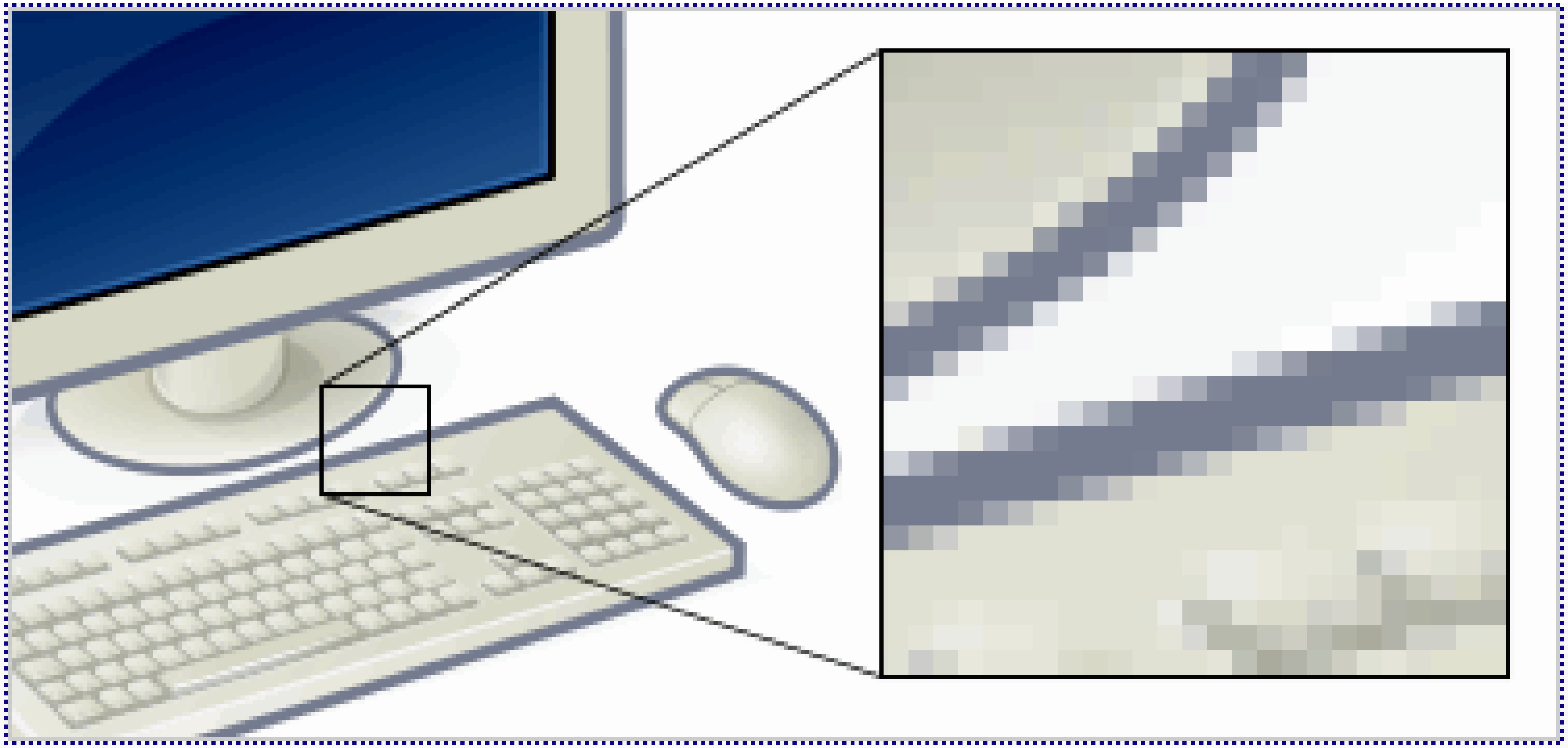
Computer Graphics
Lecture-1

Sudipto Chaki
Lecturer, CSE, BUBT
sudipto@bubt.edu.bd

Pixel

- In digital image processing, a **pixel**, or **pel** or **picture element** is a physical point in an image.
- It is the smallest addressable element in a display device; so it is t he smallest controllable element of a picture represented on the sc reen.
- The address of a pixel corresponds to its physical coordinates.
- Each pixel is a sample of an original image; more samples typicall y provide more accurate representations of the original. So picture quality is directly proportional to the picture resolution.

Pixel…

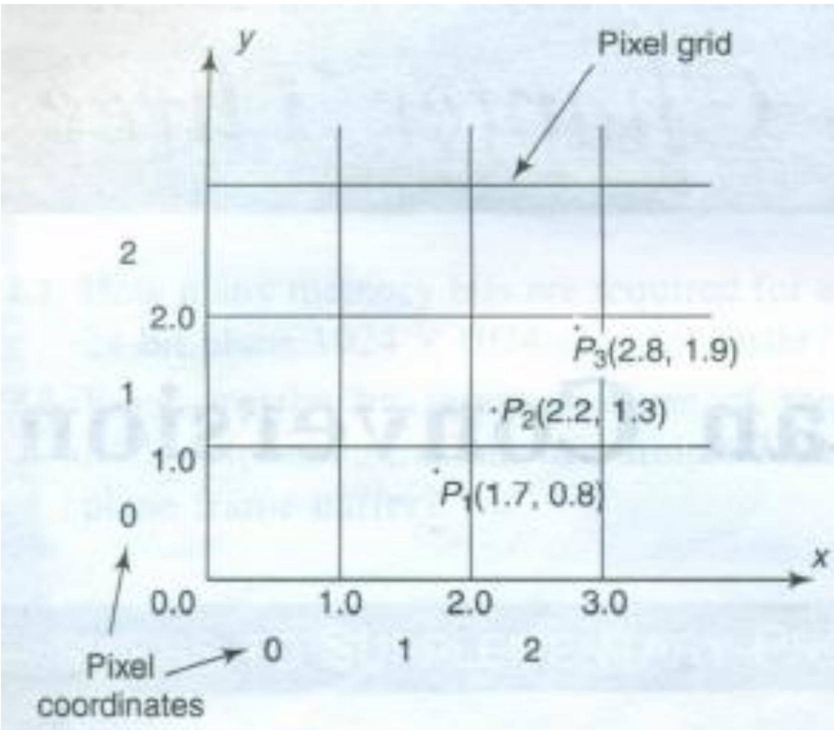


This example shows an image with a portion greatly enlarged, in which the individual pixels are rendered as small squares and can easily be seen.

