

Computer Graphics

Lectures

On

Graphics Output Primitives

Muhammad Sarfraz

Sarfraz@kfupm.edu.sa

*Information & Computer Science Department
King Fahd University of Petroleum & Minerals
Dhahran, Saudi Arabia*

Objectives

- Introduction to Primitives
- Points & Lines
- Line Drawing Algorithms
 - Digital Differential Analyzer (DDA)
 - Bresenham's Algorithm
 - Mid-Point Algorithm
- Circle Generating Algorithms
 - Properties of Circles
 - Bresenham's Algorithm
 - Mid-Point Algorithm
- Ellipse Generating Algorithms
 - Properties of Circles
 - Bresenham's Algorithm
 - Mid-Point Algorithm
- Other Curves
 - Conic Sections
 - Polynomial Curves
 - Spline Curves

Introduction

For a raster display, a picture is completely specified by:

- * intensity and position of pixels,
or/and
- * set of complex objects

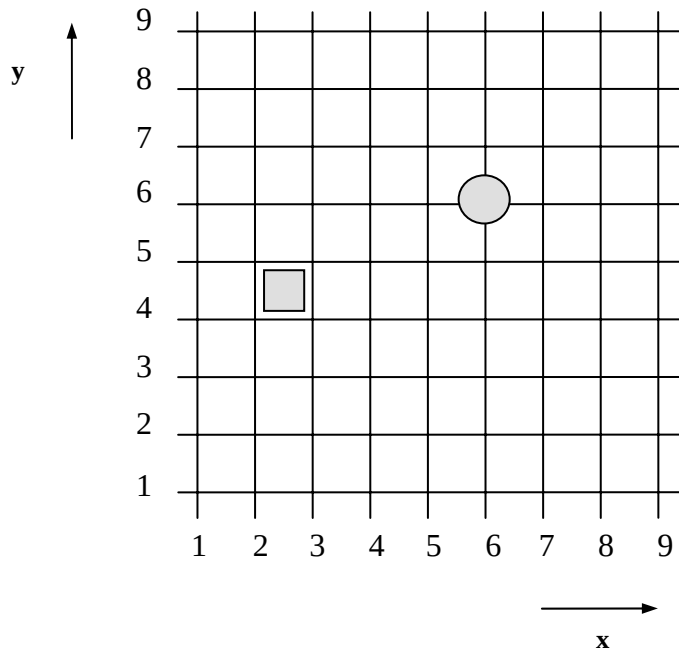
Shapes and contours can either be stored in terms of pixel patterns (bitmaps) or as a set of basic geometric structures (for example, line segments).

Output primitives are the basic geometric structures which facilitate or describe a scene/picture. Example of these include:

*points, lines, curves (circles, conics etc),
surfaces, fill colour, character string etc.*

From Points to Lines

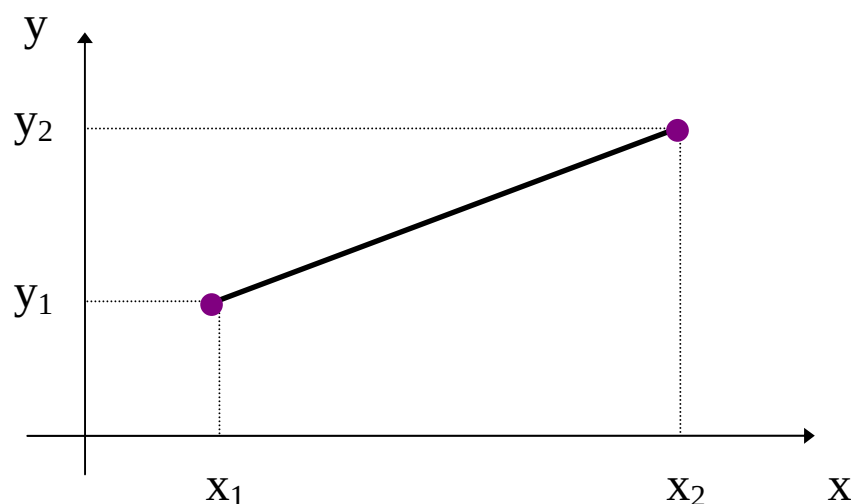
Reference to a pixel (screen point) is gained in similar fashion to using a Cartesian co-ordinate system:



A point on the screen is thus referenced by means of **x** units horizontal and **y** units vertical:
(**x**, **y**)

For raster devices, a display is produced by means of illuminating a set of pixels which “best” describe a geometric structure.

The simplest basic geometric structure is a straight line. A line segment is completely defined in terms of its two endpoints.

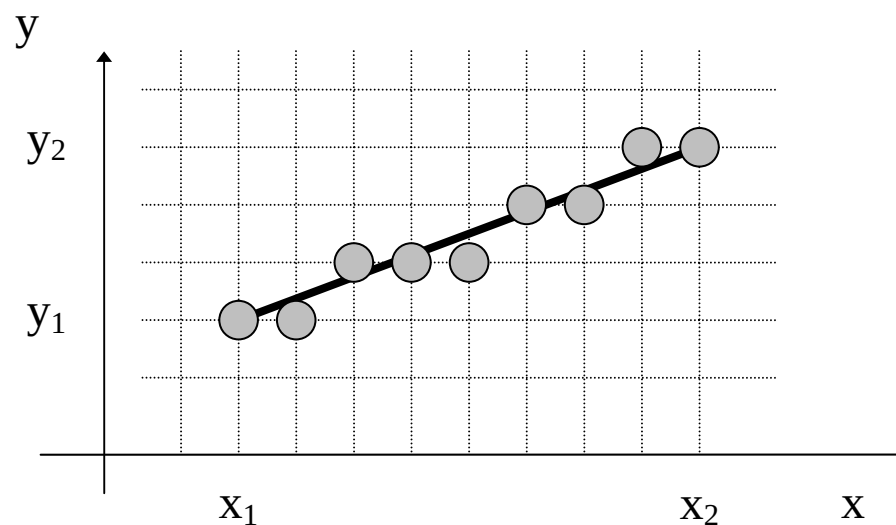


A line segment is thus defined as:

$$\text{Line_Seg} = \{ (x_1, y_1), (x_2, y_2) \}$$

For analogue devices (such as a vector pen plotter, random - scan display), a straight line can be drawn from one endpoint to the other.

