Line Rasterization

DDA algorithm for lines

Parametric Lines: the DDA algorithm
(digital differential analyzer)

$$Y_{i+1} = m\,x_{i+1} + B$$

$$= m(x_i + \Delta x) + B \qquad \Delta x = (x_{i+1} - x_i)$$

$$= y_i + m(\Delta x)$$ **<- must round to find int**

If we increment by 1 pixel in X, we turn on
[xi, Round(yi)] or same for Y if m > 1

Scan conversion for lines

DDA includes Round( ); and this is fairly slow

For Fast Lines, we want to do only integer math +,-

We do this using the **Midpoint Algorithm**

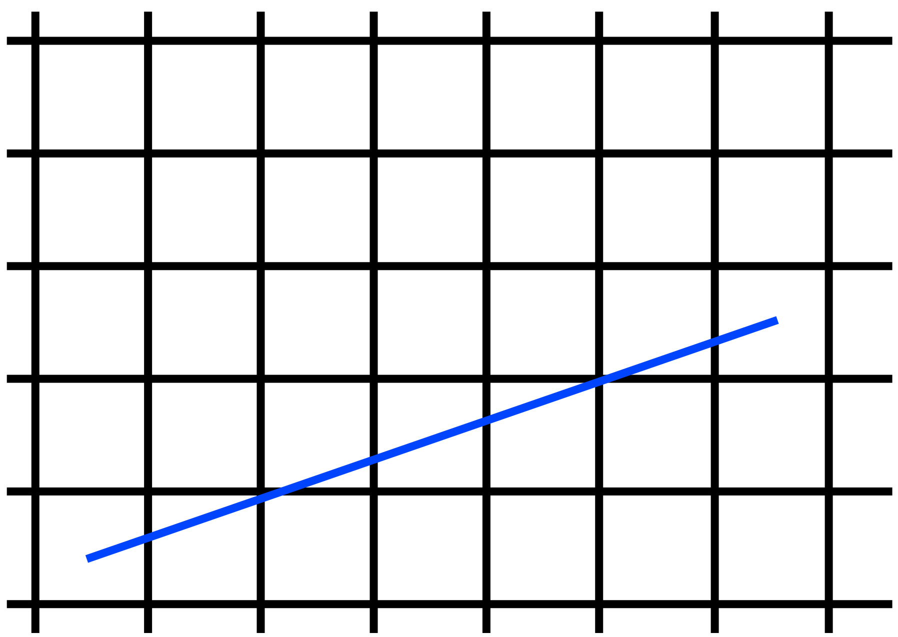
To do this, lets look at lines with y-intercept B
and with slope between 0 and 1:

$$y = (dy/dx)x + B \quad \Longrightarrow$$

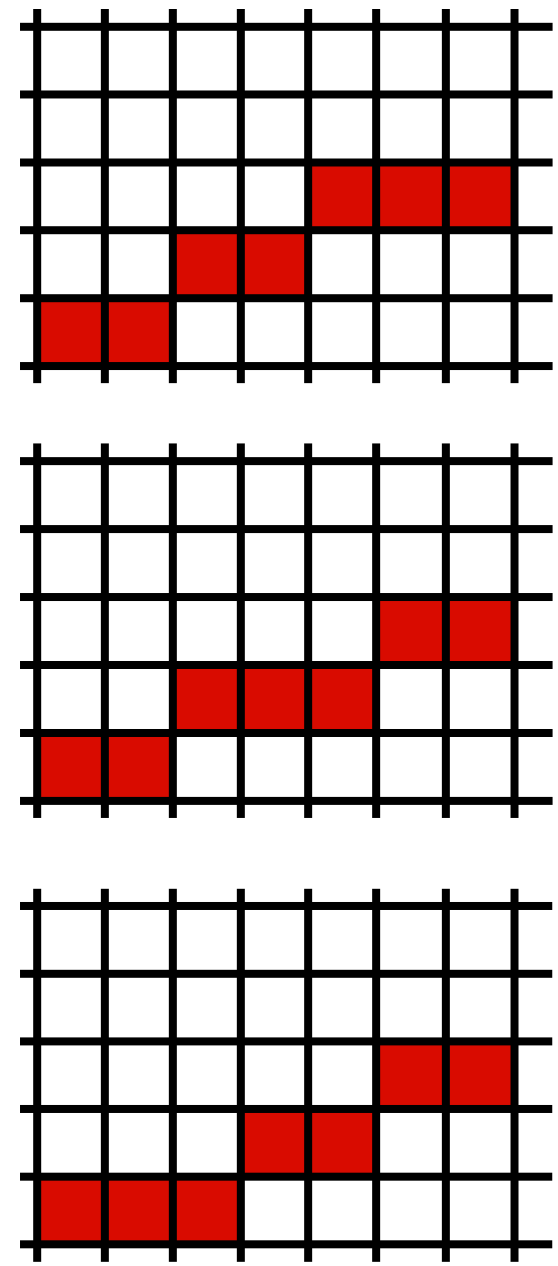
$$f(x,y) = (dy)x - (dx)y + B(dx) = 0$$

Removes the division => slope treated as 2 integers

Which pixels should be used to approximate a line?



Draw the thinnest possible line that has no gaps



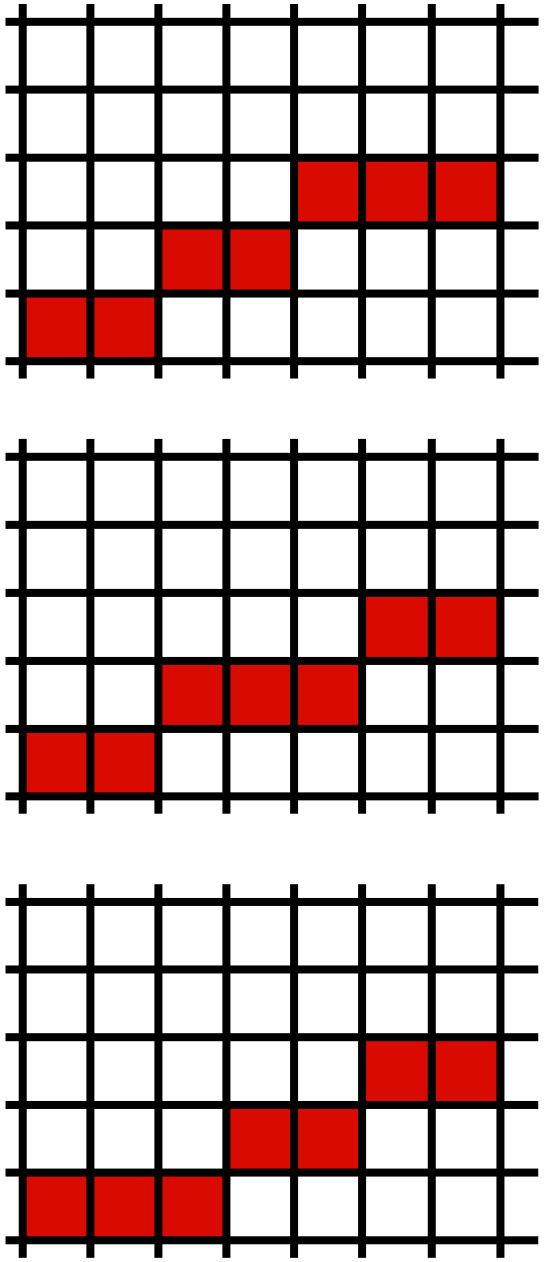
Line drawing algorithm

(case: $0 < m <= 1$)

```
y = y0
for x = x0 to x1 do
    draw(x,y)
    if (<condition>) then
        y = y+1
```

