Chapter 3

Line Drawing Algorithms

~: Learning Objectives :~

The objectives of this chapter are to acquaint you with:

☞ Scan conversion line drawing
☞ Bresenham's line drawing
☞ Drawing bar chart
☞ Problems

3.1 Prologue

In this chapter, we will explore the line drawing algorithms and their implementations in two-dimensional space. Drawing a line is so simple! More so when you draw it in your computer using two-dimensional graphics API calls, with the help of standard compilers. Most of the standard graphics libraries support usual routine with calls like *line* (x_1, y_1, x_2, y_2), where (x_1, y_1) and (x_2, y_2) are respectively the start and the end points of the line. Some graphics libraries even provide more flexibility giving the user a set of choices to select line attributes like thick or thin line, continuous or discontinuous line with various patterns. But how to select those pixels between (x_1, y_1) and (x_2, y_2) in the console screen which when turned "on", form the desired "digital" line? The term digital refers to 2D discrete or integer space representing pixels in the computer screen. Since implementation of this line drawing involves scanning of the computer screen, either row-wise or column-wise, these set of techniques are known as *scan conversion* algorithms. We explore this in the following section.

3.2 Scan Conversion Algorithm: A Simple Line Drawing Algorithm

Consider the line equation $y = mx + c$ with slope m and intercept c. Also, take the point (x_i, y_i) to be the starting point from where the line needs to be drawn. Obviously, the screen being a digital pixel space, (x_i, y_i) are integers and all subsequent "on" pixels are integer points. Given this, the y-coordinate y_{i+1} of the next pixel on the line, for a change of the x-coordinate from x_i to x_{i+1}, is given by

$$\begin{aligned} y_{i+1} &= mx_{i+1} + \mathrm{c} \\ &= m(x_i + dx) + c \\ &= mx_i + c + mdx \\ &= y_i + mdx \end{aligned} \tag{3.1}$$

The increment along x-direction is dx and for all practical purposes, we can take this to be unity, i.e. $dx = 1$. Equation (3.1) then takes the form $y_{i+1} = y_i + m$. But it does not guarantee y_{i+1} to be an integer; because, usually, the slope m would be a real number rather than an integer. To make y_{i+1} an integer, it is necessary that

$$y_{i+1} = \text{round}\,(y_i + m) \tag{3.2}$$

and typically, the rounding[1] function is given by

$$\text{round}\,(a_i) = \text{floor}\,(a_i + 0.5)$$

Therefore, the algorithm for drawing line between points (sx, sy) and (ex, ey) would be:

```
function line (int sx, int sy, int ex, int ey, BYTE color)
{
  dy = ey – sy;
  dx = ex – sx;
  m = dy/dx;
  x = sx;
  y = sy;
  OnPixel(x, y, color);
  for (x = (sx+1); x ≤ ex; x++)
  {
    y = y + m;
    OnPixel(x, round(y), color);
  }
}
```

The implicit assumption behind the above function is that $-1 \le m \le 1$. This implies that for every unit step along x-direction ($dx = 1$), there is a maximum unit step change along the

[1] Rounding a non-negative number to the nearest integer is carried out by adding 0.5 and truncating (taking the floor value). Floor function of a number is the largest integer not greater than the number. Thus, floor (3.2)=floor(3.0)=floor(3.8)=3 and round(3.0)=round(2.5)=round(2.7)=round(3.2) =3

y-direction ($y_{i+1} - y_i = dy \le 1$). This situation does not hold for $m > 1$ and $dx = 1$, $dy > 1$. How will it affect the continuity of the line? Think carefully, you still get a line, but there are chances that a few points will be scattered, and an undesired gap between two subsequent points on the line may exist. Discontinuity will appear on the line as the difference between two consecutive y-coordinate values of pixels will be greater than unity. The remedy is simple! For $m > 1$, reverse the increment order:

Increment along y-axis: $y_{i+1} = y_i + dy, \quad dy = 1$

Increment along x-axis:
$$\begin{aligned} x_{i+1} &= \frac{y_{i+1}}{m} - \frac{c}{m} \\ &= \frac{y_i + dy}{m} - \frac{c}{m} \\ &= \frac{y_i}{m} + \frac{dy}{m} - \frac{c}{m} \\ &= \left(\frac{y_i}{m} - \frac{c}{m}\right) + \frac{1}{m} \text{ (Since } dy = 1) \\ &= x_i + \frac{1}{m} \end{aligned} \tag{3.3}$$

To get the corresponding integer x-coordinate, x_{i+1} = round ($x_i + 1/m$). This process is similar to solving a differential equation by the numerical methods, and is often referred to as *digital differential analyzer* (DDA). This works well but what happens if you are conscious about the time complexity of the algorithm. Well, in that case the grey area is obviously the rounding function, especially when it is being executed quite a few times while drawing even a small line. Let us see if we can improve this aspect.

