Chapter Four: Output Primitives

4.1 Output Primitives

A computer Graphics can be anything like beautiful scenery, images, terrain, trees, or anything else that we can imagine, however, all these computer graphics are made up of the most basic components of Computer Graphics that are called Graphics Output Primitive or simply primitive. The Primitives are the simple geometric functions that are used to generate various Computer Graphics required by the User.

An **Output Primitive** is a graphics object that is essential for the creation or construction of complex images. Some most basic Output primitives are point-position (pixel), and a straight line Examples are point, line, text, filled region, images, quadric surfaces, spline curves, etc. Different Graphic packages offers different output primitives like a rectangle, conic section, circle, spline curve or may be a surface. Once it is specified what picture is to be displayed, various locations are converted into integer pixel positions within the frame buffer and various functions are used to generate the picture on the two dimensional coordinate system of output display.

Each output primitive has an associated set of attributes, such as line width and line color for lines. The programming technique is to set values for the output primitives and then call a basic function that will draw the desired primitive using the current settings for the attributes. Various graphics systems have different graphics primitives. For example, GKS defines five output primitives namely, polyline (for drawing contiguous line segments), polymarker (for marking coordinate positions with various symmetric text symbols), text (for plotting text at various angles and sizes), fill area (for plotting polygonal areas with solid or hatch fill), cell array (for plotting portable raster images). At the same time, GRPH1 has the output primitives namely Polyline, Polymarker, Text, Tone and have other secondary primitives besides these namely, Line and Arrow.

4.2 Rasterization

Rasterization is the process of converting a vertex representation to a pixel representation; rasterization is also called *scan conversion*. Included in this definition are geometric objects such as circles where you are given a center and radius.

Scan conversion algorithms use *incremental* methods that exploit *coherence*. An incremental method computes a new value quickly from an old value, rather than computing the new value from scratch, which can often be slow. Coherence

in space or time is the term used to denote that nearby objects (e.g. pixels) has qualities similar to the current object.

4.3 Line Generation Algorithm

A **Line Drawing Algorithm** is a graphical algorithm for approximating a line segment on discrete graphical media. On discrete media, such as pixel-based displays and printers, line drawing requires such an approximation (in nontrivial cases). Basic algorithms rasterize lines in one color. A line connects two points. It is a basic element in graphics. To draw a line, you need two points between which you can draw a line. We can refer the one point of line as $\boldsymbol{X_0,Y_0}$ and the second point of line as $\boldsymbol{X_1,Y_1}$.

Two line-converting algorithms are Digital Differential Algorithm and Bresenham's Algorithm are used for Line Generation

4.3.1 A **Digital Differential Analyzer** (**DDA**) is hardware or software used for interpolation of variables over an interval between start and end-point. DDAs are used for rasterization of lines, triangles and polygons. They can be extended to nonlinear functions, such as perspective correct texture mapping, quadratic curves, and traversing voxels. A line connects two points. It is a basic element in graphics. To draw a line, you need two points between which you can draw a line. In the following three algorithms, we refer the one point of line as X_0,Y_0 and the second point of line as X_1,Y_1.

In its simplest implementation for linear cases such as lines, the DDA algorithm interpolates values in interval by computing for each x_i the equations $x_i = x_{i-1} + 1$, $y_i = y_{i-1} + m$, where $\Delta x = x_{end} - x_{start}$ and $\Delta y = y_{end} - y_{start}$ and $m = \Delta y/\Delta x$.

The DDA algorithm is faster than the direct use of the line equation since it calculates points on the line without any floating-point multiplication.

The advantages of DDA Algorithm are:-

(a) It is the simplest algorithm and it does not require special skills for implementation.

(b) It is a faster method for calculating pixel positions than the direct use of equation y = mx + b. It eliminates the multiplication in the equation by making use of raster characteristics, so that appropriate increments are applied in the x or v direction to find the pixel positions along the line path.

The disadvantages of DDA Algorithm

(a) Floating-point arithmetic in DDA algorithm is still time-consuming.

(b) The algorithm is orientation dependent. Hence, end-point accuracy is poor.

The Procedure-is as follows:

Given Starting coordinates = (X_0, Y_0) and Ending coordinates = (X_n, Y_n) of a line, the points generation using DDA Algorithm involves the following steps-

Step-01:

Calculate ΔX, ΔY and M from the given input. These parameters are calculated as-

$$\Delta X = X_n - X_0,$$
$$\Delta Y = Y_n - Y_0$$
$$M = \Delta Y / \Delta X$$

Step-02:

Find the number of steps or points in between the starting and ending coordinates.

```
if (absolute (ΔX) > absolute (ΔY))
        Steps = absolute (ΔX);
else
        Steps = absolute (ΔY);
```

Step-03:

Suppose the current point is (X_p, Y_p) and the next point is (X_{p+1}, Y_{p+1}).
Find the next point by following the below three cases-

Case 1: If **m < 1**

Then **x** coordinate tends to the Unit interval.

$$\mathbf{x_{k+1} = x_k + 1}$$

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

Case 2: If **m > 1**

Then **y** coordinate tends to the Unit interval.

$$\mathbf{y_{k+1} = y_k + 1}$$

$$\mathbf{x_{k+1} = x_k + 1/m}$$

Case 3: If **m = 1**

Then **x and y** coordinate tend to the Unit interval.

$$\mathbf{x_{k+1} = x_k + 1}$$

$$\mathbf{y_{k+1} = y_k + 1}$$

Step-04:

Keep repeating Step-03 until the end-point is reached or the number of generated new points (including the starting and ending points) equals to the steps count.