- move from left to right
- choose between

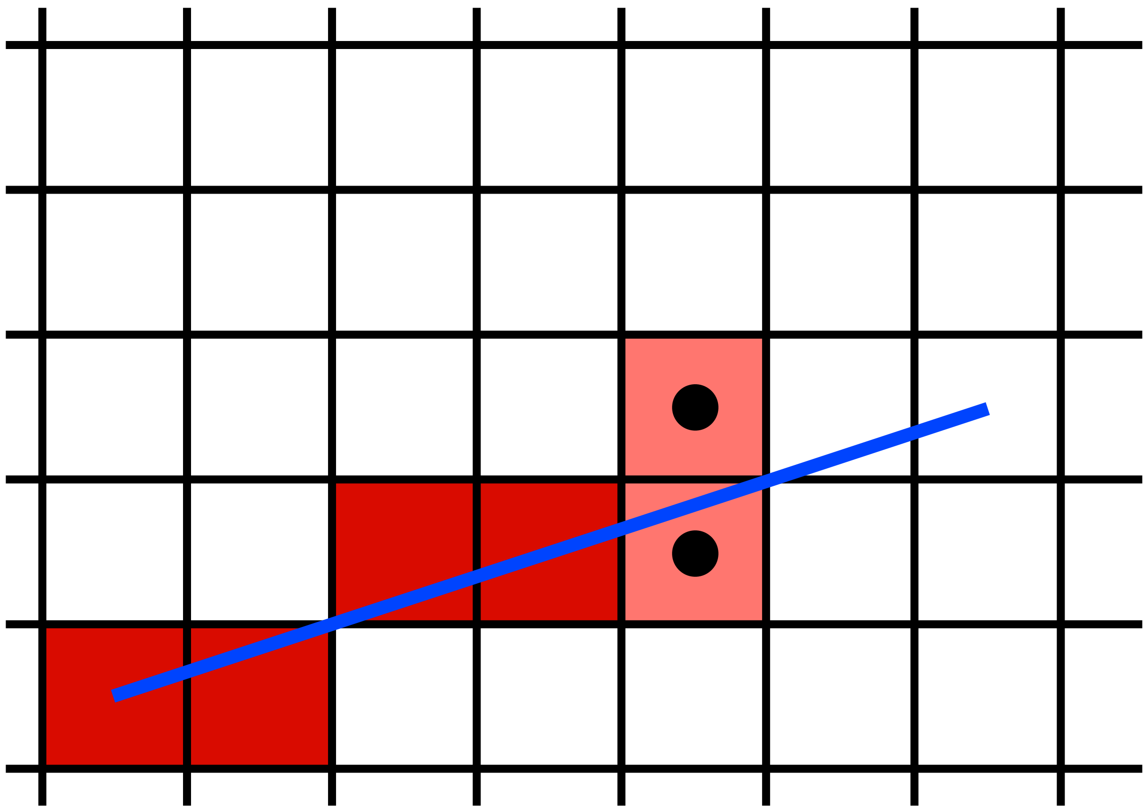
$(x+1,y)$ and $(x+1,y+1)$



Line drawing algorithm

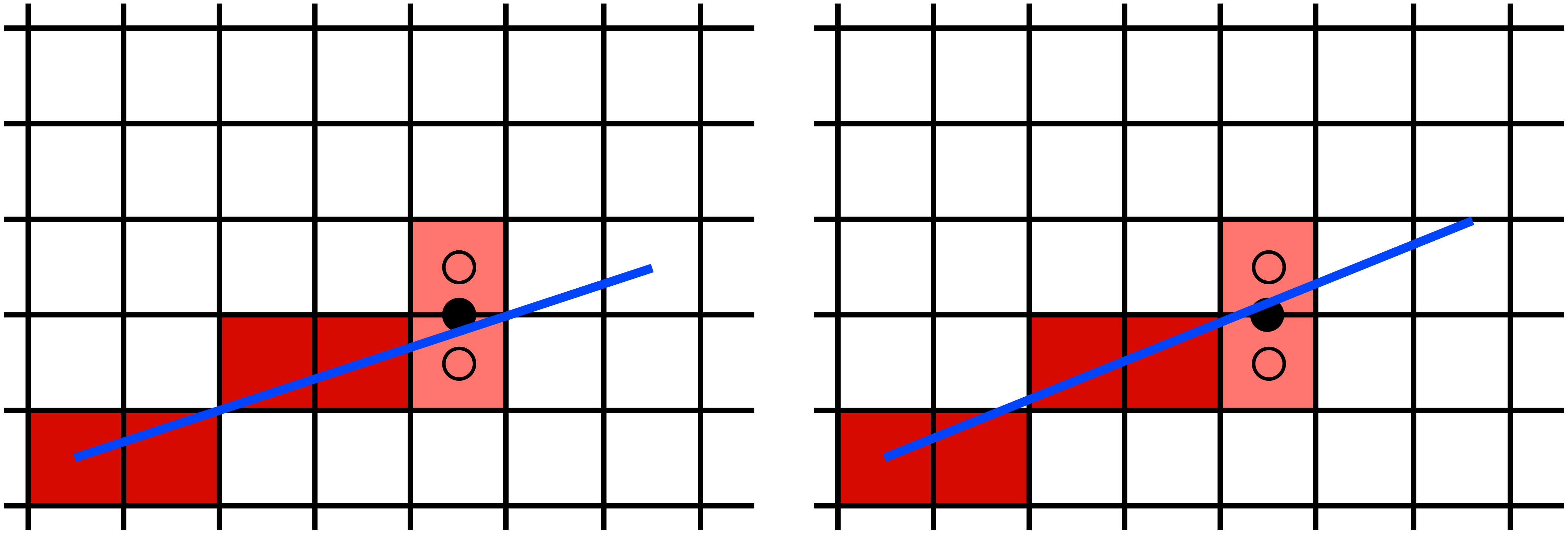
(case: 0 < m <= 1)

```
y = y0
for x = x0 to x1 do
    draw(x,y)
    if (<condition>) then
        y = y+1
```

