UNIT II

Output primitives, Line drawing algorithms, Circle Drawing algorithms, Circle drawing algorithms, Polynomials and spline curves, Area filling algorithms, character generation, Attributes of Output primitives, Line, Curve, Area fill, Character and bundled attributes, Anti aliasing techniques.

DDA Algorithm

This algorithm works on the principle of obtaining the successive pixel values based on the differential equation governing the line. Since screen pixels are referred with integer values, or plotted positions, which may only approximate the calculated coordinates – i.e., pixels which are intensified are those which lie very close to the line path if not exactly on the line path which in this case are perfectly horizontal, vertical or 45° lines only. Standard algorithms are available to determine which pixels provide the best approximation to the desired line, one such algorithm is the DDA (Digital Differential Analyser) algorithm. Before going to the details of the algorithm, let us discuss some general appearances of the line segment, because the respective appearance decides which pixels are to be intensified. It is alsoobvious that only those pixels that lie very close to the line path are to be intensified because they are the ones which best approximate the line. Apart from the exact situation of the line path, which in this case are perfectly horizontal, vertical or 45° lines (i.e., slope zero, infinite, one) only. We may also face a situation where the slope of the line is > 1 or < 1.Which is the caseshown in Figure 3.3.

In Figure 3.3, there are two lines. Line 1 (slope<1) and line 2 (slope>1). Now let us discuss the general mechanism of construction of these two lines with the DDA algorithm. As the slope of the line is a crucial factor in its construction, let us consider the algorithm in two casesdepending on the slope of the line whether it is > 1 or < 1. **Case 1**: slope (m) of line is < 1 (i.e., line 1): In this case to plot the line we have to move the direction of pixel in x by 1 unit every time and then hunt for the pixel value of the y direction which best suits the line and lighten thatpixel in order to plot the line.

So, in Case 1 i.e., 0 < m < 1 where x is to be increased then by 1 unit every time and proper y is approximated.**Case 2**: slope (m) of line is > 1 (i.e., line 2) if m > 1 i.e., case of line 2, then the most appropriate strategy would be to move towards the y direction by 1 unit every time and determine the pixel in x direction which best suits the line and get that pixel lightened to plot the line.

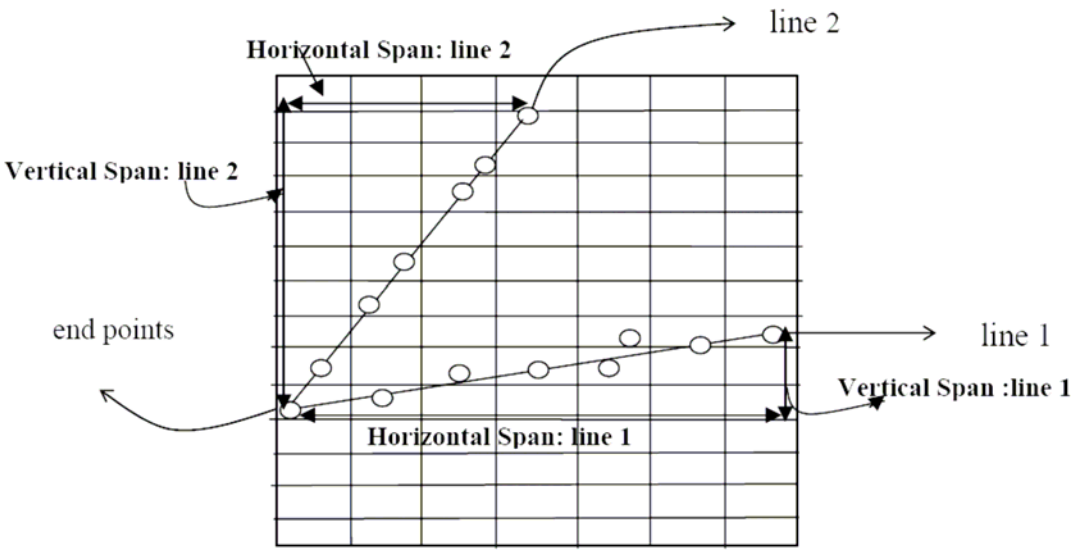


Figure 3.3 DDA Line Generation

So, in Case 2, i.e., (infinity) > m > 1 where y is to be increased by 1 unit every time and proper x is approximated.

