Points and Lines

- Point plotting is done by converting a single coordinate position furnished by an application program into appropriate operations for the output device in use.
- Line drawing is done by calculating intermediate positions along the line path between two specified endpoint positions.
- The output device is then directed to fill in those positions between the end points with some color.
- For some device such as a pen plotter or random scan display, a straight line can be drawn smoothly from one end point to other.
- Digital devices display a straight line segment by plotting discrete points between the two endpoints.
- Discrete coordinate positions along the line path are calculated from the equation of the line.
- For a raster video display, the line intensity is loaded in frame buffer at the corresponding pixel positions.
- Reading from the frame buffer, the video controller then plots the screen pixels.
- Screen locations are referenced with integer values, so plotted positions may only approximate actual line positions between two specified endpoints.
- For example line position of $(12.36, 23.87)$ would be converted to pixel position $(12, 24)$.
- This rounding of coordinate values to integers causes lines to be displayed with a stair step appearance ("the jaggies"), as represented in fig 2.1.

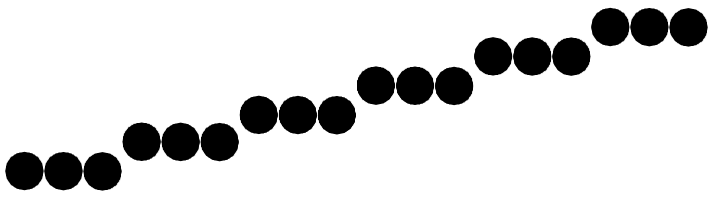

Fig. 2.1: - Stair step effect produced when line is generated as a series of pixel positions.

- The stair step shape is noticeable in low resolution system, and we can improve their appearance somewhat by displaying them on high resolution system.
- More effective techniques for smoothing raster lines are based on adjusting pixel intensities along the line paths.
- For raster graphics device-level algorithms discuss here, object positions are specified directly in integer device coordinates.
- Pixel position will referenced according to scan-line number and column number which is illustrated by following figure.

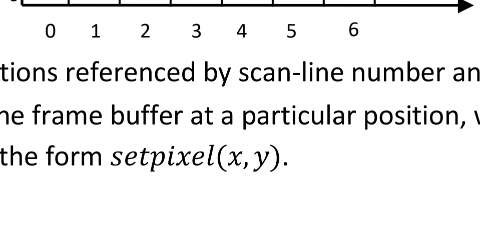

Fig. 2.2: - Pixel positions referenced by scan-line number and column number.

- To load the specified color into the frame buffer at a particular position, we will assume we have available low-level procedure of the form $setpixel(x, y)$.

- Similarly for retrieve the current frame buffer intensity we assume to have procedure $getpixel(x, y)$.

Line Drawing Algorithms

- The Cartesian slop-intercept equation for a straight line is "$y = mx + b$" with 'm' representing slop and 'b' as the intercept.
- The two endpoints of the line are given which are say $(x1, y1)$ and $(x2,y2)$.

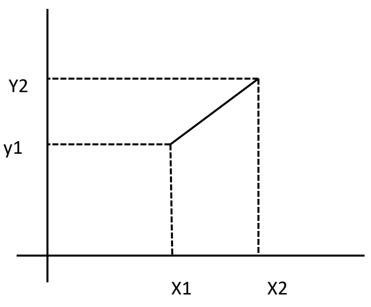

Fig. 2.3: - Line path between endpoint positions.

- We can determine values for the slope m by equation:
  $m = (y2 - y1)/(x2 - x1)$
- We can determine values for the intercept b by equation:
  $b = y1 - m * x1$
- For the given interval Δx along a line, we can compute the corresponding y interval Δy as:
  $\Delta y = m * \Delta x$
- Similarly for Δx:
  $\Delta x = \Delta y/m$
- For line with slop $|m| < 1$, Δx can be set proportional to small horizontal deflection voltage and the corresponding vertical deflection voltage is then set proportional to Δy which is calculated from above equation.
- For line with slop $|m| > 1$, Δy can be set proportional to small vertical deflection voltage and the corresponding horizontal deflection voltage is then set proportional to Δx which is calculated from above equation.
- For line with slop $m = 1$, $\Delta x = \Delta y$ and the horizontal and vertical deflection voltages are equal.