Scan Conversion

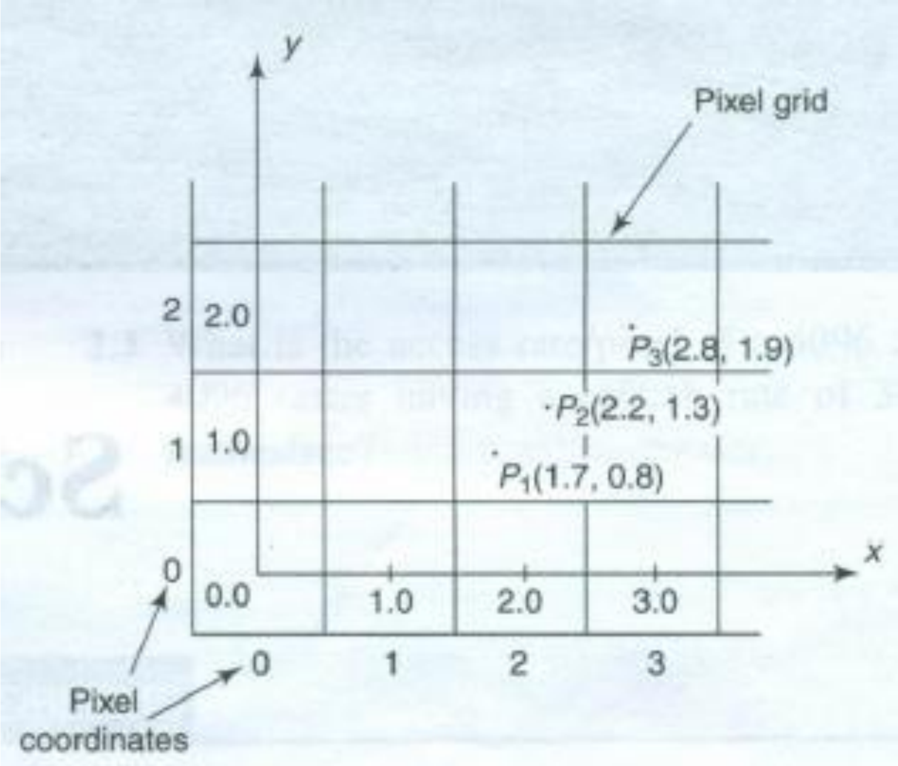
- **Rasterisation** (or **rasterization**) is the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots) for outp ut on a video display or printer, or for storage in a bitma p file format.
- This is also known as **scan conversion**.

Scan Conversion of a Point



- A point (x, y) within an image area, scan converted to a pixel at location (x', y').
- x' = Floor(x) and y' = Floor(y).
- All points satisfying $x' \leq x < x' + 1$ and $y' \leq y < y' + 1$ are mapped to pixel (x', y').
- Point P_1(1.7, 0.8) is represented by pixel (1, 0) and points $P_2(2.2, 1.3)$ and $P_3(2.8, 1.9)$ are both represented by pixel (2, 1).

Scan Conversion of a Point…

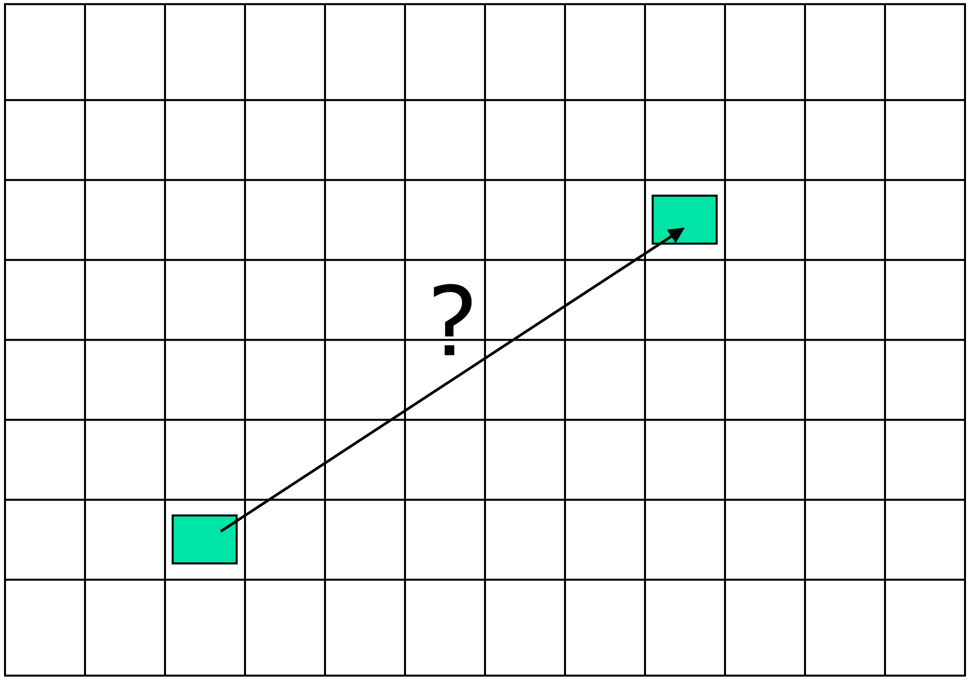


- Another approach is to align the integer values in the co-ordinate system for (*x*, *y*) with the pixel co-ordinates.
- Here *x*' = Floor(*x* + 0.5) and *y*' = Floor(*y* + 0.5)
- Points P_1 and P_2 both are now represented by pixel (2, 1) and P_3 by pixel (3, 2).

Line drawing algorithm

- Need algorithm to figure out which intermediate pixels a re on line path
- Pixel (*x*, *y*) values constrained to integer values
- Actual computed intermediate line values may be floats
- Rounding may be required. Computed point (10.48, 20.51) rounded to (10, 21)
- Rounded pixel value is off actual line path (jaggy!!)
- Sloped lines end up having jaggies
- Vertical, horizontal lines, no jaggies

Line Drawing Algorithm



Line: (3,2) -> (9,6)

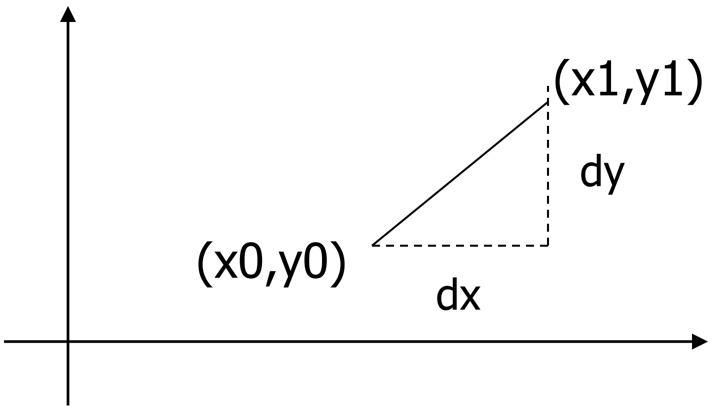
Which intermediate pixels to turn on?

