WIKIPEDIA

Bresenham's line algorithm

Bresenham's line algorithm is a line drawing 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.

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 family of algorithms extending or modifying Bresenham's original algorithm.

Contents

History

Bresenham's line algorithm is named after Jack Elton Bresenham who developed it 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 has been extended to produce circles, ellipses, cubic and quadratic bezier curves, as well as native anti-aliased versions of those.[2]

Method

The following 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
- the pixel centers have integer coordinates.

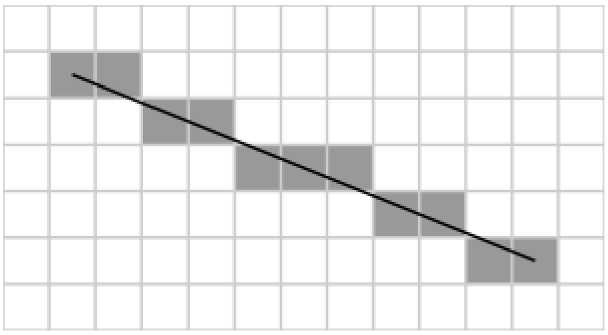


Illustration of the result of Bresenham's line algorithm. (0,0) is at the top left corner of the grid, (1,1) is at the top left end of the line and (11, 5) is at the bottom right end of the line.

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 positive slope 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 does not keep track of the y coordinate, which increases by *m* = Δ*y*/Δ*x* each time the *x* increases by one; it keeps an error bound at each stage, which represents the negative of the distance from (a) the point where the line exits the pixel to (b) the top edge of the pixel. This value is first set to $y_0 - 0.5$ (due to using the pixel's center coordinates), and is incremented by *m* each time the *x* coordinate is incremented by one. If the error becomes greater than *0.5*, we know that the line has moved upwards one pixel, and that we must increment our *y* coordinate and readjust the error to represent the distance from the top of the new pixel – which is done by subtracting one from error. [3]

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 to draw a line based on the idea of accumulation of error.

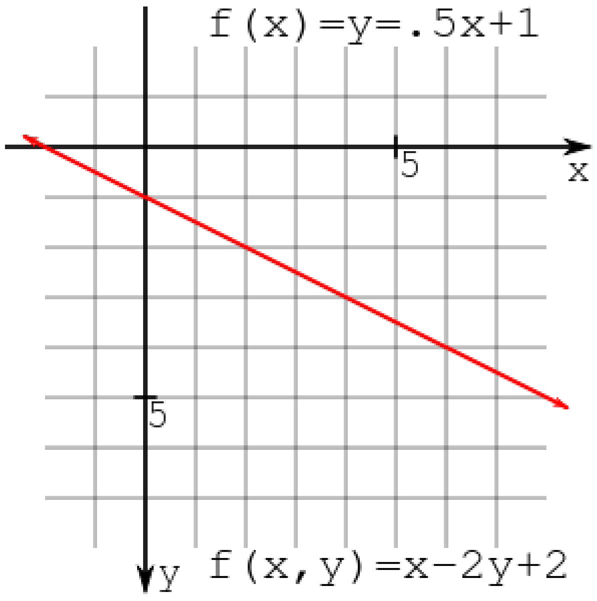
Line equation

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

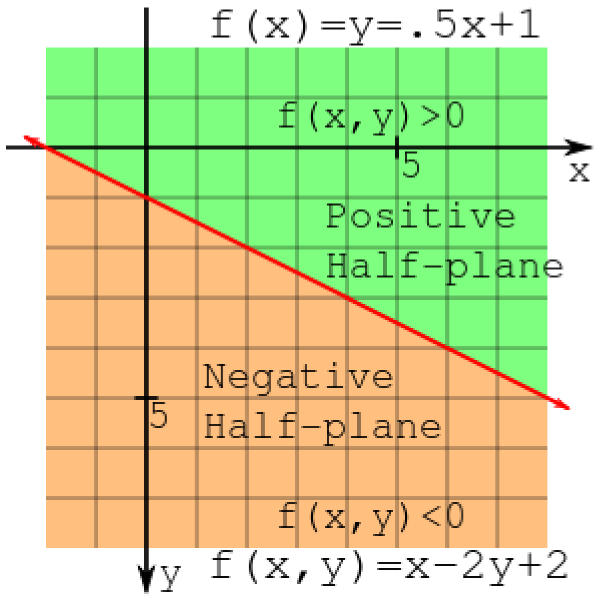
$$f(x,y) = 0 = Ax + By + C$$

where the constants are

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

The line is then defined for some constants A, B, and C anywhere $f(x,y) = 0$. For any (x,y) not on the line then $f(x,y) \neq 0$. 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



Positive and negative half-planes

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

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 be 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

