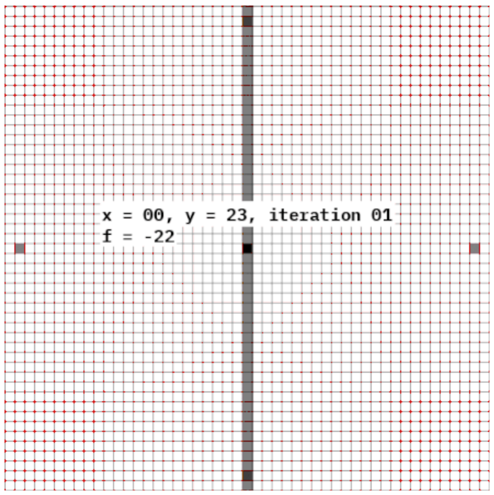
WIKIPEDIA

Midpoint circle algorithm

In computer graphics, the **midpoint circle algorithm** is an algorithm used to determine the points needed for rasterizing a circle. Bresenham's circle algorithm is derived from the midpoint circle algorithm. The algorithm can be generalized to conic sections.[1]

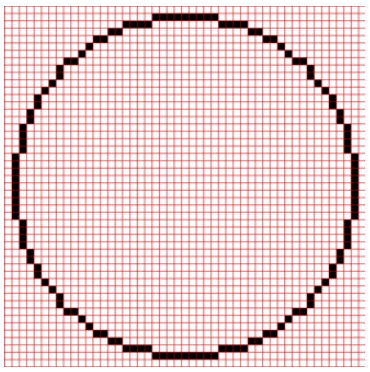
The algorithm is related to work by Pitteway[2] and Van Aken.[3]



Rasterizing a circle of radius 23 with the Bresenham midpoint circle algorithm. Only the green ☐ octant is actually calculated, it's simply mirrored eight times to form the other seven octants.

Contents

Summary

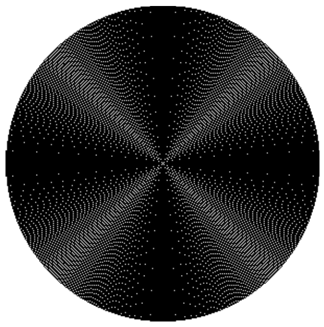


A circle of radius 23 drawn by the Bresenham algorithm

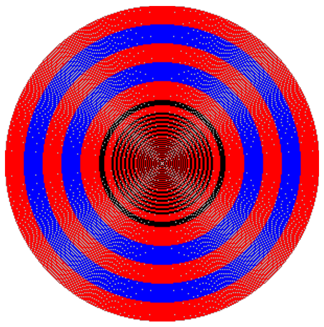
This algorithm draws all eight octants simultaneously, starting from each cardinal direction (0°, 90°, 180°, 270°) and extends both ways to reach the nearest multiple of 45° (45°, 135°, 225°, 315°). It can determine where to stop because when y = x, it has reached 45°. The reason for using these angles is shown in the above picture: As x increases, it does not skip nor repeat any x value until reaching 45°. So during the while loop, x increments by 1 each iteration, and y decrements by 1 on occasion, never exceeding 1 in one iteration. This changes at 45° because that is the point where the tangent is rise=run. Whereas rise>run before and rise<run after.

The second part of the problem, the determinant, is far trickier. This determines when to decrement y. It usually comes after drawing the pixels in each iteration, because it never goes below the radius on the first pixel. Because in a continuous function, the function for a sphere is the function for a circle with the radius dependent on z (or whatever the third variable is), it stands to reason that the algorithm for a discrete(voxel) sphere would also rely on this Midpoint circle algorithm. But when looking at a sphere, the integer radius of some adjacent circles is the same, but it is not expected to have the same exact circle adjacent to itself in the same hemisphere. Instead, a circle of the same radius needs a different determinant, to allow the curve to come in slightly closer to the center or extend out farther.

One hundred fifty concentric circles drawn with the midpoint circle algorithm.



On left, all circles are drawn black.



On right, red, black and blue are used together to demonstrate the concentricity of the circles.

Algorithm

The objective of the algorithm is to approximate the curve $x^2 + y^2 = r^2$ using pixels; in layman's terms every pixel should be approximately the same distance from the center. At each step, the path is extended by choosing the adjacent pixel which satisfies $x^2 + y^2 \leq r^2$ but maximizes $x^2 + y^2$. Since the candidate pixels are adjacent, the arithmetic to calculate the latter expression is simplified, requiring only bit shifts and additions. But a simplification can be done in order to understand the bitshift. Keep in mind that a left bitshift of a binary number is the same as multiplying with 2. Ergo, a left bitshift of the radius only produces the diameter which is defined as radius times two.

