Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
Français
ქართული
Edit links

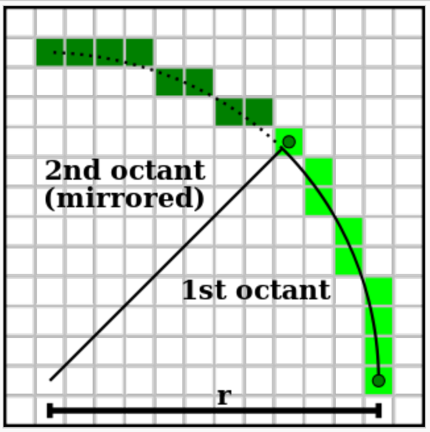
Midpoint circle algorithm

From Wikipedia, the free encyclopedia

In computer graphics, the **midpoint circle algorithm** is an algorithm used to determine the points needed for drawing 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]

Contents [hide]



Rasterisation of a circle by the Bresenham algorithm

Summary [edit]

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°). You can determine where to stop because when y = x, you have reached 45°. The reason for using these angles is shown in the above picture: As you increase y, you do not skip nor repeat any y value until you get to 45°. So during the while loop, y increments by 1 each iteration, and x 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 you decrement x. It usually comes after the drawing of the pixels in each iteration, because you never go 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 if you look at a sphere, you will find that the integer radius of some adjacent circles is the same, but you would not expect to have the same exact circle adjacent to itself in the same hemisphere. Instead, you need to have a circle of the same radius with a different determinant, to allow the curve to come in slightly closer to the center or extend out farther. The circle charts seen relating to Minecraft, like the determinant listed below, only account for one possibility.

Algorithm [edit]

This article **may be confusing or unclear to readers**. Please help us clarify the article; suggestions may be found on the talk page. *(February 2009)*

The objective of the algorithm is to find a path through the pixel grid using pixels which are as close as possible to solutions of $x^2 + y^2 = r^2$. At each step, the path is extended by choosing the adjacent pixel which satisfies $x^2 + y^2 <= r^2$ 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.

This algorithm starts with the circle equation. For simplicity, assume the center of the circle is at $(0, 0)$. We consider first only the first octant 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 we obtain the transformed equation $x^2 + y^2 - r^2 = 0$, where r^2 is computed only a single time 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 they are 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 as follows:

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

And likewise for the next point:

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

In general, it is true that:

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

So we refashion our 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, we know we will not be skipping x points as we advance in the sequence. Usually we will stay on the same x coordinate, and sometimes advance by one.

The resulting co-ordinate is then translated by adding midpoint coordinates. These frequent integer additions do not limit the performance much, as we can spare those square (root) computations 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 [edit]

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), we define the radius error to be:

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

For clarity, we derive this formula for a circle at the origin, but the algorithm can be modified for any location. We will want to start with the point $(r, 0)$ on the positive X-axis. Because the radius will be a whole number of pixels, we can see that the radius error will be zero:

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

Because we are starting in the first CCW positive octant, we will step in the direction with the greatest "travel", the Y direction, so we can say that $y_{i+1} = y_i + 1$. Also, because we are concerned with this octant only, we know that the X values only have 2 options: to stay the same as the previous iteration, or decrease by 1. We can create a decision variable that determines if the following is true:

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

If this inequality holds, we plot $(x_i - 1, y_i + 1)$; if not, then we plot $(x_i, y_i + 1)$. So how do we determine if this inequality holds? We can start with our definition of radius error:

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

The absolute value function doesn't really help us, so let's square both sides, since the square is always positive:

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

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

$$\begin{array}{lcl} 2\left[(x_i^2 + y_i^2 - r^2) + (2y_i + 1)\right] + (1 - 2x_i) & > & 0 \\ 2\left[RE(x_i, y_i) + YChange\right] + XChange & > & 0 \end{array}$$

Thus, we change our decision criterion from using floating-point operations to simple integer addition, subtraction, and bit shifting (for the multiply by 2 operations). If *2(RE+YChange)+XChange > 0*, then we decrement our X value. If *2(RE+YChange)+XChange <= 0*, then we keep the same X value. Again, by reflecting these points in all the octants, we get the full circle..

