A Fast Bresenham Type Algorithm For Drawing Ellipses

by

John Kennedy
Mathematics Department
Santa Monica College
1900 Pico Blvd.
Santa Monica, CA 90405

rkennedy@ix.netcom.com

Except for this comment explaining that it is blank for
some deliberate reason, this page is intentionally blank!

Fast Ellipse 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 published algorithms was by Jack Bresenham who worked for **I.B.M.** (1965). 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 an ellipse on a grid of discrete pixels, using only integer arithmetic.

Before reading this paper it is suggested you read the paper by the same author on drawing circles. The key ideas in the previous paper give all the details for circles, and there is much similarity between drawing ellipses and circles. This paper will discuss the basic differences between circles and ellipses but it assumes complete familiarity with the circle algorithm.

Assume $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$ represents the real variable equation of an ellipse 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 to any elliptical 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 ellipses' defining equation. In order to avoid dividing we re-write the above equation in the form

$$b^2 \cdot x^2 + a^2 \cdot y^2 = b^2 \cdot a^2$$

For a given point $P(x_i, y_i)$, the quantity

$$b^2 \cdot x_i^2 + a^2 \cdot y_i^2 - b^2 \cdot a^2$$

is a measure telling where P lies in relation to the true ellipse. If this quantity is negative it means P lies inside the true ellipse, and if this quantity is positive it means P is outside the true ellipse. When this quantity is 0 (which may be rare but does happen) P is exactly on the ellipse. The expression

$$|\, b^2 \cdot x_i^2 + a^2 \cdot y_i^2 - b^2 \cdot a^2 \,|$$

is a more practical measure of the error. The absolute value will be needed when comparing two such errors. We define a function which we call the $EllipseError$ which is an error measure for each plotted point.

$$EllipseError(x_i, y_i) = |\, b^2 \cdot x_i^2 + a^2 \cdot y_i^2 - b^2 \cdot a^2 \,|$$

The ellipse plotting algorithm differs from the circle algorithm in that the ellipses' symmetry allows only 4 simultaneous points to be plotted at a time. We still need only calculate points in the first quadrant, but we need to complete a full 90°, not just 45°. In fact, the criterion for where to make the break is determined by the slope of the tangent line to the ellipse.

In the first quadrant the ellipse tangent line slope is always negative. Starting on the x-axis and wrapping in a counterclockwise direction the slope is large and negative which means the y-coordinates increase faster than the x-coordinates. But once the tangent line slope passes through the value -1 then the x-coordinates start changing faster than the y-coordinates.

Thus we will calculate two sets of points in the first quadrant. The first set starts on the positive x-axis and wraps in a counterclockwise direction until the tangent line slope reaches -1. The second set will start on the positive y-axis and wrap in a clockwise direction until the tangent line slope again reaches the value -1. See the figure at the end of this paper.

For the first set of points we will always increment y and we will test when to decrement x. This process is very similar to that for circles. For the second set of points we will always increment x and decide when to decrement y.

Our test decision as to when to decrease x for the first set of points is based on the comparison of the two values,

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

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

In any event, we need to know how the function $EllipseError(x_i, y_i)$ changes for each possible change in its arguments. We do not need to calculate $EllipseError$ anew for each next point, if we just keep track of how the value changes as the arguments change.

For the first set of points,

$$EllipseError(x_i - 1, y_i + 1) = |\, b^2 \cdot (\, x_i - 1\,)^2 + a^2 \cdot (\, y_i + 1\,)^2 - b^2 \cdot a^2 \,|$$

$$= |\, b^2 \cdot x_i^2 - b^2 \cdot 2x_i + b^2 + a^2 \cdot y_i^2 + a^2 \cdot 2y_i + a^2 - b^2 \cdot a^2 \,|$$

$$= |\, b^2 \cdot x_i^2 + a^2 \cdot y_i^2 - b^2 \cdot a^2 + b^2 \cdot (\, 1 - 2x_i\,) + a^2 \cdot (\, 2y_i + 1\,)\,|$$

while

$$EllipseError(x_i, y_i + 1) = |\, b^2 \cdot x_i^2 + a^2 \cdot (\, y_i + 1\,)^2 - b^2 \cdot a^2 \,|$$

$$= |\, b^2 \cdot x_i^2 + a^2 \cdot y_i^2 + a^2 \cdot 2y_i + a^2 - b^2 \cdot a^2 \,|$$

$$= |\, b^2 \cdot x_i^2 + a^2 \cdot y_i^2 - b^2 \cdot a^2 + a^2 \cdot (\, 2y_i + 1\,)\,|$$

The analysis of the following ellipse inequality is almost identical to that for the circle inequality except we multiply the appropriate change factor by either a^2 or b^2. The end result is that

$$EllipseError(x_i - 1, y_i + 1) \; < \; EllipseError(x_i, y_i + 1)$$

if and only if

$$2 \cdot [\, b^2 \cdot x_i^2 + a^2 \cdot y_i^2 - b^2 \cdot a^2 + a^2 \cdot (\, 2y_i + 1\,)\,] + b^2 \cdot (\, 1 - 2x_i\,) \; > \; 0$$

When this last inequality holds 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.

$$XChange \; = \; b^2 \cdot (\, 1 - 2x_i\,)$$

$$YChange \; = \; a^2 \cdot (\, 2y_i + 1\,)$$

$$EllipseError \; = \; b^2 \cdot x_i^2 + a^2 \cdot y_i^2 - b^2 \cdot a^2$$

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 $\pm 2 \cdot b^2$ and $\pm 2 \cdot a^2$ respectively. The initial $XChange$ value is $a^2 \cdot (\, 1 - 2a\,)$. The initial $YChange$ value is a^2. The initial $EllipseError$ value is 0. The program variable $EllipseError$ neither needs nor uses an absolute value.