This algorithm starts with the circle equation. For simplicity, assume the center of the circle is at $(0,0)$. Consider first the first octant only, and draw a curve which starts at point $(r,0)$ and proceeds counterclockwise, reaching the angle of 45.

The *fast* direction here (the basis vector with the greater increase in value) is the y direction. The algorithm always takes a step in the positive y direction (upwards), and occasionally takes a step in the *slow* direction (the negative x direction).

From the circle equation is obtained the transformed equation $x^2 + y^2 - r^2 = 0$, where r^2 is computed only once during initialization.

Let the points on the circle be a sequence of coordinates of the vector to the point (in the usual basis). Points are numbered according to the order in which drawn, with $n = 1$ assigned to the first point $(r,0)$.

For each point, the following holds:

$$x_n^2 + y_n^2 = r^2$$

This can be rearranged thus:

$$x_n^2 = r^2 - y_n^2$$

And likewise for the next point:

$$x_{n+1}^2 = r^2 - y_{n+1}^2$$

Since for the first octant the next point will always be at least 1 pixel higher than the last (but also at most 1 pixel higher to maintain continuity), it is true that:

$$\begin{aligned} y_{n+1}^2 &= (y_n + 1)^2 \\ &= y_n^2 + 2y_n + 1 \end{aligned}$$

$$x_{n+1}^2 = r^2 - y_n^2 - 2y_n - 1$$

So, rework the next-point-equation into a recursive one by substituting $x_n^2 = r^2 - y_n^2$:

$$x_{n+1}^2 = x_n^2 - 2y_n - 1$$

Because of the continuity of a circle and because the maxima along both axes is the same, clearly it will not be skipping x points as it advances in the sequence. Usually it stays on the same x coordinate, and sometimes advances by one.

The resulting coordinate is then translated by adding midpoint coordinates. These frequent integer additions do not limit the performance much, as those square (root) computations can be spared in the inner loop in turn. Again, the zero in the transformed circle equation is replaced by the error term.

The initialization of the error term is derived from an offset of ½ pixel at the start. Until the intersection with the perpendicular line, this leads to an accumulated value of *r* in the error term, so that this value is used for initialization.

The frequent computations of squares in the circle equation, trigonometric expressions and square roots can again be avoided by dissolving everything into single steps and using recursive computation of the quadratic terms from the preceding iterations.

Variant with integer-based arithmetic

Just as with Bresenham's line algorithm, this algorithm can be optimized for integer-based math. Because of symmetry, if an algorithm can be found that only computes the pixels for one octant, the pixels can be reflected to get the whole circle.

We start by defining the radius error as the difference between the exact representation of the circle and the center point of each pixel (or any other arbitrary mathematical point on the pixel, so long as it's consistent across all pixels). For any pixel with a center at (x_i, y_i), the radius error is defined as:

$$RE(x_i, y_i) = \left|x_i^2 + y_i^2 - r^2\right|$$

For clarity, this formula for a circle is derived at the origin, but the algorithm can be modified for any location. It is useful to start with the point $(r, 0)$ on the positive X-axis. Because the radius will be a whole number of pixels, clearly the radius error will be zero:

$$RE(x_i, y_i) = \left|x_i^2 + 0^2 - r^2\right| = 0$$

Because it starts in the first counter-clockwise positive octant, it will step in the direction with the greatest *travel*, the Y direction, so it is clear that $y_{i+1} = y_i + 1$. Also, because it concerns this octant only, the X values have only 2 options: to stay the same as the prior iteration, or decrease by 1. A decision variable can be created that determines if the following is true:

$$RE(x_i - 1, y_i + 1) < RE(x_i, y_i + 1)$$

If this inequality holds, then plot $(x_i - 1, y_i + 1)$; if not, then plot $(x_i, y_i + 1)$. So, how to determine if this inequality holds? Start with a definition of radius error:

$$\begin{aligned} RE(x_i - 1, y_i + 1) &< RE(x_i, y_i + 1) \\ \left|(x_i - 1)^2 + (y_i + 1)^2 - r^2\right| &< \left|x_i^2 + (y_i + 1)^2 - r^2\right| \\ \left|(x_i^2 - 2x_i + 1) + (y_i^2 + 2y_i + 1) - r^2\right| &< \left|x_i^2 + (y_i^2 + 2y_i + 1) - r^2\right| \end{aligned}$$