Line Drawing Algorithm…

- Slope-intercept line equation
  - y = mx + b
  - Given two end points (x0,y0), (x1, y1), how to compute m and b?

$$m = \frac{dy}{dx} = \frac{y1 - y0}{x1 - x0}$$

$$b = y0 - m * x0$$



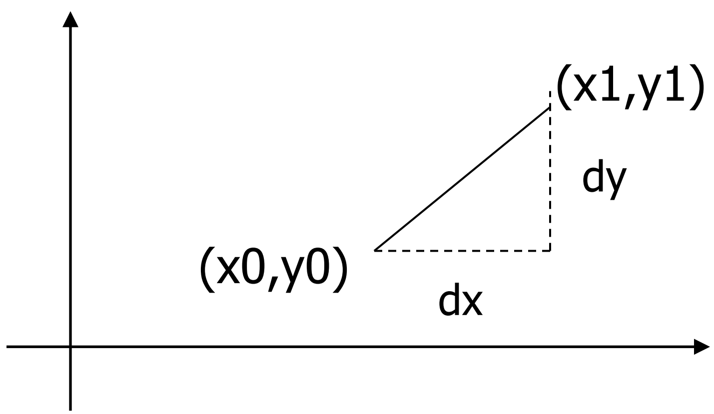
Line Drawing Algorithm…

- Numerical example of finding slope m:
- (Ax, Ay) = (23, 41), (Bx, By) = (125, 96)

$$m = \frac{By - Ay}{Bx - Ax} = \frac{96 - 41}{125 - 23} = \frac{55}{102} = 0.5392$$

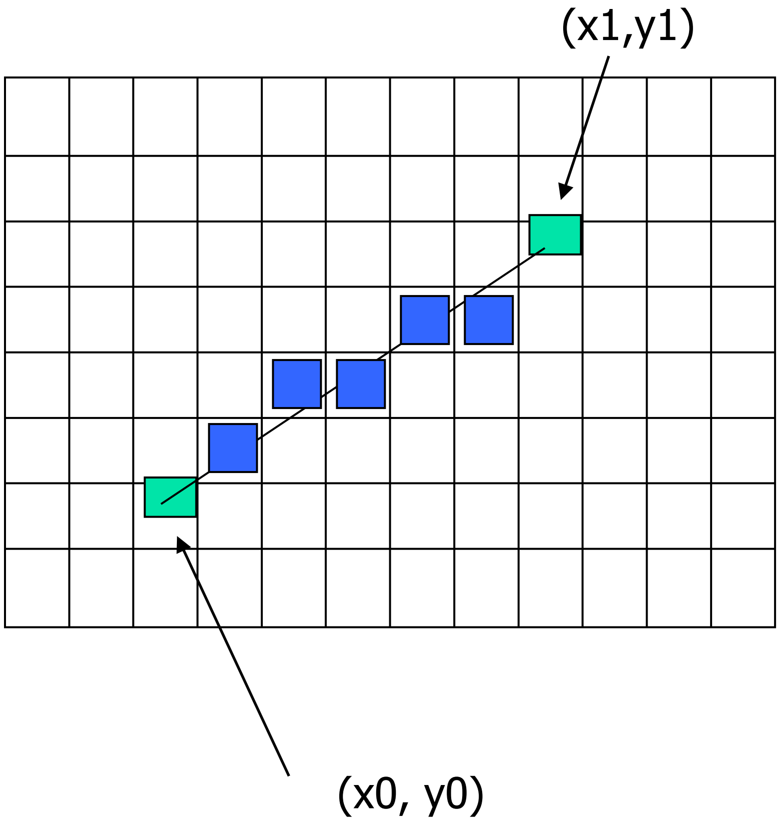
Digital Differential Analyzer (DDA): Line Drawing Algorithm

- Walk through the line, starting at (x0,y0)
- Constrain x, y increments to values in [0,1] range
- Case a: x is incrementing faster (m < 1)
  - Step in x=1 increments, compute and round y
- Case b: y is incrementing faster (m > 1)
  - Step in y=1 increments, compute and round x



DDA Line Drawing Algorithm (Case a: m < 1)

$$y_{k+1} = y_k + m$$



x = x0 y = y0

Illuminate pixel (x, round(y))

x = x0 + 1 y = y0 + 1 * m

Illuminate pixel (x, round(y))

x = x + 1 y = y + 1 * m

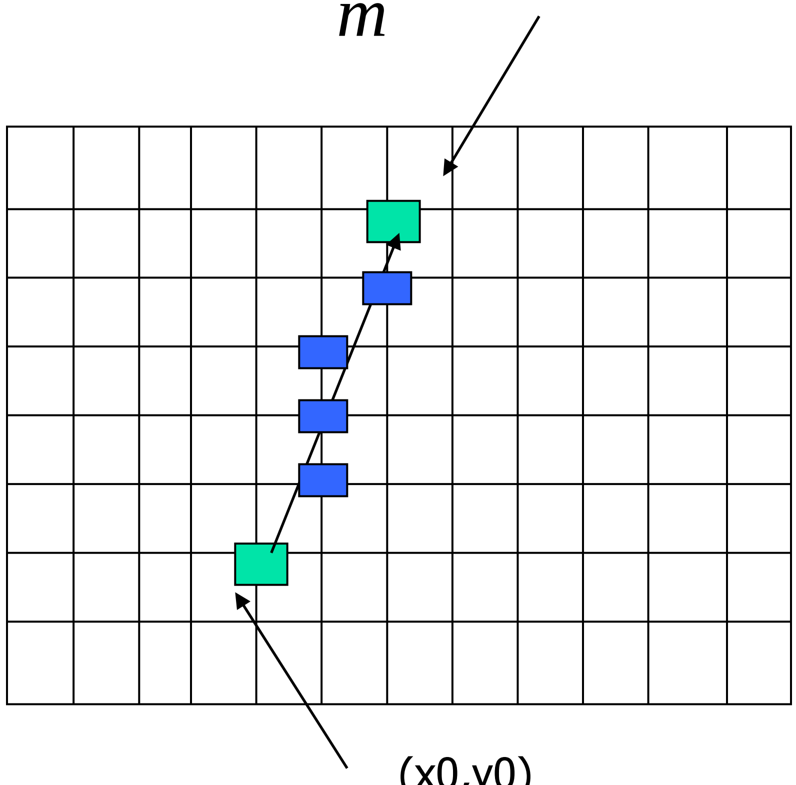
Illuminate pixel (x, round(y))

