Bresenham's Integer Only Line Drawing Algorithm

by

John Kennedy
Mathematics Department
Santa Monica College
1900 Pico Blvd.
Santa Monica, CA 90405
`rkennedy@ix.netcom.com`

Using Integer Only Arithmetic To Draw A Line

Assume $y = mx + b$ represents the real variable equation of a line which is to be plotted using a grid of pixels where the two points (x_1, y_1) and (x_2, y_2) have integer coordinates and represent two known points on the line which are to be connected by drawing a simulated line segment (or segments) which connects them. In figure 1 below these two points are respectively the lower-left and upper-right corners of the pixel grid. For example, $(0, 0)$ and $(35, 9)$.

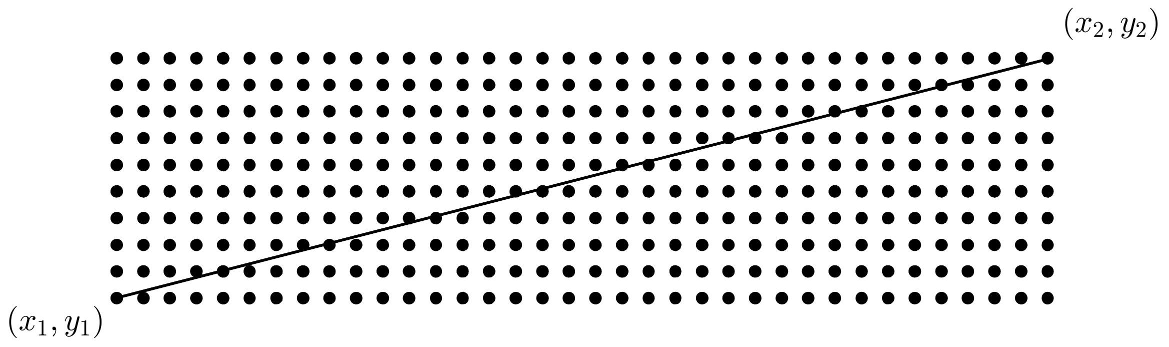

Figure 1. A true line laid out across a grid of pixels

We assume the x-pixel coordinates increase going to the right, and that the y-pixel coordinates increase going up. Thus for this example, we assume $x_1 < x_2$ and we assume $y_1 < y_2$. These restrictions are needed to describe the general setup of this explanatory example, but in no way do they impose a limitation that needs to be carried over into any actual implementation of the algorithm.

A further assumption is that the slope m is such that $0 \le m \le 1$. This assumption insures that as we move point by point from left-to-right across the graph, y will never jump up by more than one pixel and that y is increasing as x is increasing. This assumption is also necessary only to explain the idea behind the general algorithm. We will use the fraction $\dfrac{\Delta y}{\Delta x}$ to denote m.

To make an accurate plot, we keep track of the accumulated error in the y-variable. Since the first point we plot, namely (x_1, y_1), is exact, the variable $AccumulatedYError$ should be initialized with the value 0. As we move along the line from left-to-right, we will decide to increment y by 1 only when the accumulated y-error becomes greater than $\frac{1}{2}$. In this fashion we will only plot points whose y-coordinate differs from the true y-coordinate by at most 1 pixel. The accumulated y-error is a signed real value that tracks the true error between the currently plotted y and the true y, for the currently plotted x.

We also rather subtley assume that all x-coordinates are in fact exact. Thus we always increase x by 1 when we try to calculate the next point on the line by moving from left-to-right. So the only error we track is that associated with the y-variable. In fact, the accumulated y-error tells us how to correct the current y-value to get to the true y-value that we should have.

$$CurrentY + AccumulatedYError = TrueY$$

or

$$AccumulatedYError = TrueY - CurrentY$$

Given the equation $y = mx + b$, note that if x is increased by 1, then y should be increased by m. If y is not increased at all when x is increased by 1, then the accumulated error in y should be increased by m. By the same token, if y is just increased by 1, without changing x at all, then the accumulated error in y should be decreased by 1.