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

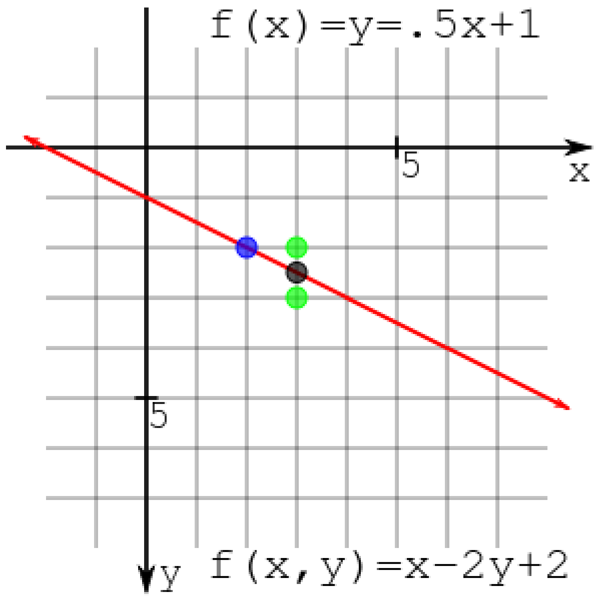
only because the line is defined to start and end on integer coordinates (though it is entirely reasonable to want to draw a line with non-integer end points).

Keeping in mind that the slope is less-than-or-equal-to one, 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:

$$f(x_0 + 1, y_0 + \tfrac{1}{2})$$

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.

The adjacent image 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.



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

Algorithm for 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 faster than using floating-point arithmetic. To derive the alternative method, define the difference to be as follows:

$$D = f(x_0+1, y_0+\tfrac{1}{2}) - f(x_0, y_0)$$

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:

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

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).

If (x_0+1, y_0) is chosen, the change in D will be:

$$\Delta D \;=\; f(x_0+2, y_0+\tfrac{1}{2}) - f(x_0+1, y_0+\tfrac{1}{2}) \;=\; A \;=\; \Delta y$$

If (x_0+1, y_0+1) is chosen the change in D will be:

$$\Delta D \;=\; f(x_0+2, y_0+\tfrac{3}{2}) - f(x_0+1, y_0+\tfrac{1}{2}) \;=\; A+B \;=\; \Delta y - \Delta x$$

If the new D 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 on each subsequent point.

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.

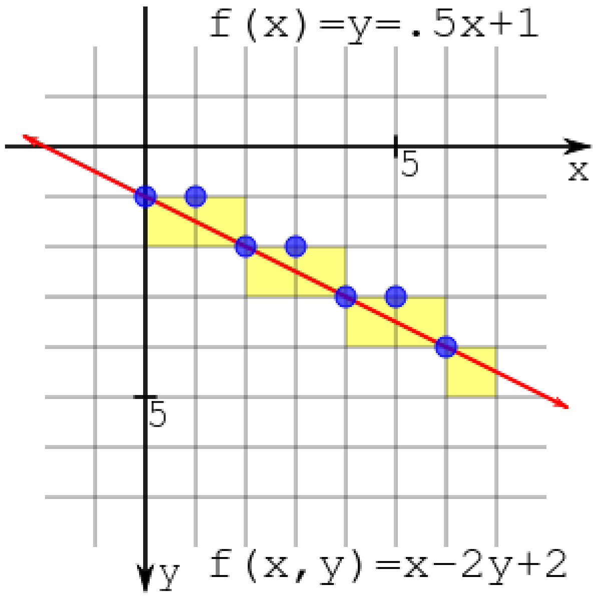
This results in an algorithm that uses only integer arithmetic.

```
plotLine(x0, y0, x1, y1)
    dx = x1 - x0
    dy = y1 - y0
    D = 2*dy - dx
    y = y0

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

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



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

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 octant zero, that is lines starting at the origin with a gradient between 0 and 1 where x increases by exactly 1 per iteration and y increases by 0 or 1.

The algorithm can be extended to cover gradients between 0 and -1 by checking whether y needs to increase or decrease (i.e. dy < 0)

```
plotLineLow(x0, y0, x1, y1)
    dx = x1 - x0
    dy = y1 - y0
    yi = 1
    if dy < 0
        yi = -1
        dy = -dy
    end if
    D = (2 * dy) - dx
    y = y0

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

By switching the x and y axis an implementation for positive or negative steep gradients can be written as

```
plotLineHigh(x0, y0, x1, y1)
    dx = x1 - x0
    dy = y1 - y0
    xi = 1
    if dx < 0
        xi = -1
        dx = -dx
    end if
    D = (2 * dx) - dy
    x = x0

    for y from y0 to y1
        plot(x, y)
        if D > 0
            x = x + xi
            D = D + (2 * (dx - dy))
        else
            D = D + 2*dx
        end if
```

A complete solution would need to detect whether x1 > x0 or y1 > y0 and reverse the input coordinates before drawing, thus

```
plotLine(x0, y0, x1, y1)
    if abs(y1 - y0) < abs(x1 - x0)
        if x0 > x1
            plotLineLow(x1, y1, x0, y0)
        else
            plotLineLow(x0, y0, x1, y1)
        end if
    else
        if y0 > y1
            plotLineHigh(x1, y1, x0, y0)
        else
            plotLineHigh(x0, y0, x1, y1)
        end if
    end if
```

In low level implementations which access the video memory directly, it would be typical for the special cases of vertical and horizontal lines to be handled separately as they can be highly optimized.

