Notes About Rasterizing Lines And Circles

by Sunshine2k

Table of Contents

1. Introduction 1
2. Rasterizing Lines 1
   2.1 Digital Differential Analyzer (DDA) 2
   2.2 Bresenham Algorithm for Lines (Floating-point numbers) 4
   2.3 Bresenham Algorithm for Lines (Integer numbers) 5
3. Rasterizing Circles 6
   3.1 Simple approach evaluating the circle equation 7
   3.2 Bresenham for Circles (floating point numbers) 9
   3.3 Bresenham for Circles (integer numbers) 11
4. References 11

1. Introduction

Here I present you my thoughts and mathematical background information for my applet about rasterizing lines and circles; although I believe that investigating and playing around with the source code enhances the comprehension. Nevertheless, my goal is to present the reader (that's you) enough information and motivation to get a good understanding of the algorithms. Remember, rasterizing lines and ellipses shape the basics of graphical algorithms, so it's a must to know them ☺
I will not describe everything in detail because there is enough information on the net, but therefore I start with simple algorithms and describe the drawbacks of them, and then slowly improve the approaches to finally come up with the famous Bresenham algorithms. This document deals with lines and circles - ellipses are dismissed. If you understand this document and are interested in plotting ellipses, go on and check [2] – a really good article.

2. Rasterizing Lines

We assume that both endpoints of the line are given and the goal is to calculate all intermediate points. Endpoint 1 is given by (x_1, y_1), endpoint 2 by (x_2, y_2).

2.1 Digital Differential Analyzer (DDA)

The simplest algorithm, called Digital Differential Analyzer [1], makes in fact use of the equation of a line $y = mx + c$, where m is the slope and is defined as $m = \frac{dy}{dx}$. dy is the difference in y-direction $dy = y_2 - y_1$, analog $dx = x_2 - x_1$, see Figure 1

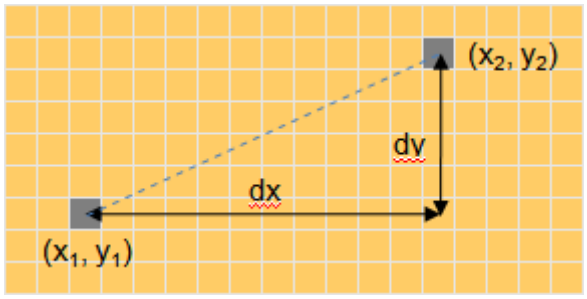

Figure 1- Line parameters

So at first, we determine in which direction the line is longer by checking if $dx > dy$ or $dy > dx$. Assume according to our diagram that dx is longer, so we set $length = dx$. Then we go along this site one by one and determine the next pixel of the line by $x = x + \frac{dx}{length}$ and $y = y + \frac{dy}{length}$, so in fact we interpolate the interval (x_1, y_1) to (x_2, y_2). See Figure 2 for a step by step example for a short line.