Digital Differential Analyzer (DDA) algorithm is the simple line generation algorithm stated below:-

Step 1:- Get the input of two end points (X_0,Y_0) and (X_1,Y_1)

.

Step 2:- Calculate the difference between two end points.

dx = X_1 - X_0

dy = Y_1 - Y_0

Step 3:- Based on the calculated difference in step-2, there is a need to identify the number of steps to put pixel. If dx > dy, then more steps are needed in x coordinate; otherwise in y coordinate.

```
if (absolute(dx) > absolute(dy))
        Steps = absolute(dx);
else
        Steps = absolute(dy);
```

Step 4:- Calculate the increment in x coordinate and y coordinate.

```
Xincrement = dx / (float) steps;
Yincrement = dy / (float) steps;
```

Step 5:- Put the pixel by successfully incrementing x and y coordinates accordingly and complete the drawing of the line.

```
for(int v=0; v < Steps; v++)
{
        x = x + Xincrement;
        y = y + Yincrement;
        putpixel(Round(x), Round(y));
}
```

PRACTICE PROBLEMS BASED ON DDA ALGORITHM-

Problem-01:

Calculate the points between the starting point (5, 6) and ending point (8, 12).

Solution-

Given-

- Starting coordinates = (X_0, Y_0) = (5, 6)
- Ending coordinates = (X_n, Y_n) = (8, 12)

Step-01:

Calculate ΔX, ΔY and M from the given input.

ΔX = X_n – X_0 = 8 – 5 = 3

ΔY =Y_n – Y_0 = 12 – 6 = 6

M = ΔY / ΔX = 6 / 3 = 2

Step-02:

Calculate the number of steps.

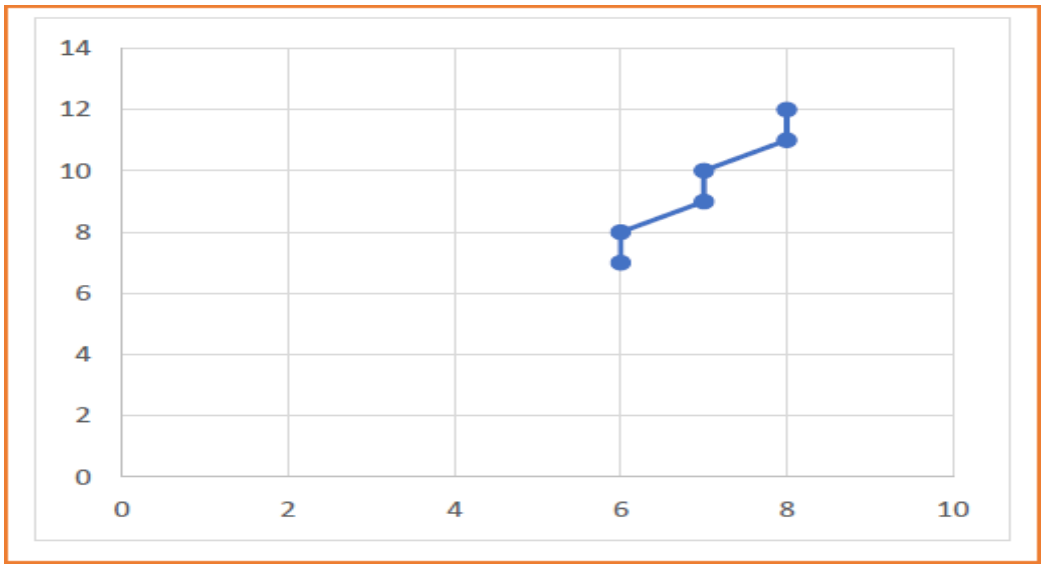
As |ΔX| < |ΔY| = 3 < 6, so number of steps = ΔY = 6

Step-03:

As M > 1, so case-03 is satisfied.

Now, Step-03 is executed until Step-04 is satisfied.

X_p	Y_p	X_{p+1}	Y_{p+1}	Round off (X_{p+1}, Y_{p+1})
5	6	5.5	7	(6, 7)
		6	8	(6, 8)
		6.5	9	(7, 9)
		7	10	(7, 10)
		7.5	11	(8, 11)
		8	12	(8, 12)



Problem-02:

Calculate the points between the starting point (5, 6) and ending point (13, 10).

Solution-

Given-

Starting coordinates = (X_0, Y_0) = (5, 6)

Ending coordinates = (X_n, Y_n) = (13, 10)

Step-01:

Calculate ΔX, ΔY and M from the given input.

$\Delta X = X_n - X_0 = 13 - 5 = 8$

$\Delta Y = Y_n - Y_0 = 10 - 6 = 4$

$M = \Delta Y / \Delta X = 4 / 8 = 0.50$

Step-02:

Calculate the number of steps.

