Line Drawing algorithms

Given the endpoints (x1, y1) and (x2, y2), we want to draw an approximation of the line connecting (x1, y1) to (x2, y2)

[1] Direct (Naïve) method

$$\frac{y-y_1}{x-x_1}=\frac{y_2-y_1}{x_2-x_1}=slope$$

$$y=y_1+(x-x_1)(y_2-y_1)/(x_2-x_1)$$

Or $$y=y_1+(x-x_1)*slope$$

Algorithm DrawLine(x_s, y_s, x_e, y_e)

1. $dx=x_e-x_s$
2. $dy=y_e-y_s$
3. $slope=dy/dx$
4. if x_s>x_e then
   a. swap x_s with x_e
   b. swap y_s with y_e
   end if
5. for x=x_s to x_e
   a. y=$y_s+(x-x_s)*slope$
   b. yi=round(y)
   c. drawpixel(x,yi)
   end for

end Algorithm

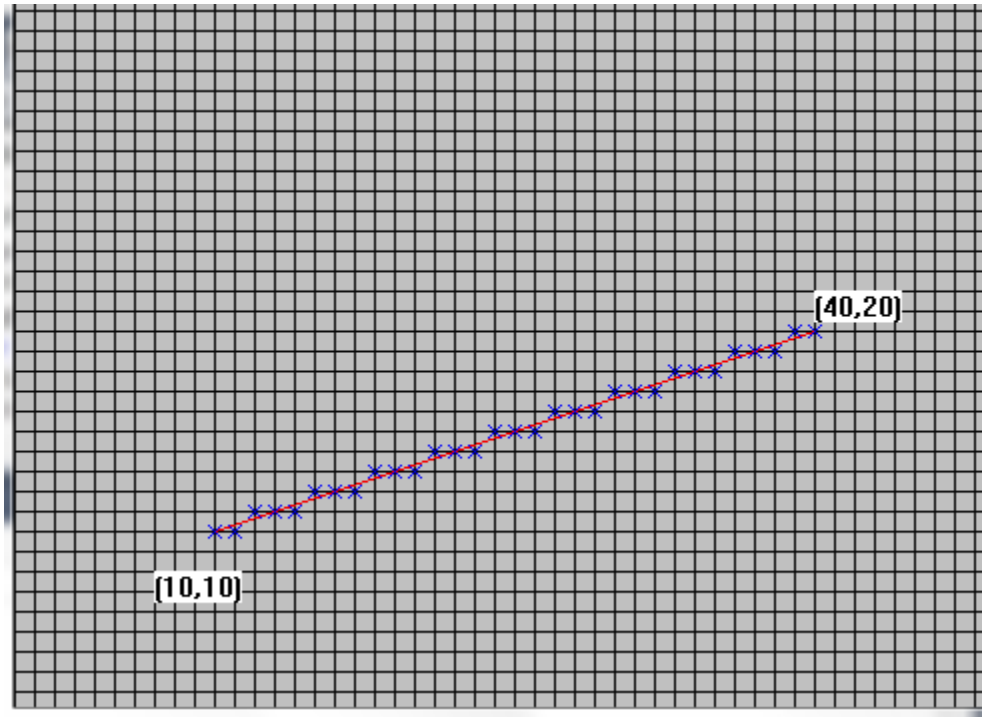
To implement this algorithm, we can first define the following utility functions

```
void swap(int& x,int& y)
{
      int tmp=x;
      x=y;
      y=tmp;
}
int round(double x)
{
      return (int)(x+0.5);
}
```