…

Until x == x1

DDA Line Drawing Algorithm (Case b: m > 1)

$$x_{k+1} = x_k + \frac{1}{m}$$



x = x0 y = y0

Illuminate pixel (round(x), y)

y = y0 + 1 x = x0 + 1 * 1/m

Illuminate pixel (round(x), y)

y = y + 1 x = x + 1 /m

Illuminate pixel (round(x), y)

…

Until y == y1

DDA Line Drawing Algorithm Pseudocode

```
compute m;
if m < 1:
{
   float y = y0;       // initial value
   for(int x = x0;x <= x1; x++, y += m)
              setPixel(x, round(y));
}
else // m > 1
{
   float x = x0;       // initial value
   for(int y = y0;y <= y1; y++, x += 1/m)
              setPixel(round(x), y);
}
```

- Note: **`setPixel(x, y)`** writes current color into pixel in column x and row y in frame buffer

DDA Example (Case a: m < 1)

- Suppose we want to draw a line starting at pixel (2,3) and ending at pixel (12,8).
- What are the values of the variables x and y at each timestep?
- What are the pixels colored, according to the DDA algorithm?

t	x	y	R(x)	R(y)
0	2	3	2	3
1	3	3.5	3	4
2	4	4	4	4
3	5	4.5	5	5
4	6	5	6	5
5	7	5.5	7	6
6	8	6	8	6
7	9	6.5	9	7
8	10	7	10	7
9	11	7.5	11	8
10	12	8	12	8

DDA Algorithm Drawbacks

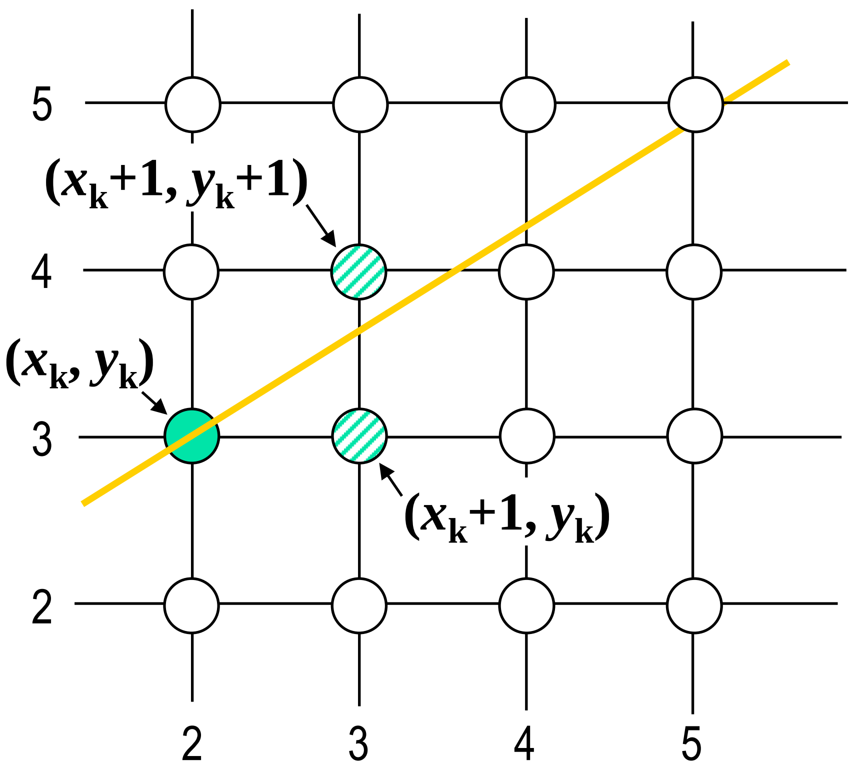
- DDA is the simplest line drawing algorithm
  - Not very efficient
  - Floating point operations and rounding operations are expensiv e.

The Bresenham Line Algorithm

- The Bresenham algorithm is another incremental scan co nversion algorithm
- The big advantage of this algorithm is that it uses only in teger calculations: integer addition, subtraction and multi plication by 2, which can be accomplished by a simple a rithmetic shift operation.

The Big Idea

- Move across the x axis in unit intervals and at each st ep choose between two different y coordinates

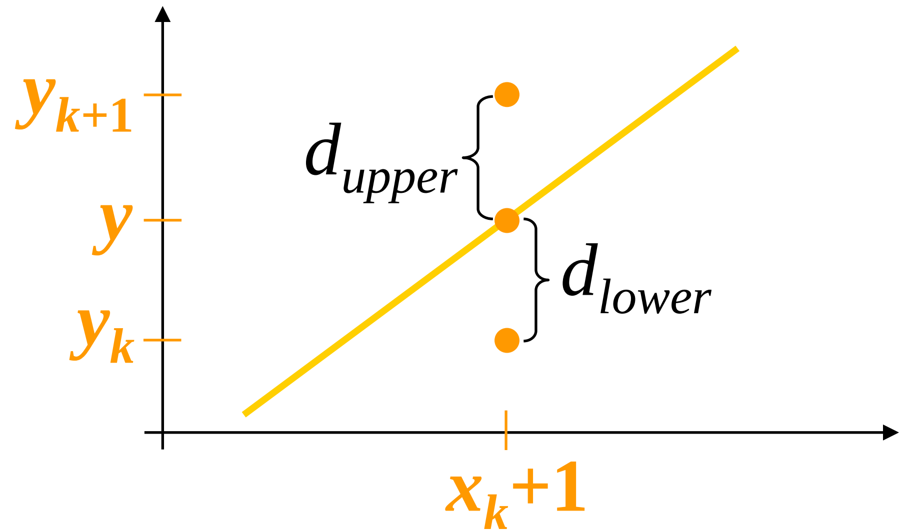


For example, from position (2, 3) we have to choose between (3, 3) and (3, 4)

We would like the point that is closer to the original line

Deriving The Bresenham Line Algorithm

- At sample position x_k+1 the vertical separations from the mathematical line are labelled d_{upper} and d_{lower}



The y coordinate on the mathematical line at x_k+1 is:

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

Deriving The Bresenham Line Algorithm…

- So, d_{upper} and d_{lower} are given as follows:

$$d_{lower} = y - y_k$$
$$= m(x_k + 1) + b - y_k$$

- and:

$$d_{upper} = (y_k + 1) - y$$
$$= y_k + 1 - m(x_k + 1) - b$$

- We can use these to make a simple decision about which pixel is closer to the mathematical line

Deriving The Bresenham Line Algorithm…

- This simple decision is based on the difference between the two pixel positions:

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1$$

- Let's substitute m with $\Delta y / \Delta x$ where Δx and Δy are the differences between the end-points:

$$\Delta x(d_{lower} - d_{upper}) = \Delta x(2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1)$$

$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + 2\Delta y + \Delta x(2b - 1)$$

