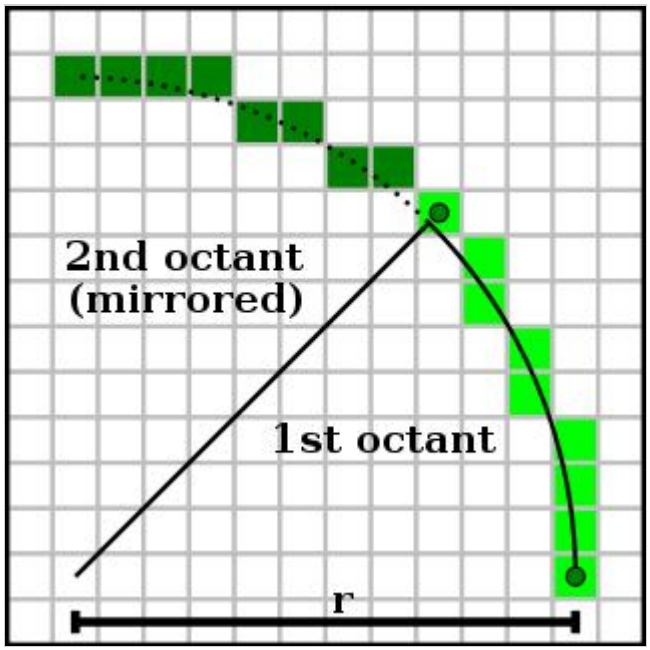
Midpoint circle algorithm

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. The algorithm is a variant of Bresenham's line algorithm, and is thus sometimes known as **Bresenham's circle algorithm**, although not actually invented by Bresenham. The algorithm can be generalized to conic sections.[1]

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



Rasterisation of a circle by the Bresenham algorithm

Contents

The algorithm

The algorithm starts accordingly with the circle equation $x^2 + y^2 = r^2$. So, the center of the circle is located at (0,0). We consider first only the first octant and draw a curve which starts at point $(r,0)$ and proceeds upwards and to the left, reaching the angle of 45°.

The "fast" direction here is the y direction. The algorithm always does a step in the positive y direction (upwards), and every now and then also has to do a step in the "slow" direction, the negative x direction.

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

(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). Let n denote which point to be looked at, starting at the one on the horizontal axis, on the first quadrant.

Then for each point also holds:

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

Likewise for the next point:

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

That is:

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

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 the circle and because is is a circle (i.e. maximum along both axes is the same) we know we will not be skipping x points as we advance in the sequence. Mostly we will either stay on the same x coordinate or advance by one.

Additionally we need to add the midpoint coordinates when setting a pixel. 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.

A possible implementation of the Bresenham Algorithm for a full circle in C. Here another variable for recursive computation of the quadratic terms is used, which corresponds with the term $2n + 1$ above. It just has to be increased by 2 from one step to the next:

```
void rasterCircle(int x0, int y0, int radius)
{
  int f = 1 - radius;
  int ddF_x = 1;
  int ddF_y = -2 * radius;
  int x = 0;
  int y = radius;

  setPixel(x0, y0 + radius);
  setPixel(x0, y0 - radius);
  setPixel(x0 + radius, y0);
  setPixel(x0 - radius, y0);
```

```
    while(x < y)
    {
      // ddF_x == 2 * x + 1;
      // ddF_y == -2 * y;
      // f == x*x + y*y - radius*radius + 2*x - y + 1;
      if(f >= 0)
      {
        y--;
        ddF_y += 2;
        f += ddF_y;
      }
      x++;
      ddF_x += 2;
      f += ddF_x;
      setPixel(x0 + x, y0 + y);
      setPixel(x0 - x, y0 + y);
      setPixel(x0 + x, y0 - y);
      setPixel(x0 - x, y0 - y);
      setPixel(x0 + y, y0 + x);
      setPixel(x0 - y, y0 + x);
      setPixel(x0 + y, y0 - x);
      setPixel(x0 - y, y0 - x);
    }
  }
```

Note: There is correlation between this algorithm and the sum of first N odd numbers, which this one basically does. That is, $1 + 3 + 5 + 7 + 9 + \cdots = \sum_{n=0}^{N} 2n + 1 = (N + 1)^2$.

```
So.
When we compare sum of N odd numbers to this algorithm we have.
ddF_y = -2 * radius       is connected to last member of sum of N odd numbers.
                          This member has index equal to value of radius (integral).
                          Since odd number is 2*n + 1 there is 1 handled elsewhere
                          or it should be -2*radius - 1
ddF_x = 0                 should be 1. Because difference between two consecutive odd numbers is 2.
                          If so f += ddF_y + 1 is f+= ddF_y. Saving one operation.
f = - radius + 1          Initial error equal to half of "bigger" step.
                          In case of saving one addition it should be either -radius or -radius + 2.
In any case there should be addition of 1 driven out of outer loop.
So.
f += ddF_y                Adding odd numbers from Nth to 1st.
f += ddF_x                Adding odd numbers from 1st to Nth. 1 is missing because it can be moved outside of l
```

Optimization

The following adaption of the circle algorithm is useful for machines where registers are scarce. Only three variables are needed in the main loop to perform the calculation. Although written in C, the code has been written to better reflect how it might be implemented in assembly code.

```
// 'cx' and 'cy' denote the offset of the circle centre from the origin.
void circle(int cx, int cy, int radius)
{
  int error = -radius;
  int x = radius;
  int y = 0;

  // The following while loop may altered to 'while (x > y)' for a
  // performance benefit, as long as a call to 'plot4points' follows
  // the body of the loop. This allows for the elimination of the
  // '(x != y') test in 'plot8points', providing a further benefit.
  //
  // For the sake of clarity, this is not shown here.
  while (x >= y)
  {
    plot8points(cx, cy, x, y);

    error += y;
    ++y;
    error += y;

    // The following test may be implemented in assembly language in
    // most machines by testing the carry flag after adding 'y' to
    // the value of 'error' in the previous step, since 'error'
    // nominally has a negative value.
    if (error >= 0)
    {
      error -= x;
      --x;
      error -= x;
    }
  }
}

void plot8points(int cx, int cy, int x, int y)
{
  plot4points(cx, cy, x, y);
  if (x != y) plot4points(cx, cy, y, x);
}

// The '(x != 0 && y != 0)' test in the last line of this function
// may be omitted for a performance benefit if the radius of the
// circle is known to be non-zero.
void plot4points(int cx, int cy, int x, int y)
{
  setPixel(cx + x, cy + y);
  if (x != 0) setPixel(cx - x, cy + y);
  if (y != 0) setPixel(cx + x, cy - y);
  if (x != 0 && y != 0) setPixel(cx - x, cy - y);
```

```
}
```

Drawing incomplete octants

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 required: one simply checks that y / x is between the desired slopes.

Ellipses

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] (http://homepage.smc.edu/kennedy_john/belipse.pdf)

References

1. ^ Donald Hearn; M. Pauline Baker. *Computer graphics* (http://books.google.com/books?id=WJiYQgAACAAJ) . Prentice-Hall. ISBN 9780131615304. http://books.google.com/books?id=WJiYQgAACAAJ.
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

External links

- The Beauty of Bresenham's Algorithm (http://free.pages.at/easyfilter/bresenham.html) – A simple implementation to plot lines, circles, ellipses and Bézier curves
- Drawing circles (https://banu.com/blog/7/drawing-circles/) - An article on drawing circles, that derives from a simple scheme to an efficient one