DDA Algorithm

- Digital differential analyzer (DDA) is scan conversion line drawing algorithm based on calculating either Δy or Δx using above equation.
- We sample the line at unit intervals in one coordinate and find corresponding integer values nearest the line path for the other coordinate.
- Consider first a line with positive slope and slope is less than or equal to 1:
  We sample at unit x interval ($\Delta x = 1$) and calculate each successive y value as follow:
  $y = m * x + b$
  $y_k = m * (x + 1) + b$
  In general $y_k = m * (x + k) + b$, &
  $y_{k+1} = m * (x + k + 1) + b$
  Now write this equation in form:
  $y_{k+1} - y_k = (m * (x + k + 1) + b) - (m * (x + k) + b)$
  $y_{k+1} = y_k + m$
  So that it is computed fast in computer as addition is fast compare to multiplication.

- In above equation k takes integer values starting from 1 and increase by 1 until the final endpoint is reached.
- As m can be any real number between 0 and 1, the calculated y values must be rounded to the nearest integer.
- Consider a case for a line with a positive slope greater than 1:
  We change the role of x and y that is sample at unit y intervals ($\Delta y = 1$) and calculate each succeeding x value as:
  $x = (y - b)/m$
  $x_1 = ((y + 1) - b)/m$
  In general $x_k = ((y + k) - b)/m$, &
  $x_{k+1} = ((y + k + 1) - b)/m$
  Now write this equation in form:
  $x_{k+1} - x_k = (((y + k + 1) - b)/m) - (((y + k) - b)/m)$
  $x_{k+1} = x_k + 1/m$
- Above both equations are based on the assumption that lines are to be processed from left endpoint to the right endpoint.
- If we processed line from right endpoint to left endpoint than:
  If $\Delta x = -1$ equation become:
  $y_{k+1} = y_k - m$
  If $\Delta y = -1$ equation become:
  $x_{k+1} = x_k - 1/m$
- Above calculated equations also used to calculate pixel position along a line with negative slope.
- **Procedure for DDA line algorithm.**

```
Void lineDDA (int xa, int ya, int xb, int yb)
{
    int dx = xb – xa, dy = yb – ya, steps, k;
    float xincrement, yincrement, x = xa, y = ya;
    if (abs(dx)>abs(dy))
    {
            Steps = abs (dx);
    }
    else
    {
            Steps = abs (dy);
    }
    xincrement = dx/(float) steps;
    yincrement = dy/(float) steps;

    setpixel (ROUND (x), ROUND (y));
    for(k=0;k<steps;k++)
    {
            x += xincrement;
            y += yincrement;
            setpixel (ROUND (x), ROUND (y));
    }
}
```

Advantages of DDA algorithm

- It is faster algorithm.
- It is simple algorithm.

Disadvantage of DDA algorithm

- Floating point arithmetic is time consuming.
- Poor end point accuracy.

Bresenham's Line Algorithm

- An accurate and efficient raster line-generating algorithm, developed by Bresenham which scan converts line using only incremental integer calculations that can be modified to display circles and other curves.
- Figure shows section of display screen where straight line segments are to be drawn.

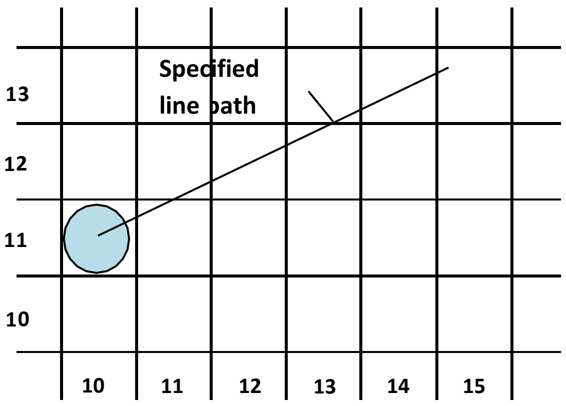

Fig. 2.4: - Section of a display screen where a straight line segment is to be plotted, starting from the pixel at column 10 on scan line 11.