$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$$

Deriving The Bresenham Line Algorithm…

- So, a decision parameter p_k for the kth step along a line is given by:

$$p_k = \Delta x(d_{lower} - d_{upper})$$
$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$$

- The sign of the decision parameter p_k is the same as that of $d_{lower} - d_{upper}$
- If p_k is negative, then we choose the lower pixel, otherwise we choose the upper pixel

Deriving The Bresenham Line Algorithm…

- Remember coordinate changes occur along the x axis in unit steps so we can do everything with integer cal culations
- At step k+1 the decision parameter is given as:

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

- Subtracting p_k from this we get:

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

Deriving The Bresenham Line Algorithm…

- But, x_{k+1} is the same as $x_k + 1$ so:

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

- where y_{k+1} - y_k is either 0 or 1 depending on the sign of p_k
- The first decision parameter p0 is evaluated at (x0, y0) is given as:

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

The Bresenham Line Algorithm…

BRESENHAM'S LINE DRAWING ALGORITHM
(for $|m| < 1.0$)

1. Input the two line end-points, storing the left end-point in (x_0, y_0)
2. Plot the point (x_0, y_0)
3. Calculate the constants Δx, Δy, $2\Delta y$, and $(2\Delta y - 2\Delta x)$ and get the first value for the decision parameter as:

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

4. At each x_k along the line, starting at $k = 0$, perform the following test. If $p_k < 0$, the next point to plot is (x_k+1, y_k) and:

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

The Bresenham Line Algorithm…

Otherwise, the next point to plot is (x_k+1, y_k+1) and:

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

5. Repeat step 4 $(\Delta x - 1)$ times

- The algorithm and derivation above assumes slopes are less than 1. for other slopes we need to adjust the algorithm slightly

Bresenham’s Line Algorithm (Example)

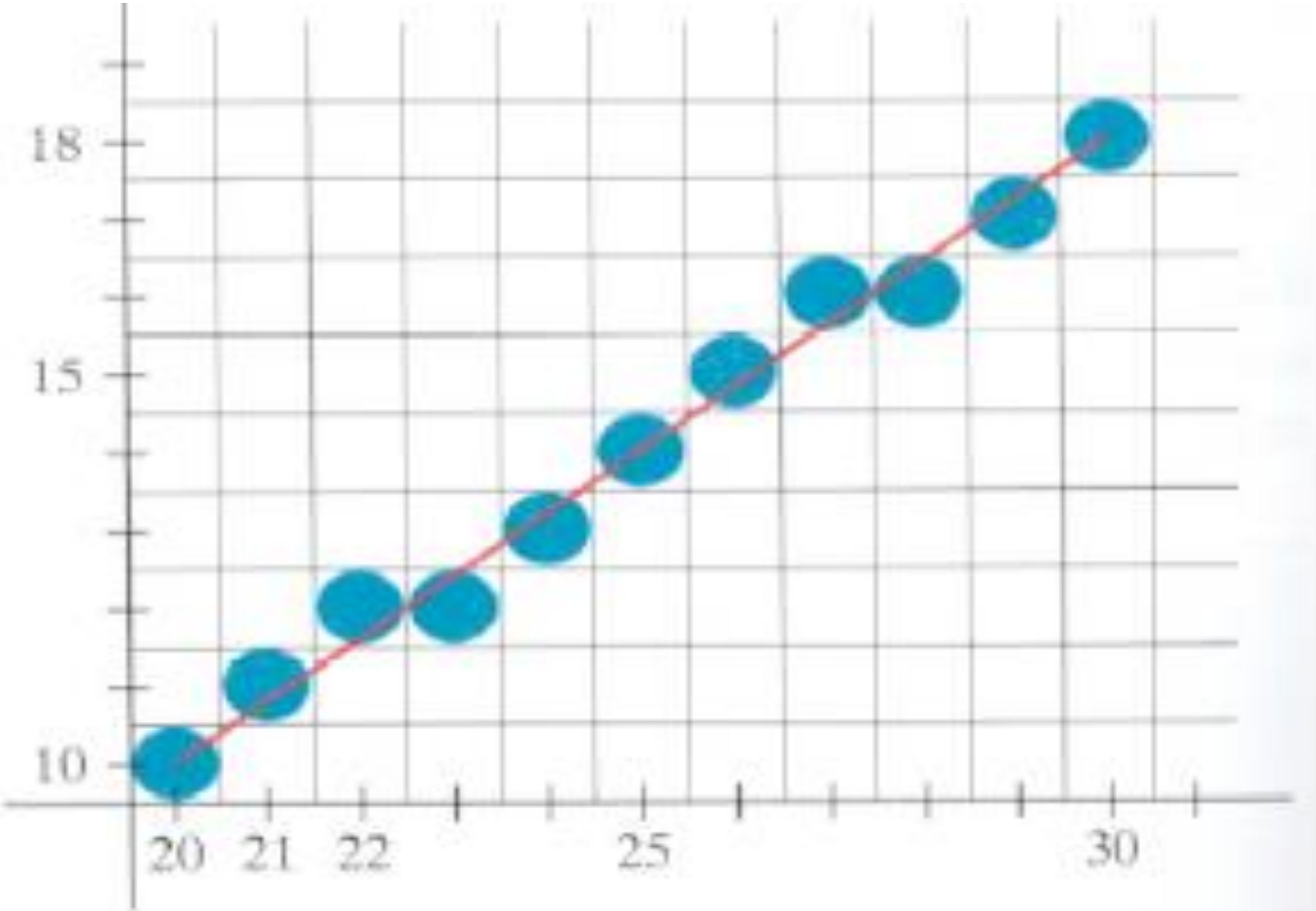
- using Bresenham’s Line-Drawing Algorithm, Digitize the line with endpoints (20,10) and (30,18).

- $\Delta y = 18 - 10 = 8$,
- $\Delta x = 30 - 20 = 10$
- $m = \Delta y / \Delta x = 0.8$
- $2*\Delta y = 16$
- $2*\Delta y - 2*\Delta x = -4$
- plot the first point $(x0, y0) = (20, 10)$
- $p0 = 2 * \Delta y - \Delta x = 2 * 8 - 10 = 6$, so the next point is (21, 11)

Example (cont.)

K	P_k	(x_{k+1}, y_{k+1})	K	P_k	(x_{k+1}, y_{k+1})
0	6	(21,11)	5	6	(26,15)
1	2	(22,12)	6	2	(27,16)
2	-2	(23,12)	7	-2	(28,16)
3	14	(24,13)	8	14	(29,17)
4	10	(25,14)	9	10	(30,18)

Example (cont.)



Bresenham's Line Algorithm (cont.)