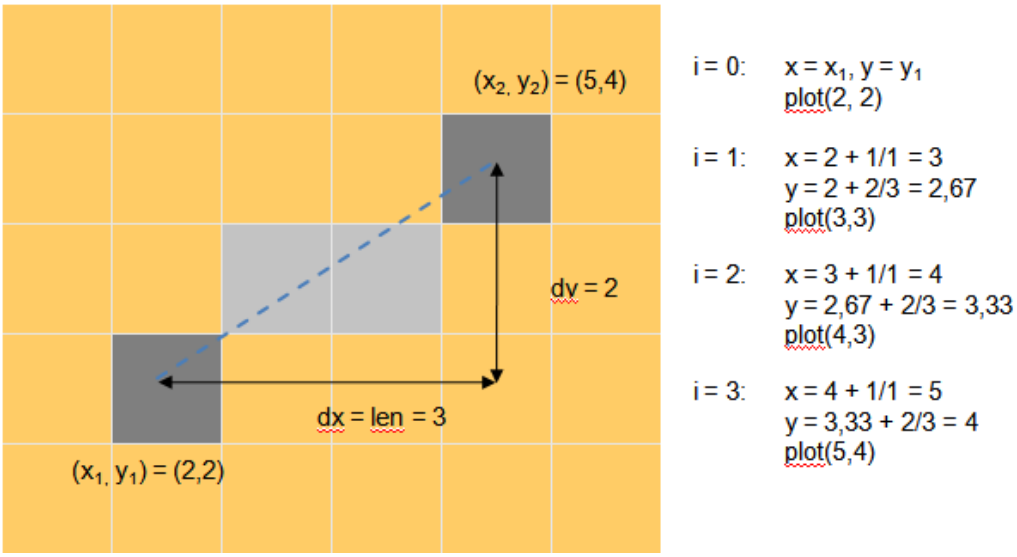

Figure 2 Simple example of DDA algorithm

```
public void DDA(int x1, int y1, int x2, int y2)
{
   if (|x2 – x1|) >= (|y2 – y1<)
     len = |x2 – x1|;
   else
     len = |y2 – y1|;

   // calculate increments
   dx = (x2 – x1) / len;
   dy = (y2 – y1) / len;

   // start point
   float x = x1;
   float y = y1;

   for (int i = 0; i < len; i++)
   {
     plot(round(x), round(y));
     x = x + dx;
     y = y + dy;
   }
   // set final pixel
   plot(x2, y2);
}
```

Listing 1 - DDA pseudo code

Listing 1 shows the pseudo code for my implementation of the Digital Differential Analyzer algorithm. Note that this works for any line, independent of direction and slope of the line (which is not self-evident), so absolute numbers are used in the length check.
In this and all following listings, I presume a function *plot(int x, int y)* which sets the pixel at the screen coordinate (x, y). The expected behavior of the *round()* function is that it returns the closest integer of the given floating point number. If you want/have to use a floor() function for getting the integer value, you may have to apply little changes. A good idea is to adjust the starting point to 'point a bit in the right direction' with something like

```
// start point
float x = x1 + 0.5 * ( (dx > 0) ? 1 : -1);
float y = y1 + 0.5 * ( (dy > 0) ? 1 : -1);
```

You can also change the loop condition from *i* < *len* to *i* <= *len* to avoid the final pixel setting but this also depends on the starting point calculation and round vs. floor function. But these are only minor details.
Ok, that was the first and simplest approach. Note that x and y has to be floating point numbers ☹ – now let's look at another approach.

2.2 Bresenham Algorithm for Lines (Floating-point numbers)

In this chapter we make the assumption that the line is in the first octant and for the slope applies 0° <= slope <= 45°, that is: x2 > x1, y2 > y1 and 0 <= dy <= dx. We will revoke this limitation in 2.3.
The Bresenham algorithm follows another approach, trying to create a preferably straight line. It proceeds in either x-direction or y-direction depending on an error value e for which applies -0.5 <= e <= 0.5 at each point in time. If e >= 0 the line proceeds in y-direction and e is decremented by 1; if e < 0 the line goes to x-direction. At the beginning e should be initialized depending on the slope to get better results – the more flat the slope angle is, the smaller should be the initial error value: E.g. a horizontal line must have an initial e smaller than zero because it must not proceed in y-direction while a slope greater than 1 should in the first step proceed in y-direction and thus requires an initial e of > 0.
Figure 3 shows the idea of the interrelation between e and the actual line, but note that this is just an outline to visualize the idea. The corresponding source code is given in
Listing 2.

