A Fast Bresenham Type Algorithm For Drawing Circles

by

John Kennedy
Mathematics Department
Santa Monica College
1900 Pico Blvd.
Santa Monica, CA 90405
rkennedy@ix.netcom.com

Fast Circle Drawing

There is a well-known algorithm for plotting straight lines on a display device or a plotter where the grid over which the line is drawn consists of discrete points or pixels. In working with a lattice of points it is useful to avoid floating point arithmetic. One of the first publications of such a straight-line algorithm was in 1965, by Jack Bresenham who worked for **I.B.M.**. The main idea in the algorithm is to analyze and manipulate the linear equation so that only integer arithmetic is used in all the calculations. Integer arithmetic has the advantages of speed and precision; working with floating point values requires more time and memory and such values would need to be rounded to integers anyway. In this paper we consider the more difficult problem of approximating the plot of a circle on a grid of discrete pixels, using only integer arithmetic.

Assume $x^2 + y^2 = r^2$ represents the real variable equation of a circle which is to be plotted using a grid of discrete pixels where each pixel has integer coordinates. There is no loss of generality here since once the points are determined they may be translated relative to any center that is not the origin (0, 0).

We will compare errors associated with the x and y coordinates of the points that we are plotting. Although we plot points of the form $P(x_i, y_i)$, these points usually do not exactly satisfy the circles' defining equation. Assuming x_i is accurate, the true y-coordinate of P is $\sqrt{r^2 - x_i^2}$. We plot $P(x_i, y_i)$, but $Q(x_i, \sqrt{r^2 - x_i^2})$ is the theoretically precise point that is on the circle. The quantity

$$\sqrt{x_i^2 + y_i^2} \; - \; \sqrt{r^2}$$

tells how far P is off the true circle. If this quantity is negative it means P lies inside the true circle, and if this quantity is positive it means P is outside the true circle. Of course when this is 0 (which may be rare but does happen) P is exactly on the circle.

The expression

$$| \, x_i^2 \; + \; y_i^2 \; - \; r^2 \, |$$

is a more practical measure of the error. The absolute value will be needed when it comes to comparing two such errors, and this new quantity avoids calculating square roots which are an expensive (time-wise) computer floating point operation. Thus we define a function which we call the $RadiusError$ which is an error measure for each plotted point.

$$RadiusError(x_i, y_i) \; = \; | \, x_i^2 \; + \; y_i^2 \; - \; r^2 \, |$$

Our plotting strategy is to start with $x_1 = r$ and $y_1 = 0$. In this case,

$$RadiusError(x_1, y_1) \; = \; | \, x_1^2 \; + \; y_1^2 \; - \; r^2 \, | \; = \; | \, r^2 \; + \; 0^2 \; - \; r^2 \, | \; = 0$$

So the first point we plot fits exactly on the true circle.

We plot our circle in the same order as would be given by the parametric equations $x = \cos(t)$ and $y = \sin(t)$ in which t starts at 0 and increases. This means the circle is drawn in a counterclockwise direction, starting on the positive horizontal x-axis. On the starting part of the graph the y-coordinates increase or change more rapidly than the x-coordinates. So at each iteration we decide to always increment y, but we need to test when we should decrement x.

$$y_{i+1} \;=\; y_i \;+\; 1, \text{ for } i = 1, 2, 3, \ldots$$

Depending on the result of our test for when to decrease x, we will have $x_{i+1} = x_i$ if the test fails, or $x_{i+1} = x_i - 1$ if the test succeeds.

Our decision as to when to decrease x is based on the comparison of the two values,

$$RadiusError(x_i - 1, y_i + 1) \text{ and } RadiusError(x_i, y_i + 1)$$

Then we choose either $x_i - 1$ or x_i as our new x-coordinate depending on which of the two $RadiusError$ values is the smallest.

In any event, we need to know how the function $RadiusError(x_i, y_i)$ changes for each possible change in its arguments, i.e, when x_i decreases and y_i increases. In other words, we do not need to calculate $RadiusError$ anew for each next point, if we just keep track of how the value changes as the arguments change.

If you just plotted the point $P(x_i, y_i)$, you would next be considering plotting either $P(x_i - 1, y_i + 1)$ or $P(x_i, y_i + 1)$ and you would base your decision of which point to plot after comparing $RadiusError(x_i - 1, y_i + 1)$ and $RadiusError(x_i, y_i + 1)$.

$$RadiusError(x_i - 1, y_i + 1) = |\,(x_i - 1)^2 + (y_i + 1)^2 - r^2\,|$$

$$= \;|\,x_i^2 - 2x_i + 1 + y_i^2 + 2y_i + 1 - r^2\,|$$

$$= |\,x_i^2 + y_i^2 - r^2 + (1 - 2x_i) + (2y_i + 1)\,|$$

The alternative is to plot $P(x_i, y_i + 1)$. The error at this point is given by:

$$RadiusError(x_i, y_i + 1) = |\,x_i^2 + (y_i + 1)^2 - r^2\,|$$

$$= |\,x_i^2 + y_i^2 + 2y_i + 1 - r^2\,|$$

$$= |\,x_i^2 + y_i^2 - r^2 + (2y_i + 1)\,|$$

Next we analyze and try to simplify the inequality:

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

This inequality holds if and only if

$$| x_i^2 + y_i^2 - r^2 + (1 - 2x_i) + (2y_i + 1) | < | x_i^2 + y_i^2 - r^2 + (2y_i + 1) |$$

if and only if

$$\sqrt{\{ [x_i^2 + y_i^2 - r^2 + (2y_i + 1)] + (1 - 2x_i) \}^2} < \sqrt{[x_i^2 + y_i^2 - r^2 + (2y_i + 1)]^2}$$

if and only if

$$\{ [x_i^2 + y_i^2 - r^2 + (2y_i + 1)] + (1 - 2x_i) \}^2 < [x_i^2 + y_i^2 - r^2 + (2y_i + 1)]^2$$

if and only if

$$[x_i^2 + y_i^2 - r^2 + (2y_i + 1)]^2 + 2 \cdot [x_i^2 + y_i^2 - r^2 + (2y_i + 1)] \cdot (1 - 2x_i) + (1 - 2x_i)^2 < [x_i^2 + y_i^2 - r^2 + (2y_i + 1)]^2$$

if and only if

$$2 \cdot [x_i^2 + y_i^2 - r^2 + (2y_i + 1)] \cdot (1 - 2x_i) + (1 - 2x_i)^2 < 0$$

if and only if

$$2 \cdot [x_i^2 + y_i^2 - r^2 + (2y_i + 1)] + (1 - 2x_i) > 0$$

In arriving at the last step we assumed the quantity $(1 - 2x_i)$ is negative so when we divide both sides of the inequality with it, the inequality is reversed.

If $2 \cdot [x_i^2 + y_i^2 - r^2 + (2y_i + 1)] + (1 - 2x_i) > 0$ then we should decrement x when we plot the next point $P(x_{i+1}, y_{i+1})$.

Having performed the above analysis we can begin to see the usefulness of defining three new quantities.

$$XChange \;=\; (1 - 2x_i)$$

$$YChange \;=\; (2y_i + 1)$$

$$RadiusError \;=\; x_i^2 + y_i^2 - r^2$$

These three quantities may also be calculated recursively (i.e., iteratively). Since when x_i and y_i change, they change by ± 1, the quantities $XChange$ and $YChange$ always change by exactly ± 2. These two quantities are always odd. $XChange$ starts counting with the negative number, $1 - 2r$, and increases toward 0. $YChange$ counts odd numbers, starting at 1 and increases as well. The initial $RadiusError$ value is 0 since $x_1 = r$ and $y_1 = 0$. During the plotting process $RadiusError$ may be positive, or negative, or 0. Thus the program's $RadiusError$ variable neither needs nor uses the absolute value that was shown earlier.

Now we can give the circle plotting algorithm. Below, CX, CY, and R refer to the circle's center point coordinates and radius value.

```
procedure PlotCircle(CX, CY, R : longint);
begin
   var  X, Y             : longint;
        XChange, YChange : longint;
        RadiusError      : longint;
   X := R;
   Y := 0;
   XChange := 1 - 2*R;
   YChange := 1;
   RadiusError := 0;
   while ( X ≥ Y ) do
      begin
         Plot8CirclePoints(X,Y);  {subroutine appears below}
         inc(Y);
         inc(RadiusError, YChange);
         inc(YChange,2);
         if ( 2*RadiusError + XChange > 0 ) then
            begin
               dec(X);
               inc(RadiusError, XChange);
               inc(XChange,2)
            end
      end
end;  {procedure PlotCircle}
```

Due to the circle's symmetry, we need only calculate points in the first 45° of the circle. Thus in the above code the main while loop terminates when Y first becomes larger than X. The subroutine called $Plot8CirclePoints$ takes advantage of the symmetry. We only calculate the points in the first 45°, but for each such point we actually plot 7 other related points at the same time as indicated in the figure below. The $Plot8CirclePoints$ subroutine would normally be defined inside the above $PlotCircle$ procedure. Also note that CX and CY refer to the circle's center point.