3.3 Bresenham's Scan Conversion Algorithm

Bresenham has provided a computationally attractive scan conversion algorithm, commonly known as *Bresenham's scan conversion algorithm* for line drawing. Following Bresenham algorithm, let us rewrite the line equation $y = mx + c$ in the form $y = (dy/dx)x + c$. Equivalently, a function $f(x, y)$ is given by

$$f(x, y) = x(dy) - y(dx) + c(dx) = 0 \tag{3.4}$$

Comparing Eq. (3.4) with a line equation of the form $Ax + By + C = 0$, the coefficients are given by $A = dy$, $B = -dx$ and $C = c(dx)$. Referring to the line in Figure 3.1, and Eq. (3.4), $f(x, y) = 0$ for the point $\mathbf{P}(x, y)$ on the line, positive for points above the line and negative for points below the line. Stating this with respect to the mid point $m(x + 1, y + 1/2)$, as shown in Figure 3.1, we have to evaluate the function of Eq. (3.4) with the following conditions in order to select the next point on the line:

- If $f(x + 1, y + 1/2) < 0$, we choose pixel NE to be on the digital line
- If $f(x + 1, y + 1/2) < 0$, we choose pixel E to be on the digital line

- If $f(x + 1, y + 1/2) = 0$, we choose any one of pixel E or NE but we should be consistent throughout the entire line drawing process.

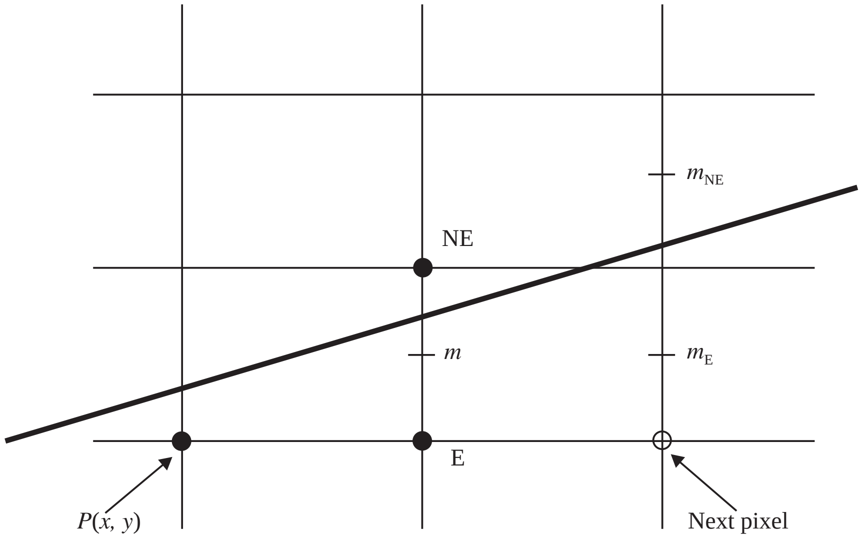

Figure 3.1 Illustration of the Bresenham algorithm (for the current point **P**(x, y) and the mid point m, if $f(m) > 0$, the NE pixel is chosen; otherwise, the choice is pixel E).

Now, let us formalize the selection of the next pixel on the line against the current selection of East (E) or North-East (NE) pixel. If E is chosen as the current pixel, the coordinate of the corresponding (next) mid point m_E is $(x + 2, y + 1/2)$.

$$f(m_E) = f\left(x + 2, y + \frac{1}{2}\right) = A(x + 2) + B\left(y + \frac{1}{2}\right) + C$$

Since

$$f(m) = A(x + 1) + B\left(y + \frac{1}{2}\right) + C$$

Therefore,

$$\Delta f_E = f(m_E) - f(m) = A \tag{3.5}$$

Instead, if NE is chosen as the current pixel following Figure 3.1, then

$$f(m_{NE}) = f\left(x + 2, y + \frac{3}{2}\right) = A(x + 2) + B\left(y + \frac{3}{2}\right) + C$$

Thus,

$$\Delta f_{NE} = f(m_{NE}) - f(m) = A + B \tag{3.6}$$

Further, if the first point on the line, that is, the starting point is (x_0, y_0), the immediate next mid point is $(x_0 + 1, y_0 + 1/2)$. The function f at this mid point is evaluated as

$$\begin{aligned} f\left(x_0+1, y_0+\frac{1}{2}\right) &= A(x_0+1)+B\left(y_0+\frac{1}{2}\right)+C \\ &= (Ax_0+By_0+C)+A+\frac{B}{2} \\ &= f(x_0, y_0)+A+\frac{B}{2} \end{aligned} \tag{3.7}$$

