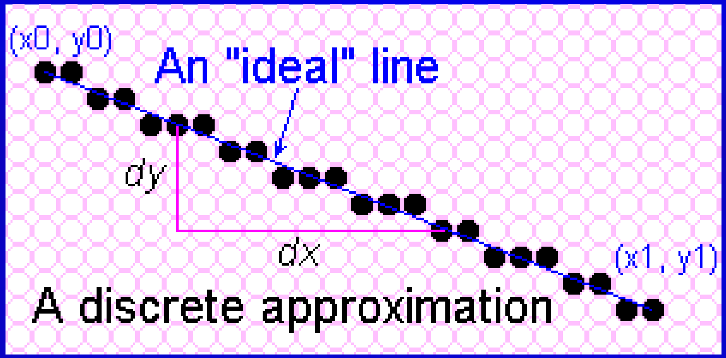
Computer Graphics

DDA & Bresenham Line Drawing

Towards the Ideal Line

- We can only do a discrete approximation



- Illuminate pixels as close to the true path as possible, consider bi-level display only
  - Pixels are either lit or not lit

What is an *ideal* line

- Must appear straight and continuous
  - Only possible axis-aligned and 45^{o} lines
- Must interpolate both defining end points
- Must have uniform density and intensity
  - Consistent within a line and over all lines
  - What about antialiasing?
- Must be efficient, drawn quickly
  - Lots of them are required!!!

Simple Line

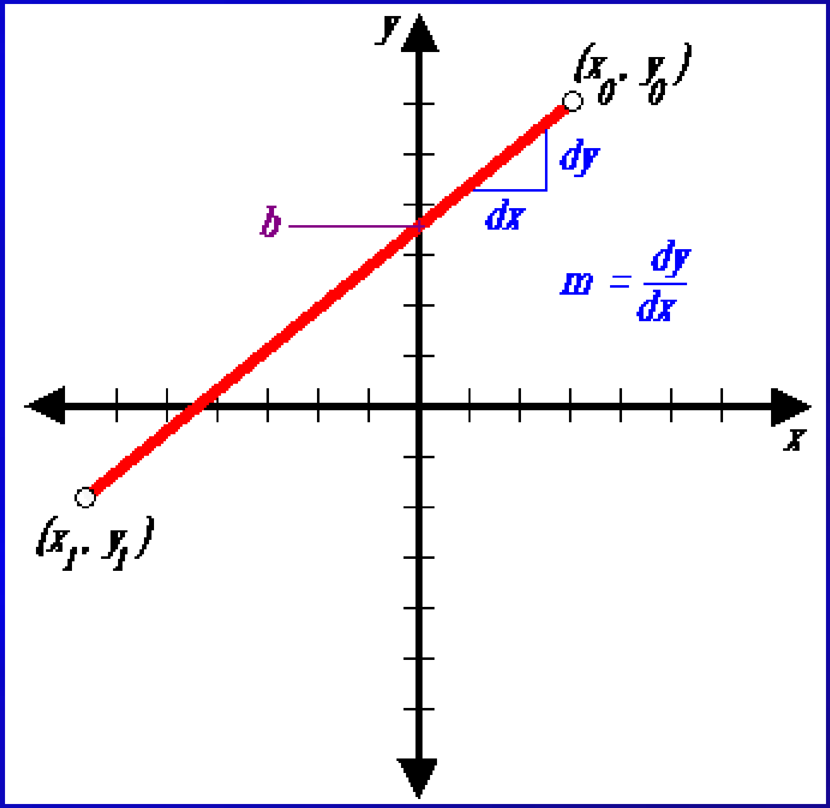
Based on *slope-intercept algorithm* from algebra:

$$y = mx + b$$

Simple approach:

increment x, solve for y

Floating point arithmetic required

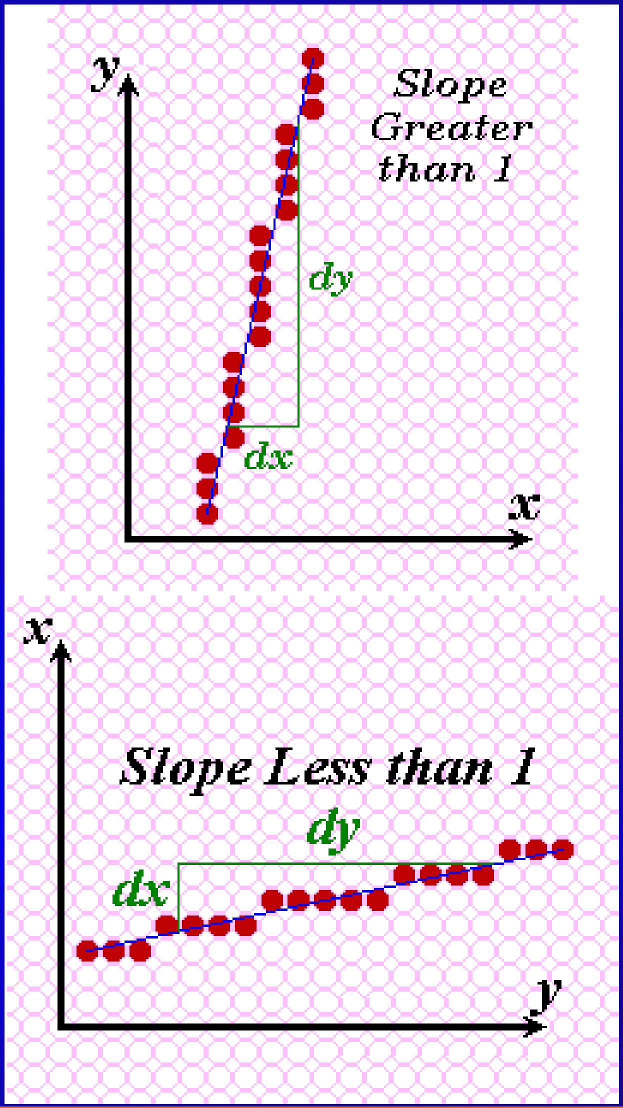


Does it Work?

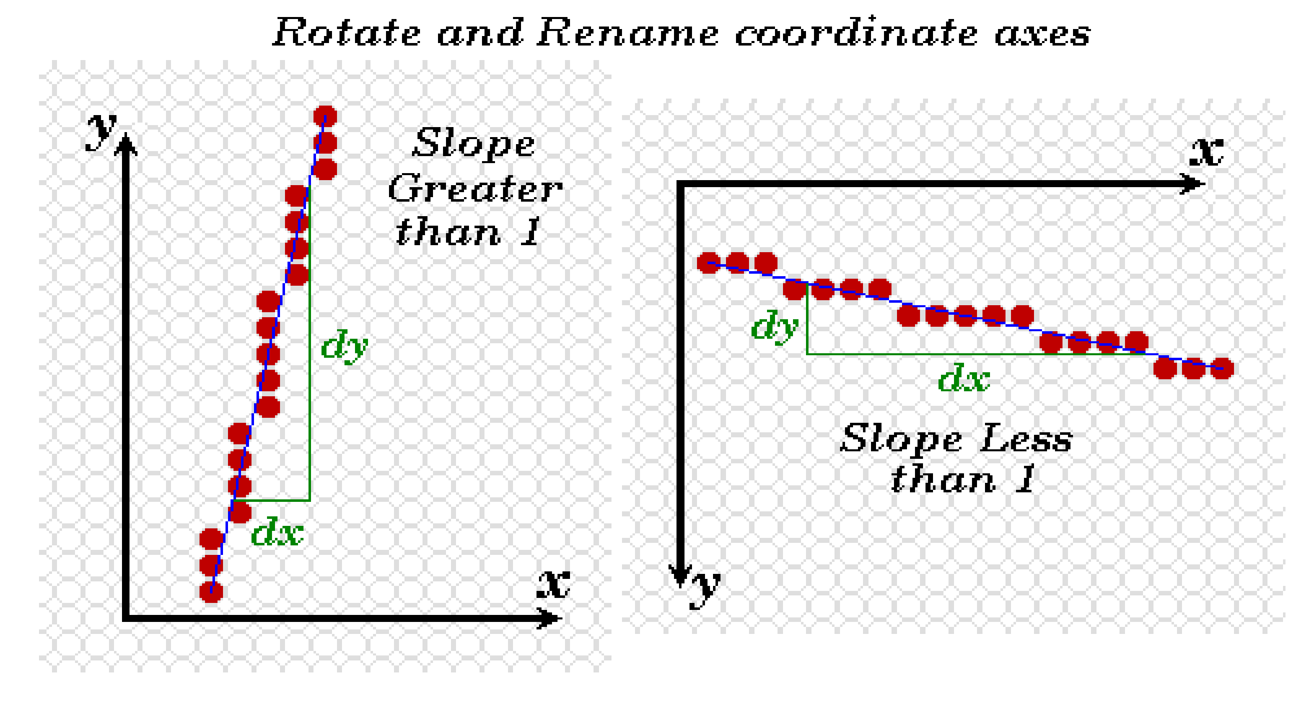
It seems to work okay for lines with a slope of 1 or less,

