• Scan-converting lines (Section 4.5)
  ◦ A scan-conversion algorithm for lines computes the coordinates of the pixels that lie on or near an ideal, infinitely thin straight line
  ◦ For lines with \(-1 \leq \text{slope} \leq 1\), exactly 1 pixel should be drawn in each column
  ◦ For lines with other slopes, exactly one pixel should be drawn in each row
Scan-converting lines (Section 4.5)

- Recall that to draw a single pixel in Java, we define a method

```java
void putPixel (Graphics g, int x, int y)
{
    g.drawLine (x, y, x, y);
}
```
Scan-converting lines (Section 4.5)

1.1 Basic incremental algorithm

- Simplest approach:
  - Slope $m = \Delta y/\Delta x$
  - Increment $x$ by 1 from left-most point (-1 ≤ $m$ ≤ 1)
  - Use line equation $y_i = x_i m + b$ and then round off
  - However this is inefficient due to floating point multiplication, addition, and rounding
Scan-converting lines (Section 4.5)

1.1 Basic incremental algorithm

- Optimized approach:
  - \( y_{i+1} = mx_{i+1} + b \)
    - \( = m(x_i + \Delta x) + b \)
    - \( = y_i + m\Delta x \)

- Since \( x = 1 \) (i.e. we are incrementing by 1 each time),
  \( y_{i+1} = y_i + m \)

- So this is called an “incremental algorithm”
Scan-converting lines (Section 4.5)

1.1 Basic incremental algorithm

- Optimized approach:
  - At each step we increment based upon the previous step

\[
\begin{align*}
&(X_{i+1}, \text{Round}(y_i)) \\
&(X_i, y_i) \\
&(X_i + 1, \text{Round}(y_i + m)) \\
&(X_i + 1, (y_i + m))
\end{align*}
\]
Scan-converting lines (Section 4.5)

1.1 Basic incremental algorithm

- Optimized approach:
  - Pseudocode: (for $|m| \leq 1$)
    
    ```
    float y, m
    int x, dx, dy
    dy = y_q - y_p
    dx = x_q - x_p
    m = (float) dy / dx

    for (x = x_p; x <= x_q; x++)
    {
        putPixel (g, x, Math.round (y))
        y = y + m
    }
    ```

Graphics Algorithms for drawing 2D primitives
Scan-converting lines (Section 4.5)

1.1 Basic incremental algorithm

- Optimized approach:
  - Because of rounding, error of inaccuracy is 
    \[-0.5 < (y_{exact} - y) \leq 0.5\]

- If $|m| > 1$, reverse the roles of $x$ and $y$, i.e.
  \[
  y_{i+1} = y_i + b \\
  x_{i+1} = x_i + \frac{1}{m}
  \]
Scan-converting lines (Section 4.5)

1.1 Basic incremental algorithm

- Optimized approach:
  - Also need to consider the special cases of horizontal, vertical, and diagonal lines
  - Major drawback: one of x and y has to be a real (float), as well as m.
  - Also, rounding takes time
• Scan-converting lines (Section 4.5)
  ◦ 1.2 Bresenham algorithm
    • Improvement over the basic incremental algorithm
    • First, get rid of the rounding operation
      ◦ Make y an integer
        • To make y an integer, we separate its integer portion from its fraction portion (i.e. m + the rounded part d)
        • d = y − Round(y)
        So, -0.5 ≤ d < 0.5
Scan-converting lines (Section 4.5)

1.2 Bresenham algorithm

- Pseudocode:
  
  ```
  float m, d = 0
  int x, y, dx, dy
  dy = y_q - y_p
  dx = x_q - x_p
  m = (float) dy / dx
  for (x = x_p; x <= x_q; x++)
  {
    putPixel (g, x, y)
    d = d + m
    if (d > 0.5)
    {
      y++
      d--
    }
  }
  ```

Graphics Algorithms for drawing 2D primitives
Scan-converting lines (Section 4.5)

1.2 Bresenham algorithm

- Second, get rid of float types m and d
  - To get rid of 0.5 for d, we double d to make it an integer
  - To get rid of division in m, we multiply m by \(x_q - x_p\)
  - So, we introduce a scaling factor \(c = 2 \times (x_q - x_p)\)
  - Main reason: d and m are used only for decision-making, not for calculation of x and y
- So,
  - \(M = cm = 2(y_q - y_p)\)
  - \(D = cd\)
  - \(H = c \times 0.5 = x_q - x_p\)
Scan-converting lines (Section 4.5)

1.2 Bresenham algorithm

Then we get an integer version of the algorithm

Pseudocode:

```
int x, y, D = 0
H = xq – xp
C = 2 * H
M = 2 * (yq – yp)
for (x = xp; x <= xq; x++)
{
    putPixel (g, x, y)
    D = D + M
    if (D > M) // d > 0.5 \(\rightarrow\) c \* d > c \* 0.5 \(\rightarrow\) D > H
    {
        y++
        D = D – C
    }
}
```
Scan-converting lines (Section 4.5)

1.2 Bresenham algorithm

- Now we generalize the algorithm to handle all angles of slope and different orders of endpoints:

\[ |dx| > |dy| \text{ a pixel is drawn on each } x \text{ increment} \]

\[ |dy| > |dx| \text{ a pixel is drawn on each } y \text{ increment} \]
Scan-converting lines (Section 4.5)

1.3 Extending the Bresenham algorithm by using double-steps

- The Bresenham algorithm:
  1. Determines slope
  2. Chooses between two pixels based on $d$

- The Double-step algorithm:
  1. Halves the number of decisions (i.e. the number of iterations of the loop) by checking for the next two pixels rather than one
  2. Is therefore faster than the Bresenham algorithm, but may not be as accurate
Scan-converting lines (Section 4.5)
  1.3 Double-step algorithm

- Pattern 1: $4dy < dx$
- Pattern 2: $4dy \geq dx$ and $2dy < dx$
- Pattern 3: $2dy \geq dx$

Note: Patterns 1 and 4 cannot occur on the same line

Graphics Algorithms for drawing 2D primitives
Scan-converting lines (Section 4.5)  

1.3 Double-step algorithm  

Algorithm:  
- D is initially set to $4dy - dx$, then check in each step:

\[  
\begin{align*}  
\text{if} & \quad \begin{cases}  
\text{d < 0, d \geq 0} & (4dy < dx) \rightarrow \text{Pattern 1, d = d + 4dy} \\
\text{d < 2dy} & \rightarrow \text{Pattern 2} \\
\text{d \geq 2dy} & \rightarrow \text{Pattern 3}  
\end{cases}  
\end{align*}  
\]

\[  
\begin{align*}  
\text{end if}  
\end{align*}  
\]

\[  
\begin{align*}  
x & = x + 2  
\end{align*}  
\]
• Scan-converting lines (Section 4.5)
  ◦ Line intensity problem
    • Question: Which line has more pixels, A or B?
• Scan-converting lines (Section 4.5)
  ◦ Line intensity problem
    • Answer: Both lines have the same number of pixels, however line A is \( \sqrt{2} \) longer than line B
    • So line A looks thinner
    • Solutions:
      (1) Set line intensity as a function of slope
      (2) Treat a line as a rectangle using anti-aliasing approach
Questions/comments/more fudge?