GRAPH PLOTTING & SCAN CONVERSION

A plot is a graphical technique for representing a data set, usually as a graph showing the relationship between two or more variables. The plot can be drawn by hand or by a mechanical or electronic plotter. Graphs are a visual representation of the relationship between variables, very useful for humans who can quickly derive an understanding which would not come from lists of values. Graphs can also be used to read off the value of an unknown variable plotted as a function of a known one. Graphs of functions are used in mathematics, sciences, engineering, technology, finance, and other areas.

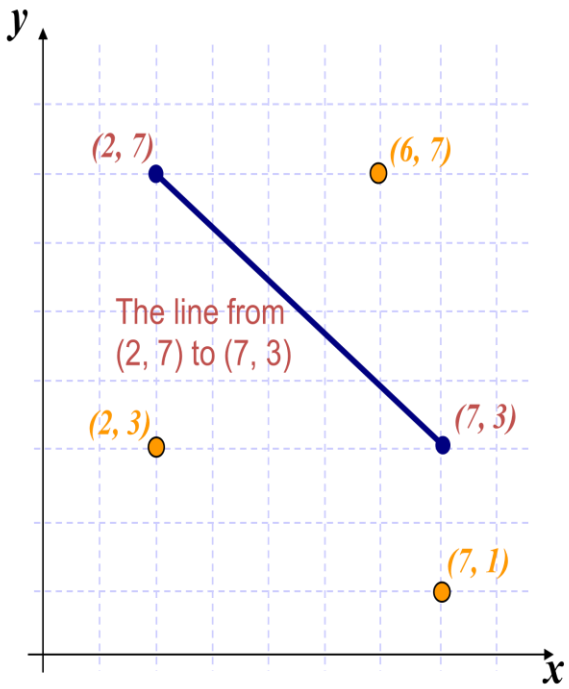
The process of representing continuous graphical objects as a collection of discrete pixels by identifying their locations and setting them on is called scan conversion. The process of determining which pixels will provide the best approximation to the desired line is properly known as rasterization.

Combined with the process of rendering the picture in scan line order it is known as scan conversion.

Scan Conversion The process of representing continuous graphics object as a collection of discrete pixels is called Scan Conversion. For e.g. a line is defined by its two end pts & the line equation, where as a circle is defined by its radius, center position & circle equation

Points and Line

- Points:
  - A point in two dimensional space is given as an ordered pair *(x, y)*
  - In three dimensions a point is given as an ordered triple *(x, y, z)*
- Lines: A line is defined using a start point and an end-point
  - In 2d: (x_{start}, y_{start}) to (x_{end}, y_{end})
  - In 3d: $(x_{start}, y_{start}, z_{start})$ to $(x_{end}, y_{end}, z_{end})$



Line

A line, or straight line, is, roughly speaking, an (infinitely) thin, (infinitely) long, straight geometrical object, i.e. a curve that is long and straight. Given two points, in Euclidean geometry, one can always find exactly one line that passes through the two points; this line provides the shortest connection between the points and is called a straight line. Three or more points that lie on the same line are called **collinear**. Two different lines can intersect in at most one point; whereas two different planes can intersect in at most one line. This intuitive concept of a line can be formalized in various ways.

A line may have three forms with respect to slope:

- slope = 1
- slope > 1
- slope < 1

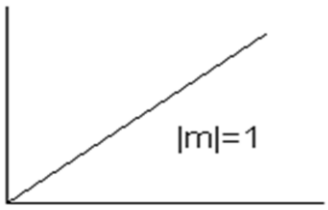


figure (a)

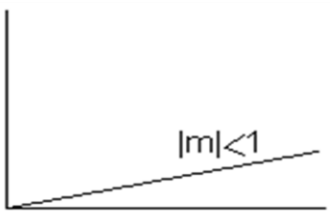


figure (b)

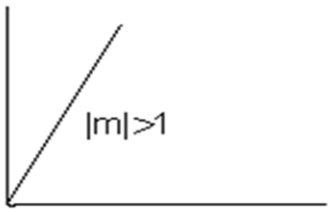


figure (c)