but doesn't work well for lines with slope greater than 1 – lines become more discontinuous in appearance and we must add more than 1 pixel per column to make it work.

Solution? - use *symmetry*.



Modify algorithm per octant



OR, increment along x-axis if dy<dx else increment along y-axis

DDA algorithm

- DDA = Digital Differential Analyser
  - finite differences
- Treat line as parametric equation in t :

Start point - (x_1, y_1)
End point - (x_2, y_2)

$$x(t) = x_1 + t(x_2 - x_1)$$
$$y(t) = y_1 + t(y_2 - y_1)$$

DDA Algorithm

$$x(t) = x_1 + t(x_2 - x_1)$$
$$y(t) = y_1 + t(y_2 - y_1)$$

- Start at $t = 0$
- At each step, increment t by dt
- Choose appropriate value for dt
- Ensure no pixels are missed:
  - Implies: $\frac{dx}{dt} < 1$ and $\frac{dy}{dt} < 1$

$$x_{new} = x_{old} + \frac{dx}{dt}$$
$$y_{new} = y_{old} + \frac{dy}{dt}$$

- Set dt to maximum of dx and dy

DDA algorithm

```
line(int x1, int y1, int x2, int y2)

{
float x,y;
int dx = x2-x1, dy = y2-y1;
int n = max(abs(dx),abs(dy));
float dt = n, dxdt = dx/dt, dydt = dy/dt;
       x = x1;
       y = y1;
       while( n-- ) {
              point(round(x),round(y));
       x += dxdt;
       y += dydt;
       }
}
```

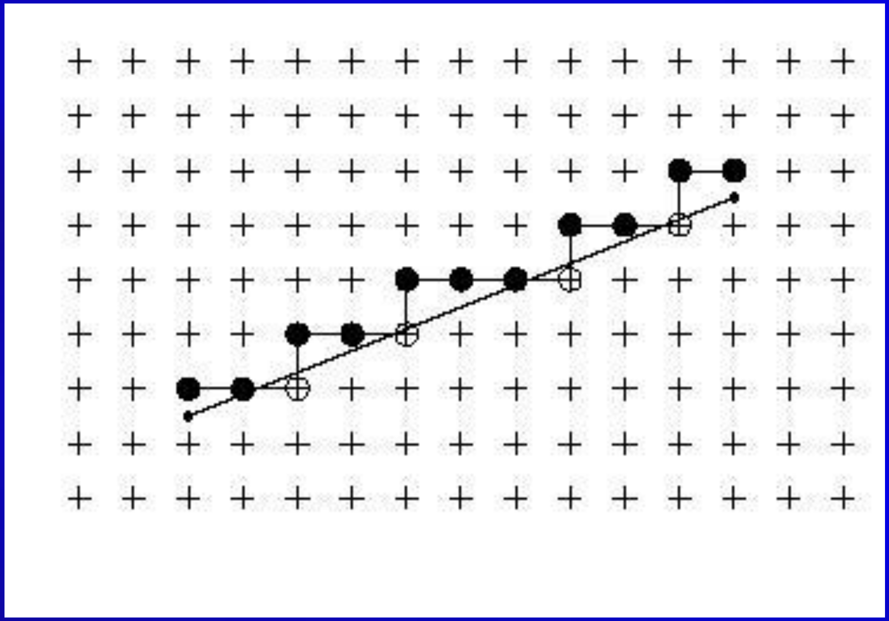
n - range of t.

