Bresenham's line algorithm

From Wikipedia, the free encyclopedia

The **Bresenham line algorithm** is an algorithm that determines which points in an *n*-dimensional raster should be plotted in order to form a close approximation to a straight line between two given points. It is commonly used to draw lines 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 one of the earliest algorithms developed in the field of computer graphics. A minor extension to the original algorithm also deals with drawing circles.

While algorithms such as Wu's algorithm are also frequently used in modern computer graphics because they can support antialiasing, the speed and simplicity of Bresenham's line algorithm means that it is still important. The algorithm is used in hardware such as plotters and in the graphics chips of modern graphics cards. It can also be found in many software graphics libraries. Because the algorithm is very simple, it is often implemented in either the firmware or the graphics hardware of modern graphics cards.

The label "Bresenham" is used today for a whole family of algorithms extending or modifying Bresenham's original algorithm. See further references below.

Contents

History

The algorithm was developed by Jack Elton Bresenham in 1962 at IBM. In 2001 Bresenham wrote:[1]

> I was working in the computation lab at IBM's San Jose development lab. A Calcomp plotter had been attached to an IBM 1401 via the 1407 typewriter console. [The algorithm] was in production use by summer 1962, possibly a month or so earlier. Programs in those days were freely exchanged among corporations so Calcomp (Jim Newland and Calvin Hefte) had copies. When I returned to Stanford in Fall 1962, I put a copy in the Stanford comp center library. A description of the line drawing routine was accepted for presentation at the 1963 ACM national convention in Denver, Colorado. It was a year in which no proceedings were published, only the agenda of speakers and topics in an issue of Communications of the ACM. A person from the IBM Systems Journal asked me after I made my presentation if they could publish the paper. I happily agreed, and they printed it in 1965.

Bresenham's algorithm was later modified to produce circles, the resulting algorithm being sometimes known as either "Bresenham's circle algorithm" or midpoint circle algorithm.

The algorithm

The common conventions will be used:

- the top-left is (0,0) such that pixel coordinates increase in the right and down directions (e.g. that the pixel at (7,4) is directly above the pixel at (7,5)), and
- that the pixel centers have integer coordinates.

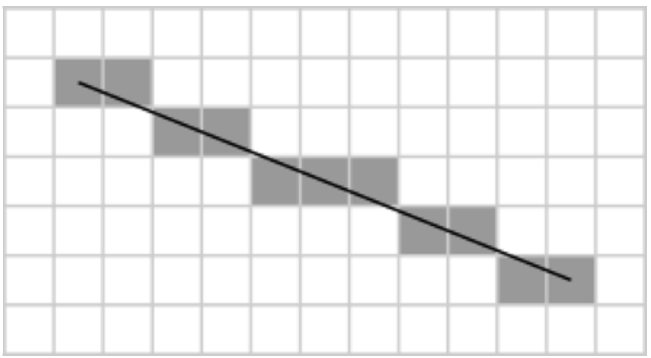


Illustration of the result of Bresenham's line algorithm. (0,0) is at the top left corner of the grid, (1,1) is at the top left end of the line and (11, 5) is at the bottom right end of the line.

The endpoints of the line are the pixels at (x_0, y_0) and (x_1, y_1), where the first coordinate of the pair is the column and the second is the row.

The algorithm will be initially presented only for the octant in which the segment goes down and to the right ($x_0 \leq x_1$ and $y_0 \leq y_1$), and its horizontal projection $x_1 - x_0$ is longer than the vertical projection $y_1 - y_0$ (the line has a negative slope whose absolute value is less than 1). In this octant, for each column x between x_0 and x_1, there is exactly one row y (computed by the algorithm) containing a pixel of the line, while each row between y_0 and y_1 may contain multiple rasterized pixels.

Bresenham's algorithm chooses the integer y corresponding to the pixel center that is closest to the ideal (fractional) y for the same x; on successive columns y can remain the same or increase by 1. The general equation of the line through the endpoints is given by:

$$\frac{y - y_0}{y_1 - y_0} = \frac{x - x_0}{x_1 - x_0}.$$

Since we know the column, x, the pixel's row, y, is given by rounding this quantity to the nearest integer:

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0.$$

The slope $(y_1 - y_0)/(x_1 - x_0)$ depends on the endpoint coordinates only and can be precomputed, and the ideal *y* for successive integer values of *x* can be computed starting from y_0 and repeatedly adding the slope.