Bresenham's Line Algorithm

Bresenham's line-drawing algorithm uses an iterative scheme. A pixel is plotted at the starting coordinate of the line, and each iteration of the algorithm increments the pixel one unit along the major, or x-axis. The pixel is incremented along the minor, or y- axis, only when a decision variable (based on the slope of the line) changes sign. A key feature of the algorithm is that it requires only integer data and simple arithmetic. This makes the algorithm very efficient and fast.

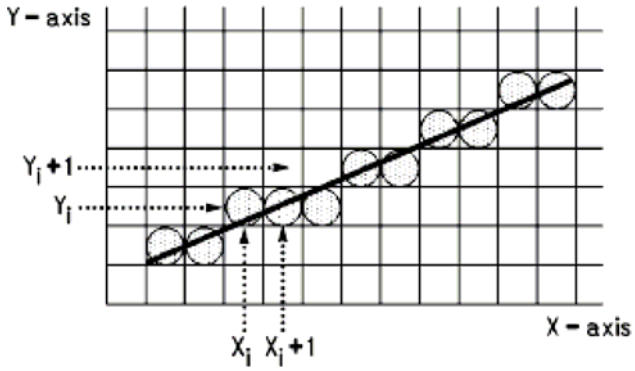


Figure 3.4

The algorithm assumes the line has positive slope less than one, but a simple change ofvariables can modify the algorithm for any slope value.

Bresenham's Algorithm for 0 < slope < 1

Figure 3.4 shows a line segment superimposed on a raster grid with horizontal axis Xand vertical axis Y. Note that xi and yi are the integer abscissa and ordinate respectively of each

pixel location on the grid. Given (xi, yi) as the previously plotted pixel location for the line segment, the next pixel to be plotted is either (xi + 1, yi) or (xi + 1, yi + 1). Bresenham's algorithm determines which of these two pixel locations is nearer to the actual line by calculating the distance from each pixel to the line, and plotting that pixel with the smaller distance. Using the familiar equation of a straight line, y = mx + b, the y value corresponding to xi + 1 is y=m(xi+1) + b The two distances are then calculated as:

d1 = y- yi

d1= m(xi + 1) + b- yi d2 = (yi+ 1)- yd2 = (yi+ 1) -

m(xi + 1)- b and,

d1 - d2 = m(xi+ 1) + b - yi - (yi + 1) + m(xi + 1) + b d1 - d2 = 2m(xi + 1) - 2yi + 2b - 1 Multiplying this result by the constant dx, defined by the slope of the line m = dy/dx, theequation becomes:

dx(d1-d2)= 2dy(xi)- 2dx(yi) + c

where c is the constant 2dy + 2dxb - dx. Of course, if d2 > d1, then (d1-d2) < 0, or conversely if d1 > d2, then (d1-d2) >0. Therefore, a parameter pi can be defined suchthat

pi = dx(d1-d2)

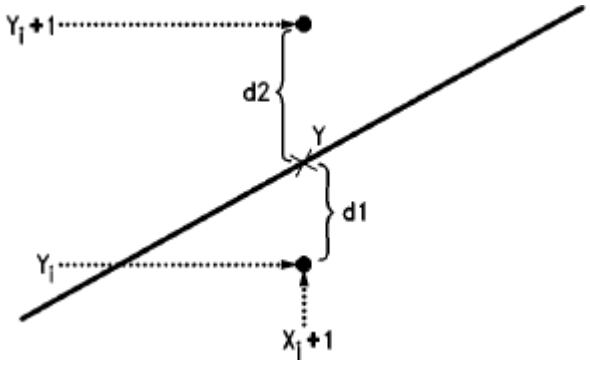


Figure 3.5

pi = 2dy(xi) - 2dx(yi) + c

If pi > 0, then d1 > d2 and y_{i+1} is chosen such that the next plotted pixel is (xi + 1, yi). Otherwise, if pi < 0, then d2 > d1 and (xi + 1, yi + 1) is plotted. (See Figure3.5 .) Similarly, for the next iteration, p_{i+1} can be calculated and compared with zero to determine the next pixel toplot. If $p_{i+1} < 0$, then the next plotted pixel is at (x_{i+1} + 1, Yi+1); if $p_{i+1} < 0$, then the next point is (x_{i+1} + 1, y_{i+1} + 1). Note that in the equation for p_{i+1}, $x_{i+1} = x_{i+1}$.

p_{i+1} = 2dy(xi + 1) - 2dx(yi + 1) + c.