- Notice that bresenham's algorithm works on lines with slope in range $0 < m < 1$.
- We draw from left to right.
- To draw lines with slope > 1, interchange the roles of x and y directions.

Code (0 < slope < 1)

```
Bresenham ( int xA, yA, xB, yB) {
    int d, dx, dy, xi, yi;
    int incE, incNE;

    dx = xB - xA;
    dy = yB - yA;
    incE = dy << 1;                         // Q
    incNE = incE - dx << 1;                 // Q + R
    d = incE - dx;                          // initial d = Q + R/2
    xi = xA; yi = yA;
    writePixel(xi, yi);
    while(xi < xB) {
         xi++;
         if(d < 0)                          // choose E
               d += incE;
         else {                             // choose NE
               d += incNE;
               yi++;
          }
          writePixel(xi, yi);
     }}
```

Bresenham Line Algorithm Summary

- The Bresenham line algorithm has the following advantages:
  - An fast incremental algorithm
  - Uses only integer calculations
- Comparing this to the DDA algorithm, DDA has the following problems:
  - Accumulation of round-off errors can make the pixel at end line drift away from what was intended
  - The rounding operations and floating point arithmetic involved are time consuming

A Simple Circle Drawing Algorithm

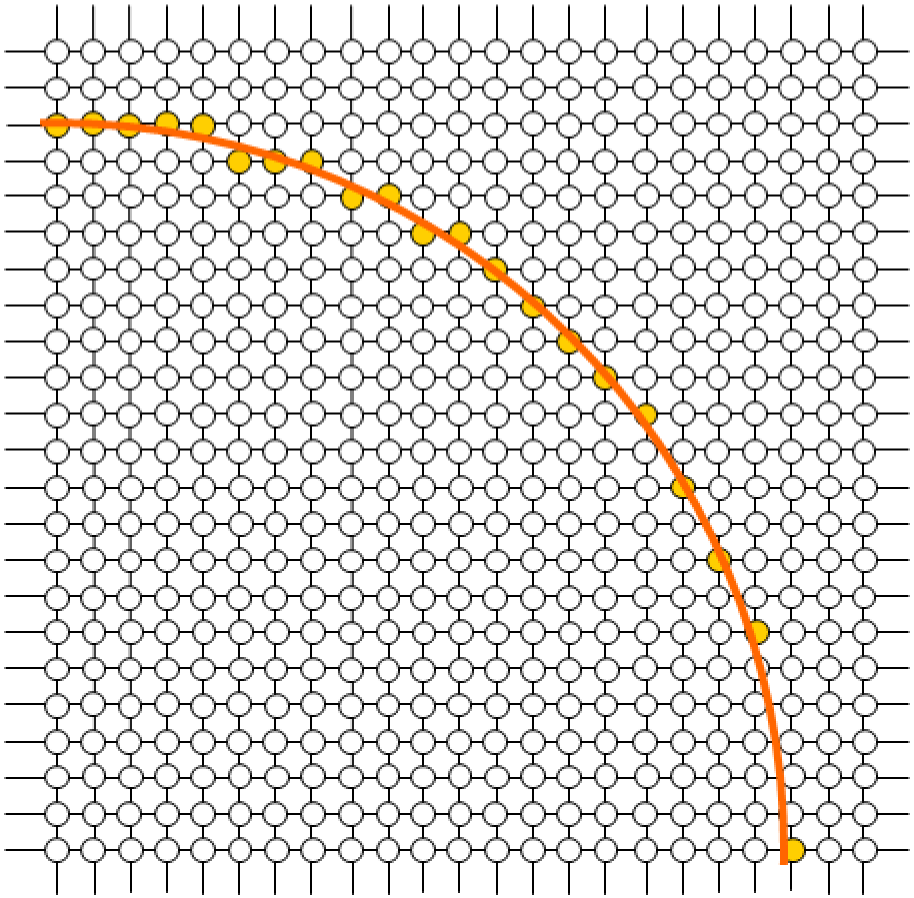
- The equation for a circle is:

$$x^2 + y^2 = r^2$$

- where r is the radius of the circle
- So, we can write a simple circle drawing algorithm by so lving the equation for y at unit x intervals using:

$$y = \pm\sqrt{r^2 - x^2}$$

A Simple Circle Drawing Algorithm (cont…)



$$y_0 = \sqrt{20^2 - 0^2} \approx 20$$

$$y_1 = \sqrt{20^2 - 1^2} \approx 20$$

$$y_2 = \sqrt{20^2 - 2^2} \approx 20$$

⋮

$$y_{19} = \sqrt{20^2 - 19^2} \approx 6$$

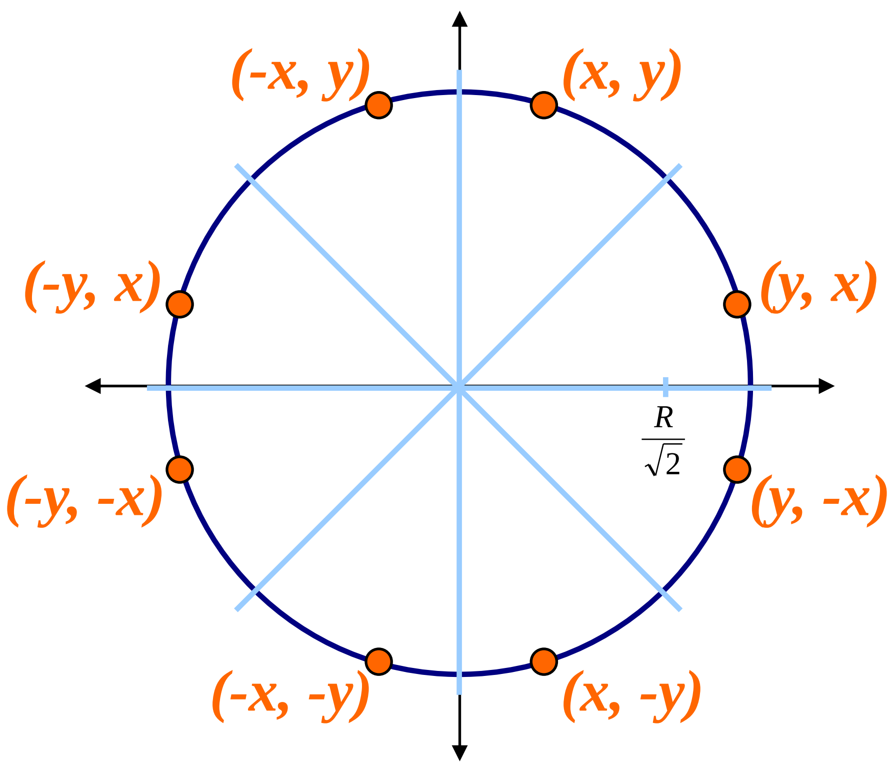
$$y_{20} = \sqrt{20^2 - 20^2} \approx 0$$