The 8 numbered regions in the figure below result after dividing each quadrant into two parts. These regions may be called octants, which like quadrants are numbered in a counterclockwise direction, with octant 1 just above the positive x-axis.

```
procedure Plot8CirclePoints(X,Y : longint);
begin
   PutPixel(CX+X, CY+Y);          {point in octant 1}
   PutPixel(CX-X, CY+Y);          {point in octant 4}
   PutPixel(CX-X, CY-Y);          {point in octant 5}
   PutPixel(CX+X, CY-Y);          {point in octant 8}
   PutPixel(CX+Y, CY+X);          {point in octant 2}
   PutPixel(CX-Y, CY+X);          {point in octant 3}
   PutPixel(CX-Y, CY-X);          {point in octant 6}
   PutPixel(CX+Y, CY-X)           {point in octant 7}
end;  {procedure Plot8CirclePoints}
```

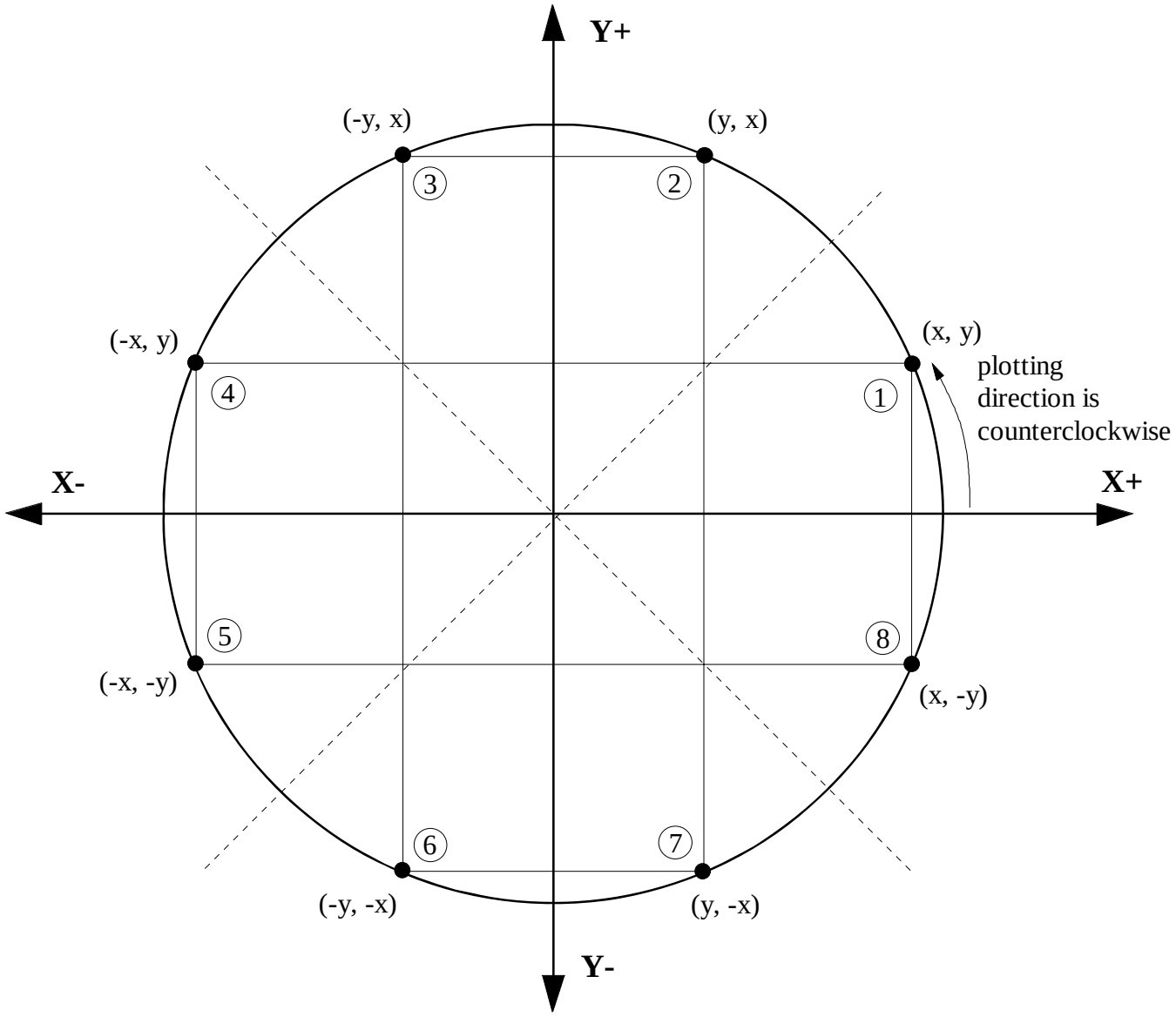

Figure 1. The 8 symmetric octants of a circle. Only octant #1 points are calculated.

REFERENCES:

1. Jack Bresenham, *Algorithm for Computer Control of a Digital Plotter*, **IBM Systems Journal**, Volume 4, Number 1, 1965, pp. 25-30.

2. Jack Bresenham, *A Linear Algorithm for Incremental Display of Circular Arcs*, **Communications of the ACM**, Volume 20, Number 2, February 1977, pp. 100-106.

3. Jerry R. Van Aken, *An Efficient Ellipse-Drawing Algorithm*, **I.E.E.E. Computer Graphics & Applications**, September 1984, pp.24-35.

4. Michael Abrash, *The Good, the Bad, and the Run-sliced*, **Dr. Dobbs Journal**, Number 194, November 1992, pp.171-176.