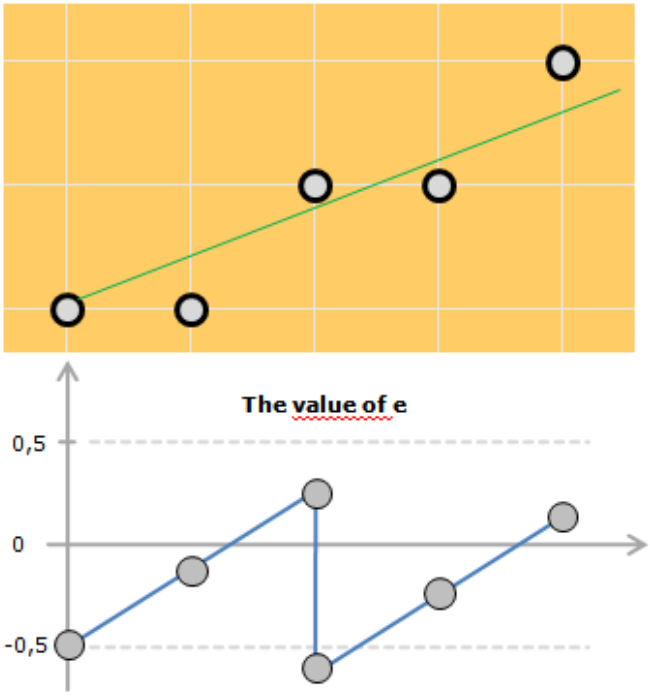

Figure 3 Relation of error value e and line proceeding

```
public void BresenhamFloat(int x1, int y1, int x2, int y2)
{
   int x = x1;
   int y = y1;

   int dx = x2 - x1;
   int dy = y2 - y1;

   float e = ((float)dy/(float)dx) - 0.5;
   for (int i = 1; i <= dx; i++)
   {
     plot(x, y);
     while (e >= 0)
     {
       y = y + 1;
       e = e - 1;
     }
     x = x + 1;
     e = e + (float)dy/(float)dx;
   }
}
```

Listing 2 Pseudo code of Bresenham Algorithm using floating point numbers

2.3 Bresenham Algorithm for Lines (Integer numbers)

The algorithm in 2.2 has two drawbacks which we going to solve in this chapter:

a) The algorithms uses floating-point numbers
b) The algorithm only works in the first octant

The only floating point value is the error value e. In order to solve a), we scale the error value by 2 * dx. So let's define $\bar{e} = 2 * dx * e$ and we just replace all three occurrences of e in
Listing 2 by $\bar{e}$.

I. $e = \frac{dy}{dx} - 0.5 \rightarrow \bar{e} = 2 * dx * \left(\frac{dy}{dx} - 0.5\right) = 2 * dy - dx$

II. $e = e - 1,\ e = \frac{\bar{e}}{2dx} \rightarrow \frac{\bar{e}}{2dx} = \frac{\bar{e}}{2dx} - 1 \rightarrow \bar{e} = \bar{e} - 2 * dx$

III. $e = e + \frac{dy}{dx}, e = \frac{\bar{e}}{2dx} \rightarrow \frac{\bar{e}}{2dx} = \frac{\bar{e}}{2dx} + \frac{dy}{dx} \rightarrow \bar{e} = \bar{e} + 2 * dy$

This substitution removes all divisions and allows $\bar{e}$ being an integer.
Applying the algorithm to all octants is not too difficult: the assumption was that dy <= dx which means the x-side is longer than the y-side – this is just because the algorithm goes along the x-side. So just interchange dx and dy if dy > dx and remember this swapping in an extra variable. Second, the direction of the line has to be considered. In 2.2 we assumed y2 > y1 and x2 > y1. In case we y1 > y2, the line

goes downwards instead of upwards, so we have to decrease y instead of increasing it; the same applies for x. In fact that's it, so
Listing 3 represents our final line drawing algorithm.

```
public void BresenhamInt(int x1, int y1, int x2, int y2)
{
   boolean changed;
   int x = x1;
   int y = y1;

   int dx = |x2 – x1|;
   int dy = |y2 – y1|;

   int signx = signum(x2 – x1);
   int signy = signum(y2 – y1);

   if (dy > dx)
   {
     swap(dx, dy);
     changed = true;
   }

   float e = 2 * dy - dx;
   for (int i = 1; i <= dx; i++)
   {
     plot(x, y);
     while (e >= 0)
     {
       if (changed)
         x = x + 1;
       else
         y = y + 1;
       e = e – 2 * dx;
     }
     if (changed)
       y += signy;
     else
       x += signx;
     e = e + 2 * dy;
   }
}
```

Listing 3 code of Bresenham Algorithm using integer numbers

3. Rasterizing Circles

Similar top chapter 2, we proceed step-by-step from a not-so-good to a really-good algorithm. A circle is defined by a midpoint (mx, my) and a radius r.