Subtracting pi from p_{i+1}, we get the recursive equation:

$p_{i+1} = pi + 2dy - 2dx(y_{i+1} - yi)$

Note that the constant c has conveniently dropped out of the formula. And, if pi < 0 theny_{i+1}= yi in the above equation, so that:

$p_{i+1} = pi + 2dy$

or, if pi > 0 then yi + 1 = yi + 1, and pi + 1 = pi + 2(dy-dx)

To further simplify the iterative algorithm, constants c1 and c2 can be initialized at the beginning of the program such that c1 = 2dy and c2 = 2(dy-dx). Thus, the actual meat of the algorithm is a loop of length dx, containing only a few integer additions and two compares (Figure 3.5) .

Step 1 − Input the two end-points of line, storing the left end-point in (x_0,y_0)

Step 2 − Plot the point (x_0,y_0)

Step 3 − Calculate the constants dx, dy, 2dy, and (2dy – 2dx) and get the first value for the decision parameter as −

$$p_0=2dy-dx$$

Step 4 − At each X_i

along the line, starting at i = 0, perform the following test −

If $pi < 0$, the next point to plot is $(xi+1,yi)$ and

$$pi_{+1}=pi+2dy$$

Otherwise,

$$(xi,yi+1)$$
$$pi_{+1}=pi+2dy-2dx$$

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 P_i will be very similar, just the x and y in the equation gets interchanged.

Scan-converting a Circle

Circle is one of the basic graphic component, so in order to understand its generation, let us go through its properties first. A circle is a symmetrical figure. Any circle- generating algorithm can take advantage of the circle's symmetry to plot of eight points for each value that the algorithm calculates. Eight-way symmetry is used by reflecting each calculated point around each 45˚ axis. For example, if point 1 in Fig.

were calculated with a circle algorithm, seven more points could be found by reflection. The reflection is accomplished by reversing the x, y coordinates as in point 2, reversing the x,y coordinates and reflecting about the y axis as in point 3, reflecting about the y axis in point 4, switching the signs of x and y as in point 5, reversing the x, y coordinates, reflecting about they axis and reflecting about the x axis.

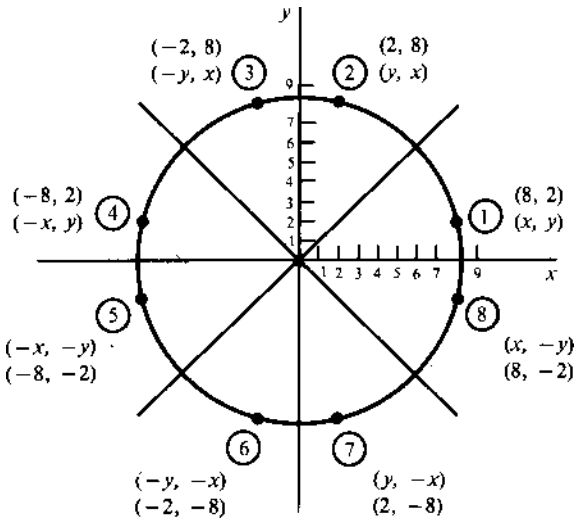


Figure 3.6: Eight–way symmetry of a circle.

As in point 6, reversing the x, y coordinates and reflecting about the y-axis as in point 7, and reflecting about the x-axis as in point 8.

To summarize:

$P_1 = (x, y)$ $P_5 = (- y,-x)$
$P_2 = (y, x)$ $P_6 = (-y, - x)$
$P_3 = (-y, x)$ $P_7 = (y, -x)$
$P_4 = (-x, y)$ $P_8 = (x, -y)$

Defining a Circle

There are two standard methods of mathematically defining a circle centered at the origin. Thefirst method defines a circle with the second-order polynomial equation (see Fig. 3.7).

$y_2 = r^2 - x^2$

Where *x* = the x coordinate y = the y coordinate r = the circle radius

With this method, each x coordinate in the sector, from 90 to 45˚, is found by stepping

x from 0 to r/ $\sqrt{2}$, and each y coordinate is found by evaluating $\sqrt{r^2 + x^2}$ for each step of x. This is a very inefficient method, however, because for each point both x and r must besquared and subtracted from each other; then the square root of the result must be found.