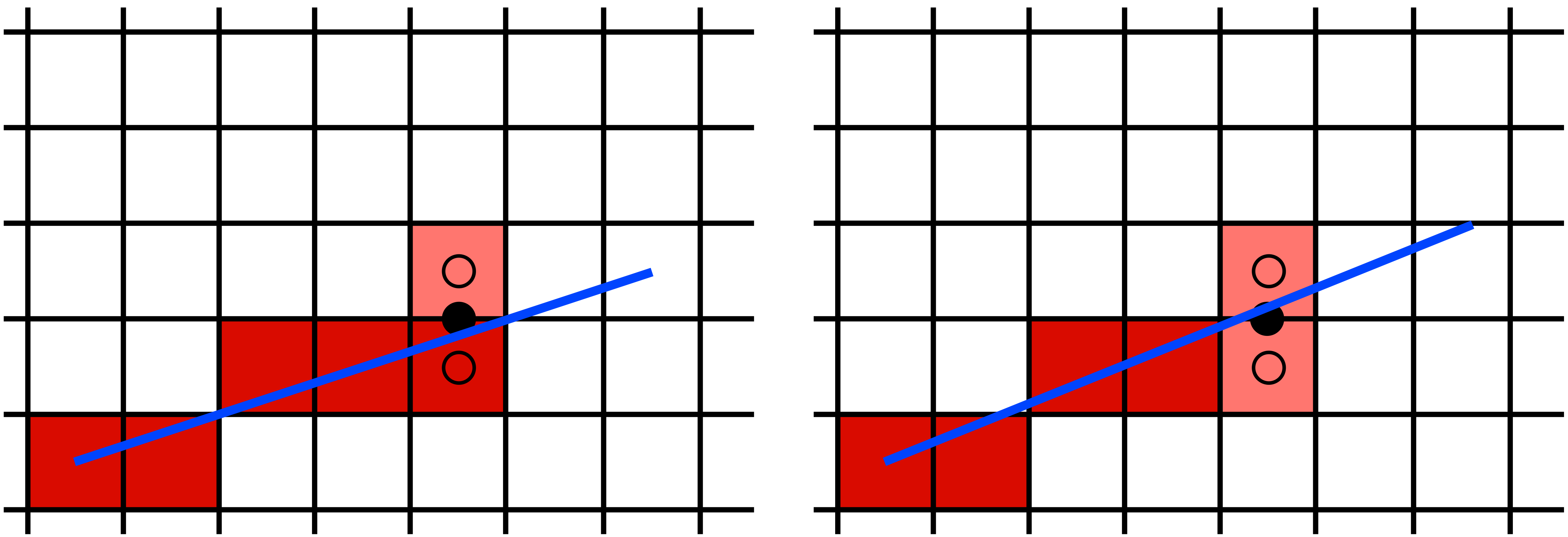


- move from left to right
- choose between (x+1,y) and (x+1,y+1)

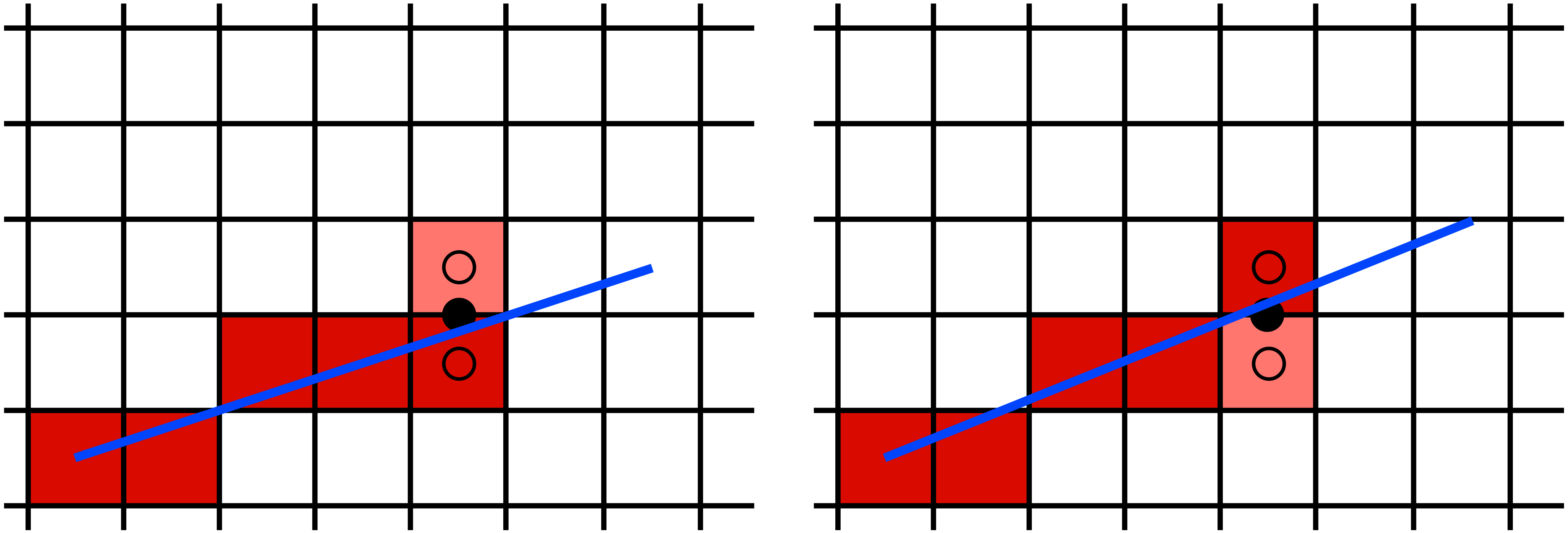
Use the midpoint between the two pixels to choose