3.1 Simple approach evaluating the circle equation

The simplest approach to rasterizing a circle bases on the circle equation

$$(x - mx)^2 + (y - my)^2 = r^2$$

Solving this for y gives

$$y = my \pm \sqrt{r^2 - (x - mx)^2}$$

This equation also reveals the basic symmetry of a circle. For each x, we get (cause of this ±) two values for y – one above the midpoint, one below, see Figure 4.

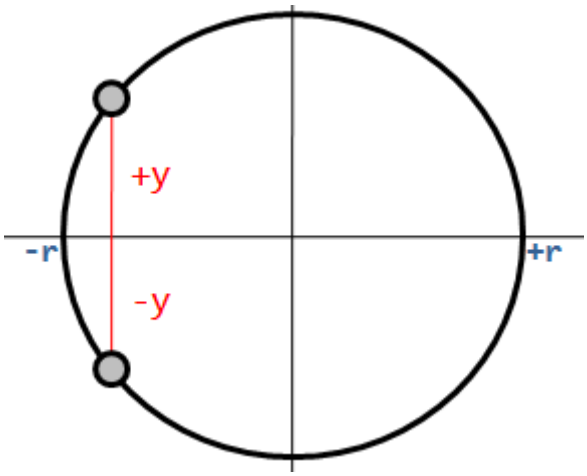

Figure 4 2-way symmetry of a circle

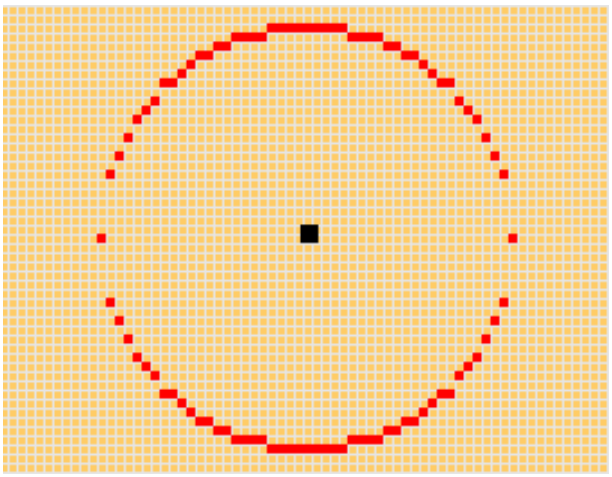

Figure 5 Result of 1-try algorithm

Based on this fact, one can easily think of an algorithm which goes step-by-step from –r till +r in x-direction, and for each x-value the two y-values of the circle are calculated and plotted:

```
for (x = -radius; x <= radius; x++)
{
  fy = sqrt( (radius*radius) - x*x);
  plot(mx + x, my + fy);  // plot circle point “above” x-axis
  plot(mx + x, my - fy);  // plot circle point “below” x-axis
}
```

Unfortunately, if you implement and try this out, you will see holes in the circle, see Figure 5. We came across a similar effect in 2.2 where we limited the slope of the line to be less than 45°. Here we see if the slope is large (the distance in to next y-value is somehow greater than to the next x-value), the holes occur. Of course, we could follow the same idea and make a check of the current slope in each step in order to decide in which direction (x or y) we should go on.

A more elegant solution is to make better use of the symmetry of a circle. By calculating one point of the circle (x,y), we have actually 8 points of the circle, see Figure 6.

