Efficient Line Drawing

Graphics 1 CMP-5010B

Dr. David Greenwood

david.greenwood@uea.ac.uk

SCI 2.16a University of East Anglia

Spring 2022

Contents

Bresenham's Line Algorithm

Improving the efficiency of the DDA line drawing algorithm.

- remove floating point operations
- minimise the number of operations

Let's make clear some assumptions:

- pixel coordinates are integers
- left to right for x
- bottom to top for y.
- $x_0 < x_1$ and $y_0 < y_1$
- the slope of the line is between 0 and 1, i.e. $0 \leq m \leq 1$

Following these assumptions, the simplest algorithm is:

```
for x = x0 to x1:
    decide y value
    draw(x, y)
```

What is an *efficient* way to decide the y value?

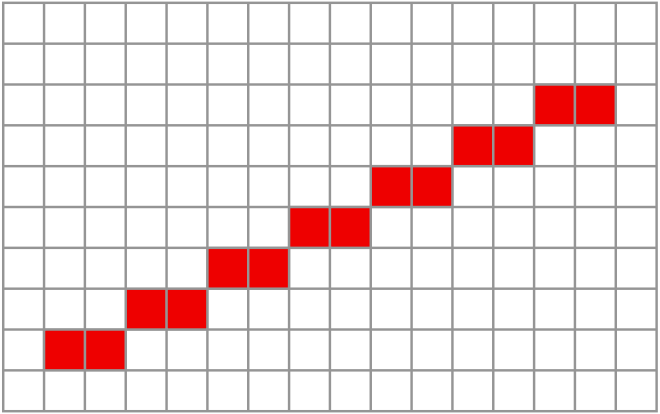

Figure 1: pixel line

As we step in the x direction, we observe that:

- y stays the same
- **or** y increases by 1.

We can include this observation in our algorithm:

```
x = x0
y = y0
draw(x, y)
while x < x1:
    x = x + 1
    if y should increment:
        y = y + 1
    draw(x, y)
```

Assuming the line is given by $y = mx + c$:

- we are setting `y = round(mx) + c`
- each unit step of x will increment y by m

Let `fraction` be the amount y has increased since the last y increase.

- We want to increment y when `fraction` is $\geq \frac{1}{2}$.

```
x = x0
y = y0
fraction = start_value
fraction_step = (y1 - y0) / (x1 - x0)
draw(x, y)
while x < x1:
    x = x + 1
    fraction = fraction + fraction_step
    if fraction >= 1/2:
        y = y + 1
        fraction = fraction - 1
    draw(x, y)
```

First we have: $m = \frac{y_1 - y_0}{x_1 - x_0}$

- To remove the fraction, we multiply by $(x_1 - x_0)$.
- To remove the comparison to $1/2$ we multiply by 2.

hence:

$$\begin{aligned} fraction\_step &= \frac{y1 - y0}{x1 - x0} \times (x1 - x0) \times 2 \\ &= 2(y1 - y0) \end{aligned}$$

We also want to set a `start_value` for `fraction`:

$$start\_value = 2(y_1 - y_0) - (x_1 - x_0)$$

```
x = x0
y = y0
fraction = 2 * (y1 - y0) - (x1 - x0)
fraction_step = 2 * (y1 - y0)
draw(x, y)
while x < x1:
    x = x + 1
    fraction = fraction + fraction_step
    if fraction >= 0:
        y = y + 1
        fraction = fraction - 2 * (x1 - x0)
    draw(x, y)
```

Bresenham's Line Algorithm

There are other approaches to deriving the Bresenham Line Algorithm. The parts are the same, but some details are presented differently.

The course text makes the decision to move up in y based on the distance between the *true* line and the nearest pixel.

- Hearn & Baker, *Computer Graphics with OpenGL*, 4th Edition, Chapter 5

Midpoint Line Algorithm

Midpoint is a variation of Bresenham's Line Algorithm.

Same improvement goals:

- remove floating point operations
- minimise the number of operations

Midpoint Line Algorithm

The midpoint algorithm uses 8 compass points to describe the *next* pixel to draw:

- E, NE, N, NW, W, SW, S, SE

Midpoint Line Algorithm

We will describe the algorithm just for the *upper right octant*.