In practice, the algorithm can track, instead of possibly large y values, a small *error value* between −0.5 and 0.5: the vertical distance between the rounded and the exact *y* values for the current *x*. Each time *x* is increased, the error is increased by the slope; if it exceeds 0.5, the rasterization *y* is increased by 1 (the line continues on the next lower row of the raster) and the error is decremented by 1.0.

In the following pseudocode sample `plot(x,y)` plots a point and `abs` returns absolute value:

```
function line(x0, x1, y0, y1)
    int deltax := x1 - x0
    int deltay := y1 - y0
    real error := 0
    real deltaerr := abs (deltay / deltax)    // Assume deltax != 0 (line is not vertical),
          // note that this division needs to be done in a way that preserves the fractional part
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if error ≥ 0.5 then
            y := y + 1
            error := error - 1.0
```

Optimization

The problem with this approach is that computers operate relatively slowly on fractional numbers like `error` and `deltaerr`; moreover, errors can accumulate over many floating-point additions. Working with integers will be both faster and more accurate. The trick we use is to multiply all the fractional numbers (including the constant 0.5) in the code above by `deltax`, which enables us to express them as integers. This results in a divide inside the main loop, however. To deal with this we modify how `error` is initialized and used so that rather than starting at zero and counting up towards 0.5, it starts at 0.5 and counts down to zero. The new program looks like this:

```
function line(x0, y0, x1, y1)
    boolean steep := abs(y1 - y0) > abs(x1 - x0)
    if steep then
        swap(x0, y0)
        swap(x1, y1)
    if x0 > x1 then
        swap(x0, x1)
        swap(y0, y1)
    int deltax := x1 - x0
    int deltay := abs(y1 - y0)
    int error := deltax / 2
    int ystep
    int y := y0
    if y0 < y1 then ystep := 1 else ystep := -1
    for x from x0 to x1
        if steep then plot(y,x) else plot(x,y)
        error := error - deltay
        if error < 0 then
            y := y + ystep
            error := error + deltax
```

Remark: If you need to control the points **in order of appearance** (for example to print several consecutive dashed lines) you will have to simplify this code by skipping the 2nd swap:

```
function line(x0, y0, x1, y1)
     boolean steep := abs(y1 - y0) > abs(x1 - x0)
     if steep then
         swap(x0, y0)
         swap(x1, y1)
     int deltax := abs(x1 - x0)
     int deltay := abs(y1 - y0)
     int error := deltax / 2
     int ystep
     int y := y0

     int inc REM added
     if x0 < x1 then inc := 1 else inc := -1 REM added

     if y0 < y1 then ystep := 1 else ystep := -1
     for x from x0 to x1 with increment inc REM changed
         if steep then plot(y,x) else plot(x,y)
         REM increment here a variable to control the progress of the line drawing
         error := error - deltay
         if error < 0 then
             y := y + ystep
             error := error + deltax
```

Simplification

It is further possible to eliminate the `swap`s in the initialisation by considering the error calculation for both directions simultaneously:

```
function line(x0, y0, x1, y1)
  dx := abs(x1-x0)
  dy := abs(y1-y0)
  if x0 < x1 then sx := 1 else sx := -1
  if y0 < y1 then sy := 1 else sy := -1
  err := dx-dy

  loop
    plot(x0,y0)
    if x0 = x1 and y0 = y1 exit loop
    e2 := 2*err
    if e2 > -dy then
      err := err - dy
      x0 := x0 + sx
    end if
    if e2 < dx then
      err := err + dx
      y0 := y0 + sy
    end if
  end loop
```

Derivation

To derive Bresenham's algorithm, two steps must be taken. The first step is transforming the equation of a line from the typical slope-intercept form into something different; and then using this new equation for a line to draw a line based on the idea of accumulation of error.