The absolute value function does not help, so square both sides, since a square is always positive:

$$\begin{aligned} \left[(x_i^2 - 2x_i + 1) + (y_i^2 + 2y_i + 1) - r^2\right]^2 &< \left[x_i^2 + (y_i^2 + 2y_i + 1) - r^2\right]^2 \\ \left[(x_i^2 + y_i^2 - r^2 + 2y_i + 1) + (1 - 2x_i)\right]^2 &< \left[x_i^2 + y_i^2 - r^2 + 2y_i + 1\right]^2 \\ \left(x_i^2 + y_i^2 - r^2 + 2y_i + 1\right)^2 + 2(1 - 2x_i)(x_i^2 + y_i^2 - r^2 + 2y_i + 1) + (1 - 2x_i)^2 &< \left[x_i^2 + y_i^2 - r^2 + 2y_i + 1\right]^2 \\ 2(1 - 2x_i)(x_i^2 + y_i^2 - r^2 + 2y_i + 1) + (1 - 2x_i)^2 &< 0 \end{aligned}$$

Since x > 0, the term $(1 - 2x_i) < 0$, so dividing gets:

$$2\left[(x_i^2+y_i^2-r^2)+(2y_i+1)\right]+(1-2x_i)>0$$
$$2\left[RE(x_i,y_i)+Y_{\text{Change}}\right]+X_{\text{Change}}>0$$

Thus, the decision criterion changes from using floating-point operations to simple integer addition, subtraction, and bit shifting (for the multiply by 2 operations). If $2(RE+Y_{\text{Change}})+X_{\text{Change}}>0$, then decrement the X value. If $2(RE+Y_{\text{Change}})+X_{\text{Change}}\leq 0$, then keep the same X value. Again, by reflecting these points in all the octants, a full circle results.

We may reduce computation by only calculating the delta between the values of this decision formula from its value at the previous step. We start by assigning E as $3-2r$ which is the initial value of the formula at $(x_0,y_0)=(r,0)$, then as above at each step if $E>0$ we update it as $E=E+2(5-2x+2y)$ (and decrement X), otherwise $E=E+2(3+2y)$ thence increment Y as usual.

Drawing incomplete octants

The implementations above always draw only complete octants or circles. To draw only a certain arc from an angle α to an angle β, the algorithm needs first to calculate the x and y coordinates of these end points, where it is necessary to resort to trigonometric or square root computations (see Methods of computing square roots). Then the Bresenham algorithm is run over the complete octant or circle and sets the pixels only if they fall into the wanted interval. After finishing this arc, the algorithm can be ended prematurely.

If the angles are given as slopes, then no trigonometry or square roots are necessary: simply check that y/x is between the desired slopes.

Generalizations

It is also possible to use the same concept to rasterize a parabola, ellipse, or any other two-dimensional curve.[4]

References

1. Donald Hearn; M. Pauline Baker (1994). *Computer graphics* (https://archive.org/details/computergraphics0000hear_r7z2). Prentice-Hall. ISBN 978-0-13-161530-4.
2. Pitteway, M.L.V., "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter (https://academic.oup.com/comjnl/article-pdf/10/3/282/1333509/100282.pdf)", Computer J., 10(3) November 1967, pp 282-289
3. Van Aken, J.R., "An Efficient Ellipse Drawing Algorithm (https://ieeexplore.ieee.org/abstract/document/4055918/)", CG&A, 4(9), September 1984, pp 24-35
4. Zingl, Alois (December 2014). "The Beauty of Bresenham's Algorithm: A simple implementation to plot lines, circles, ellipses and Bézier curves" (http://members.chello.at/~easyfilter/bresenham.html). *easy.Filter*. Alois Zingl. Retrieved 16 February 2017.

External links

- Drawing circles (https://web.archive.org/web/20120422045142/https://banu.com/blog/7/drawing-circles/) - An article on drawing circles, that derives from a simple scheme to an efficient one
- Midpoint Circle Algorithm in several programming languages (http://rosettacode.org/wiki/Bitmap/Midpoint_circle_algorithm)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Midpoint_circle_algorithm&oldid=1073593456"

This page was last edited on 23 February 2022, at 14:05 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License 3.0; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.