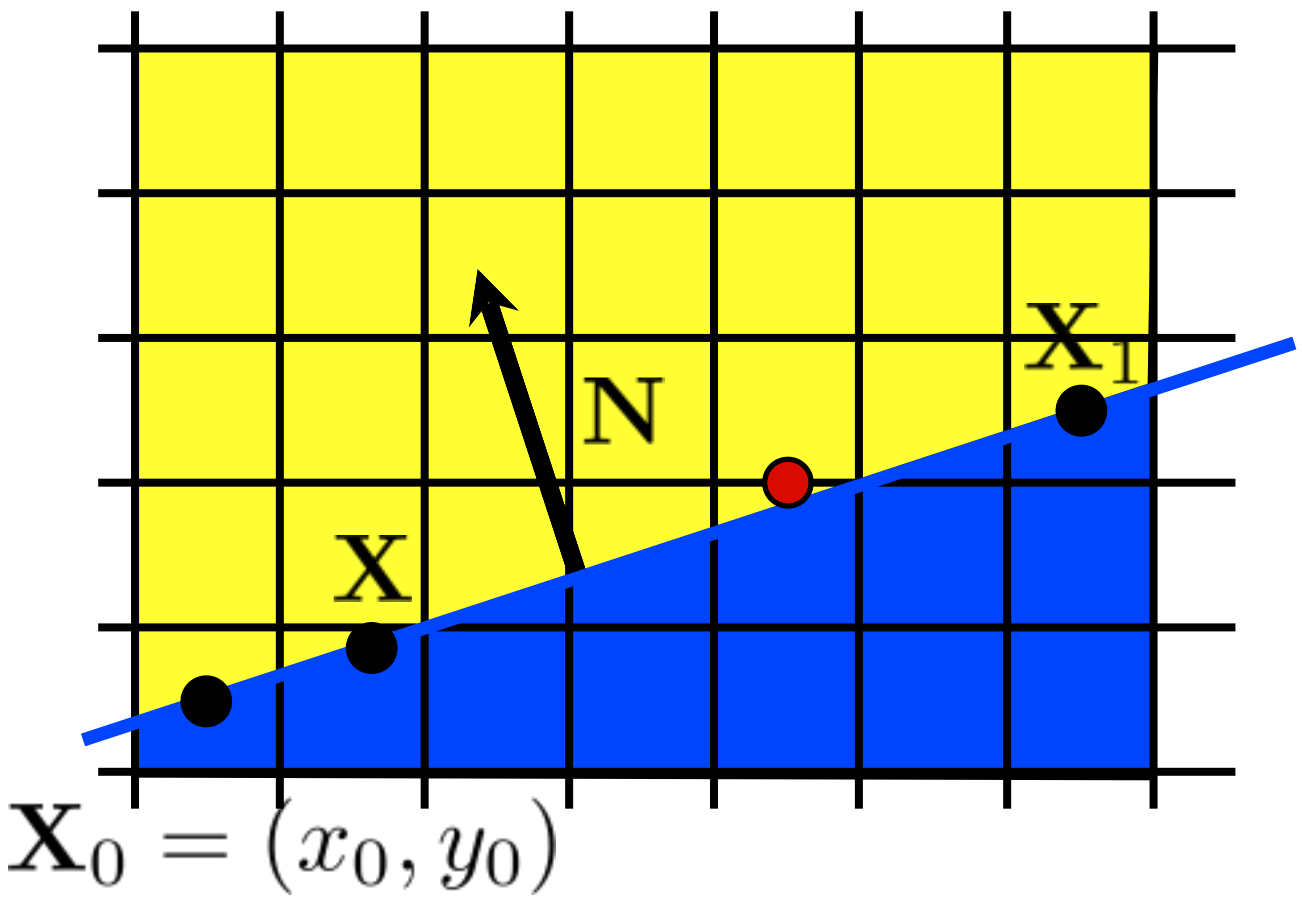
Use the midpoint between the two pixels to choose



Use the midpoint between the two pixels to choose



Use the midpoint between the two pixels to choose



implicit line equation:

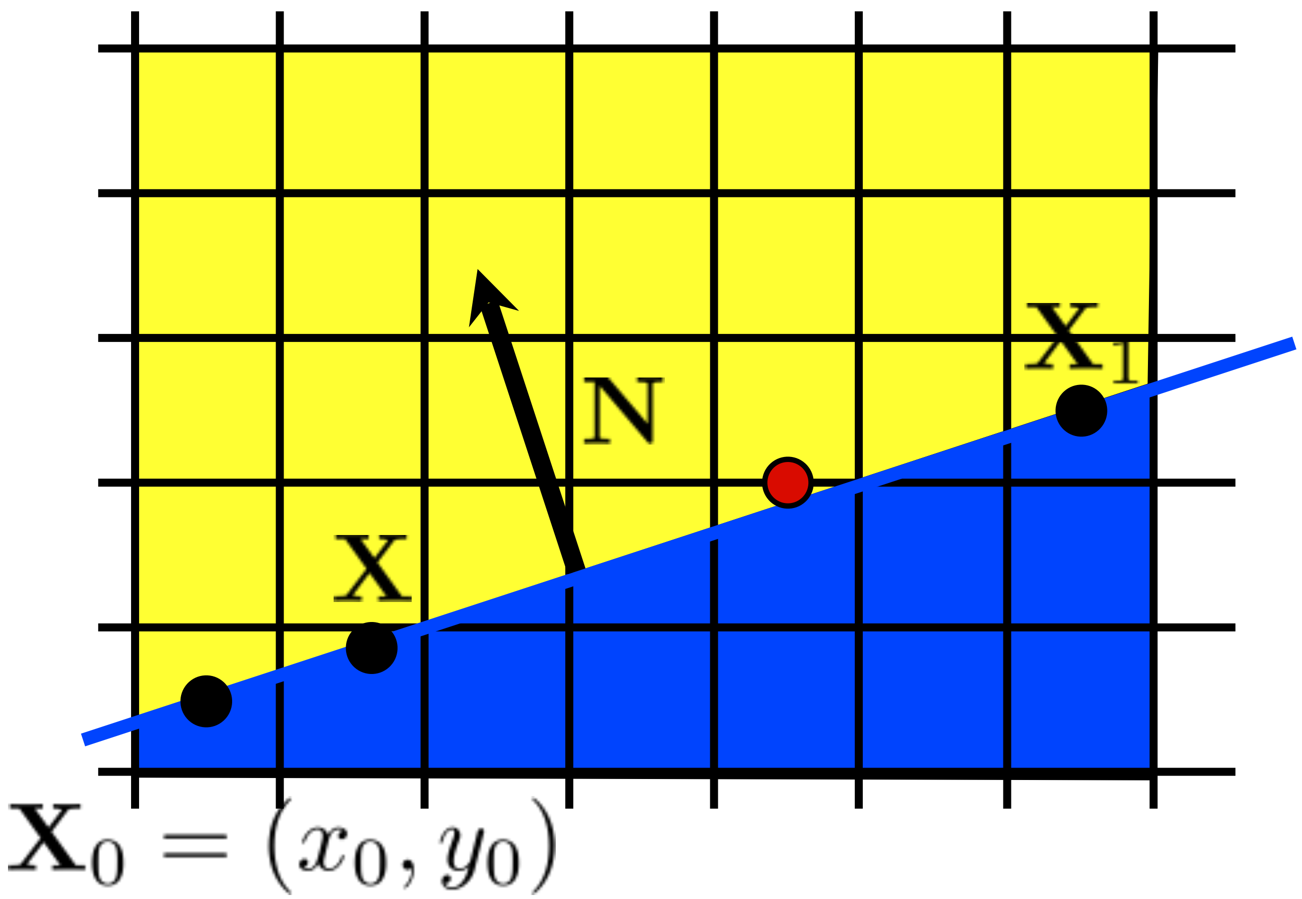
$$f(\mathbf{X}) = \mathbf{N} \cdot (\mathbf{X} - \mathbf{X}_0) = 0$$