A Simple Circle Drawing Algorithm (cont…)

- However, unsurprisingly this is not a brilliant solution!
- Firstly, the resulting circle has large gaps where the slop e approaches the vertical
- Secondly, the calculations are not very efficient
  - The square (multiply) operations
  - The square root operation – try really hard to avoid these!
- We need a more efficient, more accurate solution

Eight-Way Symmetry

- The first thing we can notice to make our circle drawing al gorithm more efficient is that circles centred at (*0*, *0*) have *eight-way symmetry*

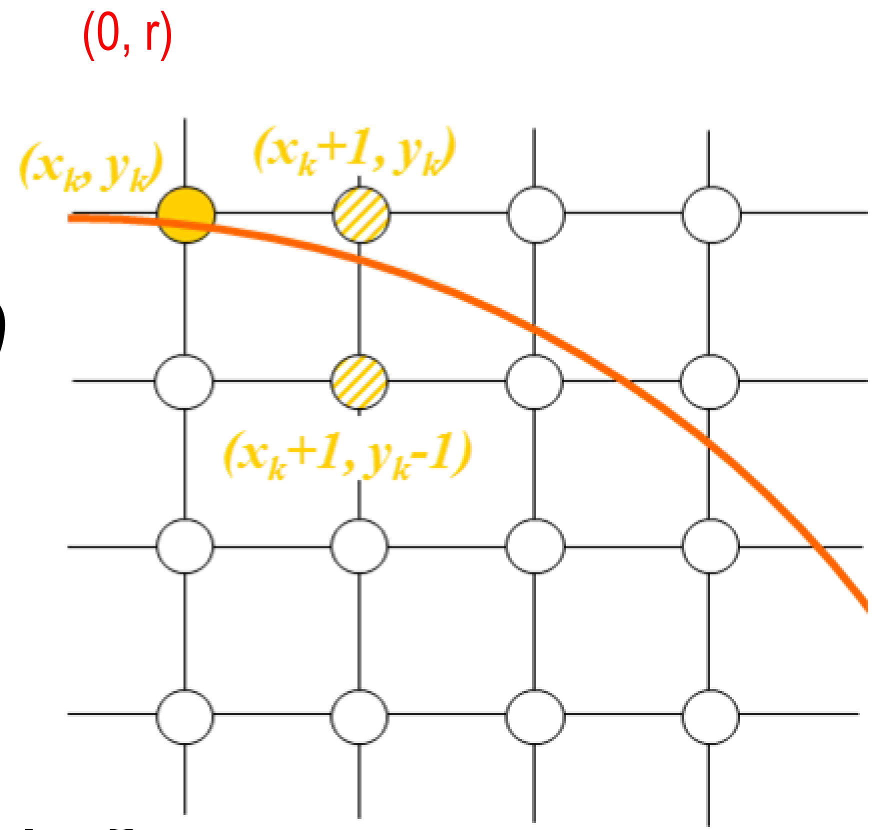


Mid-Point Circle Algorithm

- Similarly to the case with lines, there is an incremental algorithm for drawing circles – the *mid-point circle algorithm*
- In the mid-point circle algorithm we use eight-way symmetry so only ever calculate the points for the top right eighth of a circle, and then use symmetry to get the rest of the points

Mid-Point Circle Algorithm (cont…)

- Assume that we have just plotted point (x_k, y_k)
- The next point is a choice between (x_k+1, y_k) and (x_k+1, y_k-1)
- We would like to choose the point that is nearest to the actual circle
- So how do we make this choice?



Mid-Point Circle Algorithm (cont…)

- Let's re-jig the equation of the circle slightly to give us:

$$f_{circ}(x, y) = x^2 + y^2 - r^2$$

- The equation evaluates as follows:

$$f_{circ}(x, y) \begin{cases} < 0, \text{ if } (x, y) \text{ is inside the circle boundary} \\ = 0, \text{ if } (x, y) \text{ is on the circle boundary} \\ > 0, \text{ if } (x, y) \text{ is outside the circle boundary} \end{cases}$$

- By evaluating this function at the midpoint between the candidate pixels we can make our decision

Mid-Point Circle Algorithm (cont…)

- Assuming we have just plotted the pixel at (x_k, y_k) so we need to choose between (x_k+1, y_k) and (x_k+1, y_k-1)
- Our decision variable can be defined as:

$$p_k = f_{circ}(x_k + 1, y_k - \tfrac{1}{2})$$

$$= (x_k + 1)^2 + (y_k - \tfrac{1}{2})^2 - r^2$$

- If $p_k < 0$ the midpoint is inside the circle and and the pixel at y_k is closer to the circle
- Otherwise the midpoint is outside and y_k-1 is closer

Mid-Point Circle Algorithm (cont…)

- To ensure things are as efficient as possible we can do al l of our calculations incrementally
- First consider:

$$p_{k+1} = f_{circ}\left(x_{k+1}+1, y_{k+1}-\tfrac{1}{2}\right)$$

$$=[(x_k+1)+1]^2+\left(y_{k+1}-\tfrac{1}{2}\right)^2-r^2$$

- or:

$$p_{k+1} = p_k + 2(x_k+1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

- where y_{k+1} is either y_k or y_k-1 depending on the sign of p_k

Mid-Point Circle Algorithm (cont…)

- The first decision variable is given as:

$$p_0 = f_{circ}(1, r - \tfrac{1}{2})$$
$$= 1 + (r - \tfrac{1}{2})^2 - r^2$$
$$= \tfrac{5}{4} - r$$

- Then if $p_k < 0$ then the next decision variable is given as :

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

- If $p_k > 0$ then the decision variable is:

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_k + 1$$

Mid-point Circle Algorithm - Steps

1. Input radius **r** and circle center (x_c, y_c). set the first point $(x_0, y_0) = (0, r)$.

2. Calculate the initial value of the decision parameter as $p_0 = 1 - r$.

   $(p_0 = 5/4 - r \cong 1 - r)$

3. If $p_k < 0$,

   plot $(x_k + 1, y_k)$ and $p_{k+1} = p_k + 2x_{k+1} + 1$,

   Otherwise,

   plot $(x_k + 1, y_k - 1)$ and $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$,

   where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$.