Now, (x_0, y_0) being the start point on the line, $f(x_0, y_0)$ is zero. Therefore, the starting increment is $A + B/2$. Note that even the fraction in the starting increment can be eliminated by multiplying the entire line equation, i.e. Eq. (3.7) by 2: $f(x, y) = 2(Ax + By + C)$. It does not affect the decision based on the sign of the line function value $f(m)$ at the mid point but reduces the complexity by removing the fraction. Then the start increment is changed from $(A + B/2)$ to $(2A + B)$. The advantages are clear—no floating-point operation is involved in the scan conversion. The complete algorithm is, therefore:

```
function BresenhamLine (int sx, int sy, int ex, int ey, BYTE color)
{
  dx = ex - sx; // B = -dx
  dy = ey - sy; // A = dy
  d = 2dy - dx; // the first increment
  inE = 2dy; // increment for selection of E
  inNE = 2(dy - dx);
  // increment for selection of NE
  x = sx;
  y = sy;
  OnPixel(x, y, color); // the first point
  while (x <= ex)
  {
    if d <= 0 then // E is selected
    {
      d = d + inE;
      x = x + 1;
    }
    else
    {
      // NE is selected
      d = d + inNE;
      x = x + 1;
      y = y + 1;
    }
    OnPixel (x, y, color);
  }
}
```

In subsequent sections, we will write a program to implement the scan conversion algorithm and Bresenham's line drawing algorithm.

3.3.1 Implementation

During execution, first a window is displayed and it contains a menu bar with `Line`, `Clear` and `Exit` menu. `Line` menu contains `Scan Line` and `Bresenham's Line` submenus. The `Clear` menu is for erasing the content of the window and `Exit` menu is for stopping the execution process. At any point of time, if the user chooses the `Clear` menu item, the content of the screen is erased and subsequently, all menu items are enabled.

After the user clicks on `Scan Line` or `Bresenham's Line` menu item, the user starts putting the points (two end vertices of the line) on the window by clicking the left mouse button. While drawing the line, the menu item `Scan Line` or `Bresenham's Line` (whichever is used) gets disabled, and only the `Clear` and `Exit` menu items remain enabled. As the user is marking the end vertices of the line on the screen by clicking left mouse buttons, at any point of time if the user decides not to continue or start afresh, the user may choose to exit the application or clear the screen and start afresh. The second left mouse button click puts the second end point and also completes the line drawing algorithm using the chosen algorithm (`Scan Line` or `Bresenham's Line`). Upon clearing the window canvas, the menu items `Scan Line` or `Bresenham's Line` get enabled again.

The execution of the `Clear` and the `Exit` buttons are already discussed in previous chapters. In this chapter, we will follow the program example similar to Program Source Code 2.1 with certain changes as needed.

Also note that whenever the window is resized, the window area is not erased, thus the window content remains intact.

Program files used

Source files

Line.cpp	Line drawing functions (including Bresenham's line drawing) as well as some related supporting functions.
DrawLine.cpp	Other utility functions for drawing utilities and using the line drawing functions.
WMain.cpp	Almost identical as in Program Source Code 2.1; only the window title is changed.

Header files

windows.h	(system header file)
Line.h	Function declarations for using line drawing related functions
DrawLine.h	Line drawing related specific header file

Resource file

Line.rc	Line drawing specific resource script for menu items

Code

Program Source Code 3.1
Line.rc

```
/* Line Drawing */
```

```
#include "DrawLine.h"
MyMenu MENU DISCARDABLE
BEGIN
  POPUP "&Line"
  BEGIN
    MENUITEM "&Scan Line", ID_SCLINE
    MENUITEM "&Bresenham's Line", ID_BLINE
  END
  MENUITEM "&Clear", ID_CLEAR
  MENUITEM "&Exit", ID_EXIT
END
```

Line.h

```
void calculateSlope(POINT start, POINT end,
                    bool& bVerticalLine, float& m);
// For Bresenham's line drawing
void drawLineSegment(HDC hdc, POINT start, POINT end, COLORREF clr);
void drawNextLineSegment(POINT end, COLORREF clrLine);
void setupLineSegmentDrawing(HDC hdc, POINT& start, POINT& end);
bool findNextPtInLineSegment(POINT end, POINT& nextPt);
// For scan line drawing
void drawScanLineSegment(HDC hdc, POINT start,POINT end,COLORREF clr);
void drawNextScanLineSegment(POINT end, COLORREF clr);
void setupScanLineSegmentDrawing(HDC hdc, POINT& start, POINT& end);
bool findNextPtInScanLineSegment(POINT end, POINT& nextPt);
```