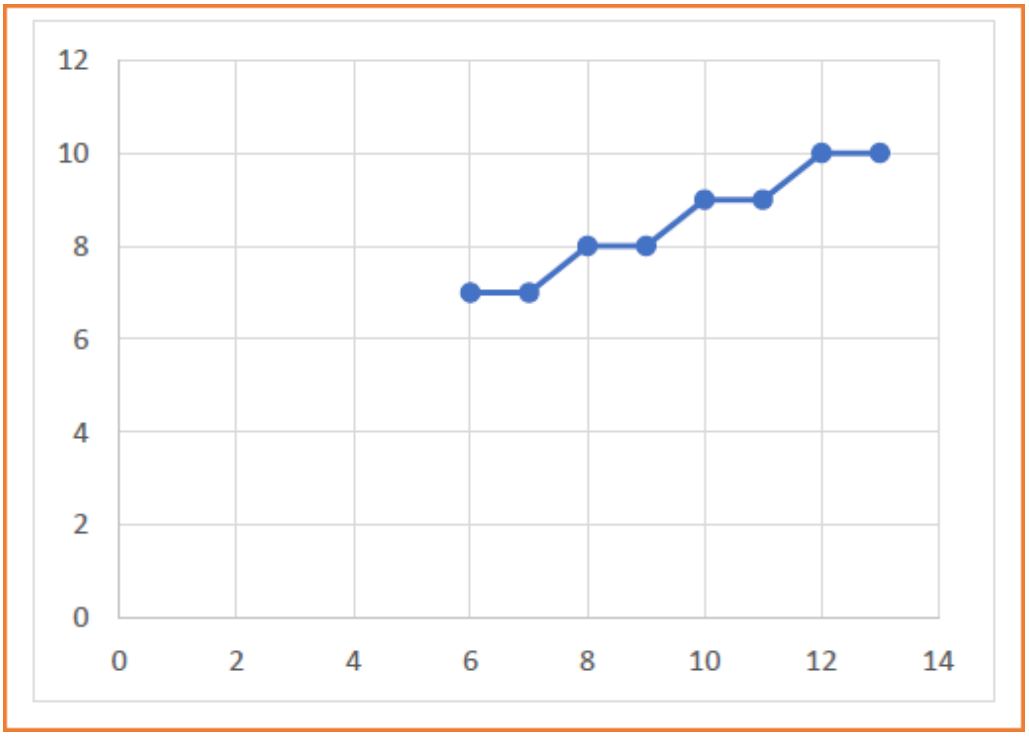
As $|\Delta X| > |\Delta Y| = 8 > 4$, so number of steps = $\Delta X = 8$

Step-03:

As M < 1, so case-01 is satisfied.

Now, Step-03 is executed until Step-04 is satisfied.

X_p	Y_p	X_{p+1}	Y_{p+1}	Round off (X_{p+1}, Y_{p+1})
5	6	6	6.5	(6, 7)
		7	7	(7, 7)
		8	7.5	(8, 8)
		9	8	(9, 8)
		10	8.5	(10, 9)
		11	9	(11, 9)
		12	9.5	(12, 10)
		13	10	(13, 10)



Problem-03:

Calculate the points between the starting point (1, 7) and ending point (11, 17).

Solution-

Given-

- Starting coordinates = $(X_0, Y_0) = (1, 7)$
- Ending coordinates = $(X_n, Y_n) = (11, 17)$

Step-01:

Calculate ΔX, ΔY and M from the given input.

$\Delta X = X_n - X_0 = 11 - 1 = 10$

$\Delta Y = Y_n - Y_0 = 17 - 7 = 10$

$M = \Delta Y / \Delta X = 10 / 10 = 1$

Step-02:

Calculate the number of steps.

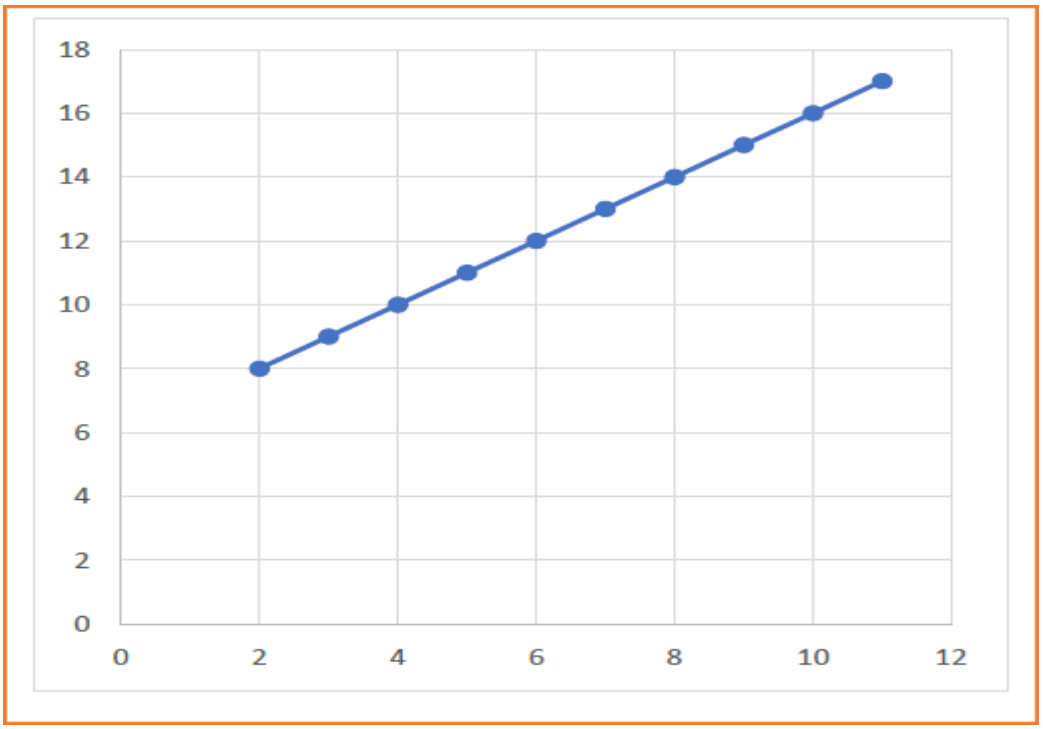
As $|\Delta X| = |\Delta Y| = 10 = 10$, so number of steps = $\Delta X = \Delta Y = 10$

Step-03:

As M = 1, so case-02 is satisfied.

Now, Step-03 is executed until Step-04 is satisfied.