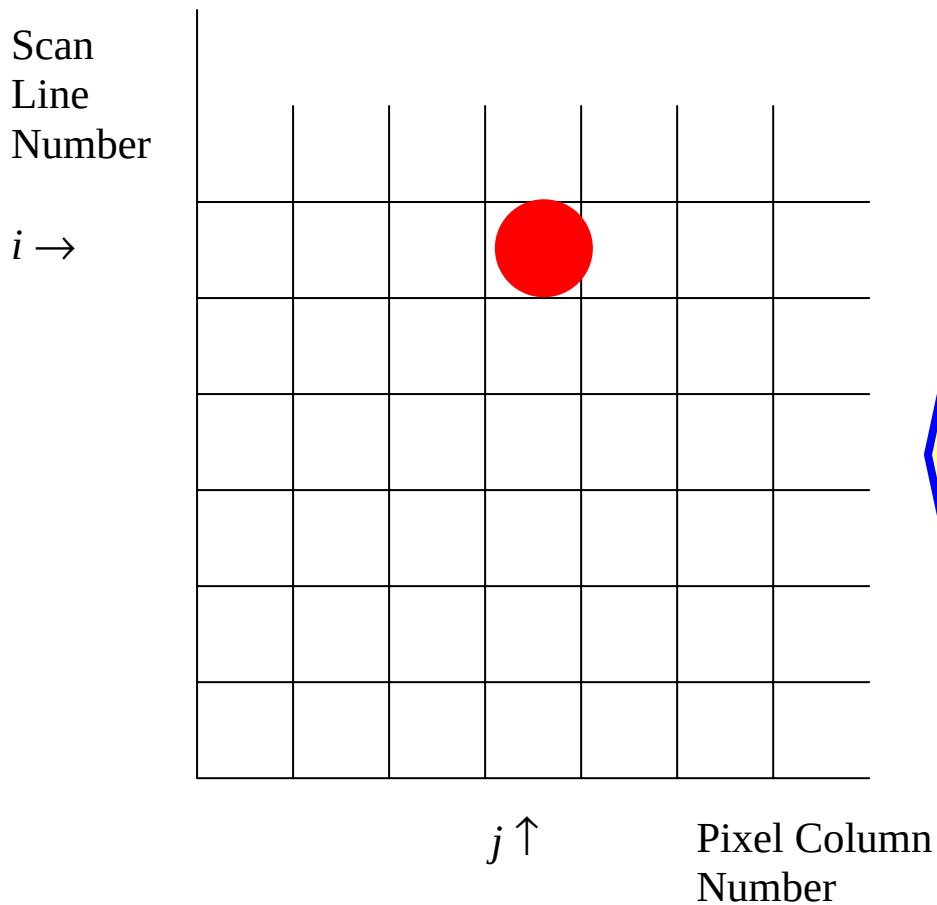
For digital devices (such as raster-scan), a line is produced by means of illuminating a set of intermediary pixels between the two endpoints.



For raster output devices, requirements for drawing straight lines can be summarised as:

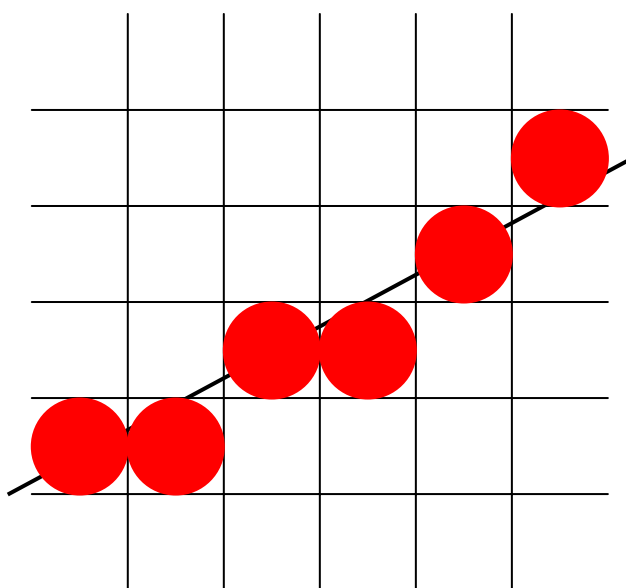
- * Must start and finish at correct position
- * Lines should appear straight
- * Lines should have constant density
- * Line density should be independent of angle
- * Generation should be fast

Points and Lines



Integer Device Coordinates:
Pixel positions referenced by scanline number and column number.