- The only possible next directions are E and NE.

Midpoint Line Algorithm

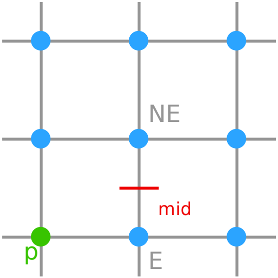

Figure 2: midpoint pixel directions

For a **previous** pixel p in the upper right octant, we label the two *candidate* pixels E and NE. We will define criteria based on the midpoint between the two candidates.

Midpoint Line Algorithm

The algorithm decides if a **true** line passes either above, below or through the midpoint.

Midpoint Line Algorithm

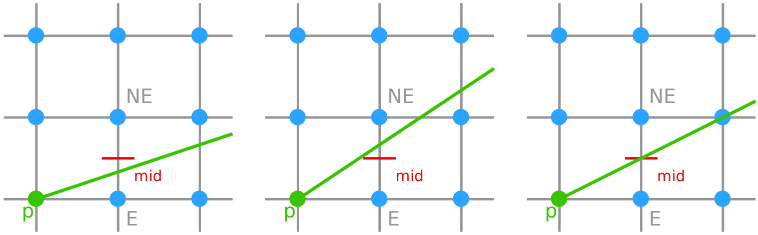

Figure 3: Three possible cases

Midpoint Line Algorithm

IF the true line is below or on the midpoint: pick the E pixel.

ELSE: pick the NE pixel.

Midpoint Line Algorithm

We will use the *implicit* line equation:

$$ax + by + c = 0$$

We know that:

$$a = \Delta y \ , b = -\Delta x \ \Rightarrow f(x, y) = x\Delta y - y\Delta x + c = 0$$

N.B. henceforth we will assume $c = 0$, and remove from the derivations.

Decision Variable

IF the line goes exactly through the midpoint then we have the *decision* variable:

$$\begin{aligned} D &= f(x_p + 1, y_p + \tfrac{1}{2}) \\ &= a_{(m)}(x_p + 1) + b_{(m)}(y_p + \tfrac{1}{2}) \\ &= 0 \end{aligned}$$

recall, in the upper right octant: $a > 0,\ b < 0$

Decision Variable

IF the line goes *below* the midpoint:

$$a < a_{(m)} \wedge b > b_{(m)} \Rightarrow D < 0 \Rightarrow E$$

The actual value of $D(E)$ is:

$$\begin{aligned} D(E) &= f(x_p + 1, y_p) \\ &= a(x_p + 1) + by_p \\ &= ax_p + a + by_p \\ &= f(x_p, y_p) + a \end{aligned}$$

Decision Variable

ELSE the line goes *above* the midpoint:

$$a > a_{(m)} \wedge b < b_{(m)} \Rightarrow D > 0 \Rightarrow NE$$

The actual value of $D(NE)$ is:

$$\begin{aligned} D(NE) &= f(x_p + 1, y_p + 1) \\ &= a(x_p + 1) + b(y_p + 1) \\ &= ax_p + a + by_p + b \\ &= f(x_p, y_p) + a + b \end{aligned}$$

Decision Variable

To avoid having to recalculate actual decision variable values each time we move one pixel in x, we can derive a decision variable *increment* instead.

We do this by looking ahead to the **next** pixel.

Decision Variable Increment

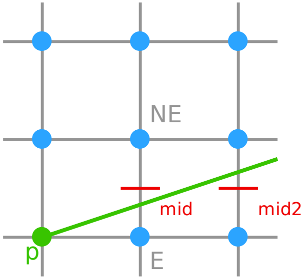

Figure 4: chosen E pixel

If we have chosen the E pixel then the next midpoint will be at:

$$
\begin{aligned}
D_{mE} &= f(x_p + 2, y_p + \tfrac{1}{2}) \\
&= a(x_p + 2) + b(y_p + \tfrac{1}{2})
\end{aligned}
$$

Subtracting the original D gives:

$$
\begin{aligned}
\Delta E &= D_{mE} - D \\
&= a \\
&= \Delta y
\end{aligned}
$$

