Drawing Lines – The Bresenham Algorithm

Graphics Tutorial
By Hexar

Introduction

A line segment is defined by an infinite set of points which lie between two points; these points have no area. A problem occurs, however, when drawing lines on a computer screen, namely that we have to approximate the line with pixels which are not infinitely small, and which are limited to fixed positions based on the screen resolution.

Problem

It's relatively easy to create an algorithm to generate the set of points which approximate a line based on some floating point math:

```
void draw_line(int x1, int y1, int x2, int y2)
{
   int dx = x2 – x1;
   int dy = y2 – y1;
   float m = dy/dx;

   for (int x = x1; x < x2; x++)
     {
      int y = m*x + y1 + 0.5;
      putpixel(x, y);
     }
}
```

Figure 1. Floating-point line-drawing algorithm

This algorithm, of course, relies on some assumptions such as:

- Non-infinite slope
- x1 < x2
- y1 < y2

But, for the most part, it works. So what's the problem? Any hardcore game-programmer will tell you: **Speed.**

The Bresenham / Midpoint Algorithm is a solution that not only correctly picks these same points, it does it with integer math only, which results in a big speed improvement.

Approximating Pixels

Let's say we want to draw a line between two points, and that we will always draw pixels on the endpoints. Let us assume that the slope m follows $0 \le m \le 1$, and that x1 < x2.

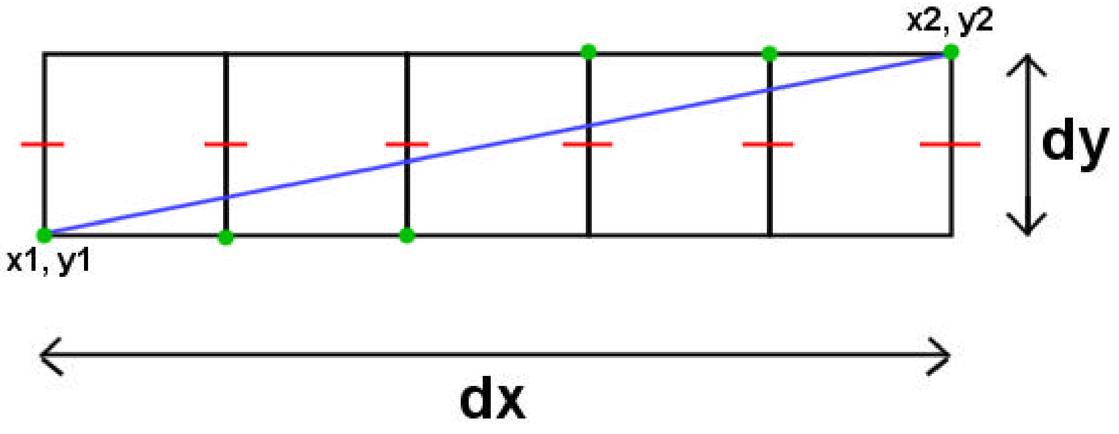

Figure 2. Selecting pixels based on midpoints

The above picture is a good representation of how one might choose the pixels to light up. The blue line is the "true" line between the endpoints, but we approximate to the green pixels based on the relative positions of the blue line and the midpoints. Namely,

```
If (BlueLine < Midpoint)
   Plot_East_Pixel();
Else
   Plot_Northeast_Pixel();
```

What if we came up with an equation, given a line and a point, that will tell us if the line is above or below that line? Let's play with the line equation.

$$y = \frac{dy}{dx}x + B$$

$$dx * y = dy * x + B * dx$$ (rewritten)

$$0 = dy * x - dx * y + B * dx$$ (rewritten again)

Notice that the left side of this equation equals zero on the line. It can be shown that for any point x, y **above** the line, the right side of the equation becomes a negative number, and any point x,y below the line becomes positive. Or, in other words, if we define a line function F:

$$F(x,y) = 2*dy*x - 2*dx*y + 2*B*dx$$

Then for any point (x,y) that represents a midpoint between the next East or Northeast pixel,

F(x,y) < 0,   when   BlueLine < Midpoint
F(x,y) > 0,   when   BlueLine > Midpoint

Note also that I changed the equation to include a factor of 2. This will be explained in a moment; but realize for now that this factor does not change the results of the equation; positive numbers remain positive, and negative numbers remain negative.

