Scan Conversion Algorithms for Drawing Straight Lines

- Task
  - Given pixel coordinates of endpoints
    P1 (x1,y1) & P2 (x2,y2)
  - Determine which pixels need to be painted
- Criteria
  - Straight as possible
  - Constant density (no gaps or bunching)
  - Density independent of orientation
  - Must be fast

Line Equations

- Differential equation:
  \( \frac{dy}{dx} = m \) (m=constant: the slope)
- Integrate (indefinite)
  \( y = mx + \text{constant} \)
  The constant (b) is called y intercept
  (value of y when x=0)
- \( y = mx + b \)
- "slope-intercept" form

Parametric Form

Express x and y linearly in terms of a parameter, t
\[
\begin{align*}
x &= ax't + bx \\
y &= ay't + by
\end{align*}
\]
a, b, ay, by are constants to be determined
Let \( t \) range between endpoints 0 (x1,y1) and 1 (x2,y2)
Determining the constants: Use endpoint values
\[
\begin{align*}
x1 &= ax'0 + bx \\
y1 &= ay'0 + by \\
x2 &= ax'1 + bx \\
y2 &= ay'1 + by
\end{align*}
\]
So \( x = \frac{x2-x1}{t} + x1 \)
\( 0 < t < 1 \)
And \( y = \frac{y2-y1}{t} + y1 \)
**Brute Force Line-Drawing Algorithm**

*Step in x direction* \((x_2 > x_1)\)

- Compute \(m = \frac{y_2 - y_1}{x_2 - x_1}\)
- \(\text{num-pts} = x_2 - x_1 + 1\)
- \(x = x_1\)
- Repeat \(\text{num-pts}\) times
  - \(y = m(x - x_1) + y_1\)
  - SetPixel(round(\(x\)), round(\(y\)))
  - \(x = x + 1\)

**Problem if \(|y_2 - y_1| > |x_2 - x_1| \Rightarrow \text{gaps}\)**

*Solution: Step in y direction*

If \(|y_2 - y_1| > |x_2 - x_1|\), *step in y* (assume \(y_2 > y_1\)):

- Compute \(\text{inv}\_m = \frac{x_2 - x_1}{y_2 - y_1}\)
- \(\text{num-pts} = y_2 - y_1 + 1\)
- \(y = y_1\)
- Repeat \(\text{num-pts}\) times
  - \(x = \text{inv}\_m(y - y_1) + x_1\)
  - SetPixel(round(\(x\)), round(\(y\)))
  - \(y = y + 1\)

**Solution: Step in y direction**

**Brute Force line algorithm, continued**

Vertical lines \((x_2 = x_1)\)

Horizontal lines \((y_2 = y_1)\)

- Handle as separate cases

**Brute Force Method is Too Slow**

- Each iteration has:
  - floating point multiply
  - floating point add
  - round() operations

**Incremental Methods--The Digital Differential Analyzer (DDA)**

*Idea: get new point from previous point*

- \(\frac{\Delta y}{\Delta x} = m\) \(\Rightarrow\) \(\frac{\Delta y}{\Delta x} \cdot \Delta x = m\) \(\Rightarrow\) \(\Delta y = m\cdot \Delta x\)

- But \(\frac{\Delta y}{\Delta x} = \frac{\Delta y_{\text{new}} - \Delta y_{\text{old}}}{\Delta x_{\text{new}} - \Delta x_{\text{old}}}\)
- And \(\Delta x = x_{\text{new}} - x_{\text{old}}\)
- So \(x_{\text{new}} = x_{\text{old}} + \Delta x\)
- and \(y_{\text{new}} = y_{\text{old}} + \Delta y\)
- i.e., \(y_{\text{new}} = y_{\text{old}} + m\cdot \Delta x\)
DDA, continued
- Can choose ? x = 1
  - stepping in x direction
- Then compute each new y value
  ynew = yold + m

DAA Algorithm (stepping in x, x2>x1)
Compute m = (y2-y1)/(x2-x1)
num-pts = x2-x1+1
x = x1
y = y1
Repeat num-pts times
  SetPixel(x,round(y))
x = x+1
y = y+m

As for the Brute force method, if |m|>1, we can step in y
DAA Algorithm (stepping in y, y2>y1):
  Compute inv_m = (x2-x1)/(y2-y1)
  num-pts = y2-y1+1
  x = x1
  y = y1
  Repeat num-pts times
    SetPixel(round(x),y)
    y = y+1
    x = x+inv_m