DDA algorithm

- Still need a lot of floating point arithmetic.
  - 2 ‘round’s and 2 adds per pixel.

- Is there a simpler way ?
- Can we use only integer arithmetic ?
  - Easier to implement in hardware.

Observation on lines.



```
while( n-- )
{
draw(x,y);
move right;
if( below line )
move up;
}
```

DDA ALGORITHM

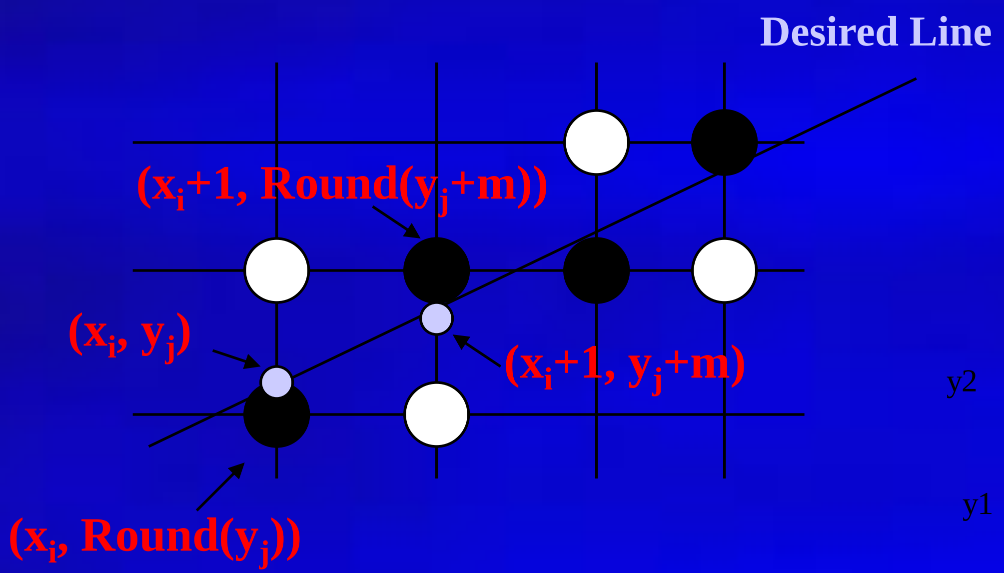
- *The digital differential analyzer (DDA)* samples the line at unit intervals in one coordinate corresponding integer values nearest the line path of the other coordinate.