{ our first attempt at an algorithm appears below }

```
procedure Bresenham( $x_1$, $y_1$, $x_2$, $y_2$ : integer );
var  AccumulatedYError  : real;
     $\Delta$y, $\Delta$x           : real;
     CurrentX, CurrentY : integer;
     FinalX             : integer;
begin
  AccumulatedYError := 0.0;
  CurrentX := $x_1$;
  CurrentY := $y_1$;
  FinalX := $x_2$;
  $\Delta$x := (real)  $x_2 - x_1$;
  $\Delta$y := (real)  $y_2 - y_1$;
  repeat
    Plot(CurrentX, CurrentY);
    CurrentX := CurrentX + 1;
    AccumulatedYError := AccumulatedYError + $\frac{\Delta y}{\Delta x}$;
    if AccumulatedYError > $\frac{1}{2}$ then
      begin
        CurrentY := CurrentY + 1;
        AccumulatedYError := AccumulatedYError $-$ 1.0
          { adding 1 to y decreases the accumulated error by 1.0 }
      end
  until  CurrentX > FinalX
end;  {procedure Bresenham}
```

The above algorithm assumes the variables $AccumulatedYError$, and Δy and Δx are all floating point real variables. In order to make the algorithm run at maximum speed, we avoid all decimals and fractions and make clever use of integer variables and values only. Note the following re-arrangement of the related equations and variables.

Writing the equation: $AccumulatedYError = AccumulatedYError + \dfrac{\Delta y}{\Delta x}$

is equivalent to writing: $\Delta x \cdot AccumulatedYError = \Delta x \cdot AccumulatedYError + \Delta y$

in which the fraction is eliminated.

Also, the conditional test: if $AccumulatedYError > \dfrac{1}{2}$

can be re-written as: if $2 \cdot AccumulatedYError \ > \ 1$

which avoids a fraction and/or a floating point or real-valued variable.

Next, we avoid the multiplication by 2 and by Δx each time through the loop by re-writing the above equation and conditional test as:

$$(\, 2 \cdot \Delta x \cdot AccumulatedYError\,) \ = \ (\, 2 \cdot \Delta x \cdot AccumulatedYError\,) \ + \ (\, 2 \cdot \Delta y\,)$$

$$\text{if } \ (\, 2 \cdot \Delta x \cdot AccumulatedYError\,) \ > \ \Delta x$$

Also note that the slope m can be written as: $m \ = \ \dfrac{\Delta y}{\Delta x} \ = \ \dfrac{y_2 - y_1}{x_2 - x_1}$.

Thus we may assume both Δy and Δx are integers where $\Delta y = y_2 - y_1$ and $\Delta x = x_2 - x_1$. We make a new variable called $TwoDxAccumulatedError$ which can also be an integer. $TwoDxAccumulatedError \ = \ 2 \cdot \Delta x \cdot AccumulatedYError$.

It is also easy to verify that in our original description of the algorithm that at the beginning or end of the repeat-until loop that

$$-\frac{1}{2} \le AccumulatedYError \le \frac{1}{2}$$

which is further evidence that our plotted point's y-coordinates are always plotted to the nearest pixel.

Our second attempt at an algorithm appears on the next page.

{ Our second attempt at an algorithm is both smaller and faster! }

```
procedure Bresenham( x1, y1, x2, y2 : integer );
var  TwoDxAccumulatedError : integer;
     DeltaX, TwoDeltaX     : integer;
     CurrentX, CurrentY    : integer;
     FinalX                : integer;
begin
  TwoDxAccumulatedError := 0;
  CurrentX := x1;
  CurrentY := y1;
  FinalX := x2;
  DeltaX := x2 − x1;
  TwoDeltaX := DeltaX + DeltaX;
  repeat
    Plot(CurrentX, CurrentY);
    inc(CurrentX);
    inc(TwoDxAccumulatedError, Y);
    if TwoDxAccumulatedError > DeltaX then
      begin
        inc(CurrentY);
        dec(TwoDxAccumulatedError, TwoDeltaX);
          {adding 1 to y decreases the accumulated error by TwoDeltaX}
      end
  until  CurrentX > FinalX
end;  {procedure Bresenham}
```

Comments About Actual Implementations.

When actually implementing the algorithm, you will want to allow for the slope to be greater than 1, or even negative, and you will not want to assume $x_1 < x_2$. The above description handles only special cases. As many as 8 cases may need to be considered if you divide each quadrant into two $45°$ sections each. When the slope is greater than 1, y can be incremented and the error on x can be tracked. This is equivalent to working with the inverse function which will have a slope less than 1.