X_p	Y_p	X_{p+1}	Y_{p+1}	Round off (X_{p+1}, Y_{p+1})
1	7	2	8	(2, 8)
		3	9	(3, 9)
		4	10	(4, 10)
		5	11	(5, 11)
		6	12	(6, 12)
		7	13	(7, 13)
		8	14	(8, 14)
		9	15	(9, 15)
		10	16	(10, 16)
		11	17	(11, 17)



Problem 04

Consider the line from (0,0) to* (4,6). *Use the simple DDA algorithm to rasterize this line.

Solution

Evaluating steps 1 to 5 in the DDA algorithm we have

$X_1 = 0$ $Y_1 = 0$

$X_2 = 4$ $Y_2 = 6$

$Length = |Y_2-Y_1| = 6$

$\Delta X = |X_2- X_1| / Length$

$= \frac{4}{6}$

$\Delta Y = |Y_2- Y_1| / Length$

$= 6/6=1$

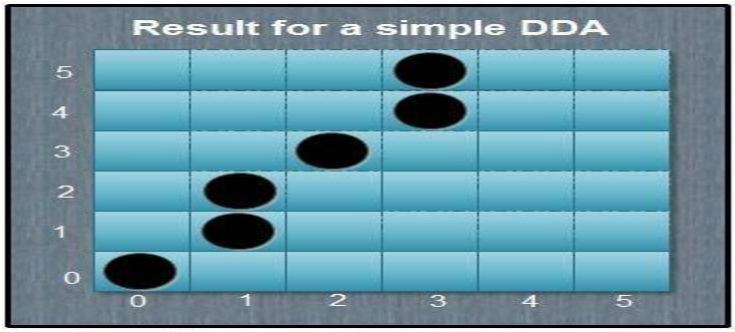
Initial value for

$X = 0 + 0.5 * Sign (\frac{4}{6}) = 0.5$

$Y = 0 + 0.5 * Sign (1) = 0.5$

Tabulating the result of each iteration in the step 6 we get,

i	Plot	x	Y
		0.5	0.5
1	(0.0)		
		1.167	1.5
2	(1, 1)		
		1.833	2.5
3	(1, 2)		
		2.5	3.5
4	(2, 3)		
		3.167	4.5
5	(3, 4)		
		3.833	5.5
6	(3.5)		
		4.5	6.5



The results are plotted as shown in the Fig. It shows that the rasterized line lies to both sides of the actual line, i.e. the algorithm is orientation dependent.

4.3.2 Bresenham's Line Generation

Bresenham's line algorithm is an algorithm that determines the points of an ***n-dimensional raster*** that should be selected in order to form a close approximation to a straight line between two points. It is commonly used to draw line primitives in a bitmap image (e.g. on a computer screen), as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures. It is an incremental error algorithm. It is one of the earliest algorithms developed in the field of computer graphics. An extension to the original algorithm may be used for drawing circles.

The Bresenham's algorithm is another incremental scan conversion algorithm. The **Bresenham's algorithm** is more accurate and efficient compared to DDA algorithm because it cleverly avoids the "Round" function and scan converts using only incremental integer calculation. The algorithm samples a line by incrementing by one unit either x or y depending on the slope of the line and selects the pixels lying at least distance from the true path at each sampling position.

DDA and Bresenham's algorithms both are efficient line drawing algorithm. Bresenham's algorithm has the following advantages on DDA:

(a) DDA uses float numbers and uses operators such as division and multiplication in its calculation. Bresenham's algorithm uses integer and only uses addition and subtraction.

(b) Due to the use of only addition, subtraction and bit shifting, Bresenham's algorithm is faster than DDA in producing the line.

(c) Fixed point DDA algorithms are generally superior to Bresenham's algorithm on modern computers. The reason is that Bresenham's algorithm uses a conditional branch in the 3 loop and this result in frequent branch mis-predictions in the CPU.

(d) Fixed point DDA also has fewer instructions in the loop body (one-bit shift, one increment and one addition to be exact. In addition to the loop instructions and the actual plotting. As CPU pipelines become deeper mis predictions penalties will become more severe.

(e) Since DDA uses rounding off of the pixel position obtained by multiplication of division, it causes an accumulation of error in the proceeding pixels whereas in Bresenham's line algorithm, the new pixel is calculated with a small unit change in one direction and checking of nearest pixel with the decision variable satisfying the line equation.

(f) Fixed point DDA does not require conditional jumps several lines in parallel can be computed with SIMD (Single Instruction Multiple Data techniques.

Advantages of Bresenham's Line Drawing Algorithm-

The advantages of Bresenham's Line Drawing Algorithm are-

- It is easy to implement.
- It is fast and incremental.
- It executes fast but less faster than DDA Algorithm.
- The points generated by this algorithm are more accurate than DDA Algorithm.
- It uses fixed points only.

