Rasterization, or "What is `glBegin(GL_LINES)` really doing?"

Course web page:

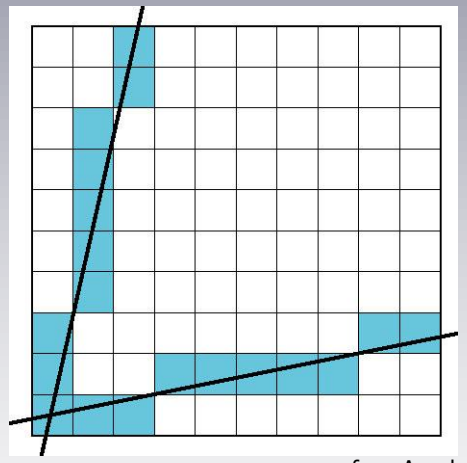
http://goo.gl/EB3aA

Outline

- Rasterizing lines
  - DDA/parametric algorithm
  - Midpoint/Bresenham's algorithm
- Rasterizing polygons/triangles

Rasterization: What is it?

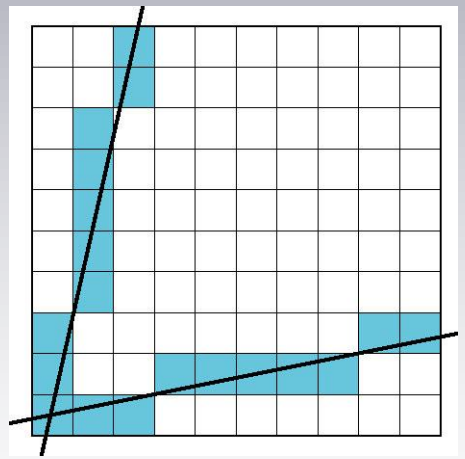
- How to go from floating-point coords of geometric primitives' vertices to integer coordinates of pixels on screen
- Geometric primitives
  - Points: Round vertex location in screen coordinates
  - Lines: Can do this for endpoints, but what about in between?
  - Polygons: How to fill area bounded by edges?



from Angel

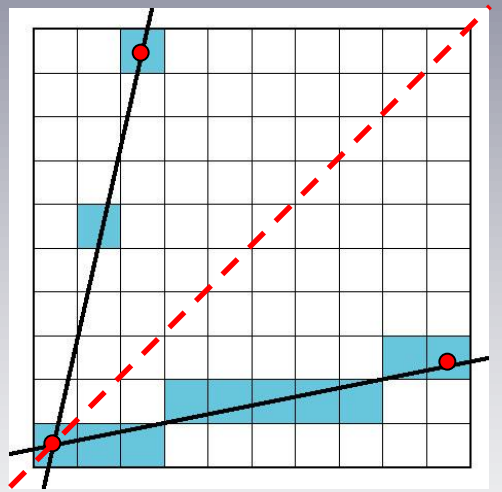
Rasterizing Lines: Goals

- Draw pixels as close to the ideal line as possible
- Use the minimum number of pixels without leaving any gaps
- Do it efficiently
- Extras
  - Handle different line styles
    - Width
    - Stippling (dotted and dashed lines)
    - Join styles for connected lines
  - Minimize aliasing ("jaggies")



DDA/Parametric Line Drawing

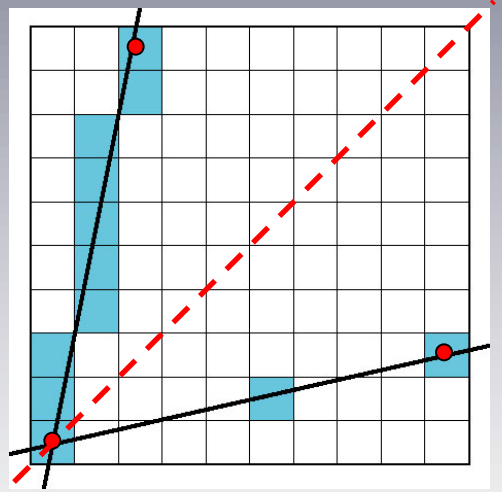
- DDA stands for Digital Differential Analyzer, the name of a class of old machines used for plotting functions
- Slope-intercept form of a line: **`y = mx + b`**
  - **`m = dy/dx`**
  - **`b`** is where the line intersects the Y axis
- DDA's basic idea: If we increment the **`x`** coordinate by 1 pixel at each step, the slope of the line tells us how to much increment **`y`** per step
  - I.e., **`m = dy/dx`**, so for **`dx = 1`**, **`dy = m`**
  - This only works if **`m <= 1`**—otherwise there are gaps
    - Solution: Reverse axes and step in Y direction. Since now **`dy = 1`**, we get **`dx = 1/m`**