Using The Line Function

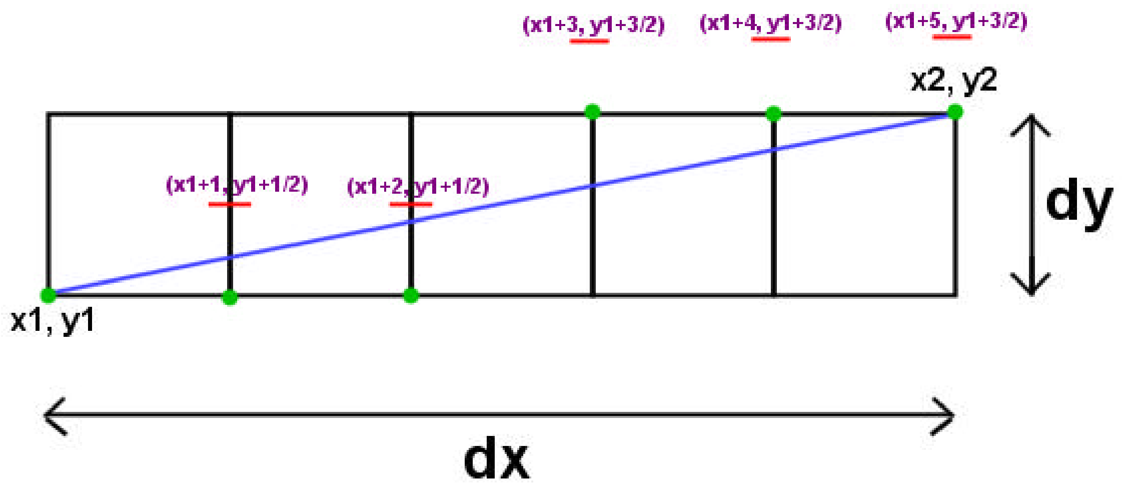

Figure 3. Midpoint locations of interest

For the first midpoint, given by (x1+1, y1+1/2), we can use our equation F(x,y) to determine if the second pixel (after x1,y1) to light up should be (x1+1, y1) or (x1+1, y1+1). So,

$$F(x1+1, y1+\tfrac{1}{2}) = 2*dy*(x1+1) - 2*dx*(y1+\tfrac{1}{2}) + 2*B*dx$$

This can be rewritten as

$$F(x1+1, y1+\tfrac{1}{2}) = 2*dy*x1 + 2*dy - 2*dx*y1 - dx + 2*B*dx$$

Next, look at the line function on (x1, y1):

$$F(x1, y1) = 2 * dy * x1 - 2 * dx * y1 + 2 * B * dx = 0$$

therefore

$$F(x1+1, y1+1/2) = \cancel{2*dy*x1} + \mathbf{2*dy} - \cancel{2*dx*y1} - \mathbf{dx} + \cancel{2*B*dx}$$
$$d0 = F(x1+1, y1+1/2) = \mathbf{2*dy - dx}$$

We call this value d0 (for "decision variable – initial value")

Now, for the next midpoint, we either evaluate F(x1+2, y1+1/2) or F(x1+2, y1+3/2), depending on whether we chose the East or Northeast pixel for the first midpoint, respectively.

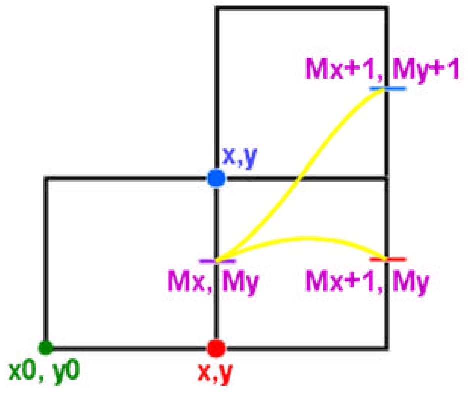

Figure 4. Incremental Line Function

As it turns out, we can exploit the nature of our Line Function F(x,y) because of the relationship between successive midpoints. Let's calculate the difference in the Line Function between midpoints, depending on our previous pixel choice:

$incE = F(Mx+1, My) - F(Mx,My) \quad = \mathbf{2*dy}$ (E pixel)

$incNE = F(Mx+1, My+1) - F(Mx, My) = \mathbf{2*dy - 2*dx}$ (NE pixel)

To clarify; each time we make a decision on the choice of East or Northeast, we must make a new calculation based on the new midpoint. As shown above, an East choice means the x-component of the midpoint increases but the y-component does not.

So now, all we really have to do is calculate the initial value of d0 and loop through the pixels, recalculating the value of F(x,y). The beauty of this is that all of the values incE, incNE, and d0 are constant integers.

Speaking of which, remember that factor of two we introduced to F(x,y)? That was added because each midpoint falls halfway between two pixels, and simply by multiplying both sides of that equation by 2, we can get an equation that uses only integers. How sweet.

On the next page, a copy of half of the Bresenham / Midpoint Algorithm is displayed. I say half because it only works for lines whose slope fulfills $-1 \leq m \leq 1$. To deal with the other case, simply rewrite the function and switch x for y.

The Code

```
void brenenham1(int x1, int y1, int x2, int y2)
{
   int slope;
   int dx, dy, incE, incNE, d, x, y;

   // Reverse lines where x1 > x2
   if (x1 > x2)
     {
      bresenham1(x2, y2, x1, y1);
      return;
     }

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

   // Adjust y-increment for negatively sloped lines
   if (dy < 0)
     {
      slope = -1;
      dy = -dy;
     }
   else
     {
      slope = 1;
     }

   // Bresenham constants
   incE = 2 * dy;
   incNE = 2 * dy - 2 * dx;
   d = 2 * dy - dx;
   y = y1;

   // Blit
   for (x = x1; x <= x2; x++)
     {
      putpixel(x, y);

      if (d <= 0)
        {
         d += incE;
        }
      else
        {
         d += incNE;
         y += slope;
        }
     }
}
```