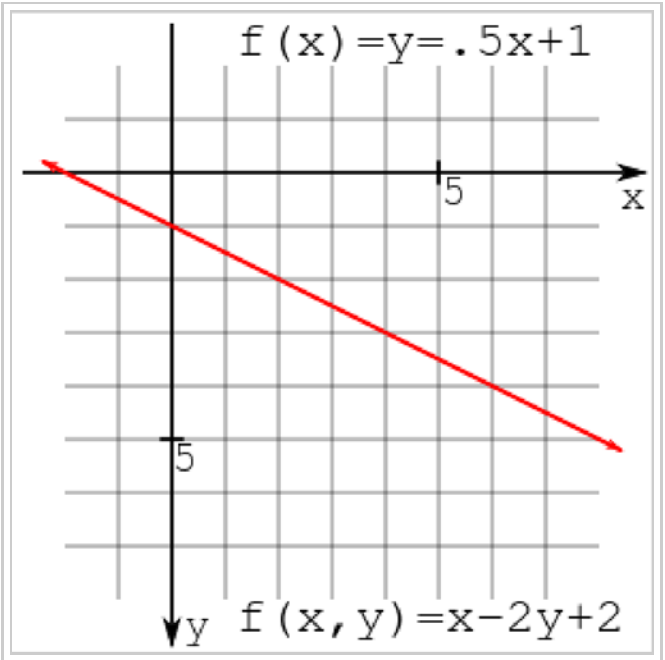
Line equation

The slope-intercept form of a line is written as

$$y = f(x) = mx + b$$

where m is the slope and b is the y-intercept. This is a function of only x and it would be useful to make this equation written as a function of both x and y. Using algebraic manipulation and recognition that the slope is the "rise over run" or $\Delta y/\Delta x$ then

$$\begin{aligned} y &= mx + b \\ y &= \frac{(\Delta y)}{(\Delta x)}x + b \\ (\Delta x)y &= (\Delta y)x + (\Delta x)b \\ 0 &= (\Delta y)x - (\Delta x)y + (\Delta x)b \end{aligned}$$



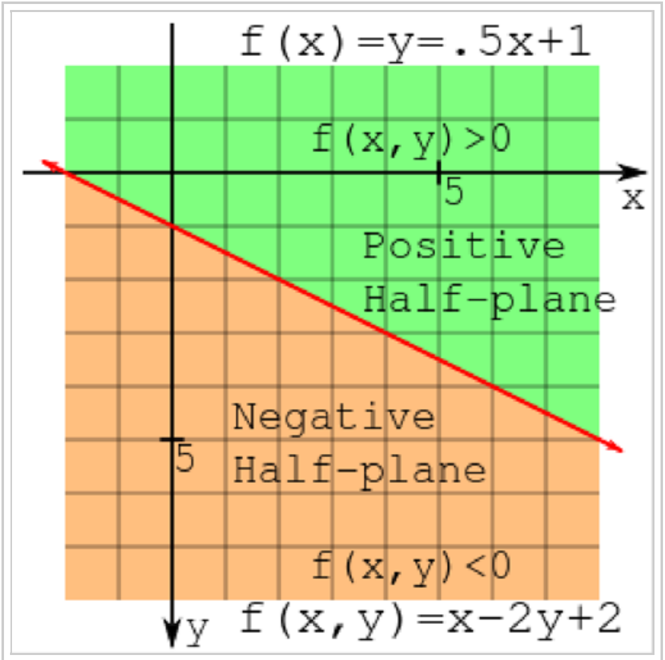
y=f(x)=.5x+1 or f(x,y)=x-2y+2

Letting this last equation be a function of x and y then it can be written as

$$f(x,y) = 0 = Ax + By + C$$

where the constants are

- $A = \Delta y$
- $B = -\Delta x$
- $C = (\Delta x)b$



Positive and negative half-planes

The line is then defined for some constants A, B, and C and anywhere $f(x,y) = 0$. For any (x,y) not on the line then $f(x,y) \neq 0$. It should be noted that everything about this form involves only integers if x and y are integers since the constants are necessarily integers.