from Angel

DDA/Parametric Line Drawing

- DDA stands for Digital Differential Analyzer, the name of a class of old machines used for plotting functions
- Slope-intercept form of a line: **`y = mx + b`**
  - **`m = dy/dx`**
  - **`b`** is where the line intersects the Y axis
- DDA's basic idea: If we increment the **`x`** coordinate by 1 pixel at each step, the slope of the line tells us how to much increment **`y`** per step
  - I.e., **`m = dy/dx`**, so for **`dx = 1`**, **`dy = m`**
  - This only works if **`m <= 1`**—otherwise there are gaps
    - Solution: Reverse axes and step in Y direction. Since now **`dy = 1`**, we get **`dx = 1/m`**



from Angel

DDA: Algorithm

1. Given endpoints $(x_0, y_0), (x_1, y_1)$
   - Integer coordinates: Round if endpoints were originally real-valued
   - Assume (x_0, y_0) is to the left of (x_1, y_1): Swap if they aren't
2. Then we can compute slope:

   $$m = dy/dx = (y_1 - y_0) / (x_1 - x_0)$$

3. Iterate
   - If $|m| <= 1$: Iterate integer x from x_0 to x_1, incrementing by 1 each step
     - Initialize real $y = y_0$
     - At each step, $y += m$, and plot point $(x, round(y))$
   - Else $|m| > 1$: Iterate integer y from y_0 to y_1, incrementing (or decrementing) by 1
     - Initialize real $x = x_0$
     - At each step, $x += 1/m$, and plot $(round(x), y)$

Midpoint/Bresenham's line drawing

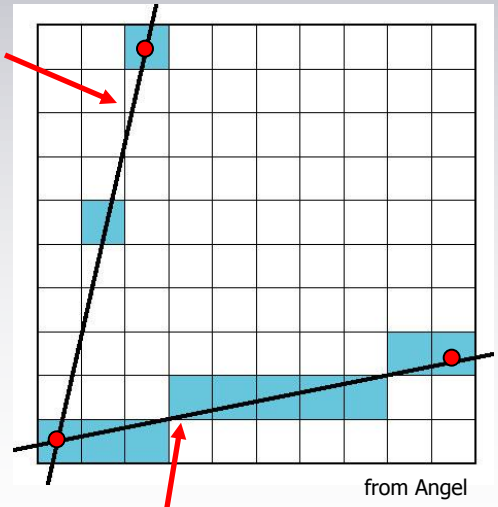
- DDA is somewhat slow
  - Floating-point calculations, rounding are relatively expensive
- Big idea: Avoid rounding, do everything with integer arithmetic for speed
- Assume slope between 0 and 1
  - Again, handle lines with other slopes by using symmetry

Midpoint line drawing: Line equation

- Recall that the slope-intercept form of the line is
  `y = (dy/dx)x + b`
- Multiplying through by **`dx`**, we can rearrange this in **implicit form**:

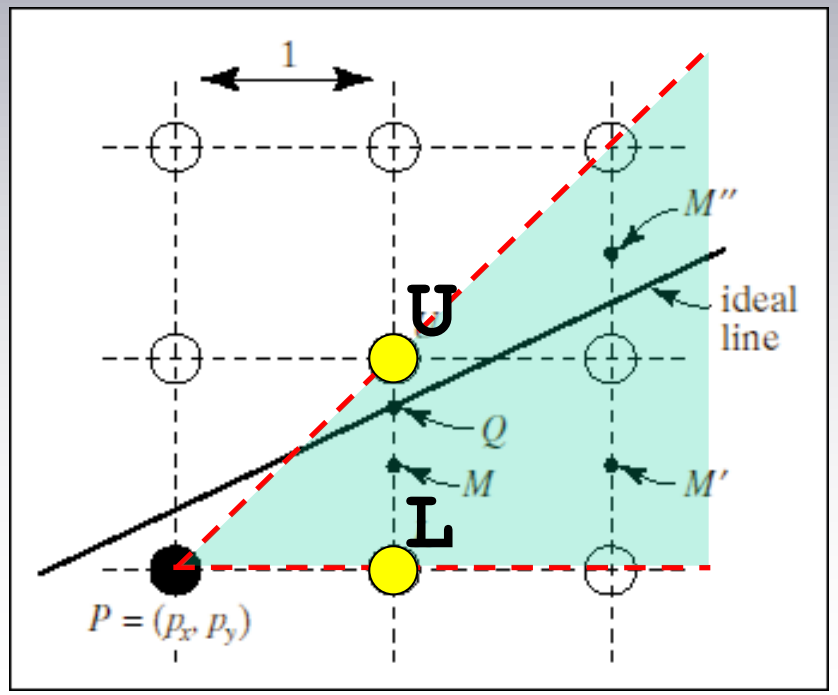
`F(x, y) = dy x - dx y + dx b = 0`

- **`F`** is:
  - Zero for points on the line
  - Positive for points **below** the line (**right** if slope > 1)
  - Negative for points **above** the line (**left** if slope > 1)
    - Examples: (0, 1), (1, 0), etc.



Midpoint line drawing: The Decision

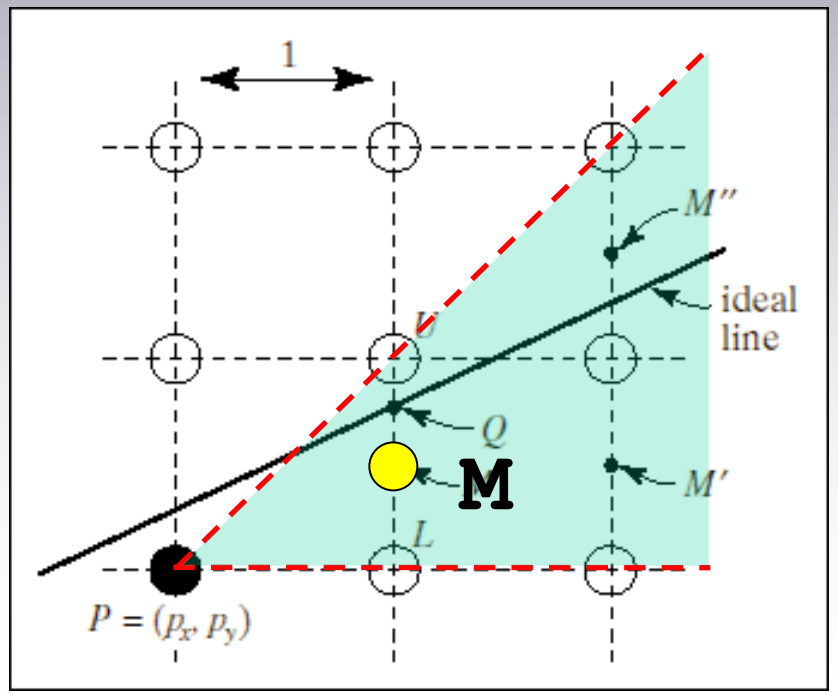
- Given our assumptions about the slope, after drawing **`(x, y)`** the only choice for the next pixel is between the upper pixel **`U = (x+1, y+1)`** and the lower one **`L = (x+1, y)`**
- We want to draw the one closest to the line