Decision Variable Increment

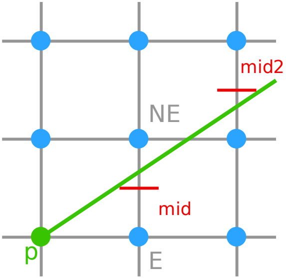

Figure 5: chosen NE pixel

If we have chosen the NE pixel then the next midpoint will be at:

$$
\begin{aligned}
D_{mNE} &= f(x_p + 2, y_p + \tfrac{3}{2}) \\
&= a(x_p + 2) + b(y_p + \tfrac{3}{2})
\end{aligned}
$$

Subtracting the original D gives:

$$
\begin{aligned}
\Delta NE &= D_{mNE} - D \\
&= a + b \\
&= \Delta y - \Delta x
\end{aligned}
$$

Initial Decision Variable

If the decision variable relies on the previous pixel, what is the decision variable for the first pixel?

Initial Decision Variable

Since the start point is on the line:

$$f(x_0, y_0) = 0$$

Substituting into the decision variable gives:

$$\begin{aligned}
D_{init} &= f(x_0 + 1, y_0 + \tfrac{1}{2}) \\
&= a(x_0 + 1) + b(y_0 + \tfrac{1}{2}) \\
&= ax_0 + by_0 + a + \tfrac{1}{2}b \\
&= f(x_0, y_0) + a + \tfrac{1}{2}b
\end{aligned}$$

Initial Decision Variable

This yields:

$$D_{init} = a + \frac{b}{2}$$

We want to remove floating point arithmetic, so we can multiply by 2, however, we must *also* do this to the decision variable *increments*.

Initial Decision Variable

Finally, our initial decision variable is:

$$D_{init} = 2a + b = 2\Delta y - \Delta x$$

and the decision variable increments are:

$$\Delta E = 2\Delta y,\ \Delta NE = 2(\Delta y - \Delta x)$$

```
void lineMid(int x0, int y0, int xEnd, int yEnd){
    int dx=xEnd-x0, dy=yEnd-y0, x=x0, y=y0;
    int E_inc = 2*dy, NE_inc = 2*(dy-dx), D = 2*dy-dx;

    setPixel(x,y);
    while(x<xEnd){
        if (D > 0){
            D += NE_inc;
            x++; y++;
        } else {
            D += E_inc;
            x++;
        }
    setPixel(x,y);
}  }
```

Aliasing

Aliasing is a distortion artifact when representing a high-resolution image at a lower resolution.

- stair steps
- jagged edges

Anti-Aliasing

To mitigate aliasing, we can use a technique called *anti-aliasing*.

We will consider a few possible approaches.

Anti-Aliasing

The first approach is to consider a higher resolution display.

– This has happened naturally, as hardware has improved.
– We can consider this a “brute force” approach.

Anti-Aliasing

We can render an artificially thick line.

- Reduce colour intensity as we move away from the true line.

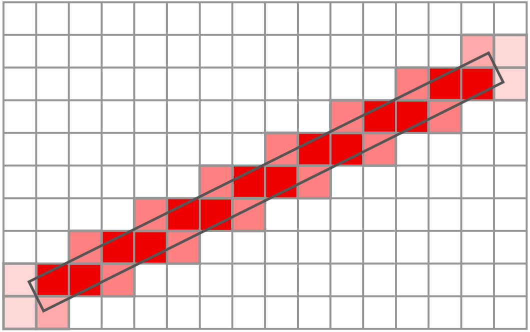

Figure 6: anti-aliased line

Anti-Aliasing

We can render to a sub-pixel grid.

- then use sampling to get the colour at the pixel.

Anti-Aliasing

We can filter the image.

- usually some *low-pass* filter
- e.g. box or gaussian filter
- filtering is performed using **convolution** with a *kernel*
- often combined with sub-pixel sampling.

Summary

- Bresenham's Line Algorithm
- Midpoint Line Algorithm
- Antialiasing

Reading:

- Hearn & Baker, *Computer Graphics with OpenGL*, 4th Edition, Chapter 5
- Bresenham, J. E. (1965) "Algorithm for computer control of a digital plotter"