Line Drawing Techniques

There are three techniques to be discussed to draw a line involving different time complexities that will be discussed along. These techniques are:

- Incremental line algorithm
- DDA line algorithm
- Bresenham line algorithm

Incremental line algorithm

This algorithm exploits simple line equation $y = m x + b$
Where $m = dy / dx$
and $b = y – m x$

Now check if:
$|m| < 1$ then
$x = x + 1$ whereas
$y = m x + b$
Now check if $|m| < 1$ then starting at the first point, simply increment x by 1 (unit increment) till it reaches ending point;
Now check if:
$|m|>1$ then
Whereas calculate y point by the equation for each x and conversely if $|m|>1$ then increment y by 1 till it reaches ending point; whereas calculate x point corresponding to each y, by the equation.

First if |m| is less than 1 then it means that for every subsequent pixel on the line there will be unit increment in x direction and there will be less than 1 increment in y direction and vice versa for slope greater than 1. Let us clarify this with the help of an example:

Suppose a line has two points p1 (10, 10) and p2 (20, 18)
Now difference between y coordinates that is dy = y2 – y1 = 18 – 10 = 8
Whereas difference between x coordinates is dx = x2 – x1 = 20 – 10 = 10
This means that there will be 10 pixels on the line in which for x-axis there will be distance of 1 between each pixel and for y-axis the distance will be 0.8.

Consider the case of another line with points p1 (10, 10) and p2 (16, 20)
Now difference between y coordinates that is dy = y2 – y1 = 20 – 10 = 10
Whereas difference between x coordinates is dx = x2 – x1 = 16 – 10 = 6
This means that there will be 10 pixels on the line in which for x-axis there will be distance of 0.6 between each pixel and for y-axis the distance will be 1.

Now having discussed this concept at length let us learns the algorithm to draw a line using above technique, called incremental line algorithm:

Incremental Line (Point p1, Point p2)

```
dx = p2.x – p1.x
dy = p2.y – p1.y
m = dy / dx
x = p1.x
y = p1.y
b = y – m * x
if |m| < 1
for counter = p1.x to p2.x
drawPixel (x, y)
x = x + 1
y = m * x + b
else
for counter = p1.y to p2.y
drawPixel (x, y)
y = y + 1
x = (y – b) / m
```

Discussion on algorithm:
Well above algorithm is quite simple and easy but firstly it involves lot of mathematical calculations that is for calculating coordinate using equation each time secondly it works only in incremental direction.

We have another algorithm that works fine in all directions and involving less calculation mostly only addition; which will be discussed in next topic.

Digital Differential Analyzer (DDA) Algorithm:

DDA abbreviated for digital differential analyzer has very simple technique. Find difference dx and dy between x coordinates and y coordinates respectively ending points of a line.

Find difference, dx and dy as:

```
dy = y2 – y1
dx = x2 – x1
if |dx| > |dy| then
        step = |dx|
else
        step = |dy|
```

Now very simple to say that step is the total number of pixels required for a line. Next step is to divide dx and dy by step to get xIncrement and yIncrement that is the increment required in each step to find next pixel value.

```
xIncrement = dx/step
yIncrement = dy/step
```

Next a loop is required that will run step times. In the loop drawPixel and add xIncrement in x1 by and yIncrement in y1.

To sum-up all above in the algorithm, we will get,

DDA_Line (Point p1, Point p2)

```
dx = p2.x – p1. x
dy = p2.y – p1. y
x1=p1.x
y1=p1.y
if |dx|>|dy| then
step = |dx|
else
step = |dy|
xIncrement = dx/step
yIncrement = dy/step
for counter = 1 to step
drawPixel (x1, y1)
x1 = x1 + xIncrement
y1 = y1 + yIncrement
```