DrawLine.h

```
#define ID_SCLINE 40001
#define ID_BLINE  40004
#define ID_CLEAR  40002
#define ID_EXIT   40003

typedef enum
{
  READY_MODE,
  SCLINE_MODE,
  BLINE_MODE,
  DRAWN_MODE
} MODE;

const int nMaxNoOfPts = 2;

typedef struct
{
  HDC hdcMem;
  HBITMAP hbmp;
  HPEN hDrawPen;
  MODE drawMode;
  POINT pts[nMaxNoOfPts];/* points of the line */
  SIZE maxBoundary;
  int nPts; /* end points of line */
} DRAWING_DATA;
```

Line.cpp

```
#include <windows.h>

typedef struct
{
  HDC hdc;
  bool bVerticalLine;
  float m;
  long d, ine, inne;
  int linetype;
  float x, y;
} LINE_SEGMENT_DATA;

typedef struct
{
  HDC hdc;
  bool bVerticalLine;
  float m;
  int linetype;
  float x, y;
} SCANLINE_SEGMENT_DATA;

// global data for scan line drawing as contiguous segments
SCANLINE_SEGMENT_DATA gscl;

// global data for Bresenham's line drawing as contiguous segments
LINE_SEGMENT_DATA gbrl;

void calculateSlope(POINT start, POINT end,
                    bool& bVerticalLine, float& m)
{
  if (end.x == start.x)
    bVerticalLine = true;
  else
  {
    m = ((float)(end.y - start.y))/((float)(end.x - start.x));
    bVerticalLine = false;
  }
}

void rearrangeOnX(long &sx, long &sy, long &ex, long &ey)
{
  long tempx, tempy;
  if(sx > ex)
  {
    tempx=ex;tempy=ey;
    ex=sx;ey=sy;
    sx=tempx;sy=tempy;
  }
}

void rearrangeOnY(long &sx, long &sy, long &ex, long &ey)
{
  long tempx, tempy;
  if(sy > ey)
  {
    tempx=ex;tempy=ey;
    ex=sx;ey=sy;
    sx=tempx;sy=tempy;
  }
}

// for Bresenham's line drawing
void setupLineSegmentDrawing(HDC hdc, POINT& start, POINT& end)
{
  // setup global line segment data for entire line stretch
  // start and end points of line get rearranged after the call

  long dx;
  long dy;

  gbrl.hdc = hdc;
  calculateSlope(start, end, gbrl.bVerticalLine, gbrl.m);

  if(gbrl.bVerticalLine)
  {
    rearrangeOnY(start.x,start.y,end.x,end.y);
    gbrl.linetype = 1;
  }
  else
  {
    // non-vertical line
    if((gbrl.m >= 0) && (gbrl.m <= 1))
    {
      rearrangeOnX(start.x,start.y,end.x,end.y);
      gbrl.linetype = 2;
    }
    else if((gbrl.m >= -1) && (gbrl.m < 0))
    {
      rearrangeOnX(start.x,start.y,end.x,end.y);
      gbrl.linetype = 3;
    }
    else if(gbrl.m > 1)
    {
      rearrangeOnY(start.x,start.y,end.x,end.y);
      gbrl.linetype = 4;
    }
    else
    {
      // gbrl.m < -1
      rearrangeOnY(start.x,start.y,end.x,end.y);
      gbrl.linetype = 5;
    }
  }

  dx = end.x - start.x;
  dy = end.y - start.y;
  switch(gbrl.linetype)
  {
    case 2:
    case 3:
      if (dy < 0)
        dy = -dy;
      gbrl.d = 2*dy-dx;
      gbrl.ine = 2*dy;
      gbrl.inne = 2*(dy-dx);
      break;

    case 4:
    case 5:
      if (dx < 0)
        dx = -dx;
      gbrl.d = 2*dx-dy;
      gbrl.ine = 2*dx;
      gbrl.inne = 2*(dx-dy);
      break;

    default:
      break;
  }
  // set current position
  gbrl.x = (float) start.x;
  gbrl.y = (float) start.y;
}

bool findNextPtInLineSegment(POINT end, POINT& nextPt)
{
  // find next pt in the line segment from current pt
  // upto and including end pt
  // return true if next point exists, false otherwise

  float ex, ey;

  ex = (float) end.x;
  ey = (float) end.y;

  nextPt.x = (int) gbrl.x;
  nextPt.y = (int) gbrl.y;

  switch(gbrl.linetype)
  {
    case 1:
      if (gbrl.y <= ey)
      {
        gbrl.y++;
        return true;
      }
      break;

    case 2:
    case 3:
```