Disadvantages of Bresenham's Line Drawing Algorithm-

The disadvantages of Bresenham's Line Drawing Algorithm are-

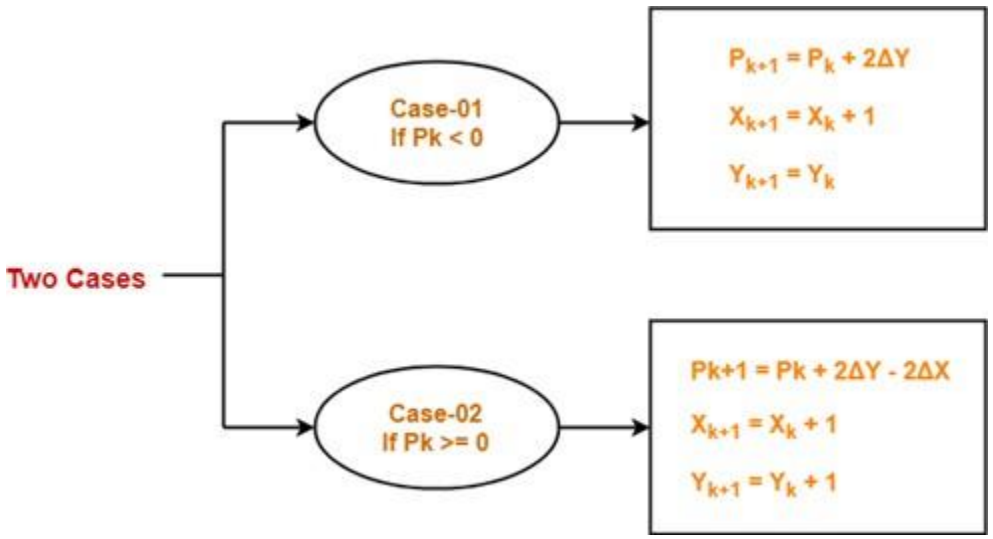
- Though, it improves the accuracy of generated points but still the resulted line is not smooth.
- This algorithm is for the basic line drawing.
- It cannot handle diminishing jaggies.

The Bresenham's procedures is stated below:-

Step 1: Calculate horizontal separation **Δx** and vertical separation **Δy** of the given line endpoints. Plot the pixels **(x_0, y_0)** i.e. the starting point

Step 2: Calculate $P_0 = 2\Delta y - \Delta x$

Step 3: At each x_k along the line, starting at k = 0, check the sign of the decision parameter P_k



if $P_k < 0$,

the next point to plot **$(x_k + 1, y_{k+1})$** 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 $P_{k+1} = P_k + 2\Delta x - 2\Delta x$

Set k = k + 1

Step 4: Repeat step 3 as long as $k < \Delta x$.

PRACTICE PROBLEMS BASED ON BRESENHAM LINE DRAWING ALGORITHM-

Problem-01:

Calculate the points between the starting coordinates (9, 18) and ending coordinates (14, 22).

Solution-

Given-

- Starting coordinates = $(X_0, Y_0) = (9, 18)$
- Ending coordinates = $(X_n, Y_n) = (14, 22)$

Step-01:

Calculate ΔX and ΔY from the given input.

- $\Delta X = X_n - X_0 = 14 - 9 = 5$
- $\Delta Y = Y_n - Y_0 = 22 - 18 = 4$

Step-02:

Calculate the decision parameter.

$$P_k = 2\Delta Y - \Delta X$$
$$= 2 \times 4 - 5$$
$$= 3$$

So, decision parameter $P_k = 3$

Step-03:

As $P_k >= 0$, so case-02 is satisfied.

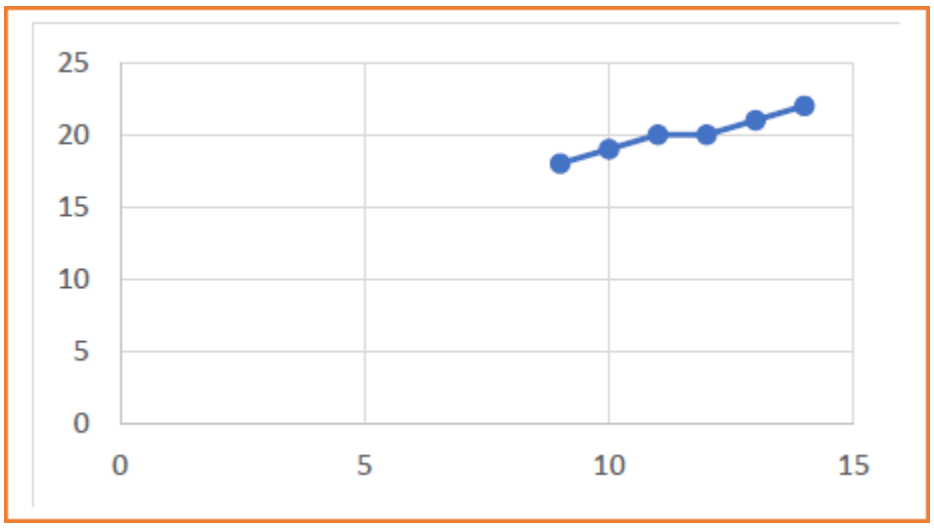
Thus,

- $P_{k+1} = P_k + 2\Delta Y - 2\Delta X = 3 + (2 \times 4) - (2 \times 5) = 1$
- $X_{k+1} = X_k + 1 = 9 + 1 = 10$
- $Y_{k+1} = Y_k + 1 = 18 + 1 = 19$

Similarly, Step-03 is executed until the end point is reached or number of iterations equals to 4 times.

(Number of iterations = $\Delta X - 1 = 5 - 1 = 4$)

P_k	P_{k+1}	X_{k+1}	Y_{k+1}
		9	18
3	1	10	19
1	-1	11	20
-1	7	12	20
7	5	13	21
5	3	14	22



Problem-02:
Calculate the points between the starting coordinates (20, 10) and ending coordinates (30, 18).

Solution-
Given-

- Starting coordinates = $(X_0, Y_0) = (20, 10)$
- Ending coordinates = $(X_n, Y_n) = (30, 18)$

Step-01:
Calculate ΔX and ΔY from the given input.