As an example, the line $y = \frac{1}{2}x + 1$ then this could be written as $f(x,y) = x - 2y + 2$. The point (2,2) is on the line

$$f(2,2) = x - 2y + 2 = (2) - 2(2) + 2 = 2 - 4 + 2 = 0$$

and the point (2,3) is not on the line

$$f(2,3) = (2) - 2(3) + 2 = 2 - 6 + 2 = -2$$

and neither is the point (2,1)

$$f(2,1) = (2) - 2(1) + 2 = 2 - 2 + 2 = 2$$

Notice that the points (2,1) and (2,3) are on opposite sides of the line and f(x,y) evaluates to positive or negative. A line splits a plane into halves and the half-plane that has a negative f(x,y) can be called the negative half-plane, and the other half can called the positive half-plane. This observation is very important in the remainder of the derivation.

Algorithm

Clearly, the starting point is on the line

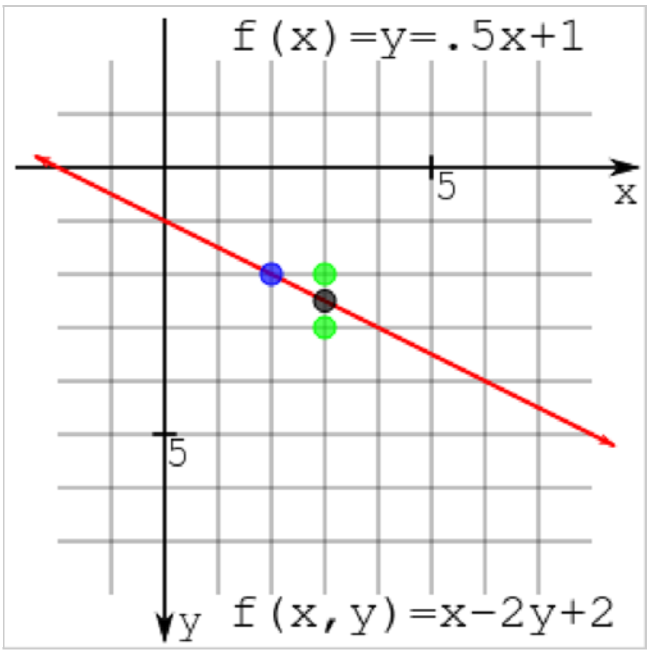
$$f(x_0, y_0) = 0$$

only because the line is defined to start and end on integer coordinates (though it is entirely reasonable to want to draw a line with non-integer end points).

Keeping in mind that the slope is less-than-or-equal-to zero, the problem now presents itself as to whether the next point should be at $(x_0 + 1, y_0)$ or $(x_0 + 1, y_0 + 1)$. Perhaps intuitively, the point should be chosen based upon which is closer to the line at $x_0 + 1$. If it is closer to the former then include the former point on the line, if the latter then the latter. To answer this, evaluate the line function at the midpoint between these two points:

$$f(x_0 + 1, y_0 + 1/2)$$

If the value of this is positive then the ideal line is below the midpoint and closer to the candidate point $(x_0 + 1, y_0 + 1)$; in effect the y coordinate has advanced. Otherwise, the ideal line passes through or above the midpoint, and the y coordinate has not advanced; in this case choose the point $(x_0 + 1, y_0)$. This observation is crucial to understand! The value of the line function at this midpoint is the sole determinant of which point should be chosen.



Candidate point (2,2) in blue and two candidate points in green (3,2) and (3,3)

The image to the right shows the blue point (2,2) chosen to be on the line with two candidate points in green (3,2) and (3,3). The black point (3, 2.5) is the midpoint between the two candidate points.