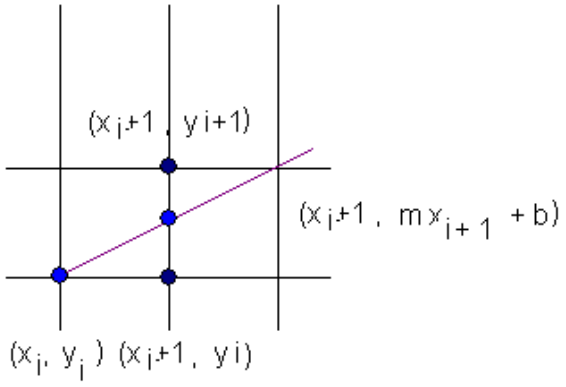
Criticism on Algorithm:

- Use of floating point calculation
- An algorithm based on integer type calculations is likely to be more efficient

Therefore, after some effort, finally we came up with an algorithm called “Bresenham Line Drawing Algorithm” which would be discussed next.

Bresenham line algorithm:

Bresenham's algorithm finds the closest integer coordinates to the actual line, using only integer math. Assuming that the slope is positive and less than 1, moving 1 step in the x direction, y either stays the same, or increases by 1. A decision function is required to resolve this choice.



If the current point is (xi, yi), the next point can be either (xi+1,yi) or (xi+1,yi+1) . The actual position on the line is (xi+1, m(xi+1)+c) . Calculating the distance between the true point, and the two alternative pixel positions available gives:

d1 = y – yi = m * (x+1)+b-yi
d2 = yi + 1 - y= yi + 1 – m (xi + 1) – b

Let us magically define a decision function p, to determine which distance is closer to the true point. By taking the difference between the distances, the decision function will be positive if d1 is larger, and negative otherwise. A positive scaling factor is added to ensure that no division is necessary and only integer math need be used.

pi = dx (d1-d2)
pi = dx (2m * (xi+1) + 2b – 2yi-1)
pi = 2 dy (xi+1) –2 dx yi + dx (2b-1) ---------- (i)
pi = 2 dy xi – 2 dx yi + k ---------- (ii)
where k=2 dy + dx (2b-1)

Then we can calculate pi+1 in terms of pi without any xi , yi or k .

pi+1 = 2 dy xi+1 – 2 dx yi+1 + k
pi+1 = 2 dy (xi + 1) - 2 dx yi+1 + k since xi+1= xi + 1
pi+1 = 2 dy xi + 2 dy- 2 dx yi+1 + k ---------- (iii)
Now subtracting (ii) from (iii), we get
pi+1 - pi = 2 dy - 2 dx (yi+1 - yi)
pi+1 = pi + 2 dy - 2 dx (yi+1 - yi)
If the next point is: (xi+1,yi) then
d1<d2 => d1-d2<0

=> pi<0

=> **pi+1= pi + 2 dy**

If the next point is: (xi+1,yi+1) then

d1>d2 => d1-d2>0

=> pi>0

=> pi+1= pi + 2 dy - 2 dx

The pi is our decision variable, and calculated using integer arithmetic from pre-computed constants and its previous value. Now a question is remaining how to calculate initial value of pi. For that use equation (i) and put values (x1, y1)

pi = 2 dy (x1+1) – 2 dx yi + dx (2b-1)

where b = y – m x implies that

pi = 2 dy x1 +2 dy – 2 dx yi + dx (2 (y1 – mx1) -1)