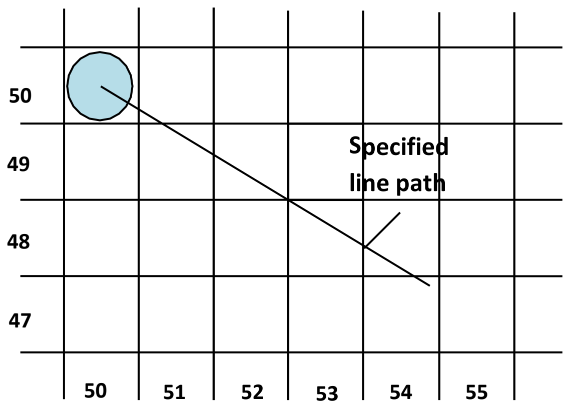

Fig. 2.5: - Section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

- The vertical axes show scan-line positions and the horizontal axes identify pixel column.
- Sampling at unit x intervals in these examples, we need to decide which of two possible pixel position is closer to the line path at each sample step.
- To illustrate bresenham's approach, we first consider the scan-conversion process for lines with positive slope less than 1.
- Pixel positions along a line path are then determined by sampling at unit x intervals.
- Starting from left endpoint (x_0, y_0) of a given line, we step to each successive column and plot the pixel whose scan-line y values is 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 $x_k + 1$.
- Our choices are the pixels at positions $(x_k + 1, y_k)$ and $(x_k + 1, y_k + 1)$.
- Let's see mathematical calculation used to decide which pixel position is light up.
- We know that equation of line is:
  $y = mx + b$
  Now for position $x_k + 1$.
  $y = m(x_k + 1) + b$
- Now calculate distance bet actual line's y value and lower pixel as d_1 and distance bet actual line's y value and upper pixel as d_2.
  $d_1 = y - y_k$

$d_1 = m(x_k + 1) + b - y_k$ ......(1)

$d_2 = (y_k + 1) - y$

$d_2 = (y_k + 1) - m(x_k + 1) - b$ ......(2)

- Now calculate $d_1 - d_2$ from equation (1) and (2).

  $d_1 - d_2 = (y - y_k) - ((y_k + 1) - y)$

  $d_1 - d_2 = \{m(x_k + 1) + b - y_k\} - \{(y_k + 1) - m(x_k + 1) - b\}$

  $d_1 - d_2 = \{mx_k + m + b - y_k\} - \{y_k + 1 - mx_k - m - b\}$

  $d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1$ ......(3)

- Now substitute $m = \Delta y/\Delta x$ in equation (3)

  $d_1 - d_2 = 2\left(\frac{\Delta y}{\Delta x}\right)(x_k + 1) - 2y_k + 2b - 1$ ......(4)

- Now we have decision parameter p_k for k^{th} step in the line algorithm is given by:

  $p_k = \Delta x(d_1 - d_2)$

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

  $p_k = 2\Delta y x_k + 2\Delta y - 2\Delta x y_k + 2\Delta x b - \Delta x$

  $p_k = 2\Delta y x_k - 2\Delta x y_k + 2\Delta y + 2\Delta x b - \Delta x$ ......(5)

  $p_k = 2\Delta y x_k - 2\Delta x y_k + C$ $(Where\ Constant\ C = 2\Delta y + 2\Delta x b - \Delta x)$ ......(6)

- The sign of p_k is the same as the sign of $d_1 - d_2$, since $\Delta x > 0$ for our example.
- Parameter c is constant which is independent of pixel position and will eliminate in the recursive calculation for p_k.
- Now if p_k is negative then we plot the lower pixel otherwise we plot the upper pixel.
- So successive decision parameters using incremental integer calculation as:

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

- Now Subtract pk from p_{k+1}

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

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

  But $x_{k+1} = x_k + 1$, so that $(x_{k+1} - x_k) = 1$

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

- Where the terms $y_{k+1} - y_k$ is either 0 or 1, depends on the sign of parameter p_k.
- This recursive calculation of decision parameters is performed at each integer x position starting at the left coordinate endpoint of the line.
- The first decision parameter p_0 is calculated using equation (5) as first time we need to take constant part into account so:

  $p_k = 2\Delta y x_k - 2\Delta x y_k + 2\Delta y + 2\Delta x b - \Delta x$

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

  Now $Substitute\ b = y_0 - mx_0$

  $p_0 = 2\Delta y x_0 - 2\Delta x y_0 + 2\Delta y + 2\Delta x(y_0 - mx_0) - \Delta x$