(10.48, 20.51) converted to pixel position (10, 21)



A scan-converted line showing intensified pixels as red circles

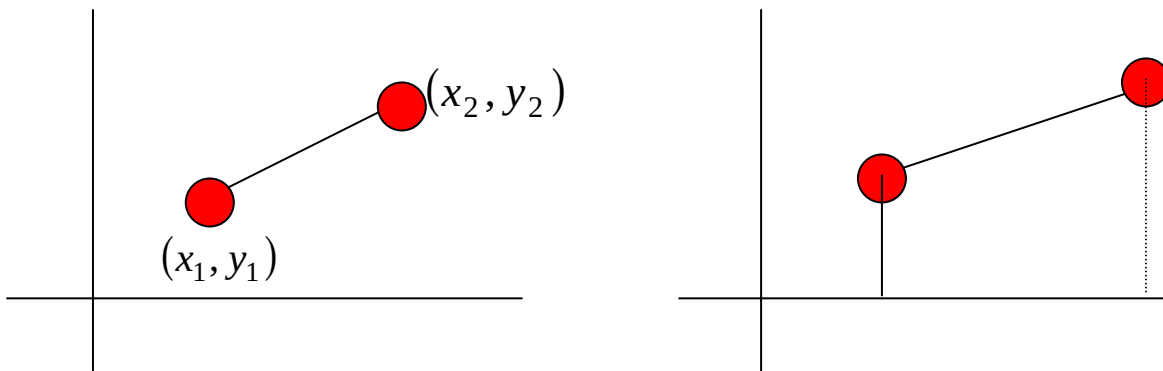
- Load screen point (x,y) by `setPixel(x,y,intensity)`
- Retrieve current setting by `getPixel(x,y)`

Line Drawing Algorithms

The process of gaining a discrete version of a continuous geometric structure is often referred to as **scan-conversion**, **digitisation** or **rasterisation**.

The process usually involves the evaluation of a position on one of the axis, whilst scanning across the other axis.

- Given Points (x_1, y_1) and (x_2, y_2)



All line drawing algorithms make use of the fundamental equations:

- Line Eqn. $y = m.x + b$
- Slope $m = (y_2 - y_1) / (x_2 - x_1) = \Delta y / \Delta x$
- y-intercept $b = y_1 - m.x_1$

x-interval $\rightarrow \Delta x$

y-interval $\rightarrow \Delta y$

Based on calculating either Δy or Δx



- $$y_{k+1} = y_k + m \quad (1)$$

- For lines with $m > 1$, reverse the role of x and y is interchanged and for $\Delta y = 1$, calculate x from

$$x_{k+1} = x_k + 1/m \quad (2)$$

- $$\Delta x = -1, y_{k+1} = y_k - m \text{ for } m \leq 1 \quad (3)$$

$$\Delta y = -1, x_{k+1} = x_k - 1/m \text{ for } m > 1 \quad (4)$$

- **Line with negative slope**

If $|m| < 1$, use(1) [provided line is calculated from left to right] and use(3) [provided line is calculated from right to left]. In case $|m| \geq 1$ use (2) or (4).

- **Merits + Demerits**

1. Faster than the direct use of line Eqn.
2. It eliminates the multiplication in line Eqn.
3. For long line segments, the true line Path may be mislead due to round off.
4. Rounding operations and floating-point arithmetic are still time-consuming.
5. The algorithm can still be improved.
6. Other algorithms, with better performance also exist.

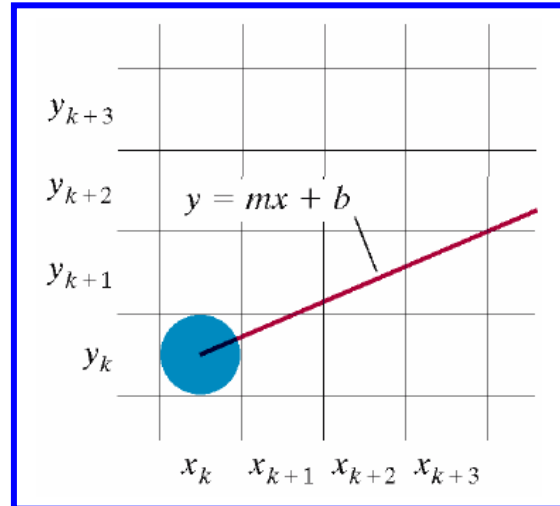
Code for DDA Algorithm

```
Procedure lineDDA(xa,ya,xb,yb:integer);
Var
    dx,dy,steps,k:integer
    xIncrement,yIncrement,x,y:real;
begin
    dx:=xb-xa;
    dy:=yb-ya;
    if abs(dx)>abs(dy) then steps:=abs(dx)
    else steps:=abs(dy);
    xIncrement:=dx/steps;
    yIncrement:=dy/steps;
    x:=xa;
    y:=ya;
    setPixel(round(x),round(y),1);
    for k:=1 to steps do
        begin
            x:=x+xIncrement;
            y:=y+yIncrement;
            setPixel(round(x),round(y),1)
        end
    end; {lineDDA}
```

Bresenham's Line Algorithm

After achieving pixel position (x_k, y_k) , what is next?

Choices are $(x_k + 1, y_k)$ and $(x_k + 1, y_k + 1)$ when $0 < m < 1$ and line is starting from left to right.



Consider d_1 and d_2 in Figure.

Consider y-value at $x_k + 1$ on the line.

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

then

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

$$\text{and } d_2 = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$$

Consider a parameter p_k such that

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

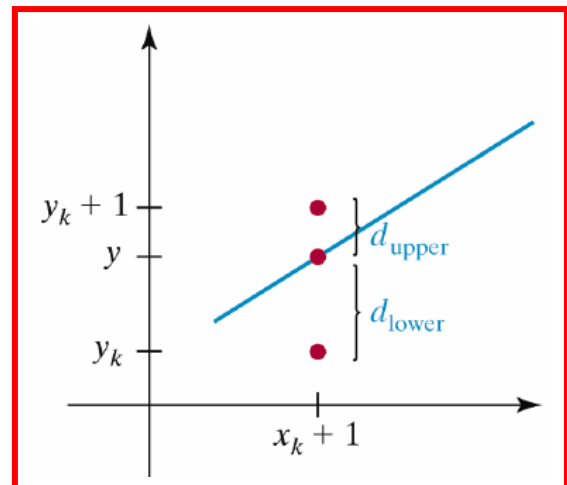
$$\Rightarrow p_k = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$$

where

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

is a constant and is independent of k.

$$\text{-ve } p_k \rightarrow d_1 < d_2$$



Hence plot lower pixel.