The second method of defining a circle makes use of trigonometric functions (see Fig. 3.8):x = r cos □ y = r sin □

where □ = current angle r = circle radius x = x coordinate y = y coordinate

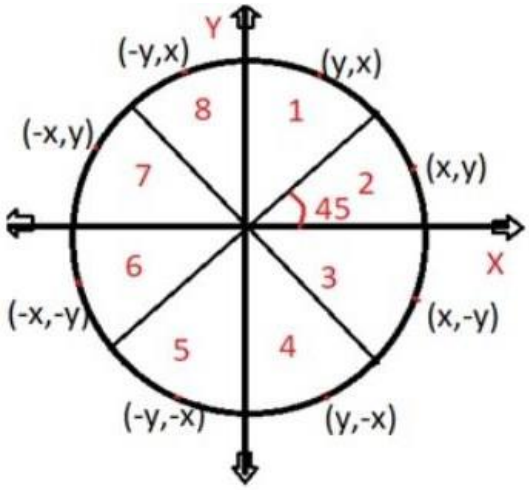
By this method, □ is stepped from □ to □/4, and each value of x and y is calculated. However, computation of the values of sin □ and cos □ is even more time-consuming than the calculations required by the first method.

There are two algorithm to do this:

1. **Mid-Point circle drawing algorithm**
2. **Bresenham's circle drawing algorithm**

Midpoint circle Algorithm

This is an algorithm which is used to calculate the entire perimeter points of a circle in a first octant so that the points of the other octant can be taken easily as they are mirror points; this isdue to circle property as it is symmetric about its center.



In this algorithm decision parameter is based on a circle equation. As we knowthat the equation of a circle is $x^2 + y^2 = r^2$ when the centre is (0, 0).

Now let us define the function of a circle i.e.: **fcircle(x,y)= $x^2 + y^2 - r^2$**

1. If **fcircle < 0** then **x**, **y** is inside the circle boundary.
2. If **fcircle > 0** then **x**, **y** is outside the circle boundary.
3. If **fcircle = 0** then **x**, **y** is on the circle boundary.

Decision parameter

p_k =fcircle(x_{k+1},$y_{k-1/2}$) where **p_k** is a decision parameter and in this **½** is taken because it is a midpoint value through which it is easy to calculate value of **y_k** and**y_{k-1}**.

I.e. **$p_k = (x_{k+1})^2 + (y_{k-1/2})^2 - r^2$**

If **p_k <0** then midpoint is inside the circle in this condition we select **y** is **y_k**
otherwise we will select next **y** as **y_{k-1}** for the condition of **p_k > 0**.**Conclusion**

1. If **p_k < 0** then **$y_{k+1}=y_k$**, by this the plotting points will be **(x_{k+1} ,y_k)**. By thisthe value for the next point will be given as:
   $P_{k+1}=p_k +2(x_{k+1}) +1$
2. If **p_k > 0** then **$y_{k+1}=y_{k-1}$**, by this the plotting points will be **(x_{k+1}, y_{k-1})**. Bythis the value of the next point will be given as:
   $P_{k+1}=p_k+2(x_{k+1}) +1-2(y_{k+1})$

Initial decision parameterP_0 = fcircle (1,

r-1/2)

This is taken because of **$(x_0, y_0) = (0, r)$**

i.e. **p_0 =5/4-r or 1-r**, (**1-r** will be taken if **r** is integer)

ALGORITHM

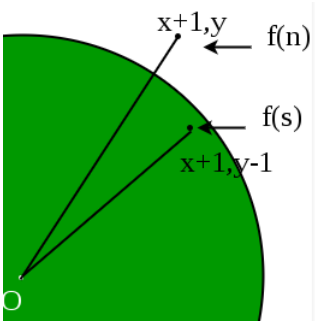
1. In this the input radius **r** is there with a centre **(x_c , y_c)**. To obtain the first point **m** the circumference of a circle is centered on the origin as**$(x_0,y_0) = (0,r)$**.
2. Calculate the initial decision parameters which are:
   p_0 =5/4-r or 1-r
3. Now at each **x_k** position starting **k=0**, perform the following task.if **$p_k < 0$** then plotting point will be **(x_{k+1} ,y_k)** and
   $P_{k+1}=p_k +2(x_{k+1}) +1$
   else the next point along the circle is (x_{k+1}, y_{k-1}) and
   $P_{k+1}=p_k+2(x_{k+1}) +1-2(y_{k+1})$
4. Determine the symmetry points in the other quadrants.
5. Now move at each point by the given centre that is:
   $x=x+x_c y=y+y_c$
6. At last repeat steps from 3 to 5 until the condition **x>=y**.