  $Now\ Substitute\ m = \Delta y/\Delta x$

  $p_0 = 2\Delta y x_0 - 2\Delta x y_0 + 2\Delta y + 2\Delta x(y_0 - (\Delta y/\Delta x)x_0) - \Delta x$

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

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

- Let's see Bresenham's line drawing algorithm for $|m| < 1$
  1. Input the two line endpoints and store the left endpoint in (x_0, y_0).
  2. Load (x_0, y_0) into the frame buffer; that is, plot the first point.
  3. Calculate constants Δx, Δy, $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting 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
   $$p_{k+1} = p_k + 2\Delta y$$
   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 Δx times.

- Bresenham's algorithm is generalized to lines with arbitrary slope by considering symmetry between the various octants and quadrants of the xy plane.
- For lines with positive slope greater than 1 we interchange the roles of the x and y directions.
- Also we can revise algorithm to draw line from right endpoint to left endpoint, both x and y decrease as we step from right to left.
- When $d_1 - d_2 = 0$ we choose either lower or upper pixel but once we choose lower than for all such case for that line choose lower and if we choose upper the for all such case choose upper.
- For the negative slope the procedure are similar except that now one coordinate decreases as the other increases.
- The special case handle separately. Horizontal line $(\Delta y = 0)$, vertical line $(\Delta x = 0)$ and diagonal line with $|\Delta x| = |\Delta y|$ each can be loaded directly into the frame buffer without processing them through the line plotting algorithm.

Parallel Execution of Line Algorithms

- The line-generating algorithms we have discussed so far determine pixel positions sequentially.
- With parallel computer we can calculate pixel position along a line path simultaneously by dividing work among the various processors available.
- One way to use multiple processors is partitioning existing sequential algorithm into small parts and compute separately.
- Alternatively we can go for other ways to setup the processing so that pixel positions can be calculated efficiently in parallel.
- Important point to be taking into account while devising parallel algorithm is to balance the load among the available processors.
- Given $\boldsymbol{n_p}$ number of processors we can set up parallel Bresenham line algorithm by subdividing the line path into $\boldsymbol{n_p}$ partitions and simultaneously generating line segment in each of the subintervals.
- For a line with slope $0 < m < 1$ and left endpoint coordinate position (x_0, y_0), we partition the line along the positive x direction.
- The distance between beginning x positions of adjacent partitions can be calculated as:
  $\Delta x_p = (\Delta x + n_p - 1)/n_p$
  Were Δx is the width of the line. And value for partition with Δx_p is computed using integer division.
- Numbering the partitions and the processors, as 0, 1, 2, up to $\boldsymbol{n_p - 1}$, we calculate the starting x coordinate for the k^{th} partition as:
  $x_k = x_0 + k\Delta x_p$
- To apply Bresenham's algorithm over the partitions, we need the initial value for the y coordinate and the initial value for the decision parameter in each partition.
- The change Δy_p in the y direction over each partition is calculated from the line slope m and partition width Δx_p:
- $\Delta y_p = m\Delta x_p$

- At the k^{th} partition, the starting y coordinate is then
- $y_k = y_0 + round(k\Delta y_p)$
- The initial decision parameter for Bresenham's algorithm at the start of the k^{th} subinterval is obtained from Equation(6):
  $p_k = 2\Delta y x_k - 2\Delta x y_k + 2\Delta y + 2\Delta x b - \Delta x$
  $p_k = 2\Delta y(x_0 + k\Delta x_p) - 2\Delta x(y_0 + round(k\Delta y_p)) + 2\Delta y + 2\Delta x(y_0 - \frac{\Delta y}{\Delta x} x_0) - \Delta x$
  $p_k = 2\Delta y x_0 - 2\Delta y k\Delta x_p - 2\Delta x y_0 - 2\Delta x round(k\Delta y_p) + 2\Delta y + 2\Delta x y_0 - 2\Delta y x_0 - \Delta x$
  $p_k = 2\Delta y k\Delta x_p - 2\Delta x round(k\Delta y_p) + 2\Delta y - \Delta x$
- Each processor then calculates pixel positions over its assigned subinterval.
- The extension of the parallel Bresenham algorithm to a line with slope greater than 1 is achieved by partitioning the line in the y direction and calculating beginning x values for the positions.
- For negative slopes, we increment coordinate values in one direction and decrement in the other.

