Line Rasterization
DDA algorithm for lines

Parametric Lines: the DDA algorithm (digital differential analyzer)

\[ Y_{i+1} = m \cdot x_{i+1} + B \]

\[ = m(x_i + \Delta x) + B \quad \Delta x = (x_{i+1} - x_i) \]

\[ = y_i + m(\Delta x) \quad <- \text{must round to find int} \]

If we increment by 1 pixel in X, we turn on \([x_i, \text{Round}(y_i)]\) 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, let's look at lines with y-intercept B and with slope between 0 and 1:

\[ y = \frac{dy}{dx}x + B \quad \Rightarrow \quad f(x,y) = (dy)x - (dx)y + B(dx) = 0 \]

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)

\[ y = y_0 \]

for \( x = x_0 \) to \( x_1 \) do

\[ \text{draw}(x, y) \]

if \( (<\text{condition}>)_x \) then

\[ y = y+1 \]

• move from left to right
• choose between \((x+1, y)\) and \((x+1, y+1)\)
Line drawing algorithm
(case: $0 < m \leq 1$)

\[ y = y_0 \]
for \( x = x_0 \) to \( x_1 \) do
  draw(\( x, y \))
  if (\(<\text{condition}>\)) 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:

\[ f(X) = N \cdot (X - X_0) = 0 \]

Evaluate \( f \) at midpoint:

\[ f(x, y + \frac{1}{2}) \neq 0 \]
Use the midpoint between the two pixels to choose

Implicit line equation:
\[ f(X) = N \cdot (X - X_0) = 0 \]

Evaluate \( f \) at midpoint:
\[ f(x, y + \frac{1}{2}) > 0 \]

\( X_0 = (x_0, y_0) \)
Line drawing algorithm

(case: $0 < m \leq 1$)

$y = y_0$
for $x = x_0$ to $x_1$ do
  draw($x, y$)
  if ($f(x + 1, y + \frac{1}{2}) < 0$) then
    $y = y + 1$
We can make the Midpoint Algorithm more efficient

\[
y = y_0 \\
\text{for } x = x_0 \text{ to } x_1 \text{ do} \\
\quad \text{draw}(x,y) \\
\quad \text{if } (f(x + 1, y + \frac{1}{2}) < 0) \text{ then} \\
\quad \quad y = y + 1
\]
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

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

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

\[ f(x + 1, y) = f(x, y) + (y_0 - y_1) \]

\[ f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \]
We can make the Midpoint Algorithm more efficient

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

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

\[ f(x + 1, y) = f(x, y) + (y_0 - y_1) \]

\[ f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \]
We can make the Midpoint Algorithm more efficient

\[ y = y_0 \]
\[ d = f(x_0 + 1, y_0 + 1/2) \]
for \( x = x_0 \) to \( x_1 \) do
  \[ \text{draw}(x, y) \]
  if \( (d < 0) \) then
    \[ y = y + 1 \]
    \[ d = d + (y_0 - y_1) + (x_1 - x_0) \]
  else
    \[ d = d + (y_0 - y_1) \]

\[ f(x + 1, y) = f(x, y) + (y_0 - y_1) \]

\[ f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \]
Adapt Midpoint Algorithm for other cases

case: $0 < m \leq 1$
Adapt Midpoint Algorithm for other cases

case: $-1 \leq m < 0$
Adapt Midpoint Algorithm for other cases

case: $l \leq m$
or $m \leq -l$
Line drawing references

- The algorithm we just described is the *Midpoint Algorithm* (Pitweway, 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