<whiteboard>

evaluate f at midpoint:

$$f(x, y + \frac{1}{2}) \ ? \ 0$$

Use the midpoint between the two pixels to choose



implicit line equation:

$$f(\mathbf{X}) = \mathbf{N} \cdot (\mathbf{X} - \mathbf{X}_0) = 0$$

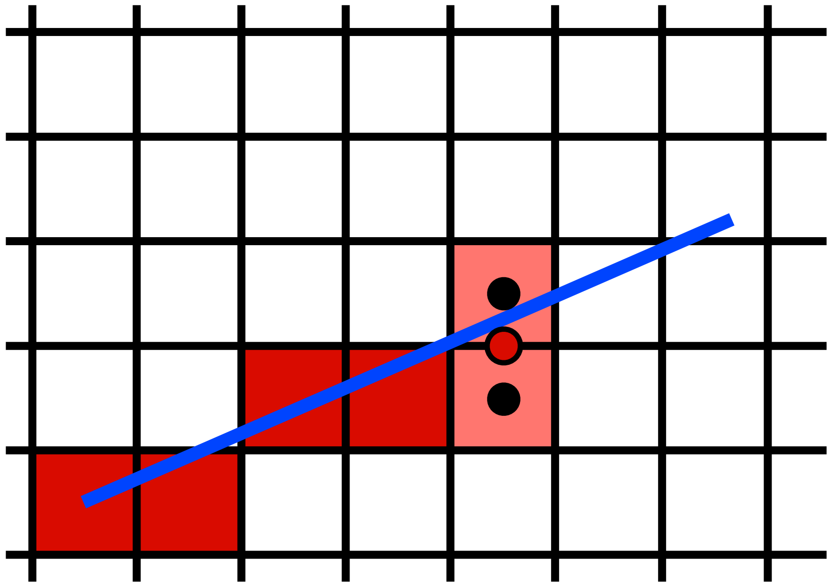
evaluate f at midpoint:

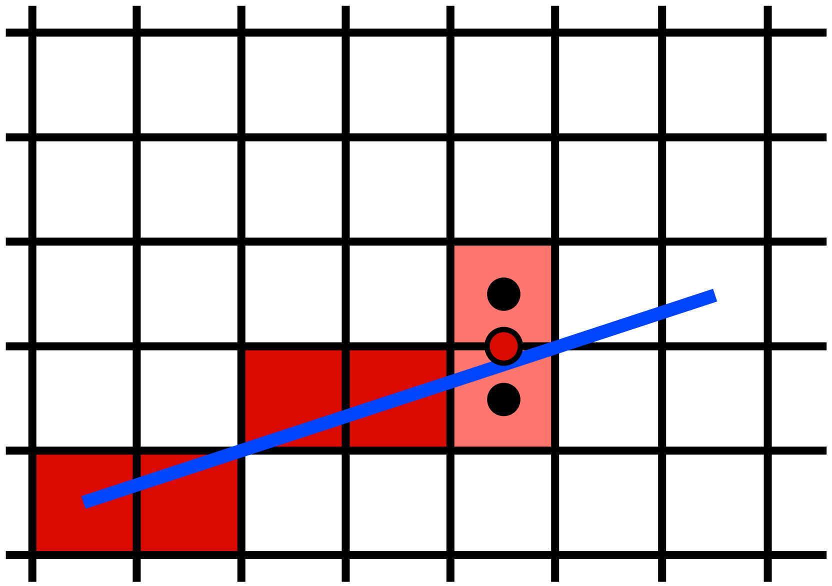
$$f(x, y + \frac{1}{2}) \; > \; 0$$

Line drawing algorithm

(case: 0 < m <= 1)

```
y = y0
for x = x0 to x1 do
    draw(x,y)
    if ( f(x+1, y+1/2) < 0) then
        y = y+1
```

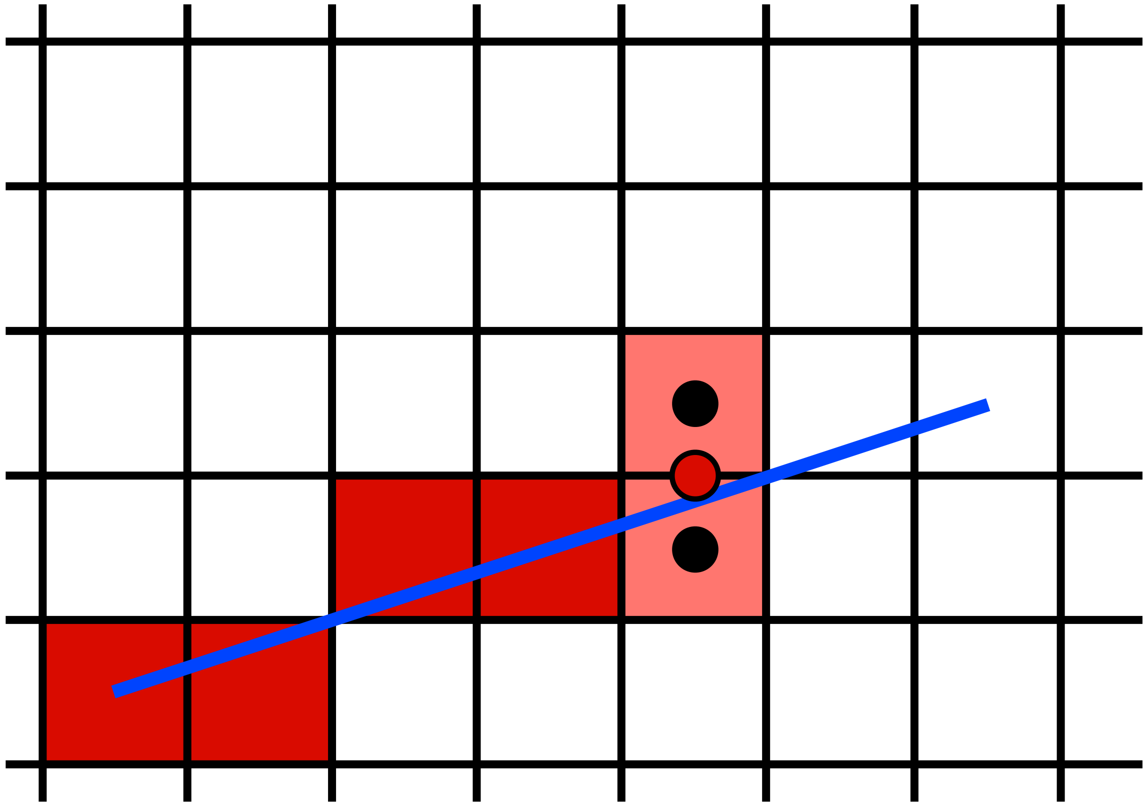




We can make the Midpoint Algorithm more efficient

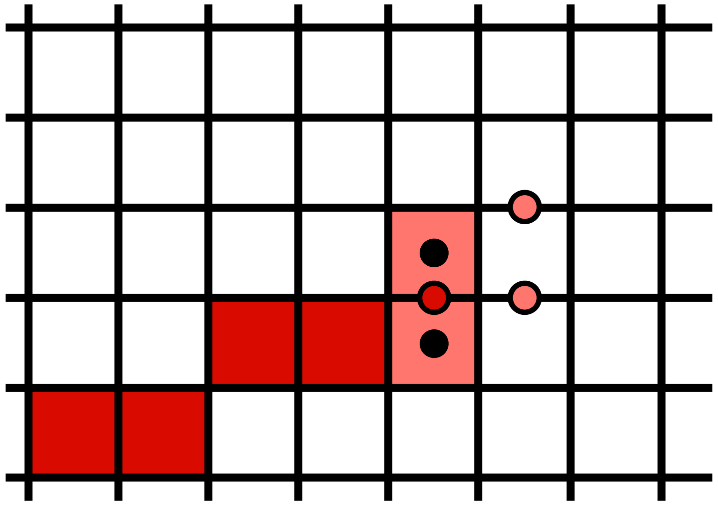
```
y = y0
for x = x0 to x1 do
    draw(x,y)
    if ( f(x+1, y+1/2) < 0 ) then
        y = y+1
```

$f(x+1, y+\frac{1}{2}) < 0$



We can make the Midpoint Algorithm more efficient

by making it incremental!