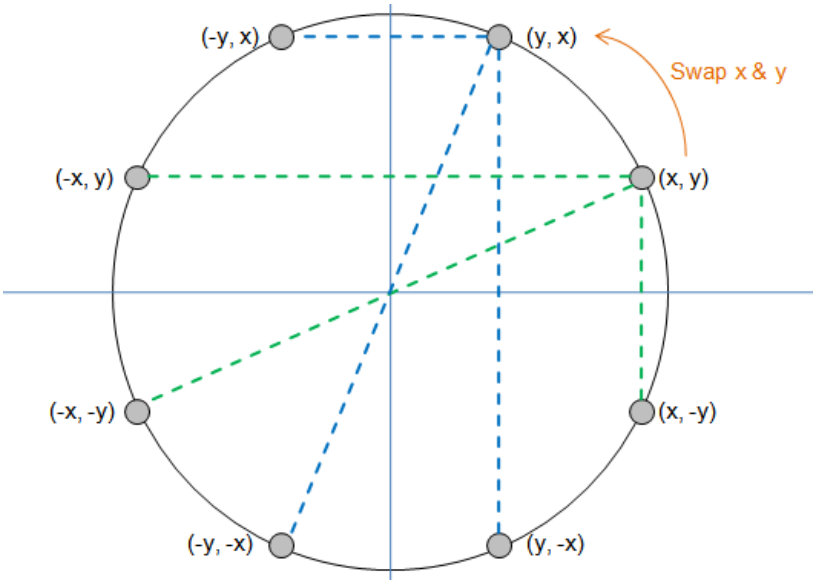

Figure 6 8-way symmetry of a circle

Using this 8-way symmetry, a correct circle can finally be drawn. Instead of proceeding the range [-radius, radius] in the loop, it's enough to go from 1 to the point where y becomes greater than x (the point where the slope get's so big that holes occur) and get the other points by symmetry. The pseudo code is given in xxx, again we add 0.5 to get the correct rounding values (not that y is a double, but for plotting a pixel it's casted to an integer):

```
public void CircleSimple(int mx, int my, int radius)
{
  double y;
  int x = 1;
  y = sqrt((radius*radius) - (x*x)) + 0.5;
  while (x <= y)
  {
    plot(mx + x, my + y);
    plot(mx + x, my - y);
    plot(mx - x, my + y);
    plot(mx - x, my - y);
    plot(mx + y, my + x);
    plot(mx + y, my - x);
    plot(mx - y, my + x);
    plot(mx - y, my - x);
    x += 1;
    y = sqrt((radius*radius) - (x*x)) + 0.5;
  }
}
```

Listing 4 Circle algorithm using 8-way symmetry

3.2 Bresenham for Circles (floating point numbers)

Although the result looks pretty good, let's try to optimize it. In each loop a square root is calculated which is quite computationally intensive. So we look for a fast, incremental algorithm that is able to draw good-quality circles.
So let's derive the algorithm step-by-step where we use a following simplification of the circle equation $x^2 + y^2 = r^2$ - that's a circle with midpoint (0,0). This is no limitation as each circle with an arbitrary midpoint can be created by a circle with midpoint (0,0) and a translation. Also we only focus on the first octant as we know the other octants can be drawn by using the symmetry. Also we start at point (0,R) and move point by point to (R,0).

So let's consider the implicit circle function $F(x,y) = x^2 + y^2 - r^2$. For each point (x,y) on the circle applies $F(x,y) = 0$; $F(x,y) < 0$ means the point is inside the circle, if $F(x,y) > 0$ the point is outside.

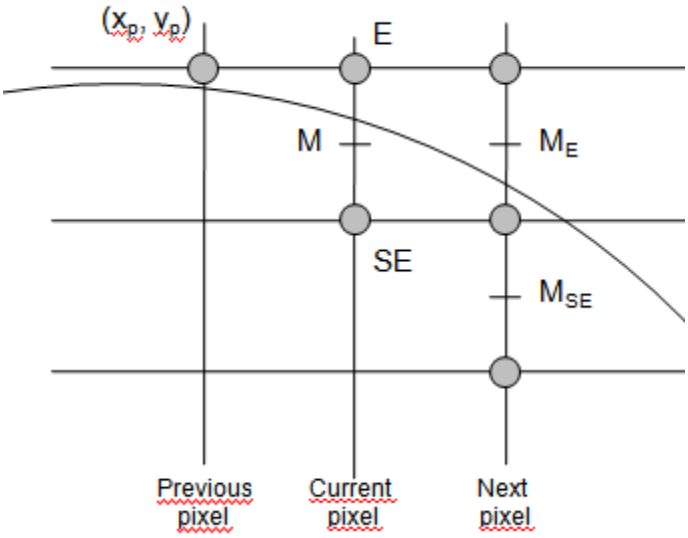

Figure 7 Decision of next circle point

Assume the current point is (x_p, y_p), see Figure 7. Whether the next point is E or SE depends on the relative position of the circle to M – that means it depends on the sign of F(M). So create a decision variable d_{old}.