Example [edit]

The following C integer implementation follows the logic very closely:

```c
void DrawCircle(int x0, int y0, int radius)
{
  int x = radius;
  int y = 0;
  int decisionOver2 = 1 - x;   // Decision criterion divided by 2 evaluated at x=r, y=0

  while( y <= x )
  {
    DrawPixel( x + x0,  y + y0);
    DrawPixel( y + x0,  x + y0);
    DrawPixel(-x + x0,  y + y0);
    DrawPixel(-y + x0,  x + y0);
    DrawPixel(-x + x0, -y + y0);
    DrawPixel(-y + x0, -x + y0);
    DrawPixel( x + x0, -y + y0);
    DrawPixel( y + x0, -x + y0);
    y++;
    if (decisionOver2<=0)
    {
      decisionOver2 += 2 * y + 1;   // Change in decision criterion for y -> y+1
    }
    else
    {
      x--;
      decisionOver2 += 2 * (y - x) + 1;   // Change for y -> y+1, x -> x-1
    }
  }
}
```

Javascript [edit]

Implementation that draws a circle in html5 canvas. For educational purposes only, there are better ways to draw circles in canvas.

```
function drawCircle(x0, y0, radius, canvas){
   var x = radius;
   var y = 0;
   var decisionOver2 = 1 - x;   // Decision criterion divided by 2 evaluated at x=r, y=0
   var imageWidth = canvas.width;
   var imageHeight = canvas.height;
   var context = canvas.getContext('2d');
   var imageData = context.getImageData(0, 0, imageWidth, imageHeight);
   var pixelData = imageData.data;
   var makePixelIndexer = function(width){
   return function(i,j){
    var index = j*(width*4) + i*4;
    //index points to the R chanel of pixel
    //at column i and row j calculated from top left
    return index;
   };
   }
   var pixelIndexer = makePixelIndexer(imageWidth);
   var drawPixel = function(x,y){
     var idx = pixelIndexer(x,y);
     pixelData[idx] = 255; //red
     pixelData[idx+1] = 0; //green
     pixelData[idx+2] = 255;//blue
     pixelData[idx+3] = 255;//alpha
   };
   while(x >= y){
     drawPixel( x + x0,  y + y0);
     drawPixel( y + x0,  x + y0);
     drawPixel(-x + x0,  y + y0);
     drawPixel(-y + x0,  x + y0);
     drawPixel(-x + x0, -y + y0);
     drawPixel(-y + x0, -x + y0);
     drawPixel( x + x0, -y + y0);
     drawPixel( y + x0, -x + y0);
     y++;
     if (decisionOver2<=0){
       decisionOver2 += 2 * y + 1;   // Change in decision criterion for y -> y+1
     }else{
       x--;
       decisionOver2 += 2 * (y - x) + 1;   // Change for y -> y+1, x -> x-1
     }
   }
   context.putImageData(imageData,0,0);
 }
```

Drawing Incomplete Octants [edit]

The implementations above always only draw 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.

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

Generalizations [edit]

Ellipse [edit]

It is possible to generalize the algorithm to handle ellipses (of which circles are a special case). These algorithms involve calculating a full quadrant of the ellipse, as opposed to an octant as explained above, since non-circular ellipses lack the x-y symmetry of a circle.

One such algorithm is presented in the paper "A Fast Bresenham Type Algorithm For Drawing Ellipses" by John Kennedy. [1]

Parabola and other curves [edit]

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

References [edit]

1. ^ Donald Hearn; M. Pauline Baker. *Computer graphics*. Prentice-Hall. ISBN 978-0-13-161530-4.
2. ^ Pitteway, M.L.V., "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter", Computer J., 10(3) November 1967, pp 282-289

3. ^ Van Aken, J.R., "An Efficient Ellipse Drawing Algorithm", CG&A, 4(9), September 1984, pp 24-35
4. ^ http://members.chello.at/~easyfilter/bresenham.html

External links [edit]

- The Beauty of Bresenham's Algorithm – A simple implementation to plot lines, circles, ellipses and Bézier curves
- Drawing circles - An article on drawing circles, that derives from a simple scheme to an efficient one.
- Midpoint Circle Algorithm in several programming languages

Categories: Geometric algorithms | Digital geometry | Articles with example JavaScript code