Also note that if $\Delta x = 0 = \Delta y$ then only one point needs to be plotted. You must also avoid an infinite repeat-until loop in the cases where the incremented Δ-value is 0. When these conditions are tested for and avoided, then the plotting within the repeat-until loop takes place at the end of the loop, as compared to the beginning of the loop as in the above sample algorithm. The code below is an actual implementation in Turbo Pascal version 7.0.

```
{  TEST BRESENHAM'S INTEGER LINE ALGORITHM  }

program TESTLINE(Input,Output);

uses dos, crt, bgidriv, bgifont, graph;


var  X1,Y1,X2,Y2 : integer;
     grMode      : integer;
     grDriver    : integer;
     ErrorCode   : integer;


procedure PlotLine(X1,Y1,X2,Y2 : integer);
var  CurrentX, CurrentY    : integer;
     Xinc, Yinc            : integer;
     Dx, Dy                : integer;
     TwoDx, TwoDy          : integer;
     TwoDxAccumulatedError : integer;
     TwoDyAccumulatedError : integer;
begin
  Dx := (X2-X1);
  Dy := (Y2-Y1);
  TwoDx := Dx + Dx;
  TwoDy := Dy + Dy;
  CurrentX := X1;     { start at (X1,Y1) and move towards (X2,Y2) }
  CurrentY := Y1;
  Xinc := 1;      { X and/or Y are incremented/decremented by 1 only }
  Yinc := 1;
```

{ the program listing continues on the next page }

```
  if Dx<0 then
    begin
      Xinc := -1;     { decrement X's }
      Dx := -Dx;
      TwoDx := -TwoDx
    end;
  if Dy<0 then         { insure Dy >= 0 }
    begin
      Yinc := -1;
      Dy := -Dy;
      TwoDy := -TwoDy
    end;
  putpixel(X1,Y1,1);           { the first point is a special case }
  if (Dx<>0) or (Dy<>0) then      { are other points on the line ? }
    begin
      if Dy<=Dx then                         { is the slope <= 1 ? }
        begin
          TwoDxAccumulatedError := 0;       { initialize the error }
          repeat
            inc(CurrentX, Xinc);      { consider X's from X1 to X2 }
            inc(TwoDxAccumulatedError, TwoDy);
            if TwoDxAccumulatedError>Dx then
              begin
                inc(CurrentY, Yinc);
                dec(TwoDxAccumulatedError, TwoDx)
              end;
            putpixel(CurrentX,CurrentY,1)     { process next point }
          until CurrentX=X2
        end
        else     { then the slope is large, reverse roles of X & Y }
          begin
            TwoDyAccumulatedError := 0;     { initialize the error }
            repeat
              inc(CurrentY, Yinc);    { consider Y's from Y1 to Y2 }
              inc(TwoDyAccumulatedError, TwoDx);
              if TwoDyAccumulatedError>Dy then
                begin
                  inc(CurrentX, Xinc);
                  dec(TwoDyAccumulatedError, TwoDy)
                end;
              putpixel(CurrentX,CurrentY,1)    { process next point }
            until CurrentY=Y2
          end
    end
end;  {procedure PlotLine}
```

{ the program listing continues on the next page }

```
{ start of program TESTLINE }
begin
  if RegisterBGIDriver(@EGAVGADriverProc)<0 then Exit;
  grDriver := Detect;
  InitGraph(grDriver,grMode,'');
  ErrorCode := GraphResult;
  if ErrorCode<>grOk then
    begin
      writeln('Error = ',ErrorCode);
      delay(3000);
      Exit
    end;
  repeat
    RestoreCRTMode;
    TextMode(CO80);       { go into text mode if not already there }
    writeln('To quit enter X1=100, Y1=100.');
    writeln;
    writeln;
    write('X1 = ? ');
    readln(X1);
    write('Y1 = ? ');
    readln(Y1);
    writeln;
    write('X2 = ? ');
    readln(X2);
    write('Y2 = ? ');
    readln(Y2);
    SetGraphMode(grMode);
    PlotLine(X1,Y1,X2,Y2);  { use the algorithm to draw a test line }
    delay(3000);
    delay(150);
    line(X1,Y1,X2,Y2);   { compare the test line with Pascal's draw }
    delay(2000);
    if (x1=100) and (y1=100) then CloseGraph
  until (X1=100) and (Y1=100);   { quit when 1st point = (100,100) }
  TextMode(CO80);                { quit in text mode }
  clrscr
end.  {program TESTLINE.PAS}
```

{ this is the end of the program listing }

REFERENCES:

1. Jack Bresenham, *Algorithm for Computer Control of a Digital Plotter*, **IBM Systems Journal**, Volume 4, Number 1, 1965, pp. 25-30.

2. Jack Bresenham, *A Linear Algorithm for Incremental Display of Circular Arcs*, **Communications of the ACM**, Volume 20, Number 2, February 1977, pp. 100-106.

3. Jerry R. Van Aken, *An Efficient Ellipse-Drawing Algorithm,* **I.E.E.E. Computer Graphics & Applications**, September 1984, pp.24-35.

4. Michael Abrash, *The Good, the Bad, and the Run-sliced*, **Dr. Dobbs Journal**, Number 194, November 1992, pp.171-176.