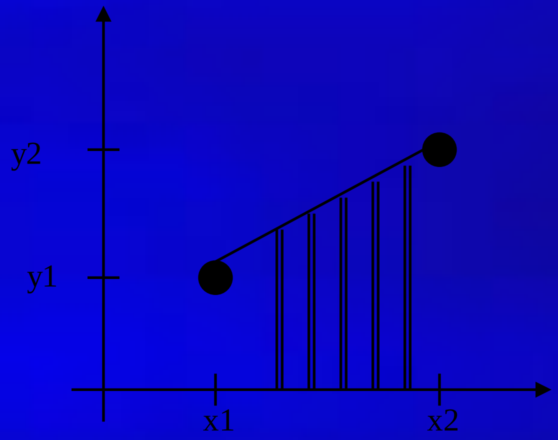
- The following is thus the basic *incremental scan-conversion(DDA)* algorithm for line drawing

```
for x from x0 to x1
Compute y=mx+b
Draw_fn(x, round(y))
```

- Major deficiency in the above approach :
  - Uses floats
  - Has rounding operations

DDA Illustration





Bresenham's Line Algorithm

- **An accurate, efficient raster line drawing algorithm developed by Bresenham, scan converts lines using only *incremental integer* calculations that can be adapted to display circles and other curves.**

- **Keeping in mind the symmetry property of lines, lets derive a more efficient way of drawing a line.**

  Starting from the left end point (x_0,y_0) of a given line , we step to each successive column (x position) and plot the pixel whose scan-line y value closest to the line path

  Assuming we have determined that the pixel at (x_k,y_k) is to be displayed, we next need to decide which pixel to plot in column x_{k+1}.

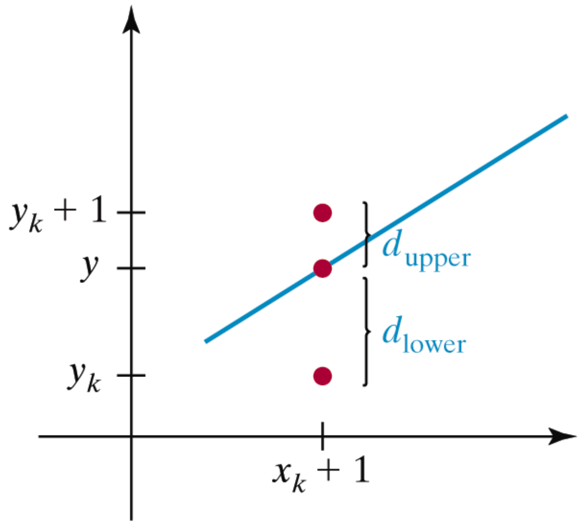


Figure 3-11

Vertical distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

Computer Graphics with Open GL, Third Edition, by Donald Hearn and M.Pauline Baker.
ISBN 0-13-0-15390-7

$y = m(x_k + 1) + b$ ------------ 1

Choices are $(x_k + 1, y_k)$ **and** $(x_k + 1, y_K + 1)$

$d_{lower} = d_1$

$= y - y_k$ substitute $y = m(x_k + 1) + b$

$= m(x_k + 1) + b - y_k$

$= mx_k + m + b - y_k$

$d_{upper} = d_2$

$= (y_k + 1) - y$ substitute $y = m(x_k + 1) + b$

$= y_k + 1 - [m(x_k + 1) + b]$

$= y_k + 1 - [mx_k + m + b]$

$= y_k + 1 - mx_k - m - b$

$$d1 - d2 = [mx_k + m + b - y_k] - [y_k + 1 - mx_k - m - b]$$
$$= mx_k + m + b - y_k - y_k - 1 + mx_k + m + b$$
$$= 2mx_k + 2m + 2b - 2y_k - 1$$
$$= 2m(x_k + 1) - 2y_k + 2b - 1 \qquad \text{-------} \; 2$$

A decision parameter p_k for the k^{th} step in the line algorithm can be obtained by rearranging above equation so that it involves only *integer calculations*

substitute m = $\Delta y / \Delta x$ ***in above eqn 2***

$$d1 - d2 = 2\,\Delta y / \Delta x\,(x_k + 1) - 2y_k + 2b - 1$$
$$\Delta x\,(d1 - d2) = 2\,\Delta y\,(x_k + 1) - \Delta x\,(2y_k) + \Delta x\,(2b - 1)$$
$$= 2\,\Delta y\,x_k + 2\,\Delta y - 2\Delta x\,y_k + \Delta x\,(2b - 1)$$
$$= 2\,\Delta y\,x_k - 2\Delta x\,y_k + 2\,\Delta y + \Delta x\,(2b - 1)$$