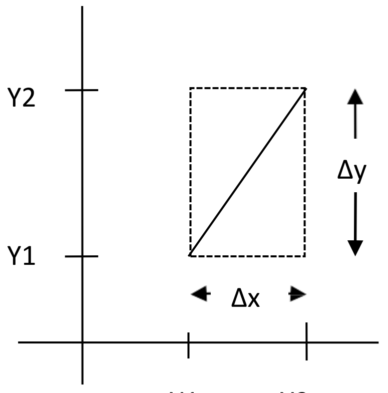

Fig. 2.6: - Bounding box for a line with coordinate extents Δx and Δy.

- Another way to set up parallel algorithms on raster system is to assign each processor to a particular group of screen pixels.
- With sufficient number of processor we can assign each processor to one pixel within some screen region.
- This approach can be adapted to line display by assigning one processor to each of the pixels within the limit of the bounding rectangle and calculating pixel distance from the line path.
- The number of pixels within the bounding rectangle of a line is $\Delta x \times \Delta y$.
- Perpendicular distance d from line to a particular pixel is calculated by:
  $d = Ax + By + C$
  Where
  $A = -\Delta y / linelength$
  $B = -\Delta x / linelength$
  $C = (x_0\Delta y - y_0\Delta x)/linelength$
  With
  $linelength = \sqrt{\Delta x^2 + \Delta y^2}$
- Once the constant A, B, and C have been evaluated for the line each processors need to perform two multiplications and two additions to compute the pixel distance d.
- A pixel is plotted if d is less than a specified line thickness parameter.
- Instead of partitioning the screen into single pixels, we can assign to each processor either a scan line or a column a column of pixels depending on the line slope.

- Each processor calculates line intersection with horizontal row or vertical column of pixels assigned to that processor.
- If vertical column is assign to processor then x is fix and it will calculate y and similarly is horizontal row is assign to processor then y is fix and x will be calculated.
- Such direct methods are slow in sequential machine but we can perform very efficiently using multiple processors.

Circle

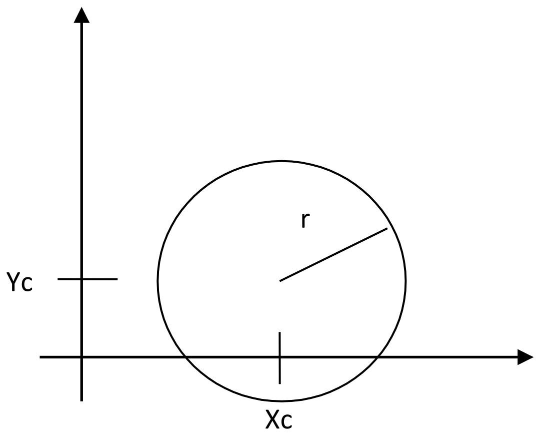

Fig. 2.7: - Circle with center coordinates (x_c, y_c) and radius r.

- A circle is defined as the set of points that are all at a given distance r from a center position say (x_c, y_c).

Properties of Circle

- The distance relationship is expressed by the Pythagorean theorem in Cartesian coordinates as:
  $(x - x_c)^2 + (y - y_c)^2 = r^2$
- We could use this equation to calculate circular boundary points by incrementing 1 in x direction in every steps from $x_c - r$ to $x_c + r$ and calculate corresponding y values at each position as:
  $(x - x_c)^2 + (y - y_c)^2 = r^2$
  $(y - y_c)^2 = r^2 - (x - x_c)^2$
  $(y - y_c) = \pm\sqrt{r^2 - (x_c - x)^2}$
  $y = y_c \pm \sqrt{r^2 - (x_c - x)^2}$
- But this is not best method for generating a circle because it requires more number of calculations which take more time to execute.
- And also spacing between the plotted pixel positions is not uniform as shown in figure below.