Otherwise

Plot the upper pixel.

At steps k and k+1

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

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

Subtraction gives

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

OR

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

where the term $y_{k+1} - y_k$ is 0 or 1, depending on the sign of parameter p_k .

p_k has now recursive definition.

p_0 can be determined at the starting pixel position (x_0, y_0) as:

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

The constants $2\Delta y$ and $2\Delta y - 2\Delta x$ are calculated once for each line to be scan converted. Hence the arithmetic involves only integer addition and subtraction of these two constants.

Bresenham's Line-Drawing Algorithm for $|m| < 1$

1. Input the two line end points and store the left end point in (x_0, y_0) .
2. Load (x_0, y_0) into the frame buffer; that is, plot the first point.
3. Calculate constants $\Delta x, \Delta y, 2\Delta y$, 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, $\Delta x - 1$ times.

Example:

To illustrate the algorithm, we digitize the line with endpoints (20,10) and (30,18). This line has slope of 0.8, with

$$\Delta x = 10, \Delta y = 8$$

The initial decision parameter has the value

$$\begin{aligned} p_0 &= 2\Delta y - \Delta x \\ &= 6 \end{aligned}$$

and the increments for calculating successive decision parameters are

$$\begin{aligned} 2\Delta y &= 16, \\ 2\Delta y - 2\Delta x &= -4 \end{aligned}$$

We plot the initial point $(x_0, y_0) = (20, 10)$ and determine successive pixel positions along the line path from the decision parameter as

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)

A plot of the pixels generated along this line path is shown in Fig.

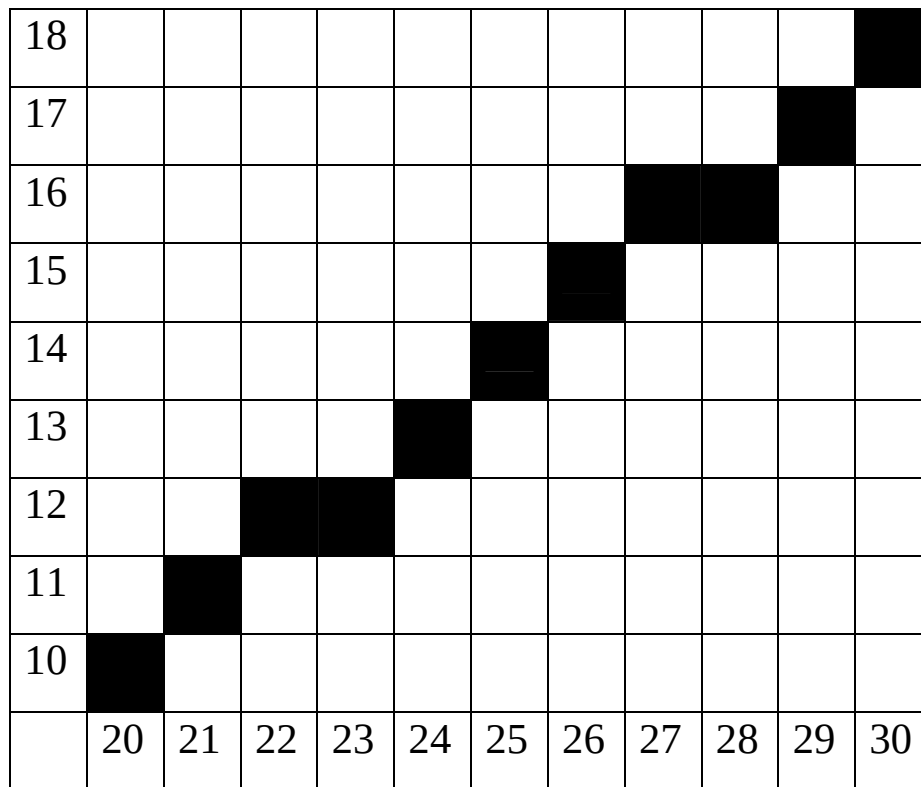


Figure: The Bresenham line from point (20,10) to point (30,18)

What about the lines with arbitrary slopes?

See the book by Hearn and Baker.

Midpoint Line Algorithm

Assume pixel P at (x_k, y_k) is selected in the line of figure below. Next pixel chosen must be either E (East) or NE (North-East) depending on whether the midpoint M is above or below the line. In Bresenham's formulation this choice is dependent on the sign of the difference of the distances from the intersection point Q.