pi = 2 dy x1 +2 dy – 2 dx yi + 2 dx y1 – 2 dy x1 - dx

pi = 2 dy x1 +2 dy – 2 dx yi + 2 dx y1 – 2 dy x1 - dx

there are certain figures will cancel each other shown in same different colour

pi = 2 dy - dx

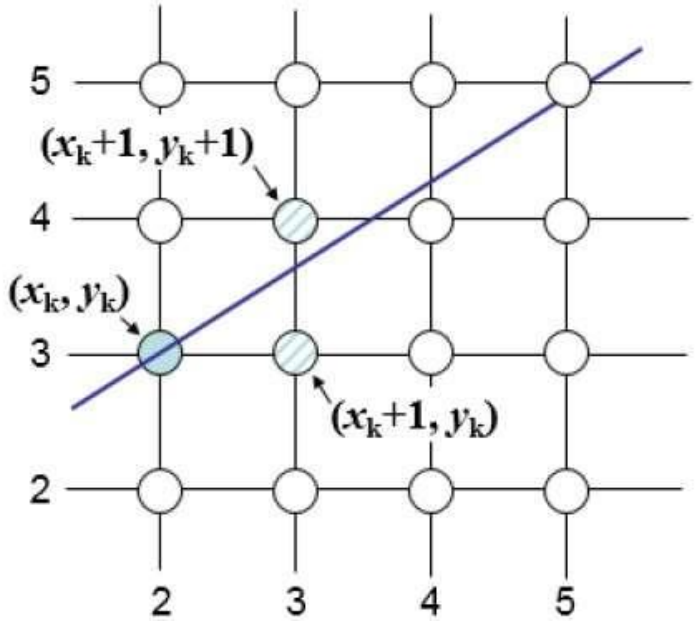
Thus Bresenham's line drawing algorithm is as follows:

```
dx = x2-x1
dy = y2-y1
p = 2dy-dx
c1 = 2dy
c2 = 2(dy-dx)
x = x1
y = y1
plot (x,y,colour)
while (x < x2 )
x++;
if (p < 0)
p = p + c1
else
p = p + c2
y++
plot (x,y,colour)
```

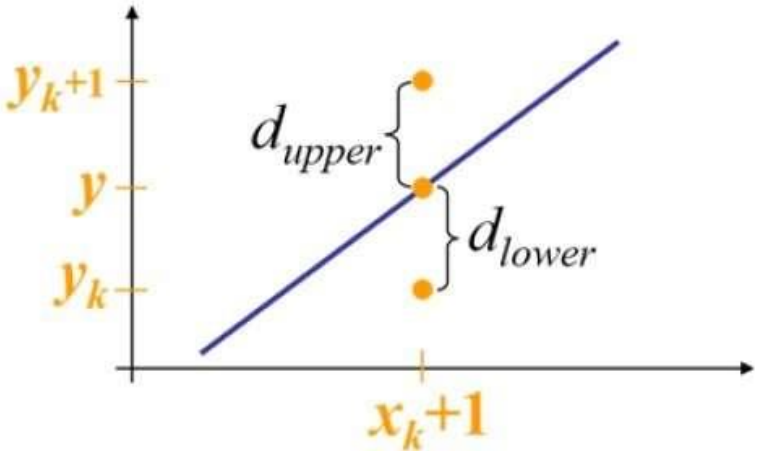
Again, this algorithm can be easily generalized to other arrangements of the end points of the line segment, and for different ranges of the slope of the line.

ANOTHER WAY TO UNDERSTAND BRESENHAM'S LINE DRAWING ALGORITHM

The Bresenham algorithm is another incremental scan conversion algorithm. The big advantage of this algorithm is that, it uses only integer calculations. Moving across the x axis in unit intervals and at each step choose between two different y coordinates.
For example, as shown in the following illustration, from position 2, 3 you need to choose between 3, 3 and 3, 4. You would like the point that is closer to the original line.



At sample position Xk + 1, the vertical separations from the mathematical line are labelled as dupper and dlower.



From the above illustration, the y coordinate on the mathematical line at xk + 1 is −
Y = mXk + 1 + b
So, dupper and dlower are given as follows −
dlower = y − yk
= m(Xk + 1) + b − Yk
and
dupper = (yk + 1) − y
= Yk + 1 − m(Xk + 1) − b