We can analyze the tangent line slope by implicitly differentiating the equation

$$b^2 \cdot x^2 + a^2 \cdot y^2 \; = \; b^2 \cdot a^2$$

$$2 \cdot b^2 \cdot x + 2 \cdot a^2 \cdot y \cdot y' \; = \; 0$$

Solving for y' yields

$$y' \; = \; \frac{-2 \cdot b^2 \cdot x}{2 \cdot a^2 \cdot y}$$

Although one might normally expect to reduce the above fraction by dividing by 2, it turns out to be more efficient to ***not*** reduce this fraction.

Now we can determine where to break the two sets of calculated points in the first quadrant. The first set corresponds to where $y' > -1$. So we consider the inequality

$$\frac{-2 \cdot b^2 \cdot x}{2 \cdot a^2 \cdot y} \; > \; -1$$

and we decide to continue to plot points while

$$2 \cdot b^2 \cdot x \;+\; 2 \cdot a^2 \cdot y \;>\; 0.$$

Rather than perform the multiplications in this inequality each time through the while-loop test, we again take advantage of the fact that whenever x and y change, they change by exactly 1 unit. So we define two new values called $StoppingX$ and $StoppingY$. Although we compute these values recursively (iteratively), these values are really represented by the following equations.

$$StoppingX \;=\; 2 \cdot b^2 \cdot x_i \quad \text{and} \quad StoppingY \;=\; 2 \cdot a^2 \cdot y_i$$

When plotting the first set of points the initial values are given by $StoppingX = 2 \cdot b^2 \cdot a$ and $StoppingY = 0$. For the first set of points x decreases by 1 at each stage so $StoppingX$ has a corresponding decrease of $2 \cdot b^2$. For the first set of points $StoppingY$ increases by $2 \cdot a^2$.

When plotting the second set of points the initial values are given by $StoppingX = 0$ and $StoppingY = 2 \cdot a^2 \cdot b$. For the second set of points x increases by 1 at each stage so $StoppingX$ has a corresponding increase of $2 \cdot b^2$. For the second set of points $StoppingY$ decreases by $2 \cdot a^2$ each time y decreases by 1.

Now we can give the ellipse plotting algorithm. Below, CX, CY, and $XRadius$ and $YRadius$ refer to the ellipse's center point coordinates and its horizontal and vertical radial values. So $XRadius = a$ and $YRadius = b$. The constants $TwoBSquare$ and $TwoASquare$ are the pre-computed values $2 \cdot b^2$ and $2 \cdot a^2$.

```
procedure PlotEllipse(CX, CY, XRadius, YRadius : longint);
begin
   var  X, Y                       : longint;
        XChange, YChange           : longint;
        EllipseError               : longint;
        TwoASquare, TwoBSquare : longint;
        StoppingX, StoppingY   : longint;

   TwoASquare := 2*XRadius*XRadius;
   TwoBSquare := 2*YRadius*YRadius;
   X : =  XRadius;
   Y := 0;
   XChange : =  YRadius*YRadius*(1 − 2*XRadius);
   YChange : =  XRadius*XRadius;
   EllipseError := 0;
   StoppingX := TwoBSquare*XRadius;
   StoppingY := 0;

   { algorithm continues on the next page }
   while ( StoppingX ≥ StoppingY ) do     {1st set of points,
y′> − 1}
      begin
         Plot4EllipsePoints(X,Y);          {subroutine appears later}
         inc(Y);
         inc(StoppingY, TwoASquare);
         inc(EllipseError, YChange);
         inc(YChange,TwoASquare);
         if ((2*EllipseError + XChange) > 0 ) then
            begin
               dec(X);
               dec(StoppingX, TwoBSquare);
               inc(EllipseError, XChange);
               inc(XChange,TwoBSquare)
            end
      end;

   { 1st point set is done; start the 2nd set of points }

   X := 0;
   Y := YRadius;
   XChange := YRadius*YRadius;
   YChange := XRadius*XRadius*(1 − 2*YRadius);
   EllipseError := 0;
   StoppingX := 0;
   StoppingY := TwoASquare*YRadius;
   while ( StoppingX ≤ StoppingY ) do  {2nd set of points, y′< − 1}
      begin
         Plot4EllipsePoints(X,Y);       {subroutine appears later}
         inc(X);
         inc(StoppingX, TwoBSquare);
         inc(EllipseError, XChange);
         inc(XChange,TwoBSquare);
         if ((2*EllipseError + YChange) > 0 ) then
            begin
               dec(Y);
               dec(StoppingY, TwoASquare);
               inc(EllipseError, YChange);
               inc(YChange,TwoASquare)
            end
      end
end;  {procedure PlotEllipse}
```