Represent the line by implicit function

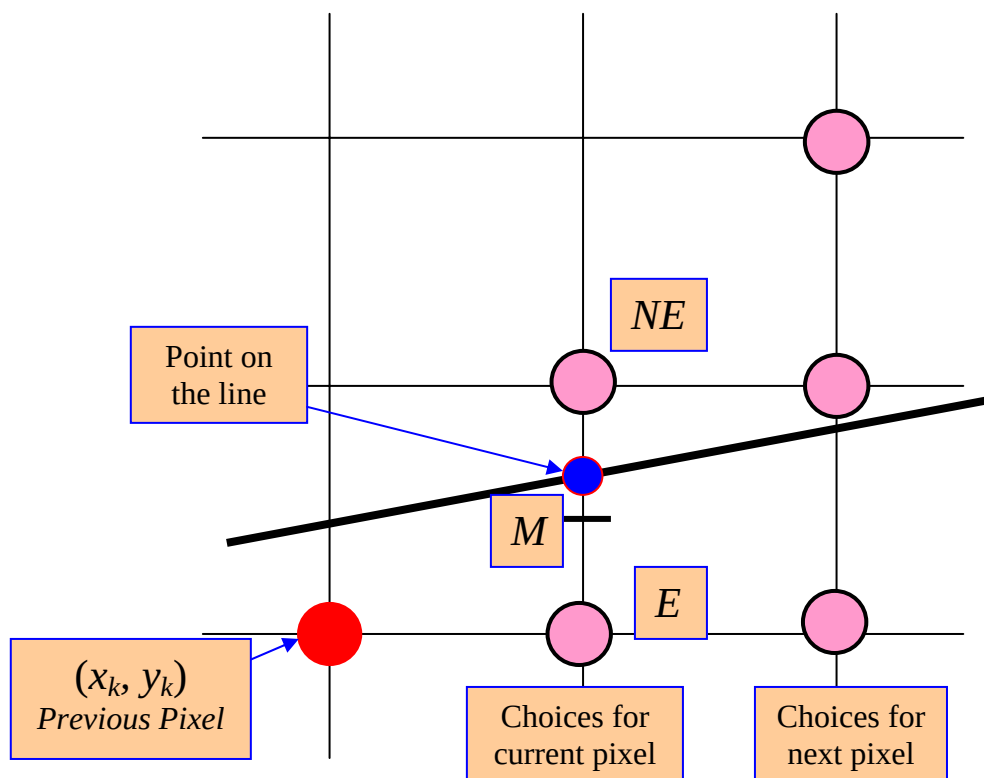
$$F(x, y) = \Delta y \cdot x - \Delta x \cdot y + b \cdot \Delta x = 0$$

Now,

> 0 , if (x, y) is above the line

$F(x, y) = 0$, if (x, y) is on the line

< 0 , if (x, y) is below the line



Only requirement for the Mid point criterion is to compute

$$F(M) = F(x_k + 1, y_k + 1/2)$$

and to test its sign.

Let $d_k = F(M)$. If $d_k > 0$, choose NE, else choose E.

If E is chosen, $M_{k+1} = (x_k + 2, y_k + 1/2)$. Then,

$$d_{k+1} = F(M_{k+1}) = \Delta y.(x_k + 2) + \Delta x.(y_k + 1/2) + b.\Delta x$$

but

$$d_k = F(M_k) = \Delta y.(x_k + 1) + \Delta x.(y_k + 1/2) + b.\Delta x$$

Subtraction gives

$$d_{k+1} = d_k + \Delta y \quad (1)$$

Similarly, if NE is chosen,

$$M_{k+1} = (x_k + 2, y_k + 3/2)$$

and so

$$d_{k+1} = d_k + \Delta y - \Delta x \quad (2)$$

Notation:

$$\Delta_E = \Delta y$$

$$\Delta_{NE} = \Delta y - \Delta x$$

d_k has now recursive definition. d_0 can be determined at the starting line-point (x_0, y_0) as:

$$d_0 = F(x_0 + 1, y_0 + 1/2) = \Delta y - \Delta x/2 + F(x_0, y_0)$$

or

$$d_0 = \Delta y - \Delta x/2, \because (F(x_0, y_0) = 0) \quad (3)$$

From (1),(2) and (3), it follows that if a pixel position is determined then next is determined by just simple addition of either Δ_E or Δ_{NE} . These quantities are determined once throughout the algorithm.

To eliminate the fraction in d_0 , F can be defined as

$$F(x, y) = 2(\Delta y.x - \Delta x.y + b.\Delta x) = 0.$$

It does not affect the sign of the decision variable.

Remark: The arithmetic needed to evaluate d for any step is simple addition. No time consuming multiplication is involved.

Algorithm

```
Procedure MidpointLine( $x_0, y_0, x_1, y_1, value : \text{integer}$ );
Var
     $dx, dy, incrE, incrNE, d, x, y : \text{integer}$ ;
begin
     $dx := x_1 - x_0$ ;
     $dy := y_1 - y_0$ ;
     $d := 2 * dy - dx$ ;           {Initial value of  $d$ }
     $incrE := 2 * dy$ ;           {Increment used for move to  $E$ }
     $incrNE := 2 * (dy - dx)$ ; {Increment used for move to  $NE$ }
     $x := x_0$ ;
     $y := y_0$ ;
    WritePixel( $x, y, value$ ); {The start pixel}
    While  $x < x_1$  do
        begin
            If  $d \leq 0$  then {Choose  $E$ }
                begin
                     $d := d + incrE$ ;
                     $x := x + 1$ ;
                     $y := y$ ;
                end
            else
                begin {Choose  $NE$ }
                     $d := d + incrNE$ ;
                     $x := x + 1$ ;
                     $y := y + 1$ ;
                end
        end;
    WritePixel( $x, y, value$ ) {The selected pixel closest to line}
end {while}
end; {MidpointLine}
```

Exercise: This version of the algorithm works for only those lines with slope between 0 and 1. Generalize the algorithm for arbitrary slopes.

Example 1: Line from (2,2) to (19,9)

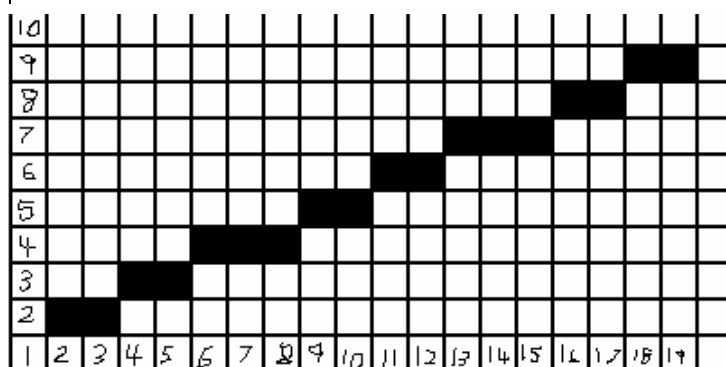
The code generates the x , y , and d values for the line from (2,2) to (19,9). d is the value that determines the row for the next pixel. The values of incrE and incrNE are added to d to change its value.