$$d_{old} = F(M) = F\left(x_p + 1, y_p - \frac{1}{2}\right) = (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - R^2$$

If $d_{old} < 0$, M is inside the circle, so the circle is nearer to E and E is the next point to choose. Again the question occurs what is the next point – or equivalent, if M_E is inside the circle or not.

$$d_{new} = F(M_E) = F\left(x_p + 2, y_p - \frac{1}{2}\right)$$
$$= (x_p + 2)^2 + \left(y_p - \frac{1}{2}\right)^2 - R^2$$

$$= d_{old} + 2x_p + 3$$

So the increment if $d_{old} < 0$ (proceeding in x-direction) is $2x_p + 3$.

If $d_{old} > 0$, M is outside the circle, so the circle is nearer to SE and SE is the next point to choose. Again the question occurs what is the next point – or equivalent, if M_{SE} is inside the circle or not.

$$d_{new} = F(M_E) = F\left(x_p + 2, y_p - \frac{3}{2}\right)$$
$$= \left(x_p + 2\right)^2 + \left(y_p - \frac{3}{2}\right)^2 - R^2$$
$$= d_{old} + 2x_p - 2y_p + 5.$$

So the increment if $d_{old} > 0$ (proceeding in y-direction) is $2x_p - 2y_p + 5$. In both cases, the next step is defined by a linear term.

In the last step the starting condition is required. As we start at (0,R), the point M is in this case $(1, R - \frac{1}{2})$. So d_{old} has to be initialized as

$$d_{old} = F(M) = F\left(1, R - \frac{1}{2}\right) = 1^2 + \left(R - \frac{1}{2}\right)^2 - R^2$$
$$= 1 + R^2 - R + \frac{1}{4} - R^2$$
$$= \frac{5}{4} - R$$

Using this, we can formulate our algorithm:

```
public void CircleBresenhamFloat(int mx, int my, int radius)
{
  int x = 0;
  int y = radius;
  double d = 1.25 - radius;
  while (x < y)
  {
    if (d < 0)
    {
      d = d + 2 * x + 3;
      x += 1;
    }
    else
    {
      d = d + 2 * (x-y) + 5;
      x += 1;
      y -= 1;
    }
    plot(mx + x, my + y);
    plot(mx + x, my - y);
    plot(mx - x, my + y);
    plot(mx - x, my - y);
```

```
    plot(mx + y, my + x);
    plot(mx + y, my - x);
    plot(mx - y, my + x);
    plot(mx - y, my - x);
  }
}
```

Listing 5 Bresenham circle algorithm using floating point numbers

3.3 Bresenham for Circles (integer numbers)

Similar to 2.3, we want to eliminate the floating point numbers in order to use only integers. The only variable to be changed is d. So let's define $h = d - \frac{1}{4}$ ($d = h + \frac{1}{4}$) and replace d by h.

i. $d = \frac{5}{4} - R$ becomes $h = 1 - R$.

ii. The comparison $d < 0$ becomes $h < -\frac{1}{4}$ but because h is an integer, the comparison remains $h < 0$.

That's it. So to get the plotting algorithm for circles only using integer numbers, solely replace the line

```
  double d = 1.25 - radius;
```

in Listing 5 by

```
  int d = 1 - radius;
```

Finished!

4. References

[1] http://en.wikipedia.org/wiki/Digital_Differential_Analyzer_%28graphics_algorithm%29
[2] http://homepage.smc.edu/kennedy_john/bcircle.pdf
[3] University of Karlsruhe, Einführung in die graphische Datenverarbeitung

Sunshine, December 2010

www.sunshine2k.de || www.bastian-molkenthin.de