DDA is Better, but Still Not Fast Enough
- Floating point multiply gone from loop
- But loop still has a floating point add
- And a round()
- WE CAN DO BETTER!
- Best performance:
  - Only integer adds/subtracts inside loop

Bresenham's Line-drawing Algorithm
- Used in most graphics packages
- Often implemented in hardware
- Also incremental (new pixel from old)
- Uses only integer operations

Basic Idea of Bresenham Algorithm:
- All lines can be placed in one of four categories:
  A. Steep positive slope (m>1)
  B. Gradual positive slope (0<m<=1)
  C. Steep negative slope (m<-1)
  D. Gradual negative slope (0>=m>=-1)
- In each case, there are only 2 choices for the next pixel to be plotted!
The Four Bresenham Cases

We need a "Predictor" function to choose which pixel is next.
Should use only simple integer math.
Look at Case-A (Steep positive slope).
Also assume P1 is to the left of P2 (x1<x2).
– If not true, points can be swapped.
delta_y > delta_x ==> stepping in y

Assume last pixel plotted (r,s).
– Next pixel must be at (r,s+1) "left" pixel.
– or at (r+1,s+1) "right" pixel.
Mathematical point at (r', s+1).
Define two horizontal distances:
dl=distance from math. Pt. to "left" pixel.
dr=distance from math. Pt. and "right" pixel.
dl = r'-r
dr = r+1-r'

If dl<dr,
– Pt is closer to actual point than Pr.
I.e., if dl-dr<0, choose "left" pixel.
Criterion for choosing Pt is:
dl-dr = r'-r - (r+1-r') < 0
or: 2*r' - 2*r -1 < 0

Equation for line:
y = m*x + b
define ? y=y2-y1, ?x=x2-x1
Mathematical point lies on line, so:
s+1 = (?y/?x)*r' + b
r' = (s+1-b)?x/?y
dl-dr = 2*r' - 2*r -1 < 0
So:
dl-dr = (2s+1)x + 2s*x - 2b*?x)/?y - 2*r -1 < 0
If dl-dr is negative, choose “left” pixel

But this “predictor” is too complex
Multiply by \( ?y \)
(always positive for Case-A lines)
Call result the “predictor”, \( P \)

\[
P = ?y^* (dl-dr) = 2^n * x_{(s+1-b)} - 2^n * ?y - ?y
\]

Divide is gone—but it's still too complex

---

**Bresenham’s Contribution**

We’re trying to get new point from old point, so:

- Maybe we can get a new predictor value from its old value in a simple way
- Need a recurrence relation for \( P \)
- Call \( P_n \) the new value, and \( P_o \) the old value
  - Then \( P_n = P_o + ?P \)
- Call \( s_n \) & \( s_o \) the new & old values of \( s \)
- Call \( r_n \) & \( r_o \) the new & old values of \( r \)

---

**Change in Predictor:**

\[
?P = P_n - P_o, \text{ so: } \\
P_n = P_o + ?P \\
\text{Two cases:} \\
\text{left case (} r_n=r_o \text{ and } s_n=s_o+1) \\
\text{right case (} r_n=r_o+1 \text{ and } s_n=s_o+1) \\
\text{For both cases:} \\
P_o = 2^n * x_{(s_o+1-b)} - 2^n * ?y - ?y
\]

---

**Left case:**

\[
P_n = 2^n * x_{((s_o+1)+1-b)} - 2^n * r_o * ?y - ?y \\
P_o = 2^n * x_{(s_o+1-b)} - 2^n * ?y - ?y \\
\text{Subtracting } P_o \text{ from } P_n \text{ gives } ?P \\
\text{Result:} \\
?P = 2^n * x
\]

**Right case:**

\[
P_n = 2^n * x_{((s_o+1)+1-b)} - 2^n * (r_o+1) * ?y - ?y \\
P_o = 2^n * x_{(s_o+1-b)} - 2^n * ?y - ?y \\
\text{Again subtracting } P_o \text{ from } P_n \text{ gives } ?P: \\
?P = 2^n (?x - ?y)
\]

---

- Both results are very simple (Integers!!)
- Look at current value of the predictor:
  
  If \( P < 0 \) // left case
  
  \[
P = P + 2^n * x \\
x = x \\
y = y + 1
\]
  
  If \( P > 0 \) // right case
  
  \[
P = P + 2^n (?x - ?y) \\
x = x + 1 \\
y = y + 1
\]