$$P_k = \Delta x\,(d_1 - d_2) = 2\Delta y x_k - 2\Delta x y_k + c \quad \text{------------} \; 3$$

where $c = 2\Delta y + \Delta x\,(2b - 1)$

- **The sign of P_k is the same as the sign of $d_1 - d_2$, since $\Delta x > 0$.**
  Parameter c **is a constant and has the value $2\Delta y + \Delta x(2b-1)$**
  (independent of pixel position)

- **If *pixel at* y_k is closer to line-path than pixel at $y_k + 1$**
  (i.e, if $d_1 < d_2$***)*** **then p_k is negative. We plot lower pixel in such a case.**
  Otherwise , upper pixel will be plotted.

- **At step *k + 1*, the decision parameter can be evaluated as, from eqn 3**

$$p_{k+1} = 2\Delta y x_{k+1} - 2\Delta x y_{k+1} + c$$

- **Taking the difference of p_{k+1} and p_k we get the following.**

$$p_{k+1} - p_k = [2\Delta y x_{k+1} - 2\Delta x y_{k+1} + c] - [2\Delta y x_k - 2\Delta x y_k + c]$$

$$= 2\Delta y x_{k+1} - 2\Delta x y_{k+1} + c - 2\Delta y x_k + 2\Delta x y_k - c$$

$$= 2\Delta y (x_{k+1} - x_k) - 2\Delta x (y_{k+1} - y_k)$$

but $x_{k+1} = x_k + 1$

$x_{k+1} - x_k = 1$ **substitute in above eqn**

$$= 2\Delta y (1) - 2\Delta x (y_{k+1} - y_k)$$

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

Where the term ($y_{k+1} - y_k$) is either 0 or 1, depending on the sign of *parameter* p_k

$$p_{k+1} = p_k + 2\Delta y - 2\,\Delta x(0)$$

$$p_{k+1} = p_k + 2\Delta y$$

$$p_{k+1} = p_k + 2\Delta y - 2\,\Delta x(1)$$

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

$$P_k = 2\Delta y x_k - 2\Delta x y_k + c \quad \text{-------------} \; 3$$

- **The first parameter p_0 is directly computed from eqn 3**

$$p_0 = 2\,\Delta y x_0 - 2\,\Delta x y_0 + c$$

$$= 2\Delta y x_0 - 2\Delta x y_0 + 2\Delta y + \Delta x\,(2b - 1)$$

$$= 2\Delta y x_0 - 2\Delta x y_0 + 2\Delta y + 2\Delta x b - \Delta x$$

- **Since (x_0, y_0) satisfies the line equation , we also have**

substitute in above eqn $y_0 = \Delta y / \Delta x * x_0 + b$

$$= 2\Delta y x_0 - 2\Delta x(\Delta y / \Delta x * x_0 + b\,) + 2\Delta y + 2\Delta x b - \Delta x$$

$$= 2\Delta y x_0 - 2\Delta y\, x_0 - 2\Delta x\, b + 2\Delta y + 2\Delta x\, b - \Delta x$$

$$p_0 = 2\Delta y - \Delta x$$

The constants $2\Delta y$ and $2\Delta y - 2\Delta x$ are calculated once for each time to be scan converted

Bresenham's Line Algorithm

- **So, the arithmetic involves only integer addition and subtraction of 2 constants**

1. Input the two end points and store the left end point in (x_0,y_0)

2. Load (x_0,y_0) into the frame buffer ***(plot the first point)***

3. Calculate the constants Δx, Δy, 2Δy and 2Δy-2Δx and obtain the starting value for the decision parameter as

$$p_0 = 2\Delta y - \Delta x$$

Bresenham's Line Algorithm

4. At each x_k along the line, starting at k=0, perform the following test:

If $p_k < 0$, the next point is (x_k+1, y_k) and

$$p_{k+1} = p_k + 2\Delta y$$

Otherwise

Point to plot is (x_k+1, y_k+1)