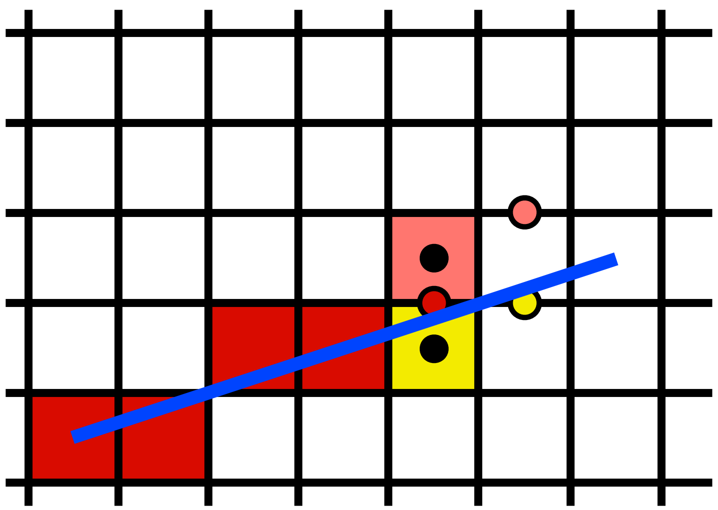
$$f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0$$

$$f(x+1, y) = f(x, y) + (y_0 - y_1)$$

$$f(x+1, y+1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0)$$

We can make the Midpoint Algorithm more efficient

$$f(x+1, y+\frac{1}{2}) > 0$$



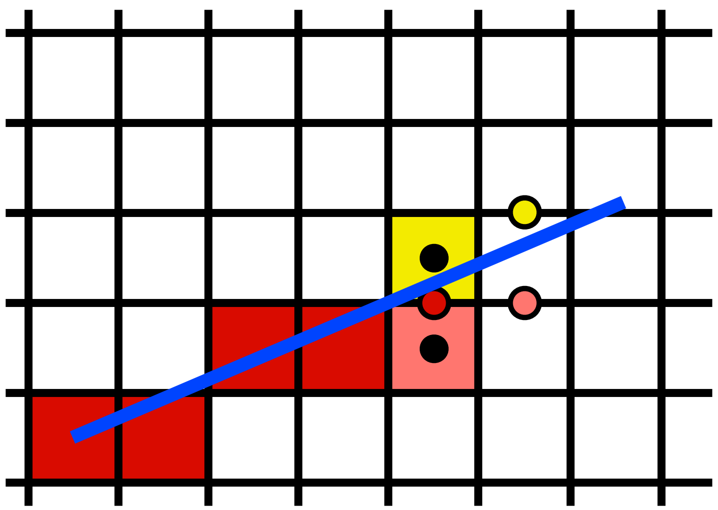
$$f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0 = 0$$

$$f(x+1, y) = f(x,y) + (y_0 - y_1)$$

$$f(x+1, y+1) = f(x,y) + (y_0 - y_1) + (x_1 - x_0)$$

We can make the Midpoint Algorithm more efficient

$$f(x+1, y+\frac{1}{2}) < 0$$



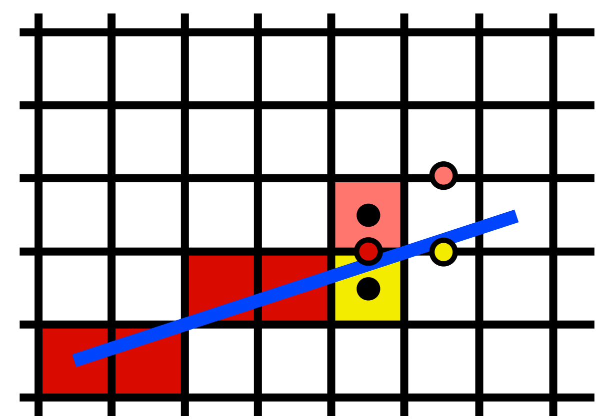
$$f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0$$

$$f(x+1, y) = f(x,y) + (y_0 - y_1)$$

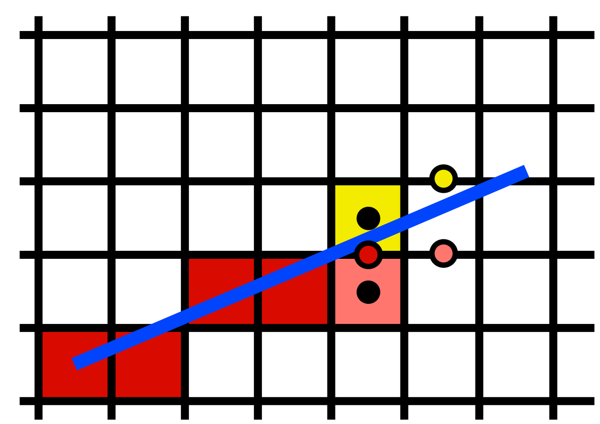
$$f(x+1, y+1) = f(x,y) + (y_0 - y_1) + (x_1 - x_0)$$

We can make the Midpoint Algorithm more efficient

```
y = y0
d = f(x0+1,y0+1/2)
for x = x0 to x1 do
    draw(x,y)
    if (d<0) then
        y = y+1
        d = d+(y0-y1)+(x1-x0)
    else
        d = d+(y0-y1)
```