- $\Delta X = X_n - X_0 = 30 - 20 = 10$
- $\Delta Y = Y_n - Y_0 = 18 - 10 = 8$

Step-02:
Calculate the decision parameter.

$$P_k = 2\Delta Y - \Delta X$$
$$= 2 \times 8 - 10$$
$$= 6$$

So, decision parameter $P_k = 6$

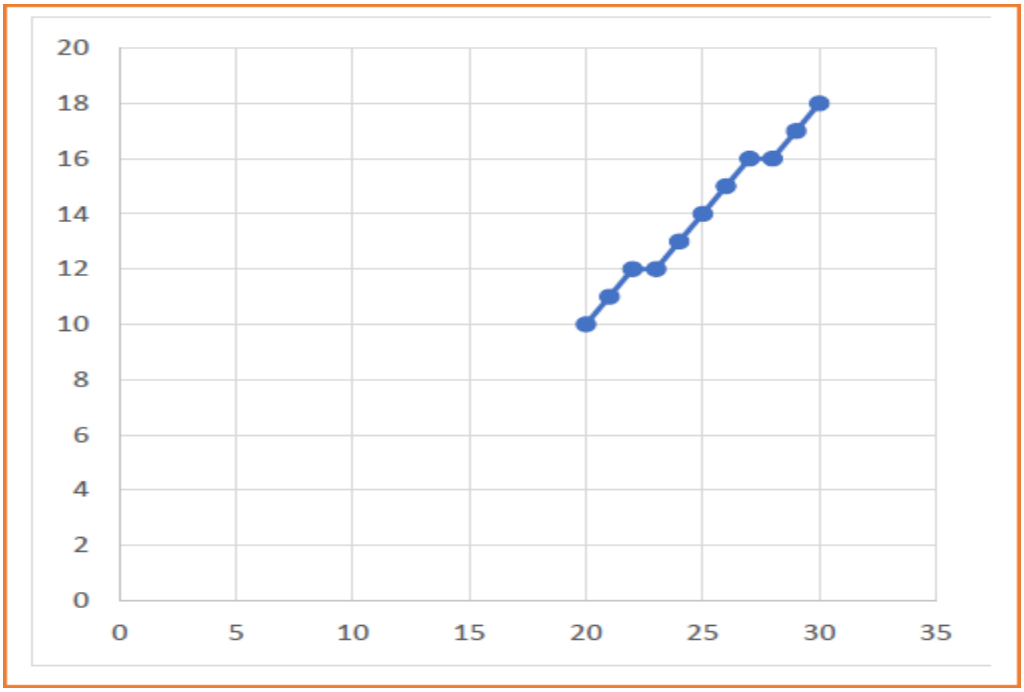
Step-03:
As $P_k >= 0$, so case-02 is satisfied.

Thus,

- $P_{k+1} = P_k + 2\Delta Y - 2\Delta X = 6 + (2 \times 8) - (2 \times 10) = 2$
- $X_{k+1} = X_k + 1 = 20 + 1 = 21$
- $Y_{k+1} = Y_k + 1 = 10 + 1 = 11$

Similarly, Step-03 is executed until the end point is reached or number of iterations equals to 9 times.
(Number of iterations = ΔX – 1 = 10 – 1 = 9)

P_k	P_{k+1}	X_{k+1}	Y_{k+1}
		20	10
6	2	21	11
2	-2	22	12
-2	14	23	12
14	10	24	13
10	6	25	14
6	2	26	15
2	-2	27	16
-2	14	28	16
14	10	29	17
10	6	30	18



Problem-03:

Digitize a line from (0,2) to point (4.5) using Bresenham's Algorithm and plot the graph.

Solution

$(x_0, y_0) = (0, 2)$

$\Delta y = 5 - 2, \Delta x = 4 - 0$

Slope $m = 5 - 2/4 - 0 = ¾ < 1$

Now calculate the successive decision parameter P_k and corresponding (x_{k+1}, y_{k+1}) closest to the line path as follows:

$P_0 = 2\Delta y - \Delta x = 2 > 0$, $x_1 = 1, y_1 = y_o + 1 = 3$

$P_1 = P_0 + 2\Delta y - 2\Delta x = 2 + 2(3) - 2(4) = 0$

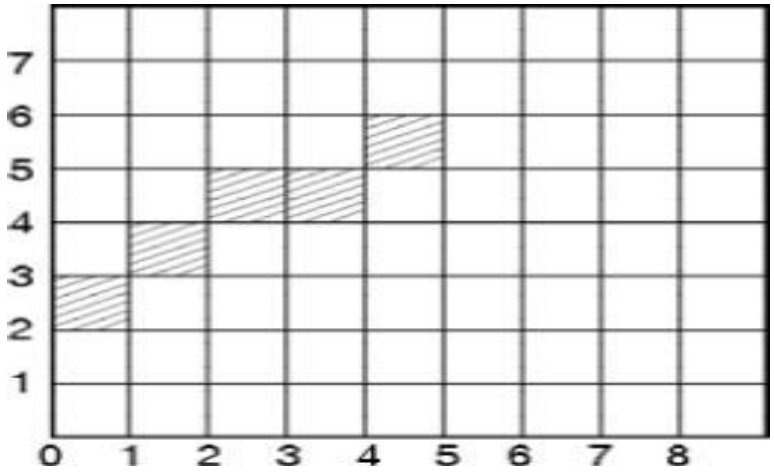
$x_2 = 2, y_2 = y_1 + 1 = 4, P_2 = P_1 + 2\Delta y - 2\Delta x = 0 + 2(3) - 2(4) = -2 < 0$

$x_3 = 3, y_3 = y_2 = 4, P_3 = P_2 + 2\Delta y = -2 + 2(3) = 4 > 0$

$x_4 = 4, y_4 = y_3 + 1 = 5$

We stop further calculation because the endpoint (4, 5) is reached.

