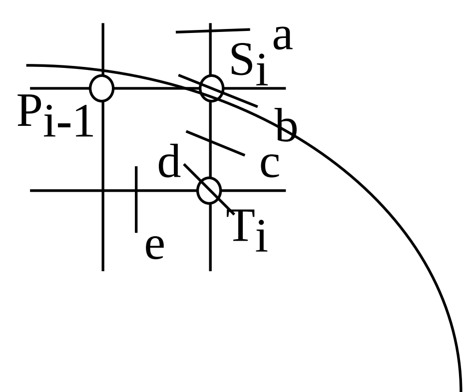
start from x=0, and go to x=y

Since our points $P_{i-1} = (x_{i-1},y_{i-1})$ will be integers, there will be errors

$e(P_{i-1}) = (x_{i-1}^2 + y_{i-1}^2) - R^2$

what sign is $e(P_{i-1})$ when (x,y) inside circle?

outside circle?



Only two possible next points in this octant

S_i and T_i

choose one with smallest error

Create error difference

$d_i = e(S_i) + e(T_i)$

So if $d_i < 0$, choose S_i

else, choose T_i

Only five possible cases: show it works in each

for c: S is outside, T is inside

errors will have opposite signs, S's positive

if d < 0, T's is larger, choose S

for a and b: S is on or inside circle, T inside

T's is negative, S's is negative or 0

d < 0, choose S

for d and e: T is on or outside; S is outside

S's error positive, T's is positive

d > 0, choose T

Incremental version of decision variable:

$d_{i+1} = d_i + 4x_{i-1} + 6 + 2(y_i^2 - y_{i-1}^2) - 2(y_i - y_{i-1})$

(should be able to derive this)

if $d_i < 0$, then y didn't change

$d_{i+1} = d_i + 4x_{i-1} + 6$

else, y changed by -1

$d_{i+1} = d_i + 4(x_{i-1}) + 6 - 2\,(2y_{i-1} - 2) - 2$

$= d_i + 4(x_{i-1} - y_{i-1}) + 10$

Initial Conditions:

$x_0 = 0, \qquad y_0 = ?$

$S_1 = (1, R) \qquad T_1 = (1, R-1)$

$d_1 = (x_s^2 + y_s^2) - R^2 + (x_t^2 + y_t^2 - R^2)$

$= 1 + R^2 - R^2 + (1 + R^2 - 2R + 1 - R^2)$

$= 3 - 2R$

Michner's Circle Algorithm:

```
x = 0
y = R
d = 3 - 2R
while x <= y
    {
    DrawPixel(x,y)
    if d < 0
        d = d + 4x +6
    else
        {
        d = d + 4(x-y) + 10
        y = y - 1
        }
    x = x +1
    }
```

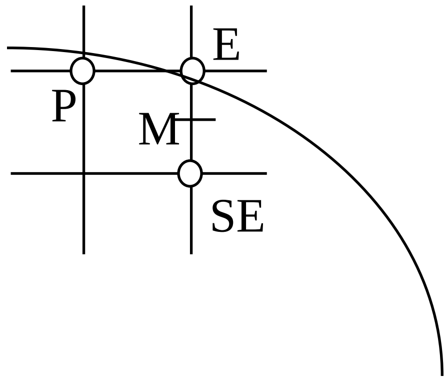
Problems:

multiplication

doesn't generalize to all conics

Midpoint Circle Algorithm

Do analysis based on whether midpoint is inside or outside of circle



See derivations in text for: midpoint circle algorithm
integer midpoint circle algorithm
integer second-order difference midpoint circle algorithm

Midpoint Circle Algorithm

```
x = 0
y = R
d = 1-R
dE = 3
dSE = 5 - 2R
DrawPixel(x,y)
while (y > x){
    if d < 0{
        d = d + dE
        dE = dE + 2
        dSE = dSE + 2
        x = x + 1
    }else{
        d = d + dSE
        dE = dE + 2
        dSE = dSE + 4
        x = x + 1
        y = y -1
        }
    DrawPixel(x,y)
}
```

What is initial sign of d?

How can d ever go from positive to negative?