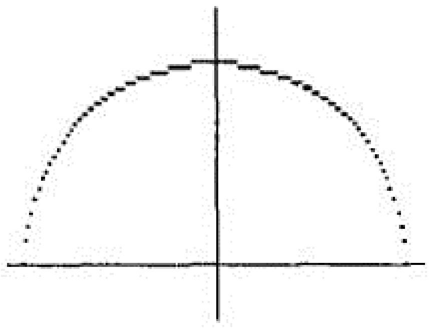

Fig. 2.8: - Positive half of circle showing non uniform spacing bet calculated pixel positions.

- We can adjust spacing by stepping through y values and calculating x values whenever the absolute value of the slop of the circle is greater than 1. But it will increases computation processing requirement.
- Another way to eliminate the non-uniform spacing is to draw circle using polar coordinates 'r' and 'θ'.
- Calculating circle boundary using polar equation is given by pair of equations which is as follows.
  $x = x_c + r\cos\theta$
  $y = y_c + r\sin\theta$
- When display is produce using these equations using fixed angular step size circle is plotted with uniform spacing.
- The step size 'θ' is chosen according to application and display device.
- For a more continuous boundary on a raster display we can set the step size at $1/r$. This plot pixel position that are approximately one unit apart.
- Computation can be reduced by considering symmetry city property of circles. The shape of circle is similar in each quadrant.
- We can obtain pixel position in second quadrant from first quadrant using reflection about y axis and similarly for third and fourth quadrant from second and first respectively using reflection about x axis.
- We can take one step further and note that there is also symmetry between octants. Circle sections in adjacent octant within one quadrant are symmetric with respect to the 45^0 line dividing the two octants.
- This symmetry condition is shown in figure below where point (x, y) on one circle sector is mapped in other seven sector of circle.

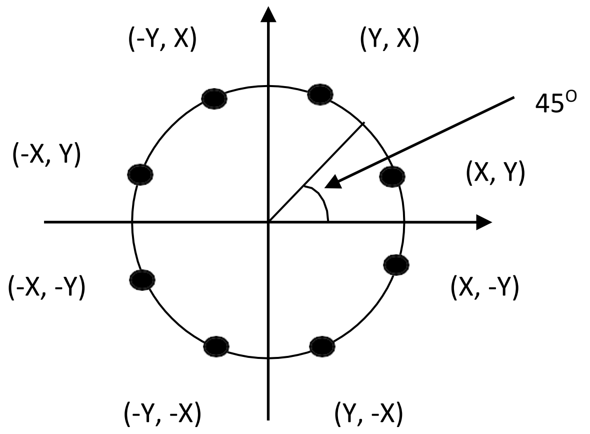

Fig. 2.9: - symmetry of circle.

- Taking advantage of this symmetry property of circle we can generate all pixel position on boundary of circle by calculating only one sector from $x = 0$ to $x = y$.
- Determining pixel position along circumference of circle using any of two equations shown above still required large computation.
- More efficient circle algorithm are based on incremental calculation of decision parameters, as in the Bresenham line algorithm.
- Bresenham's line algorithm can be adapted to circle generation by setting decision parameter for finding closest pixel to the circumference at each sampling step.
- The Cartesian coordinate circle equation is nonlinear so that square root evaluations would be required to compute pixel distance from circular path.
- Bresenham's circle algorithm avoids these square root calculation by comparing the square of the pixel separation distance.

- A method for direct distance comparison to test the midpoint between two pixels to determine if this midpoint is inside or outside the circle boundary.
- This method is easily applied to other conics also.
- Midpoint approach generates same pixel position as generated by bresenham's circle algorithm.
- The error involve in locating pixel positions along any conic section using midpoint test is limited to one-half the pixel separation.

Midpoint Circle Algorithm

- Similar to raster line algorithm we sample at unit interval and determine the closest pixel position to the specified circle path at each step.
- Given radius 'r' and center (x_c, y_c)
- We first setup our algorithm to calculate circular path coordinates for center (0, 0). And then we will transfer calculated pixel position to center (x_c, y_c) by adding x_c to x and y_c to y.
- Along the circle section from $x = 0$ to $x = y$ in the first quadrant, the slope of the curve varies from 0 to -1 so we can step unit step in positive x direction over this octant and use a decision parameter to determine which of the two possible y position is closer to the circular path.
- Position in the other seven octants are then obtain by symmetry.
- For the decision parameter we use the circle function which is:
  $f_{circle}(x, y) = x^2 + y^2 - r^2$
- Any point which is on the boundary is satisfied $f_{circle}(x, y) = 0$ if the point is inside circle function value is negative and if point is outside circle the function value is positive which can be summarize as below.