k	**P_k**	**Coordinate of pixel to be plotted**	
		x_{k+1}	**y_{k+1}**
		$x_0 = 0$	$y_0 = 2$ start pixel
0	$P_0 = 2$	$x_1 = 2$	$y_1 = 3$
1	$P_1 = 0$	$x_2 = 2$	$y_2 = 4$
2	$P_2 = -2$	$x_3 = 3$	$y_3 = 4$
3	$P_3 = 4$	$x_4 = 4$	$y_4 = 5$ pixel end



4.4 Character Recognition

Letters, numbers, and other characters can be displayed in a variety of sizes and styles. The overall design style for a set (or family) of characters is called a *typeface*. Today, there are hundreds of typefaces available for computer applications. Examples of a few common typefaces are *Courier, Helvetica, New York, Palatino,* and *Zapf Chancery*. Originally, the term font referred to a set of cast metal character forms in a particular size and format, such as 10-point Courier Italic or *12*- point Palatino Bold. Now, the terms font and typeface are

often used interchangeably, since printing is no longer done with cast metal forms. The three basic methods to generate characters on a computer screen are:

- ✓ Hardware-based
- ✓ Vector-based
- ✓ Bit map-based methods.

4.4.1 Hardware-based Methods

In the Hardware-based method, the logic for generating character is built into the graphics terminal. Though the generation time is less, the typefaces are limited due to hardware restrictions.

4.4.2 Vector-based Methods

In the vector-based method, the characters are developed using a set of polylines and splines that approximates the character outline. This form of character representation is completely device-independent and memory requirement is less as boldface, italics or different sizes can be produced by manipulating the curves outlining the character shapes, as it does not require separate memory blocks for each variation.

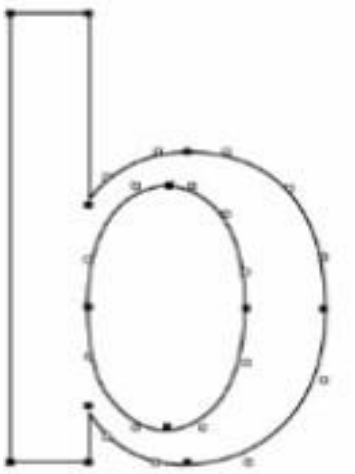


Figure 6.1: Vector-Based Font Generated with Line and Spline Passing Through Control Points

4.4.3 Bit map-based methods

In the bitmap-based method, small rectangular bitmap called character mask (containing binary values 1 and 0) is used to store pixel representation of each character in a framebuffer area known as *font cache*. Relative pixel locations corresponding to a character bitmap are marked depending on the size, face and style (font) of character. Size of each character masks range from 5 × 7 to 10 × 12. A single font in 10 different font sizes and 4 aces (normal, bold, italic, bold italic) would require 40 *font caches*. Characters are actually generated on display

by copying the appropriate bitmaps from the frame buffer to the desired screen positions. A mask is referenced by the coordinate of the origin (lower left corner) of the mask with respect to frame buffer addressing system.

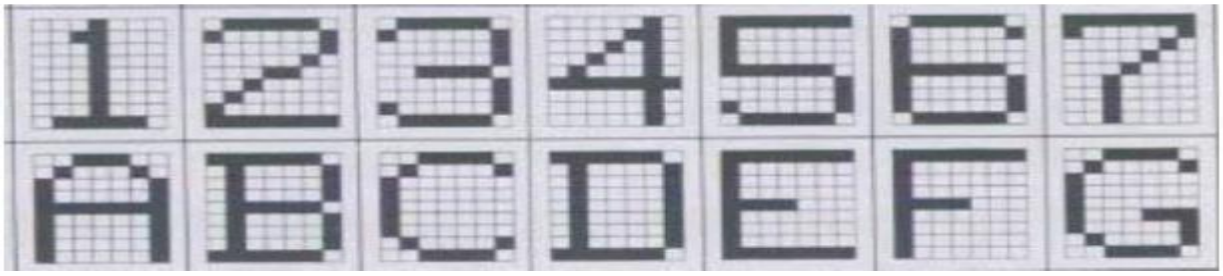


Figure 6.2: Bitmapped Font

In bitmap-based method, a bold face character is obtained by writing the corresponding 'normal' character mask in consecutive frame buffer x-locations. Italics character is produced by necessary skewing of 'normal' character mask while being written in the frame buffer. In fact, a typeface designer can create from scratch new fonts using a program like WindowsPaint. The overall design style (font and face) for a set of characters is called a *typeface.*

4.5 Anti- Aliasing

When implementing the various scan conversion algorithms discussed so far in a computer, one will soon discover that sometimes same pixel is unnecessarily set to same intensity multiple times; sometimes some objects appear dimmer or brighter though all are supposed to have same intensity; and most of the objects generated are not smooth and appear to have rough edges. All these are standard side effects of scan conversion: the first type is called *over striking effect,* the second is *unequal intensity effect* while the third type which is the most common and most pronounced, known as *aliasing effect*. In fact, aliasing is a typical image quality problem on all pixel devices, including computer screen. Aliasing occurs when real-world objects, which comprise of smooth continuous curves are rasterized using pixels.

Aliasing is the stair-step effect or jaggies on the edges of objects displayed on computer screens in which the diagonal and curved lines are displayed as a series of little zigzag horizontal and vertical lines and can be extremely distracting for PC users. Among the other effects of aliasing, jagged profiles and disintegrating textures are very common.

Aliasing *Notice how the letters have jagged edges (except i, l and the horizontal portion of letter A which are either perfectly horizontal or vertical).*	 *If we zoom in on the letter A, it is easy to see what is happening.*

Figure 6.3: The Aliasing Effect

Figure 6.4: Aliasing is prominent in the entity primitives drawn –inclined line, circle and ellipse.