Algorithm with Integer Arithmetic

Alternatively, the difference between points can be used instead of evaluating f(x,y) at midpoints. This alternative method allows for integer-only arithmetic, which is generally considered faster than using floating-point arithmetic. To derive the alternative method, define the difference to be as follows:

$$D = f(x_0+1, y_0+1/2) - f(x_0, y_0)$$

For the first decision, this formulation is equivalent to the midpoint method since $f(x_0, y_0) = 0$ at the starting point. Simplifying this expression yields:

$$\begin{aligned} D &= \left[A(x_0+1) + B(y_0+\tfrac{1}{2}) + C\right] - [Ax_0 + By_0 + C] \\ &= \left[Ax_0 + By_0 + C + A + \tfrac{1}{2}B\right] - [Ax_0 + By_0 + C] \\ &= A + \tfrac{1}{2}B \end{aligned}$$

Just as with the midpoint method, if D is positive, then choose (x_0+1, y_0+1), otherwise choose (x_0+1, y_0).

The decision for the second point can be written as

$$f(x_0+2, y_0+1/2) - f(x_0+1, y_0+1/2) = A = \Delta y$$
$$f(x_0+2, y_0+3/2) - f(x_0+1, y_0+1/2) = A + B = \Delta y - \Delta x$$

If the difference is positive then (x_0+2, y_0+1) is chosen, otherwise (x_0+2, y_0). This decision can be generalized by accumulating the error.

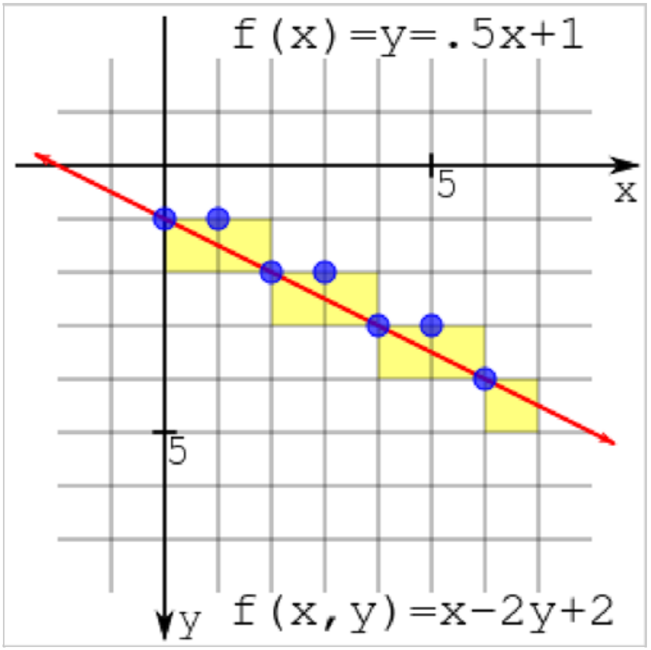
All of the derivation for the algorithm is done. One performance issue is the 1/2 factor in the initial value of D. Since all of this is about the sign of the accumulated difference, then everything can be multiplied by 2 with no consequence.

This results in an algorithm that uses only integer arithmetic.

```
plotLine(x0,y0, x1,y1)
  dx=x1-x0
  dy=y1-y0

  D = 2*dy - dx
  plot(x0,y0)
  y=y0

  for x from x0+1 to x1
    if D > 0
      y = y+1
      plot(x,y)
      D = D + (2*dy-2*dx)
    else
      plot(x,y)
      D = D + (2*dy)
```



Plotting the line from (0,1) to (6,4) showing a plot of grid lines and pixels

Running this algorithm for $f(x,y) = x - 2y + 2$ from (0,1) to (6,4) yields the following differences with dx=6 and dy=3:

- D=2*3-6=0
- **plot(0,1)**
- Loop from 1 to 6
  - x=1: D≤0: **plot(1,1)**, D=6
  - x=2: D>0: y=2, **plot(2,2)**, D=6+(6-12)=0
  - x=3: D≤0: **plot(3,2)**, D=6
  - x=4: D>0: y=3, **plot(4,3)**, D=6+(6-12)=0
  - x=5: D≤0: **plot(5,3)**, D=6
  - x=6: D>0: y=4, **plot(6,4)**, D=6+(6-12)=0

The result of this plot is shown to the right. The plotting can be viewed by plotting at the intersection of lines (blue circles) or filling in pixel boxes (yellow squares). Regardless, the plotting is the same.