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

- Above equation we calculate for the mid positions between pixels near the circular path at each sampling step and we setup incremental calculation for this function as we did in the line algorithm.
- Below figure shows the midpoint between the two candidate pixels at sampling position $x_k + 1$.

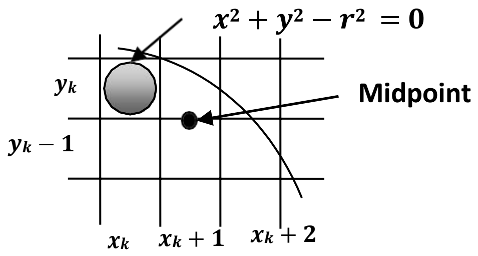

Fig. 2.10: - Midpoint between candidate pixel at sampling position $x_k + 1$ along circle path.

- Assuming we have just plotted the pixel at (x_k, y_k) and next we need to determine whether the pixel at position '$(x_k + 1, y_k)$' or the one at position' $(x_k + 1, y_k - 1)$' is closer to circle boundary.
- So for finding which pixel is more closer using decision parameter evaluated at the midpoint between two candidate pixels as below:

$$p_k = f_{circle}\left(x_k + 1, y_k - \frac{1}{2}\right)$$

$$p_k = (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2$$

- If $p_k < 0$ this midpoint is inside the circle and the pixel on the scan line y_k is closer to circle boundary. Otherwise the midpoint is outside or on the boundary and we select the scan line $y_k - 1$.
- Successive decision parameters are obtain using incremental calculations as follows:

$$p_{k+1} = f_{circle}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right)$$
$$p_{k+1} = [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2$$

- Now we can obtain recursive calculation using equation of p_{k+1} and p_k as follow.

$$p_{k+1} - p_k = \left([(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2\right) - \left((x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2\right)$$
$$p_{k+1} - p_k = (x_k + 1)^2 + 2(x_k + 1) + 1 + y_{k+1}^2 - y_{k+1} + \frac{1}{4} - r^2 - (x_k + 1)^2 - y_k^2 + y_k - \frac{1}{4} + r^2$$
$$p_{k+1} - p_k = 2(x_k + 1) + 1 + y_{k+1}^2 - y_{k+1} - y_k^2 + y_k$$
$$p_{k+1} - p_k = 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$
$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

- In above equation y_{k+1} is either y_k or $y_k - 1$ depending on the sign of the p_k.
- Now we can put $2x_{k+1} = 2x_k + 2$ and when we select $y_{k+1} = y_k - 1$ we can obtain $2y_{k+1} = 2y_k - 2$.
- The initial decision parameter is obtained by evaluating the circle function at the start position $(x_0, y_0) = (0, r)$ as follows.

$$p_0 = f_{circle}\left(0 + 1, r - \frac{1}{2}\right)$$
$$p_0 = 1 + \left(r - \frac{1}{2}\right)^2 - r^2$$
$$p_0 = 1 + r^2 - r + \frac{1}{4} - r^2$$
$$p_0 = \frac{5}{4} - r$$

Algorithm for Midpoint Circle Generation

1. Input radius r and circle center (x_c, y_c), and obtain the first point on the circumference of a circle centered on the origin as
$$(x_0, y_0) = (0, r)$$
2. calculate the initial value of the decision parameter as
$$p_0 = \frac{5}{4} - r$$
3. At each x_k position, starting at $k = 0$, perform the following test:
If $p_k < 0$, the next point along the circle centered on $(0, 0)$ is $(x_k + 1, y_k)$ &
$$p_{k+1} = p_k + 2x_{k+1} + 1$$
Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ &
$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$
Where $2x_{k+1} = 2x_k + 2$, & $2y_{k+1} = 2y_k - 2$.
4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position (x, y) onto the circular path centered on (x_c, y_c) and plot the coordinate values:
$$x = x + x_c,\ y = y + y_c$$
6. Repeat steps 3 through 5 until $x \geq y$.