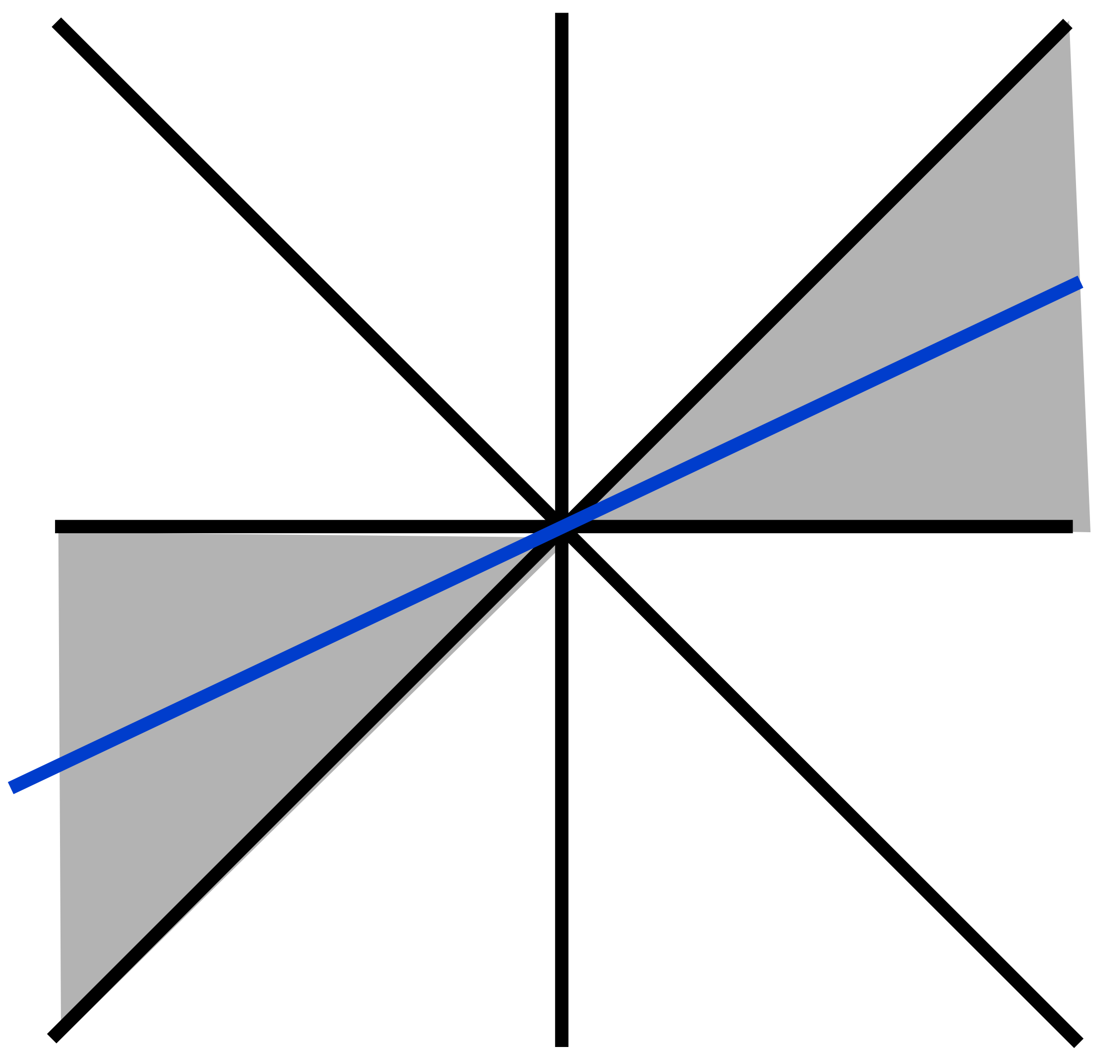


$$f(x+1,y) = f(x,y) + (y_0 - y_1)$$

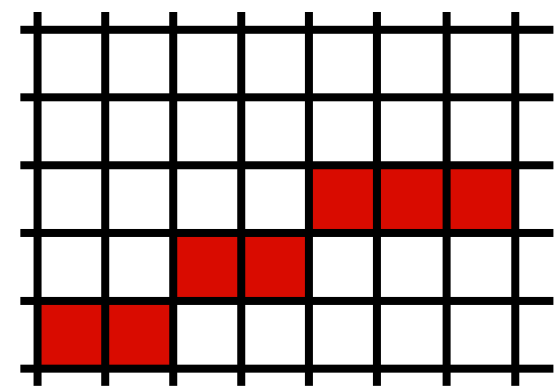


$$f(x+1,y+1) = f(x,y) + (y_0 - y_1) + (x_1 - x_0)$$

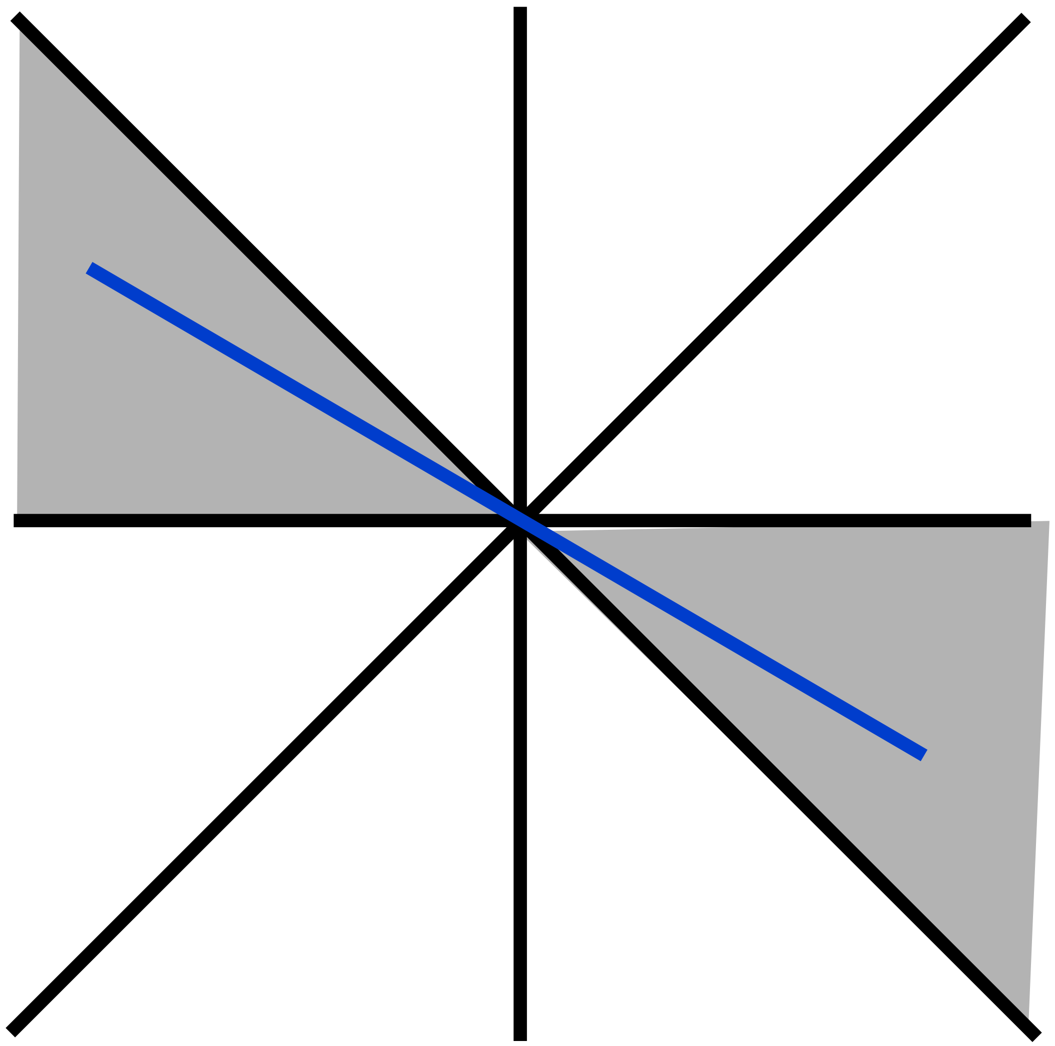
Adapt Midpoint Algorithm for other cases