The subroutine called $Plot4EllipsePoints$ takes advantage of the symmetry in the ellipse. We only calculate the points in the first quadrant, but for each such point we actually plot 4 other points at the same time as indicated in the figure below. The $Plot4EllipsePoints$ subroutine would normally be defined inside the above $PlotEllipse$ procedure. Also note that CX and CY refer to the ellipse's center point.

```
procedure Plot4EllipsePoints(X,Y : longint);
begin
   PutPixel(CX+X, CY+Y);          {point in quadrant 1}
   PutPixel(CX-X, CY+Y);          {point in quadrant 2}
   PutPixel(CX-X, CY-Y);          {point in quadrant 3}
   PutPixel(CX+X, CY-Y)           {point in quadrant 4}
end;  {procedure Plot4EllipsePoints}
```

As shown in the figure below, when this subroutine plots the first set of points, the four points would be like those numbered 1, 4, 5, and 8 in the figure. When plotting the second set of points, the four points would be like those numbered 2, 3, 6, and 7 in the figure.

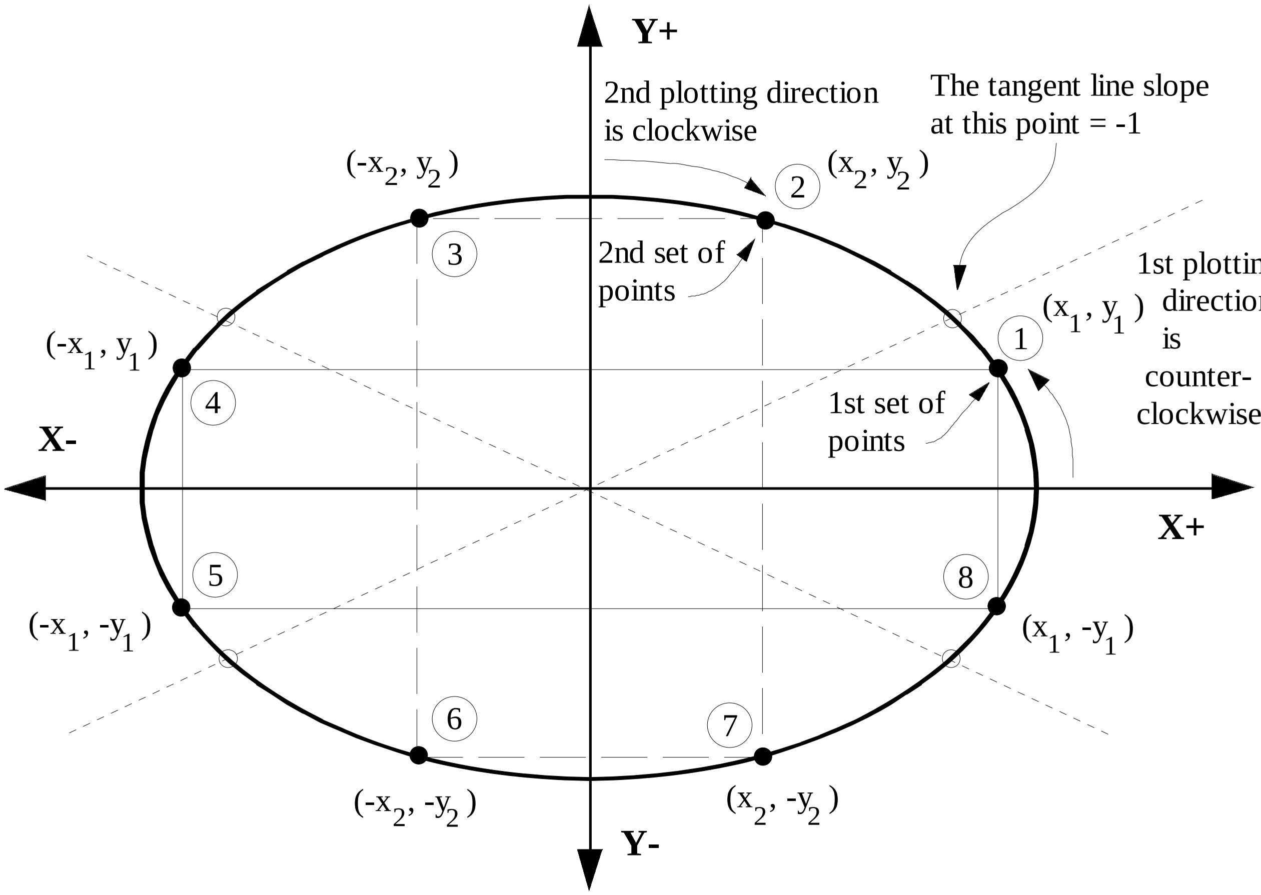

Figure 1. This figure indicates the two sets of points in the first quadrant that get plotted. The plotting algorithm uses two sets with 4-point symmetry.

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.