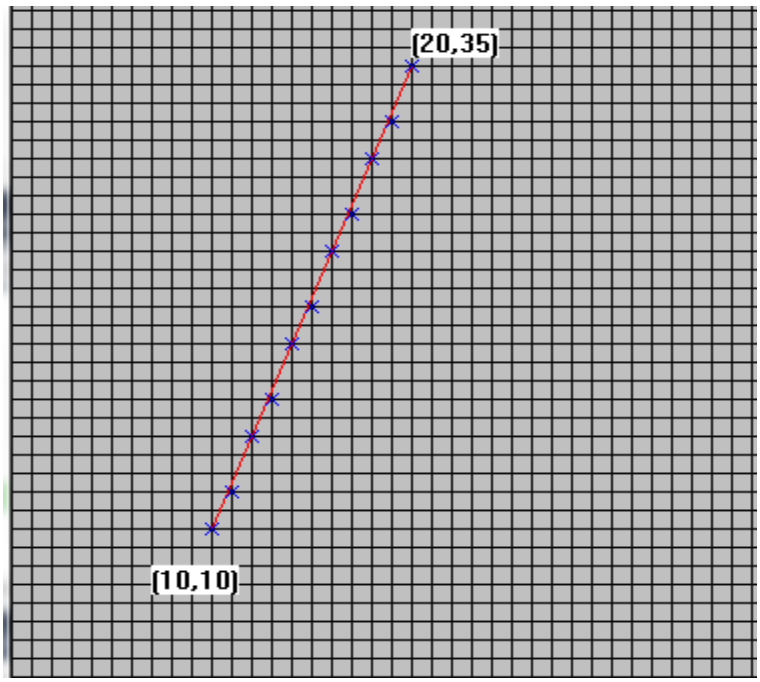
The visual C++ implementation is hence as follows:

```
void DrawLine(HDC hdc,int xs,int ys,int xe,int ye,COLORREF color)
{
	int dx=xe-xs;
	int dy=ye-ys;
	double slope=(double)dy/dx;
	if(xs>xe)
	{
		swap(xs,xe);
		swap(ys,ye);
	}
	for(int x=xs;x<=xe;x++)
	{
		int y=round(ys+(x-xs)*slope);
		SetPixel(hdc,x,y,color);
	}
}
```

The following figure shows the point generated by the above algorithm when called with the endpoints (10, 10) and (40, 20).



You might note that the algorithm will draw a disconnected lie if the slope of the line is greater than 1. To show this, try to call the function with the end points (10, 10), (20, 35) where the generated points are as shown in the following figure.



The reason is that in each iteration of the 'for' loop, x is incremented by 1 and y is incremented by the of the slope and is then rounded. If the slope>1, y will increase by more than 1. To fix this problem, the following is a modified version of the algorithm:

```
void DrawLine2(HDC hdc,int xs,int ys,int xe,int ye,COLORREF color)
{
	int dx=xe-xs;
	int dy=ye-ys;
	if(abs(dy)<=abs(dx))
	{
		double slope=(double)dy/dx;
		if(xs>xe)
		{
			swap(xs,xe);
			swap(ys,ye);
		}
		for(int x=xs;x<=xe;x++)
		{
			int y=round(ys+(x-xs)*slope);
			SetPixel(hdc,x,y,color);
		}
	} else
	{
		double islope=(double)dx/dy;
		if(ys>ye)
		{
			swap(xs,xe);
			swap(ys,ye);
		}
		for(int y=ys;y<=ye;y++)
```

```
            {
                  int x=round(xs+(y-ys)*islope);
                  SetPixel(hdc,x,y,color);
            }

      }
}
```

[2] Simple Digital Differential analyzer method (simple DDA)

When x is incremented (decremented) by 1, y is incremented (decremented) by the slope of the line. Likewise, when y is incremented (decremented) by 1, x is incremented (decremented) by the inverse of the slope. We can thus express the line points that will be drawn as (x_k, y_k) with the following iterative definition:

Case: |slope|<1

$$x_{k+1} = x_k \pm 1$$

$$y_{k+1} = y_k \pm slope$$

Case: |slope|>1

$$y_{k+1} = y_k \pm 1$$

$$x_{k+1} = x_k \pm 1/slope$$

With initial values:

$$x_{1=x_s}, y_{1=y_s}$$

The visual C++ implementation of the simple DDA algorithm is given below

```
void SimpleDDA(HDC hdc,int xs,int ys,int xe,int ye,COLORREF color)
{
      int dx=xe-xs;
      int dy=ye-ys;
      SetPixel(hdc,xs,ys,color);
      if(abs(dx)>=abs(dy))
      {
            int x=xs,xinc= dx>0?1:-1;
            double y=ys,yinc=(double)dy/dx*xinc;
            while(x!=xe)
            {
            x+=xinc;
            y+=yinc;
            SetPixel(hdc,x,round(y),color);
            }
      }
```

```
      else
      {
            int y=ys,yinc= dy>0?1:-1;
            double x=xs,xinc=(double)dx/dy*yinc;
            while(y!=ye)
            {
            x+=xinc;
            y+=yinc;
            SetPixel(hdc,round(x),y,color);
            }
      }
}
```

Note that the simple DDA algorithm is faster than the direct (naïve) algorithm since the loop in the direct algorithm contains floating point multiplication addition and rounding while it contains only addition and rounding in the simple DDA algorithm.

[3] Bresenham (midpoint) algorithm (integer DDA)

Line equation can be written as:

$$\frac{y-y_s}{x-x_s}=\frac{y_e-y_s}{x_e-x_s}=\frac{\Delta y}{\Delta x}$$

Or:

$$(y-y_s)\Delta x-(x-x_s)\Delta y=0$$

Consider the function:

$$f(x,y)=(y-y_s)\Delta x-(x-x_s)\Delta y$$

Any point (x, y) has one of three cases depending on the function f(x, y) provided that Δx is positive:

$$f(x,y)=\begin{cases}0 & (x,y)\ is\ on\ the\ line\\ <0 & (x,y) is\ under\ the\ line\\ >0 & (x,y) is\ above\ the\ line\end{cases}$$

Also provided that Δy is positive:

$$f(x,y)=\begin{cases}0 & (x,y)\ is\ on\ the\ line\\ <0 & (x,y) is\ to\ the\ right\ of\ the\ line\\ >0 & (x,y) is\ to\ the\ left\ of\ the\ line\end{cases}$$

Consider the case when Δx is positive and 0<slope<1:

In this case the algorithm increments x by 1 at each iteration. The variable y is incremented by 1 if the next midpoint (x+1, y+1/2) is under the line (i.e. when f(x+1, y+1/2)<0)

1. $\Delta x = x_e - x_s$
2. $\Delta y = y_e - y_s$
3. $x = x_s$
4. $y = y_s$
5. Drawpixel(x,y)
6. while x<x_e
    a. d=f(x+1,y+1/2)
    b. if d<0 then
        y=y+1
       end if
    c. x=x+1
    d. DrawPixel(x,y)

   End while

In the above algorithm, the computation of the variable d inside the loop is given by:

$$d(x,y) = f\left(x+1, y+\frac{1}{2}\right) = \left(y+\frac{1}{2}-y_s\right)\Delta x - (x+1-x_s)\Delta y$$

This includes addition and multiplication operations. To reduce the complexity of computing d, we can use an iterative approach as follows:

Initial value of d (before the loop):

$$\boldsymbol{d_{initial} = d(x_s, y_s)} = \left(y_s+\frac{1}{2}-y_s\right)\Delta x - (x_s+1-x_s)\Delta y = \frac{\Delta x}{2} - \Delta y$$

Change in d

If d<0 then

$$\begin{aligned}\Delta d &= d(x+1, y+1) - d(x,y)\\ &= \left(y+\frac{3}{2}-y_s\right)\Delta x - (x+2-x_s)\Delta y - \left[\left(y+\frac{1}{2}-y_s\right)\Delta x - (x+1-x_s)\Delta y\right]\\ &= \Delta x - \Delta y\end{aligned}$$

Else

$$\begin{aligned}\Delta d &= d(x+1, y) - d(x,y)\\ &= \left(y+\frac{1}{2}-y_s\right)\Delta x - (x+2-x_s)\Delta y - \left[\left(y+\frac{1}{2}-y_s\right)\Delta x - (x+1-x_s)\Delta y\right]\\ &= -\Delta y\end{aligned}$$

End

The previous algorithm can thus be reduced to the following:

1. $\Delta x = x_e - x_s$
2. $\Delta y = y_e - y_s$
3. $x = x_s$

4. $y = y_s$
5. $d = \frac{\Delta x}{2} - \Delta y$
6. $change1 = \Delta x - \Delta y$
7. $change2 = -\Delta y$
8. Drawpixel(x,y)
9. while x<x_e
    a. if d<0 then
        d=d+change1
        y=y+1
      else
        d=d+change2
      end if
    b. x=x+1
    c. DrawPixel(x,y)
   End while

Note that step 5 of the above algorithm contains division by 2 leading to floating point results. We can avoid this by multiplying the variable d by 2. The algorithm will not be affected by this multiplication since it considers only the sign of d not its value. Of course the sign of d will not change with multiplication by any positive value. The final version of the algorithm is given next:

Algorithm MidPoint(x_s, y_s, x_e, y_e)

1. $\Delta x = x_e - x_s$
2. $\Delta y = y_e - y_s$
3. $x = x_s$
4. $y = y_s$
5. $d = \Delta x - 2\Delta y$
6. $change1 = 2(\Delta x - \Delta y)$
7. $change2 = -2\Delta y$
8. Drawpixel(x,y)
9. while x<x_e
    a. if d<0 then
        d=d+change1
        y=y+1
      else
        d=d+change2
      end if
    b. x=x+1
    c. DrawPixel(x,y)
   End while

The algorithm uses only integer operations and hence it is faster than the other algorithms.

For the case when slope> 1 and Δy is positive, the algorithm will increment y in every loop iteration and x will be incremented only if the middle point (x+1/2,y+1) is to the left of the line (i.e. f(x+1/2,y+1)>0).

1. $\Delta x = x_e - x_s$
2. $\Delta y = y_e - y_s$
3. $x = x_s$
4. $y = y_s$
5. Drawpixel(x,y)
6. while x<x_e
    a. d=f(x+1/2,y+1)
    b. if d>0 then
        x=x+1
       end if
    c. y=y+1
    d. DrawPixel(x,y)

   End while

In the above algorithm, the computation of the variable d inside the loop is given by:

$$d(x,y) = f\left(x + \frac{1}{2}, y + 1\right) = (y + 1 - y_s)\Delta x - \left(x + \frac{1}{2} - x_s\right)\Delta y$$

Again, to reduce the complexity of computing d, we can use an iterative approach as follows:

Initial value of d (before the loop):

$$\boldsymbol{d_{initial} = d(x_s, y_s)} = (y_s + 1 - y_s)\Delta x - \left(x_s + \frac{1}{2} - x_s\right)\Delta y = \Delta x - \frac{\Delta y}{2}$$

Change in d

If d>0 then

$$\begin{aligned}\Delta d &= d(x+1, y+1) - d(x,y)\\ &= (y + 2 - y_s)\Delta x - \left(x + \frac{3}{2} - x_s\right)\Delta y - \left[(y + 1 - y_s)\Delta x - \left(x + \frac{1}{2} - x_s\right)\Delta y\right]\\ &= \Delta x - \Delta y\end{aligned}$$

Else

$$\begin{aligned}\Delta d &= d(x, y+1) - d(x,y)\\ &= (y + 2 - y_s)\Delta x - \left(x + \frac{1}{2} - x_s\right)\Delta y - \left[(y + 1 - y_s)\Delta x - \left(x + \frac{1}{2} - x_s\right)\Delta y\right]\\ &= \Delta x\end{aligned}$$

End

Multiplying d by 2 as before we reach at the following:

Algorithm MidPoint(x_s, y_s, x_e, y_e)

1. $\Delta x = x_e - x_s$
2. $\Delta y = y_e - y_s$

3. $x = x_s$
4. $y = y_s$
5. $d = 2\Delta x - \Delta y$
6. $change1 = 2(\Delta x - \Delta y)$
7. $change2 = 2\Delta x$
8. Drawpixel(x,y)
9. while y<y_e
    a. if d>0 then
        d=d+change1
        x=x+1
      else
        d=d+change2
      end if
    b. y=y+1
    c. DrawPixel(x,y)
   End while