$\text{incrE} = 14$

$\text{incrNE} = -20$

k	x	y	d
0	2	2	-3
1	3	2	11
2	4	3	-9
3	5	3	5
4	6	4	-15
5	7	4	-1
6	8	4	13
7	9	5	-7

8	10	5	7
9	11	6	-13
10	12	6	1
11	13	7	-19
12	14	7	-5
13	15	7	9
14	16	8	-11



Example 2: Line from (5,8) to (9,11)

For a line from point (5,8) to point (9,11), the successive values of d are 2,0,6, and 4, resulting in the selection of NE , E , NE , and then NE , respectively, as shown in Fig.

12							
11						■	
10					■		
9			■	■			
8		■					
7							
	4	5	6	7	8	9	10

Figure: The Mid-point line from point (5,8) to point (9,11)

The line appears abnormally jagged because of the enlarged scale of the drawing and the artificially large interpixel spacing used to make the geometry of the algorithm clear. For the same reason, the drawings in the following sections also make the primitives appear blockier than they look on the actual screen.

Exercises:

1. Check whether the same output appears when Bresenham's algorithm is implemented.
2. Check whether the same output appears when DDA algorithm is implemented.

Line Function

In PHIGS,GKS and some other packages, two-dimensional line function is :

```
polyline(n,wcPoints);
```

n: number of coordinate positions to be input.

wcPoints: the array of input-coordinate values for line segment endpoints.

This function is used to define a set of $n-1$ connected straight line segments. As a special case for $n=2$, a single straight line is displayed.

Example: The program segment

```
wcPoints.x[1]=50;  
wcPoints.y[1]=100;  
wcPoints.x[2]=150;  
wcPoints.y[2]=250;  
wcPoints.x[3]=250;  
wcPoints.y[3]=100;  
Polyline(3,wcPoints);
```

Causes to generate TWO line segments(connected) with endpoints at (50,100),(150,250) and (250,100).

Problem: Can the points, which are outside the screen coordinate system, be managed? How?

Exercise: Implement the polyline to draw any polygon (open or closed). No given or computed points should be repeated.

Exercise: Read article 19.2.3 of the TEXT.

Circle Generating Algorithms

Equation of the circle in Figure 1 below.

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad (1)$$

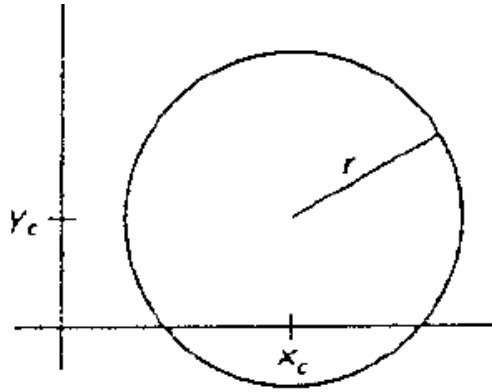


Figure 1: Circle centered at (x_c, y_c) with radius r .

For $x_c - r \leq x \leq x_c + r$ in unit steps,

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2}$$

Problems:

- 1) Considerable computation at each step.
- 2) Non-uniform spacing between plotted pixels as in Figure 2.

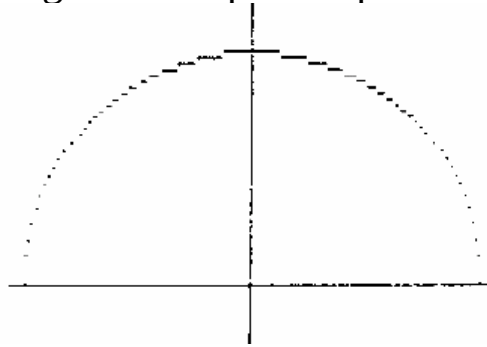


Figure 2:

Problem 2 can be removed using the polar form:

$$x = x_c + r \cos \theta$$

$$y = y_c + r \sin \theta$$

using a fixed angular step size.

Problem 1 can be overcome (upto some extent) by considering the symmetry of circles as in Figure 3. But it still requires a good deal of computation time.

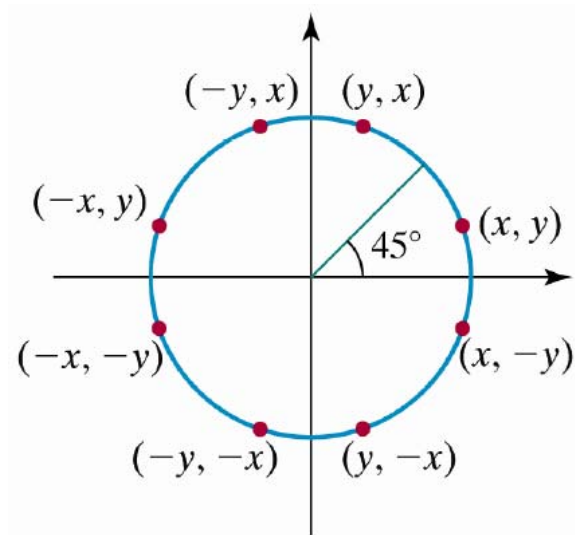


Figure 3: Symmetry of a Circle.