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat (above step 4) Δx-1 times

Bresenham algorithm

```
void BresenhamLine(int  x0,int y0, int xEnd, int yEnd)
{
int dx=fabs(xEnd-x0);
int dy=fabs(yEnd-y0);
int p=2*dy-dx;
int twoDy=2*dy;
int twoDyMinusDx=2*(dy-dx);
 int x, y;
 if(x0 > xEnd) {
 x = xEnd; y = yEnd;
 xEnd = x0; }
 else { x = x0; y = y0;}
```

```
 setPixel(x,y);
while (x < xEnd) {
     x++;
      if (p < 0)
     {
       p+=twoDy;
       }
      else {
        y++;
        p += twoMinusDx;
        }
      setPixel(x,y);
   }
   }
```

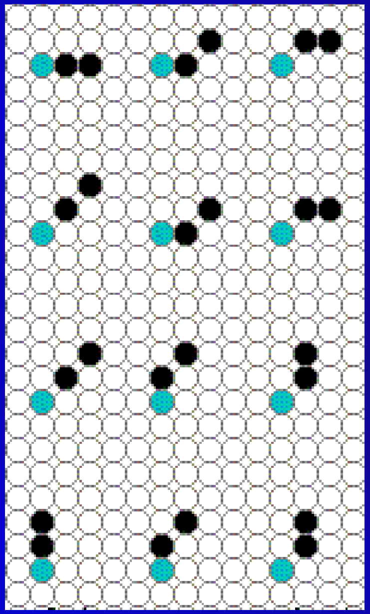
Bresenham was (not) the end!

2-step algorithm by Xiaolin Wu:

(see Graphics Gems 1, by Brian Wyvill)

Treat line drawing as an automaton , or finite state machine, ie. looking at next two pixels of a line, easy to see that only a finite set of possibilities exist.

The 2-step algorithm exploits symmetry by simultaneously drawing from both ends towards the midpoint.



Two-step Algorithm

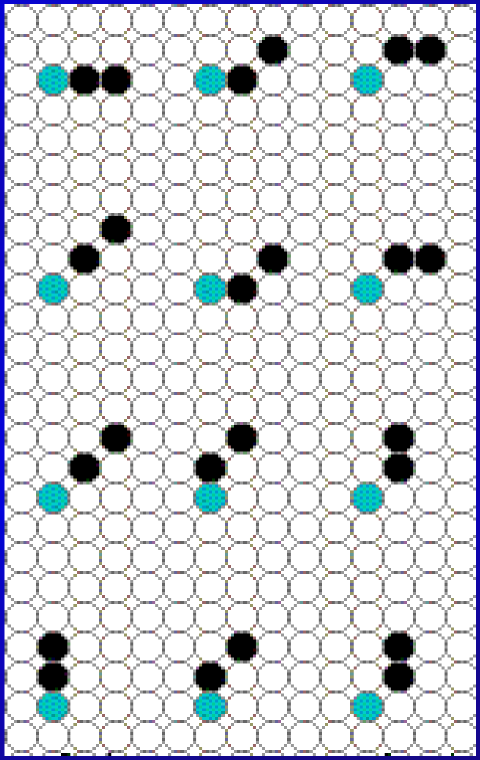
Possible positions of next two pixels dependent on slope – current pixel in blue:

Slope between 0 and ½

Slope between ½ and 1

Slope between 1 and 2

Slope greater than 2



Summary of line drawing so far.

- Explicit form of line
  - Inefficient, difficult to control.
- Parametric form of line.
  - Express line in terms of parameter *t*
  - DDA algorithm
- Implicit form of line
  - Only need to test for ‘side’ of line.
  - Bresenham algorithm.
  - Can also draw circles.