You can use these to make a simple decision about which pixel is closer to the mathematical line.
This simple decision is based on the difference between the two pixel positions.
dlower − dupper = 2m(xk + 1) − 2yk + 2b − 1
Let us substitute m with dy/dx where dx and dy are the differences between the end-points.
dx(dlower − dupper) = dx(2
dy
dx(xk + 1) − 2yk + 2b − 1)
= 2dy. xk − 2dx. yk + 2dy + 2dx(2b − 1)
= 2dy. xk − 2dx. yk + C
So, a decision parameter Pk for the kth step along a line is given by −
pk = dx(dlower − dupper)
= 2dy. xk − 2dx. yk + C
The sign of the decision parameter Pk is the same as that of dlower − dupper. If pk is negative, then choose the lower pixel, otherwise choose the upper pixel.
Remember, the coordinate changes occur along the x axis in unit steps, so you can do everything with integer calculations. At step k+1, the decision parameter is given as −
pk+1 = 2dy. xk+1 − 2dx. yk+1 + C
Subtracting pk from this we get −
pk+1 − pk = 2dy(xk+1 − xk) − 2dx(yk+1 − yk)
But, xk+1 is the same as xk+1. So − pk+1 = pk + 2dy − 2dx(yk+1 − yk)
Where, Yk+1– Yk is either 0 or 1 depending on the sign of Pk.
The first decision parameter p0 is evaluated at (x0, y0) is given as − p0 = 2dy − dx
Now, keeping in mind all the above points and calculations, here is the Bresenham algorithm for slope m < 1 −
Step 1 − Input the two end-points of line, storing the left end-point in (x0, y0).
Step 2 − Plot the point (x0, y0).
Step 3 − Calculate the constants dx, dy, 2dy, and 2dy– 2dx and get the first value for the decision parameter as −
p0 = 2dy − dx
Step 4 − At each Xk along the line, starting at k = 0, perform the following test −
If pk < 0, the next point to plot is (xk + 1, yk) and pk+1 = pk + 2dy
Otherwise,
pk+1 = pk + 2dy − 2dx
Step 5 − Repeat step 4 dx– 1 times.
For m > 1, find out whether you need to increment x while incrementing y each time. After solving, the equation for decision parameter Pk will be very similar, just the x and y in the equation gets interchanged.

Different examples to display the lines:

CHECKING GRAPHIC FILE

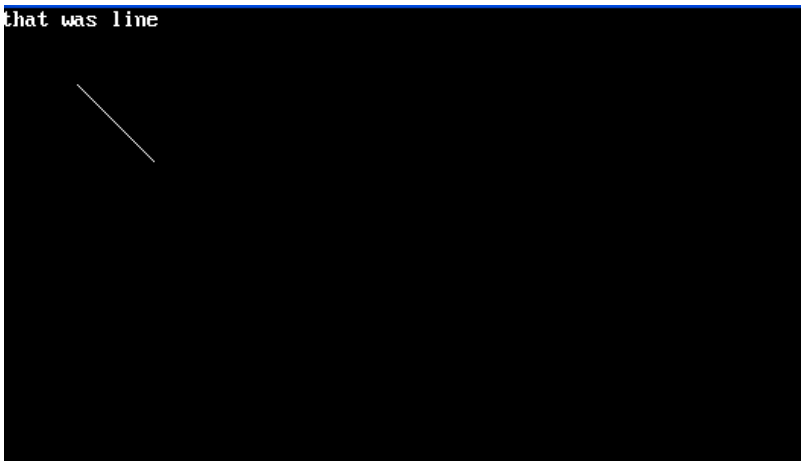
```
int main(void)
{
  /* request auto detection */
  int gdriver = DETECT, gmode, errorcode;
  int xmax, ymax;

  /* initialize graphics and local variables */
  initgraph(&gdriver, &gmode, "");
  /* read result of initialization */
  errorcode = graphresult();
  /* an error occurred */
  if (errorcode != grOk)
  {
    printf("Graphics error: %s\n", grapherrormsg(errorcode));
    printf("Press any key to halt:");
    getch();
    exit(1);   }
```