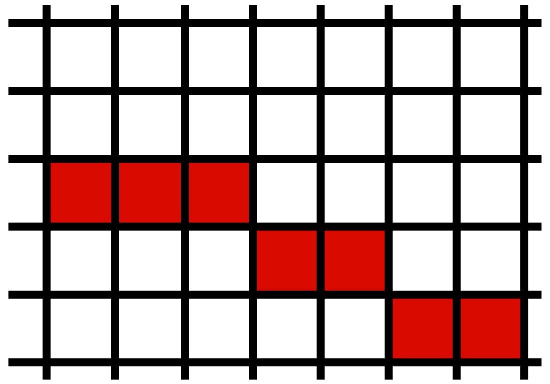
case: 0 < m <= 1



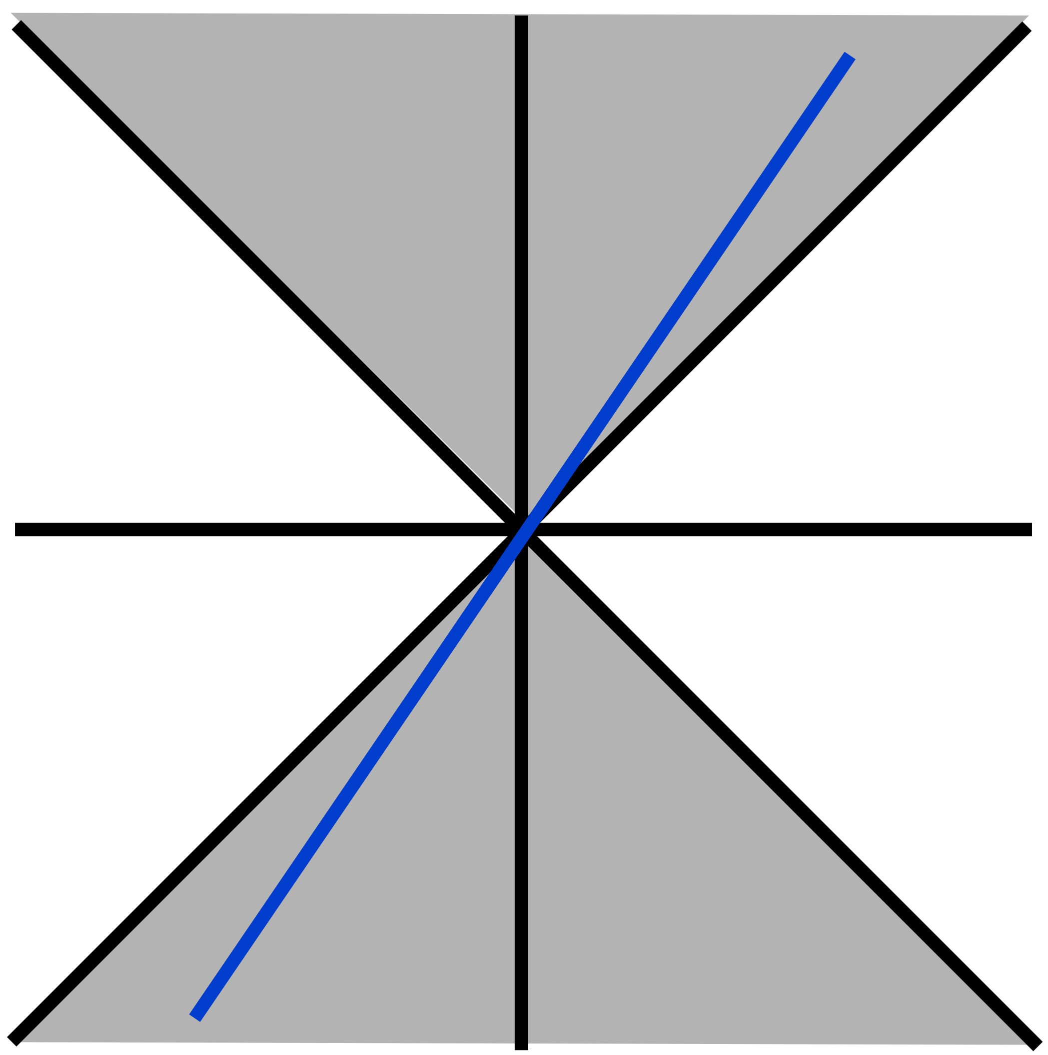
Adapt Midpoint Algorithm for other cases



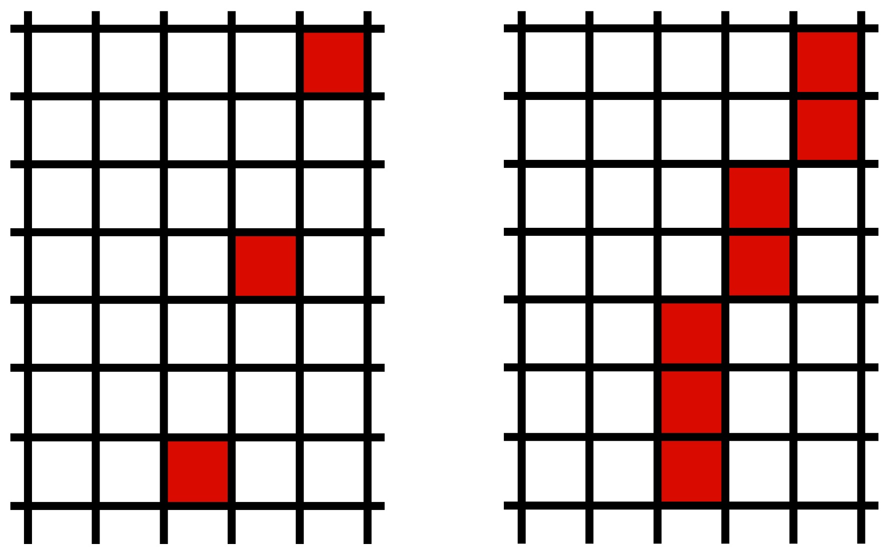
case: -1 <= m < 0



Adapt Midpoint Algorithm for other cases



case: 1 <= m
or m <= -1



Line drawing references

- The algorithm we just described is the *Midpoint Algorithm* (Pitteway, 1967), (van Aken and Novak, 1985)
  - Handles floating point coordinates
- Draws the same lines as the *Bresenham Line Algorithm* (Bresenham, 1965)
  - Simpler, cheaper
  - Integer coordinates only