Bresenham's Circle drawing algorithm

It is not easy to display a continuous smooth arc on the computer screen as our computer screen is made of pixels organized in matrix form. So, to draw a circle on a computer screen we should always choose the nearest pixels from a printed pixel so as they could form an arc.

The Bresenham's circle drawing algorithm. Both of these algorithms uses the key feature of circle that it is highly symmetric. So, for whole 360 degree of circle we will divide it in 8-parts each octant of 45 degree. In order to that we will use Bresenham's Circle Algorithm for calculation of the locations of the pixels in the first octant of 45 degrees. It assumes that the circle is centered on the origin. So for every pixel (x, y) it calculates, we draw a pixel in each of the 8 octants of the circle as shown below :

Now, we will see how to calculate the next pixel location from a previously known pixel location (x, y). In Bresenham's algorithm at any point (x, y) we have two option either to choose the next pixel in the east i.e. (x+1, y) or in the south east i.e. (x+1, y-1).



And this can be decided by using the decision parameter d as:

- If d > 0, then (x+1, y-1) is to be chosen as the next pixel as it will be closerto the arc.
- else (x+1, y) is to be chosen as next pixel.

Now to draw the circle for a given radius _r' and centre (xc, yc) We will start from (0, r) and move in first quadrant till x=y (i.e. 45 degree). We should startfrom listed initial condition:

d = 3 - (2 * r)x = 0
y = r

Now for each pixel, we will do the following operations:

1. **Set initial values of (xc, yc) and (x, y)**
2. **Set decision parameter d to d = 3 – (2 * r).**
3. **call drawCircle(int xc, int yc, int x, int y) function.**
4. **Repeat steps 5 to 8 until x < = y**
5. **Increment value of x.**
6. **If d < 0, set d = d + (4*x) + 6**
7. **Else, set d = d + 4 * (x – y) + 10 and decrement y by 1.**
8. **call drawCircle(int xc, int yc, int x, int y) function**

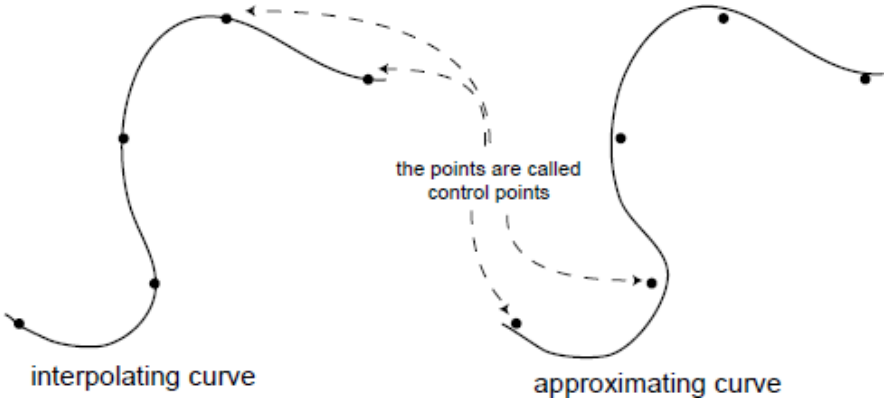
Below is C implementation of above approach.

```
#include    <stdio.h>    #include    <dos.h>
#include <graphics.h>

// Function to put pixels
// at subsequence points
voiddrawCircle(intxc,  intyc,  intx,  inty)
{
        putpixel(xc+x,  yc+y,  RED); putpixel(xc-x,  yc+y,  RED);
        putpixel(xc+x,  yc-y,  RED); putpixel(xc-x,  yc-y,  RED);
        putpixel(xc+y,  yc+x,  RED); putpixel(xc-y,  yc+x,  RED);
        putpixel(xc+y, yc-x, RED);putpixel(xc-y, yc-x, RED);
}

// Function for circle-generation
// using Bresenham's algorithm
voidcircleBres(intxc,  intyc,  intr)
{
        intx = 0, y = r; intd = 3 - 2 * r;
        drawCircle(xc, yc, x, y);while(y >= x)
        {
                // for each pixel we will
                // draw all eight pixelsx++;

                // check for decision parameter
                // and correspondingly
                // update d, x, yif(d > 0)
                {
                        y--;
                        d = d + 4 * (x - y) + 10;
                }
                else
                        d = d + 4 * x + 6;
```

```
            drawCircle(xc, yc, x, y);

                delay(50);
        }
}
```