Efficient Solutions

- 1) Bresenham's Circle Algorithm
- 2) Midpoint Circle Algorithm

Mid point Circle Algorithm

To apply the midpoint method, we define a circle function:

$$f_{circle}(x, y) = x^2 + y^2 - r^2 = 0 \quad (2)$$

Any point (x, y) on the boundary of the circle with radius r satisfies the equation $f_{circle}(x, y) = 0$. If the point is in the interior of the circle, the circle function is negative. And if the point is outside the circle, the circle function is positive. To summarize, the relative position of any point (x, y) can be determined by checking the sign of the circle function:

$$\begin{aligned} &< 0 && \text{if } (x, y) \text{ is inside the circle boundary} \\ f_{circle}(x, y) &= 0 && \text{if } (x, y) \text{ is on the circle boundary} \\ &> 0 && \text{if } (x, y) \text{ is outside the circle boundary} \end{aligned} \quad (3)$$

The circle function tests in (3) are performed for the mid positions between pixels near the circle path at each sampling step. Thus, the circle function is the decision parameter in the midpoint algorithm, and we can set up incremental calculations for this function as we did in the line algorithm.

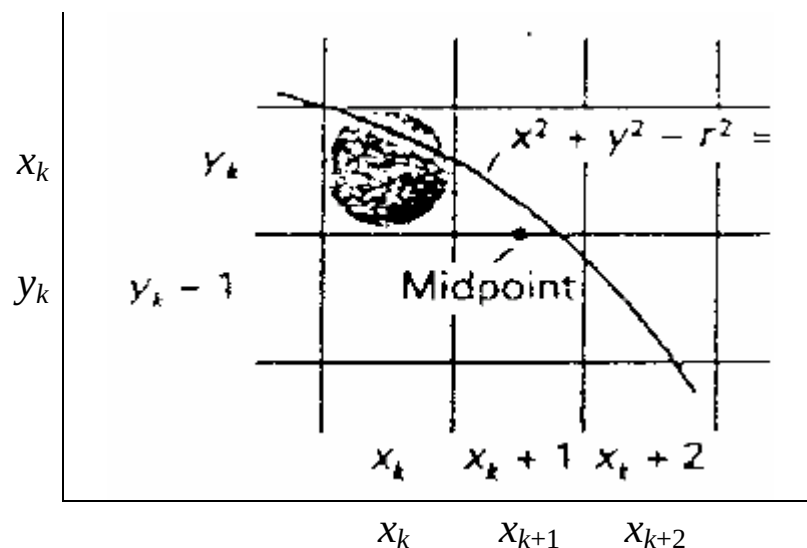


Figure 4

Figure 4 shows the midpoint between the two candidate pixels at sampling position $x_k + 1$. Assuming we have just plotted the pixel at (x_k, y_k) , we next 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 the circle. Our decision parameter is the circle function (2) evaluated at the midpoint between these two pixels:

$$\begin{aligned} p_k &= f_{circle}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \end{aligned} \tag{4}$$

If $p_k < 0$, this midpoint is inside the circle and the pixel on scan line y_k is closer to the circle boundary. Otherwise, the midposition is outside or on the circle boundary, and we select the pixel on scanline $y_k - 1$.

Successive decision parameters are obtained using incremental calculations. We obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{k+1} + 1 = x_k + 2$:

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

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 .

Increments for obtaining p_{k+1} are either $2x_{k+1} + 1$ (if p_k is negative) or $2x_{k+1} + 1 - 2y_{k+1}$. Evaluation of the terms $2x_{k+1}$ and $2y_{k+1}$ can also be done incrementally as:

$$2x_{k+1} = 2x_k + 2$$

$$2y_{k+1} = 2y_k - 2$$

At the start position $(0, r)$, these two terms have the values 0 and $2r$, respectively. Each successive value is obtained by adding 2 to the previous value of $2x$ and subtracting 2 from the previous value of $2y$.

The initial decision parameter is obtained by evaluating the circle function at the start position $(x_0, y_0) = (0, r)$:

$$\begin{aligned} p_0 &= f_{circle}\left(1, r - \frac{1}{2}\right) \\ &= 1 + \left(r - \frac{1}{2}\right)^2 - r^2 \end{aligned}$$

or

$$p_0 = \frac{5}{4} - r$$

If the radius r is specified as an integer, we can simply round p_0 to

$$p_0 = 1 - r \quad (\text{for } r \text{ an integer})$$

since all increments are integers.

Summary of the Algorithm

As in Bresenham's line algorithm, the midpoint method calculates pixel positions along the circumference of a circle using integer additions and subtractions, assuming that the circle parameters are specified in screen coordinates. We can summarize the steps in the midpoint circle algorithm as follows.

Algorithm

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) and

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

Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and

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

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

4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position (x, y) onto the circular path centered on (x_0, y_0) and plot the coordinate values:
 $x = x + x_c, y = y + y_c$

6. Repeat steps 3 through 5 until $x \geq y$.

Example

Given a circle radius $r = 10$, we demonstrate the midpoint circle algorithm by determining positions along the circle octant in the first quadrant from $x = 0$ to $x = y$. The initial value of the decision parameter is

$$p_0 = 1 - r = -9$$

For the circle centered on the coordinate origin, the initial point is $(x_0, y_0) = (0, 10)$, and initial increment terms for calculating the decision parameters are

$$2x_0 = 0, 2y_0 = 20$$

Successive decision parameter values and positions along the circle path are calculated using the midpoint method as

k	p_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
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

A plot of the generated pixel positions in the first quadrant is shown in Figure 5.