THIS PROGRAM DISPLAY A LINE WITH DIFFERENT POINTS

```
#include<iostream.h> #include <graphics.h>
#include <stdlib.h>     #include <stdio.h>
#include <conio.h>
int main(void)
{
  /* request auto detection */
  int gdriver = DETECT, gmode, errorcode;
  int xmax, ymax;
  /* initialize graphics and local variables */
  initgraph(&gdriver, &gmode, "");
  /* read result of initialization */
  errorcode = graphresult();
  /* an error occurred */
  if (errorcode != grOk)
  {
    printf("Graphics error: %s\n", grapherrormsg(errorcode));
    printf("Press any key to halt:");
    getch();
```

```
    exit(1);
  }
  setcolor(getmaxcolor());
  xmax = getmaxx();
  ymax = getmaxy();
  /* draw a diagonal line */
  line(50, 50, 100,100);
 cout<<"that was line";
  getch();
  closegraph();
  return 0;
}
```



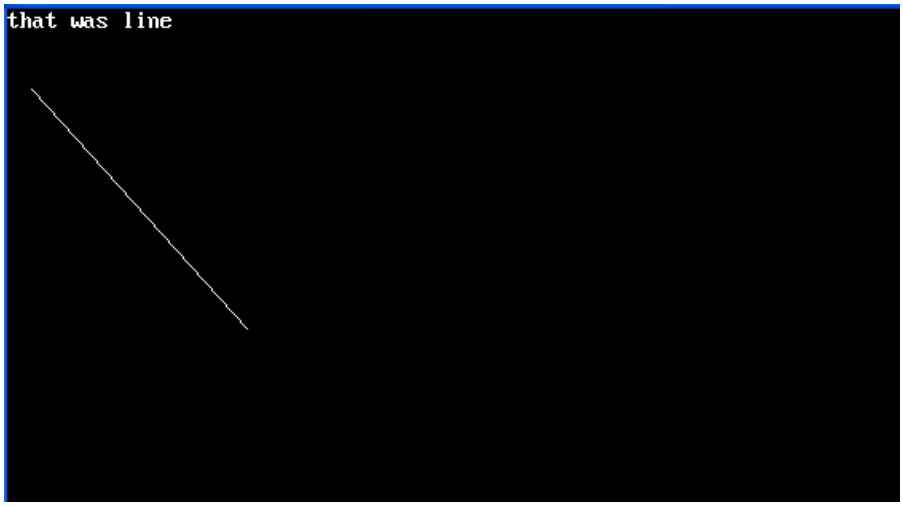
THIS PROGRAM DISPLAY A LINE WITH DIFFERENT POINTS

```
#include<iostream.h> #include <graphics.h>
#include <stdlib.h>     #include <stdio.h>
#include <conio.h>
int main(void)
{
 int gdriver = DETECT, gmode, errorcode;
  int xmax, ymax;
  /* initialize graphics and local variables */
  initgraph(&gdriver, &gmode, "");
  /* read result of initialization */
  errorcode = graphresult();
  /* an error occurred */
  if (errorcode != grOk)
  {
    printf("Graphics error: %s\n", grapherrormsg(errorcode));
```

```
    printf("Press any key to halt:");
    getch();
    exit(1);
  }
 setcolor(getmaxcolor());
  xmax = getmaxx();
  ymax = getmaxy();

  /* draw a diagonal line */
  line(15, 50, 150,200);

  /* clean up */
  cout<<"that was line";
  getch();
  closegraph();
  return 0;
}
```



THIS PROGRAM DISPLAY DIFFERENT LINE WITH FOR LOOP

```
#include<iostream.h> #include <graphics.h>
#include <stdlib.h>    #include <stdio.h>
#include <conio.h>
int main(void)
{
  /* request auto detection */
  int gdriver = DETECT, gmode, errorcode;
  int xmax, ymax;
  /* initialize graphics and local variables */
```

```
  initgraph(&gdriver, &gmode, "");
  /* read result of initialization */
  errorcode = graphresult();
    if (errorcode != grOk)
  {printf("Graphics error: %s\n", grapherrormsg(errorcode));
    printf("Press any key to halt:");
    getch();
    exit(1); }
  setcolor(getmaxcolor());
  xmax = getmaxx();
  ymax = getmaxy();
  for(int i=10;i<xmax;i+=10)
  line(i,0,i,ymax);
  for(i=10;i<ymax;i+=10)
  line(0,i,xmax,i);
      getch();
  closegraph();
  return 0;
}
```