Spline Curves

A *spline curve* is a mathematical representation for which it is easy to build an interface that will allow a user to design and control the shape of complex curves and surfaces. The general approach is that the user enters a sequence of points, and a curve is constructed whose shape closely follows this sequence. The points are called *control points*. A curve that actually passes through each control point is called an *interpolating curve*; a curve that passes near to the control pointsbut not necessarily through them is called an *approximating curve*. Once we establish this interface, then to change the shape of the curve we just move the control points.



The easiest example to help us to understand how this works is to examine a curve that is like the graph of a function, like y = x2. This is a special case of a polyno- mial function.

Polynomial curves

Polynomials have the general form:

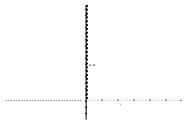
$y = a + bx + cx^2 + dx^3 + \ldots$

The *degree* of a polynomial corresponds with the highest coefficient that is non- zero. For example if *c* is non-zero but coefficients *d* and higher are all zero, the

polynomial is of degree 2. The shapes that polynomials can make are as follows:

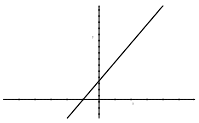
degree 0: *Constant*, only *a* is non-zero.

Example: $y = 3$



A constant, uniquely defined by one point.

degree 1: *Linear*, *b* is highest non-zero coefficient.

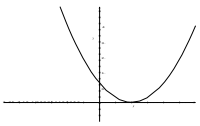


Example: $y = 1 + 2x$

A line, uniquely defined by two points.

degree 2: *Quadratic*, *c* is highest non-zero coefficient.

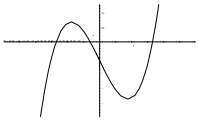
Example: $y = 1 - 2x + x^2$



A parabola, uniquely defined by three points.

degree 3: *Cubic*, *d* is highest non-zero coefficient.

Example: $y = -1 - 7/2x + 3/2x^3$



A cubic curve (which can have an inflection, at $x = 0$ in this example), uniquely defined by four points.

The degree three polynomial – known as a *cubic polynomial* – is the one that is most typically chosen for constructing smooth curves in computer graphics. It is used because

1. it is the lowest degree polynomial that can support an inflection – so we can make interesting curves,and
2. it is very well behaved numerically – that means that the curves will usually be smooth like this:and not jumpy like this:

So, now we can write a program that constructs cubic curves. The user entersfour control points, and the program solves for the four coefficients *a, b, c* and *d* which cause the polynomial to pass through the four control points. Below, we work through a specific example.

Typically the interface would allow the user to enter control points by clicking them in with the mouse. For example, say the user has entered control points (1, 2), (0, 0), (1, 2), (2, 0) as indicated by the dots in the figure to the left.

Then, the computer solves for the coefficients *a, b, c, d* and might draw the curveshown going through the control points, using a loop something like this:

```
glBegin(GL LINE STRIP);for(x = -3; x
<= 3; x += 0.25)
    glVertex2f(x, a + b * x + c * x * x + d * x * x * x); glEnd();
```

Note that the computer is not really drawing the curve. Actually, all it is doing is drawing straight line segments through a sampling of points that lie on the curve. If the sampling is fine enough, the curve will appear to the user as a continuous smooth curve.

The solution for *a, b, c, d* is obtained by simultaneously solving the 4 linear equations below, that are obtained by the constraint that the curve must pass through the 4 points:

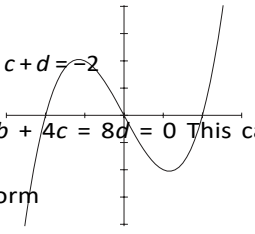
general form: $a + bx + cx^2 + dx^3 = y$