Mid-point Circle Algorithm - Steps

4. Determine symmetry points on the other seven octants.

5. Move each calculated pixel position (x, y) onto the circular path c entered on (x_c, y_c) and plot the coordinate values: $\boldsymbol{x = x + x_c}$ **,** $\boldsymbol{y = y + y_c}$

6. Repeat steps 3 though 5 until $\boldsymbol{x \geq y}$.

7. For all points, add the center point (x_c, y_c)

Mid-point Circle Algorithm - Steps

- Now we drew a part from circle, to draw a complete circle, we must plot the other points.
- We have $(x_c + x , y_c + y)$, the other points are:
  - $(x_c - x , y_c + y)$
  - $(x_c + x , y_c - y)$
  - $(x_c - x , y_c - y)$
  - $(x_c + y , y_c + x)$
  - $(x_c - y , y_c + x)$
  - $(x_c + y , y_c - x)$
  - $(x_c - y , y_c - x)$

Mid-point circle algorithm (Example)

- Given a circle radius r = 10, demonstrate the midpoint circle algorithm by determining positions along the circle octant in the first quadrant from x = 0 to x = y.

<u>Solution:</u>

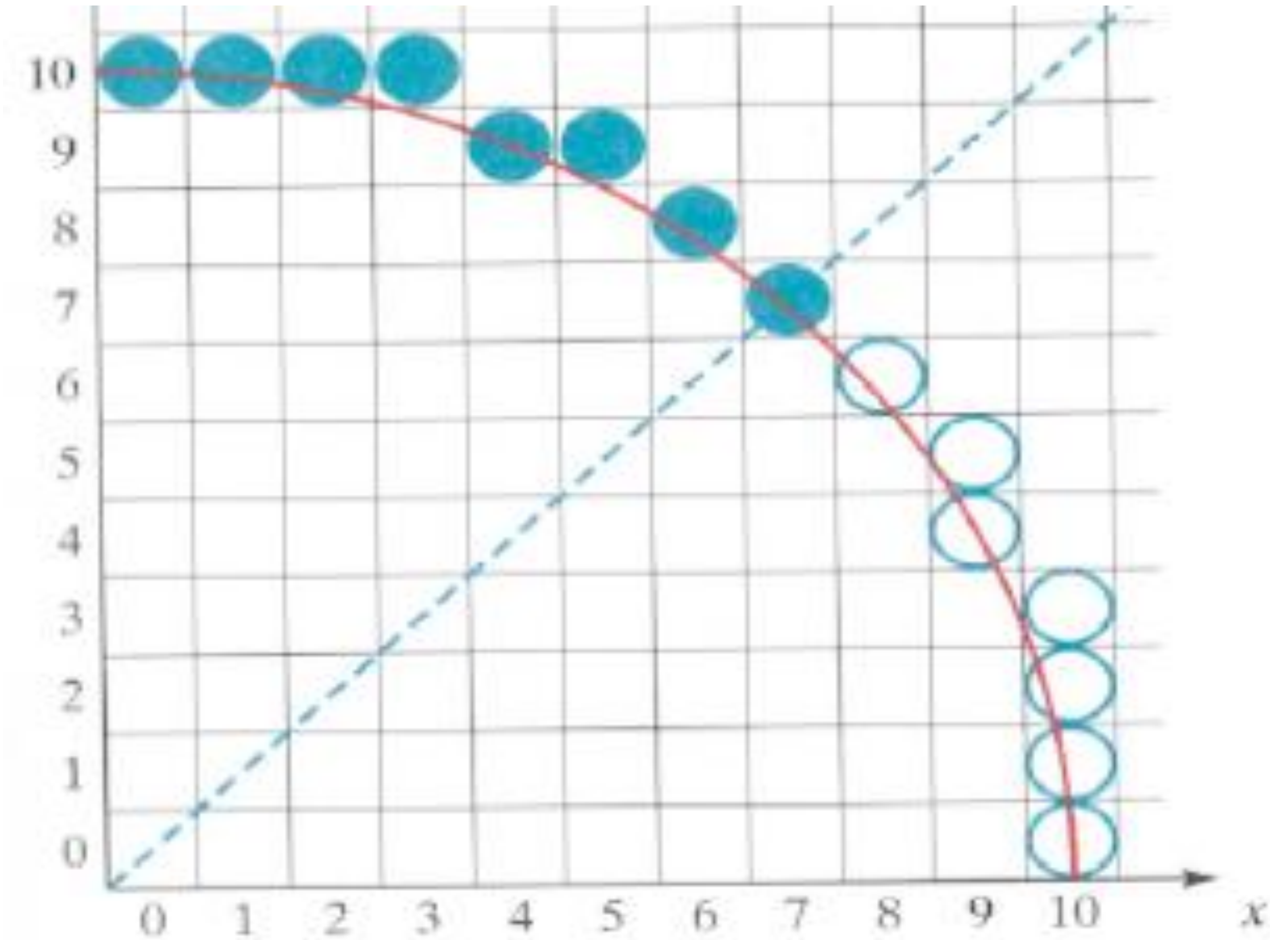
- $p_0 = 1 - r = -9$
- Plot the initial point $(x_0, y_0) = (0, 10)$,

- $2x_0 = 0$ and $2y_0 = 20$.
- Successive decision parameter values and positions along the circle path are calculated using the midpoint method as appear in the next table:

Mid-point circle algorithm (Example)

K	P_k	(x_{k+1}, y_{k+1})	$2\,x_{k+1}$	$2\,y_{k+1}$
0	– 9	(1, 10)	2	20
1	– 6	(2, 10)	4	20
2	– 1	(3, 10)	6	20
3	6	(4, 9)	8	18
4	– 3	(5, 9)	10	18
5	8	(6,8)	12	16
6	5	(7,7)	14	14

Mid-point circle algorithm (Example)



Mid-point Circle Algorithm – Example (2)

- Given a circle radius r = 15, demonstrate the midpoint circle algorithm by determining positions along the circle octant in the first quadrant from x = 0 to x = y.

Solution:

- $p_0 = 1 – r = – 14$
- plot the initial point $(x_0 , y_0) = (0, 15)$,
- $2x_0 = 0$ and $2y_0 = 30$.
- Successive decision parameter values and positions along the circle path are calculated using the midpoint method as:

Mid-point Circle Algorithm – Example (2)

K	P_k	(x_{k+1}, y_{k+1})	$2\,x_{k+1}$	$2\,y_{k+1}$
0	– 14	(1, 15)	2	30
1	– 11	(2, 15)	4	30
2	– 6	(3, 15)	6	30
3	1	(4, 14)	8	28
4	– 18	(5, 14)	10	28

Mid-point Circle Algorithm – Example (2)

K	P_k	(x_{k+1}, y_{k+1})	$2\,x_{k+1}$	$2\,y_{k+1}$
5	– 7	(6,14)	12	28
6	6	(7,13)	14	26
7	– 5	(8,13)	16	26
8	12	(9,12)	18	24
9	7	(10,11)	20	22
10	6	(11,10)	22	20