These jaggies are essentially caused by the problem of trying to map a continuous image onto a discrete grid of pixels. This continuous-to-discrete transformation (known as scan conversion) is performed by sampling the continuous line, curve, etc. at discrete points(integer pixel positions) only followed by generating image pixels at integer locations that only approximate the true location of the sampled points. Pixels so generated at alias locations constitute aliases of the true objects or object edges. Therefore, we can say aliasing occurs as a result of an insufficient sampling rate and approximation error (or more specifically quantization error). In fact, aliasing is a potential problem whenever an analog signal is point sampled to convert it into a digital signal. It can occur in audio sampling, for example, in converting music to digital forms to be stored on a CD-ROM or other digital devices. Whenever an analog audio signal is not sampled at a high enough frequency, aliasing manifests itself in the form of spurious low frequencies.

Antialiasing is a technique used in computer graphics to remove the *aliasing effect.* The aliasing effect is the appearance of jagged edges or "jaggies" in a rasterized image (an image rendered using pixels). *Antialiasing* implies the techniques used to diminish the jagged edges of an image so that the image appears to have smoother lines. The problem of jagged edges technically occurs due to distortion of the image when scan conversion is done with sampling at a low frequency, which is also known as *Undersampling.*

Undersampling results in loss of information of the picture and occurs when sampling is done at a frequency lower than *Nyquist sampling frequency*. To avoid this loss, we need to have our sampling frequency at least twice that of highest frequency occurring in the object. This minimum required frequency is referred to as *Nyquist sampling frequency (fs) and is given by*:

$$f_s = 2 * f_{max}$$

What antialiasing does is to change the pixels around the edges to intermediate colours or grayscales. This has the effect of making the edges look smoother although it also makes them fuzzier.

There are quite a few standard methods for antialiasing and some of them are:

- Using high-resolution display,
- Post filtering (Supersampling),
- Pre-filtering (Area Sampling),
- Pixel phasing.

There are other methods such as *Adaptive Sampling, Stochastic Sampling, Raytracing, etc*. However, as an introduction to this topic, we may stop here now. *However, it should be noted that aliasing is an inherent part of any discrete process, its effect can be minimised but not eliminated*.

4.5.1 Using high-resolution display

As *Aliasing* problem is due to low sampling rate for low resolution. One easy solution is to increase the resolution, causing sample points to occur more frequently. Using high resolution, the jaggies become so small that they become indistinguishable by the human eye. This in turn will reduce the size of the pixels. The size of the 'jaggy' or stair-step error is never larger than the size of the actual pixel. Hence, reducing the size of the pixel reduces the size of these steps. Hence, jagged edges get blurred out and edges appear smooth. Using this method, the cost of image production becomes higher with increase in resolution as it calls for more graphics memory. The computational overhead increases for maintaining a high frame rate (*60 fps*) for bigger frame buffer. Thus, within the present limitations of hardware technology, increased screen resolution is not a feasible solution. As an example of applications of increased resolution to resolve aliasing problem, retina displays in Apple devices, OLED (Organic light-emitting diode) displays have high pixel density due to which jaggies formed are so small that they blurred and indistinguishable by our eyes.

4.5.2 Post filtering (Supersampling),

One of the methods of antialiasing is *Supersampling or Postfiltering*. In this method, more than one sample is sampled per pixel. Every pixel area on the display surface is assumed to be subdivided into a grid of smaller subpixels (*supersamples*). Thus, the screen is treated as having higher resolution than what actually is. A virtual image is calculated at this higher spatial resolution and then mapped (displayed) to the actual frame resolution after combining (averaging) the results of the subpixels. The intensity value of a pixel is the average of the intensity values of all the sampled subpixels within that pixel.

This method is also known as *postfiltering* because filtering is carried out after sampling. *Filtering* means eliminating the high frequencies, i.e., combining the supersamples to compute a pixel colour. This type of filtering is known as *unweighted filtering* because each supersample in a pixel, irrespective of its position, has equal influence in determining the pixel's colour. In other words, an unweighted filter computes an unweighted average. The other type of filter is a *weighted filter*. Through this filter, each *supersample* is multiplied by its corresponding weight and the products are summed to produce a weighted average, which is used as the pixel colour. The weighting given to each sample should depend in some way on its distance from the centre of the pixel. The centre sample within a pixel has maximum weight. An array of values specifying the relative importance (weights) of subpixels can be set for different-sized grids and is often referred to as *Pixel-Weighting Masks.*

4.5.3 Pre-filtering (Area Sampling)

The other standard method of antialiasing is *Area Sampling or Prefiltering*. *Prefiltering* method treats a pixel as an area, and computes pixel colour based on the overlap of the scene's objects with a pixel's area. These techniques compute the shades of grey based on how much of a pixel's area is covered by an object. For example, a modification to Bresenham's algorithm was developed by Pitteway and Watkinson. In this algorithm, each pixel is given an intensity depending on the area of overlap of the pixel and the line.

Compared to postfiltering, prefiltering technique does not take into account calculating subpixel intensities (thus eliminates higher frequencies) before determining pixel intensities. A better technique, weighted area sampling uses the same basic approach, but gives greater weight to area near the centre of the pixel. Prefiltering thus amounts to sampling the shape of the object very densely within a pixel region. For shapes other than polygons, this can be very computationally intensive.

4.5.4 Pixel phasing.

Pixel phasing is an anti-aliasing technique in which stair steps (pixel positions) are smoothed out by moving the electron beam to more nearly approximate positions specified by the object geometry. Some systems allow the size of individual pixels to be adjusted for distributing intensities, which is helpful in pixel phasing.