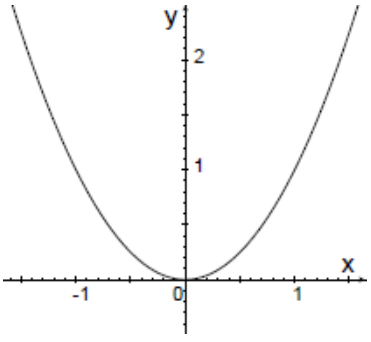
point (−1, 2)**:** $a - b + c - d = 2$

point (0, 0)**:** $a = 0$

point (1, −2)**:** $a + b + c + d = -2$

point (2, 0)**:** $a + 2b + 4c = 8d = 0$ This can be written in matrix form



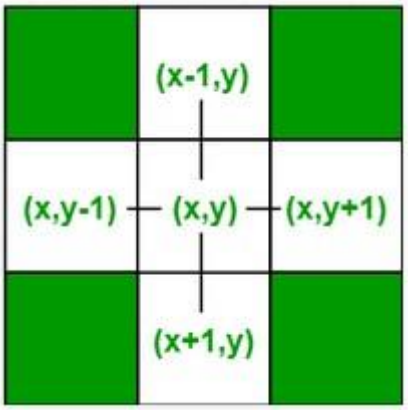


Fill Algorithm

Boundary Fill Algorithm is recursive in nature. It takes an interior point(x, y), a fill color, and a boundary color as the input. The algorithm starts by checking the color of (x, y). If it's color is not equal to the fill color and the boundary color, then it is painted with the fill color and the function is called for all the neighbours of (x, y). If a point is found to be of fill color or of boundary color, the function does not call its neighbours and returns. This process continues until all points up to the boundary color for the region have been tested.

The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.

4-connected pixels : After painting a pixel, the function is called for four neighboring points. These are the pixel positions that are right, left, above and below the current pixel. Areas filled by this method are called 4-connected. Below given is the algorithm :



Algorithm :

```
void boundaryFill4(int x, int y, int fill_color,int boundary_color)
{
    if(getpixel(x, y) != boundary_color &&getpixel(x, y) != fill_color)
    {
        putpixel(x, y, fill_color);
        boundaryFill4(x + 1, y, fill_color, boundary_color); boundaryFill4(x, y + 1, fill_color,
        boundary_color); boundaryFill4(x - 1, y, fill_color, boundary_color); boundaryFill4(x, y - 1,
        fill_color, boundary_color);
    }
}
```

Flood-fill Algorithm

By this algorithm, we can recolor an area that is not defined within a single color boundary.In this, we can paint such areas by replacing a color instead of searching for a boundary color value. This whole approach is termed as flood fill algorithm. This procedure can also be used to reduce the storage requirement of the stack by filling pixel spans.

```
floodfill4 (x, y, fillcolor, oldcolor: integer)begin
        if getpixel (x, y) = old color thenbegin
                setpixel (x ,y, fillcolor)
                floodfill4 (x+1, y, fillcolor, oldcolor) floodfill4 (x-1, y, fillcolor,
                oldcolor) floodfill4 (x, y+1, fillcolor, oldcolor) floodfill4 (x, y-1,
                fillcolor,

end.
```

Character Generation

```
setCharacterHeight (ch)
```

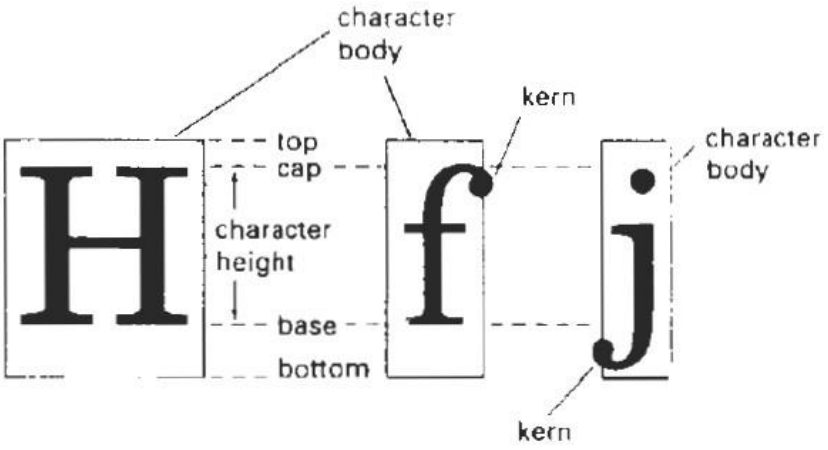


Figure 4-23
Character body.