Some versions use Bresenham's principles of integer incremental error to perform all octant line draws, balancing the positive and negative error between the x and y coordinates.[2] Take note that the order is not necessarily guaranteed; in other words, the line may be drawn from (x0, y0) to (x1, y1) or from (x1, y1) to (x0, y0).

```
plotLine(int x0, int y0, int x1, int y1)
    dx =  abs(x1-x0);
    sx = x0<x1 ? 1 : -1;
    dy = -abs(y1-y0);
    sy = y0<y1 ? 1 : -1;
    err = dx+dy;  /* error value e_xy */
    while (true)   /* loop */
        plot(x0, y0);
        if (x0 == x1 && y0 == y1) break;
        e2 = 2*err;
        if (e2 >= dy) /* e_xy+e_x > 0 */
            err += dy;
            x0 += sx;
        end if
        if (e2 <= dx) /* e_xy+e_y < 0 */
            err += dx;
            y0 += sy;
        end if
    end while
```

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.[4] 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. This method has been represented in a number of US patents:

5,815,163		Method and apparatus to draw line slices during calculation
5,740,345		Method and apparatus for displaying computer graphics data stored in a compressed format with an efficient color indexing system
5,657,435		Run slice line draw engine with non-linear scaling capabilities
5,627,957		Run slice line draw engine with enhanced processing capabilities
5,627,956		Run slice line draw engine with stretching capabilities
5,617,524		Run slice line draw engine with shading capabilities
5,611,029		Run slice line draw engine with non-linear shading capabilities
5,604,852		Method and apparatus for displaying a parametric curve on a video display
5,600,769		Run slice line draw engine with enhanced clipping techniques

An extension to the algorithm that handles thick lines was created by Alan Murphy at IBM.[5]

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. https://xlinux.nist.gov/dads/HTML/bresenham.html
2. Zingl, Alois "A Rasterizing Algorithm for Drawing Curves" (2012) http://members.chello.at/~easyfilter/Bresenham.pdf
3. Joy, Kenneth. "Bresenham's Algorithm" (http://graphics.idav.ucdavis.edu/education/GraphicsNotes/Bresenhams-Algorithm.pdf) (PDF). Visualization and Graphics Research Group, Department of Computer Science, University of California, Davis. Retrieved 20 December 2016.
4. [1] (https://patents.google.com/patent/US5739818A/en), "Apparatus and method for performing perspectively correct interpolation in computer graphics", issued 1995-05-31
5. "Murphy's Modified Bresenham Line Algorithm" (http://homepages.enterprise.net/murphy/thickline/index.html). *homepages.enterprise.net*. Retrieved 2018-06-09.

References

- Bresenham, J. E. (1965). "Algorithm for computer control of a digital plotter" (https://web.archive.org/web/20080528040104/http://www.research.ibm.com/journal/sj/041/ibmsjIVRIC.pdf) (PDF). *IBM Systems Journal*. **4** (1): 25–30. doi:10.1147/sj.41.0025 (https://doi.org/10.1147%2Fsj.41.0025). Archived from the original (http://www.research.ibm.com/journal/sj/041/ibmsjIVRIC.pdf) (PDF) on May 28, 2008.
- "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* (https://archive.org/details/michaelabrashsgr00abra/page/654). Albany, NY: Coriolis. pp. 654–678 (https://archive.org/details/michaelabrashsgr00abra/page/654). 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
- Zingl, Alois (2012). "A Rasterizing Algorithm for Drawing Curves" (http://members.chello.at/~easyfilter/Bresenham.pdf) (PDF)., The Beauty of Bresenham's Algorithms

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.

rather than *[which]* for circle extension use: Technical Report 1964 Jan-27 -11- Circle Algorithm TR-02-286 IBM San Jose Lab or A Linear Algorithm for Incremental Digital Display of Circular Arcs February 1977 Communications of the ACM 20(2):100-106 DOI:10.1145/359423.359432

External links

- Michael Abrash's Graphics Programming Black Book Special Edition: Chapter 35: Bresenham Is Fast, and Fast Is Good (http://www.phatcode.net/res/224/files/html/ch35/35-01.html)
- *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 (https://xlinux.nist.gov/dads/HTML/bresenham.html)
- Calcomp 563 Incremental Plotter Information (http://www.pdp8online.com/563/563.shtml)
- 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