All cases

However, as mentioned above this is only for the first octant. This means there are eight possible cases to consider. The simplest way to extend the same algorithm, if implemented in hardware, is to flip the co-ordinate system on the input and output of the single-octant drawer.

```
function switchToOctantZeroFrom(octant, x, y)
  switch(octant)
    case 0: return (x,y)
    case 1: return (y,x)
    case 2: return (-y, x)
    case 3: return (-x, y)
    case 4: return (-x, -y)
    case 5: return (-y, -x)
    case 6: return (y, -x)
    case 7: return (x, -y)
```

```
Octants:
 \2|1/
 3\|/0
---+---
 4/|\7
 /5|6\
```

Similar algorithms

The Bresenham algorithm can be interpreted as slightly modified digital differential analyzer (using 0.5 as error threshold instead of 0, which is required for non-overlapping polygon rasterizing).

The principle of using an incremental error in place of division operations has other applications in graphics. It is possible to use this technique to calculate the U,V co-ordinates during raster scan of texture mapped polygons. The voxel heightmap software-rendering engines seen in some PC games also used this principle.

Bresenham also published a Run-Slice (as opposed to the Run-Length) computational algorithm.

An extension to the algorithm that handles thick lines was created by Alan Murphy at IBM.

See also

- Digital differential analyzer (graphics algorithm), a simple and general method for rasterizing lines and triangles
- Xiaolin Wu's line algorithm, a similarly fast method of drawing lines with antialiasing
- Midpoint circle algorithm, a similar algorithm for drawing circles

Notes

1. ^ Paul E. Black. *Dictionary of Algorithms and Data Structures,* NIST. http://www.nist.gov/dads/HTML/bresenham.html

References

- Bresenham, J. E. (1 January 1965). "Algorithm for computer control of a digital plotter" (http://www.research.ibm.com/journal/sj/041/ibmsjIVRIC.pdf). *IBM Systems Journal* **4** (1): 25–30. doi:10.1147/sj.41.0025 (http://dx.doi.org/10.1147%2Fsj.41.0025).
- "The Bresenham Line-Drawing Algorithm" (http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html), by Colin Flanagan
- Abrash, Michael (1997). *Michael Abrash's graphics programming black book* (http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/graphics-programming-black-book-r1698). Albany, NY: Coriolis. pp. 654–678. ISBN 978-1-57610-174-2. A very optimized version of the algorithm in C and assembly for use in video games with complete details of its inner workings

Further reading

- Patrick-Gilles Maillot's Thesis (https://sites.google.com/site/patrickmaillot/english) an extension of the Bresenham line drawing algorithm to perform 3D hidden lines removal; also published in MICAD '87 proceedings on CAD/CAM and Computer Graphics, page 591 - ISBN 2-86601-084-1.
- Line Thickening by Modification To Bresenham's Algorithm (http://homepages.enterprise.net/murphy/thickline/index.html), A.S. Murphy, IBM Technical Disclosure Bulletin, Vol. 20, No. 12, May 1978.

External links

- Didactical Javascript implementation (http://gpolo.awardspace.info)
- *The Bresenham Line-Drawing Algorithm* by Colin Flanagan (http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html)
- National Institute of Standards and Technology page on Bresenham's algorithm (http://www.nist.gov/dads/HTML/bresenham.html)
- Calcomp 563 Incremental Plotter Information (http://www.pdp8online.com/563/563.shtml)
- 3D extension (http://www.cobrabytes.com/index.php?topic=1150.0)
- Bresenham 2D, 3D up to 6D (http://sites.google.com/site/proyectosroboticos/bresenham-3d)
- Bresenham Algorithm in several programming languages (http://rosettacode.org/wiki/Bitmap/Bresenham's_line_algorithm)
- The Beauty of Bresenham's Algorithm (http://members.chello.at/~easyfilter/bresenham.html) – A simple implementation to plot lines, circles, ellipses and Bézier curves
- Draw a Line using Bresenham Algorithm (http://www.etechplanet.com/codesnippets/computer-graphics-draw-a-line-using-bresenham-algorithm.aspx)
- Java implementation (https://github.com/fragkakis/bresenham)