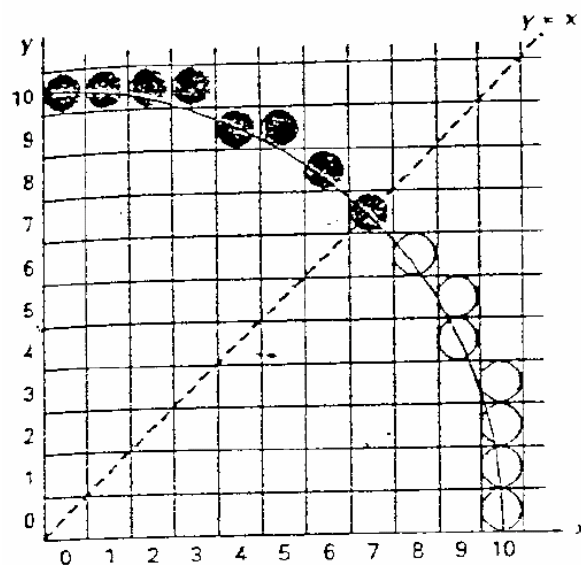


Figure 5

Implementation of the algorithm

The following procedure displays a raster circle on a bilevel monitor using the midpoint algorithm. Input to the procedure are the coordinates for the circle center and the radius. Intensities for pixel positions along the circle circumference are loaded into the frame-buffer array with calls to the setPixel routine.

Procedure circleMidpoint(xCenter, yCenter, radius:integer);

var

 p,x,y:integer;

 procedure plotpoints;

 begin

 setPixel (xCenter + x, yCenter + y, 1);

 setPixel (xCenter - x, yCenter + y, 1);

 setPixel (xCenter + x, yCenter - y, 1);

 setPixel (xCenter - x, yCenter - y, 1);

 setPixel (xCenter + y, yCenter + x, 1);

 setPixel (xCenter - y, yCenter + x, 1);

 setPixel (xCenter + y, yCenter - x, 1);

 setPixel (xCenter - y, yCenter - x, 1);

 end; plotPoints

begin

 x := 0;

 y := radius;

 plotPoints;

 p := 1 - radius;

 while x < y do

 begin

 if p < 0 then

 x := x + 1

 else

 begin

 x := x + 1;

 y := y - 1

 end;

 if p < 0 then

```

        p := p + 2 * x + 1
    else
        p := p + 2 * (x - y) + 1;
    plotPoints
end;
end;    { circleMidpoint }

```

Exercise: Device the Bresenham's circle algorithm.

Ellipse

Remark: Loosely stated, an ellipse is an elongated circle. Therefore, elliptical curves can be generated by modifying circle-drawing procedures to take into account the different dimensions of an ellipse along the major and minor axes.

Exercise: Modify the Bresenham's and Midpoint Circle Algorithms to draw ellipses.

Curve Functions

Routines for circles, splines, etc. are included in many graphics packages separately. PHIGS includes the following general curve function:

GeneralizedDrawingPrimitive($n, wcPoints, id, datalist$)

Where

n : number of points to be input

$wcPoints$: list of n points

$datalist$: non-coordinate data values

id : the code for the desired function

Example: A circle may have:

$id = 1$

$wcPoints.xc[1] = 4$

(Center Coordinate)

$wcPoints.yc[1] = -3$

$datalist.r = 5$ (radius)

Example:

$id = 2$

$wcPoints.xc[1] = 0$

(Center Coordinates)

$wcPoints.yc[1] = 0$

$datalist.rx = 3$

(Minor+Major axes)

$datalist.ry = 5$

Other Curves

Commonly encountered curves:

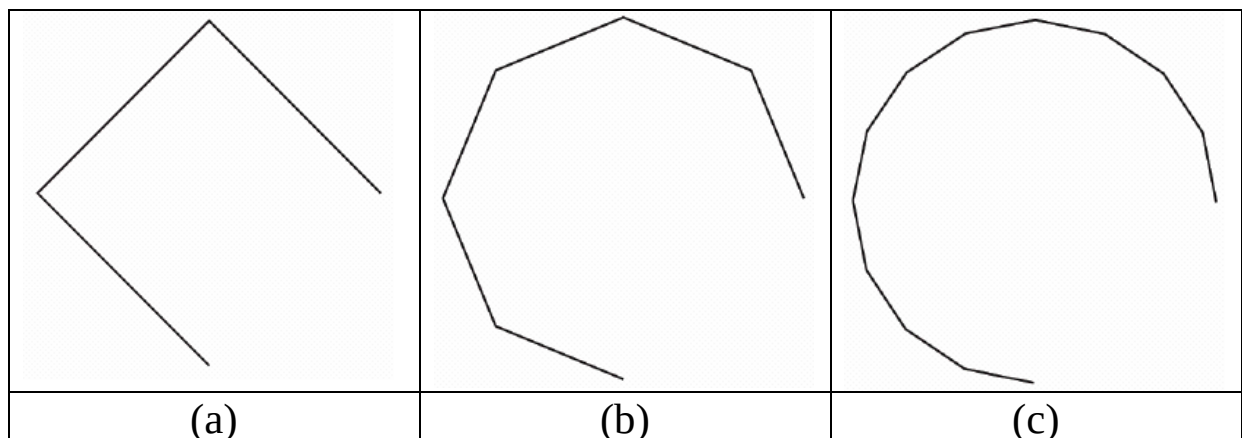
- Conics
- Trigonometric and Exponential Functions
- Probability Distributions
- General Polynomials
- Spline Functions

Various Applications:

- Object Modeling
- Animation Path Specification
- Data and Function Graphing
- Other Graphics Application

Display of curves:

- Direct manipulation
- Bresenham's or Midpoint-like method
- Approximation with Straight Line Segments



- A circular arc approximated with
- (a) Three straight line segments,
 - (b) Six straight line segments,
 - (c) Twelve straight line segments.