from Hill

Midpoint line drawing: Midpoint decision

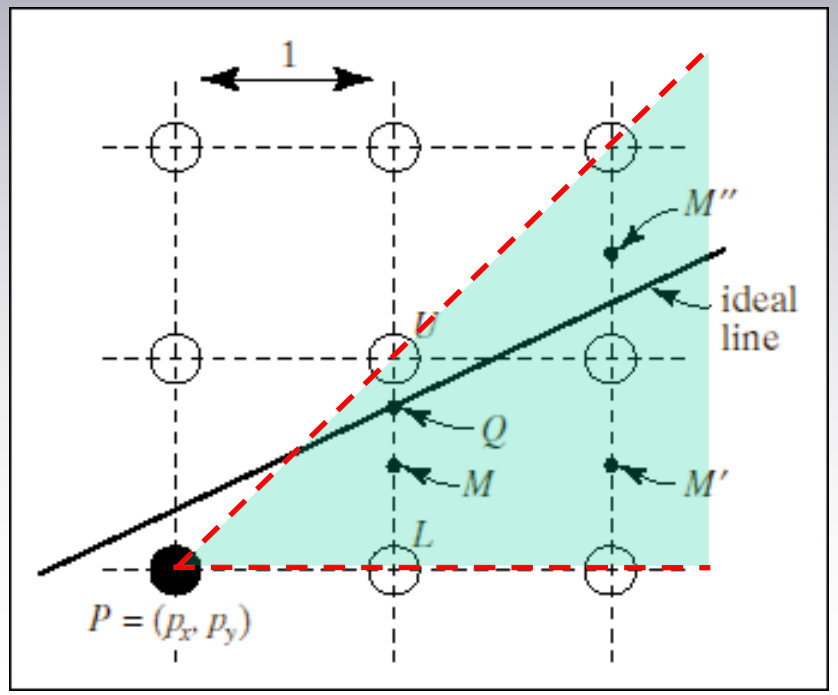
- After drawing **`(x, y)`**, in order to choose the next pixel to draw we consider the **midpoint `M = (x+1, y+0.5)`**
  - If it's **on** the line, then **`U`** and **`L`** are equidistant from the line
  - If it's **below** the line, pixel **`U`** is closer to the line than pixel **`L`**
  - If it's **above** the line, then **`L`** is closer than **`U`**



from Hill

Midpoint line drawing: Midpoint decision

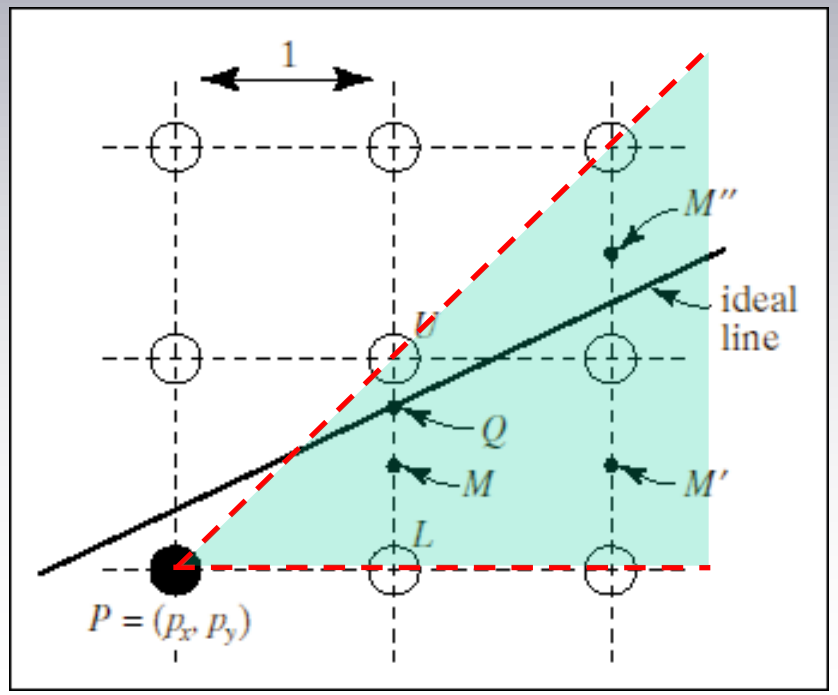
- So **`F`** is a **decision function** about which pixel to draw:
  - If **`F(M) = F(x+1, y+0.5) > 0`** (**`M`** below the line), pick **`U`**
  - If **`F(M) = F(x+1, y+0.5) <= 0`** (**`M`** above or on line), pick **`L`**



from Hill

Midpoint line drawing: Implementation

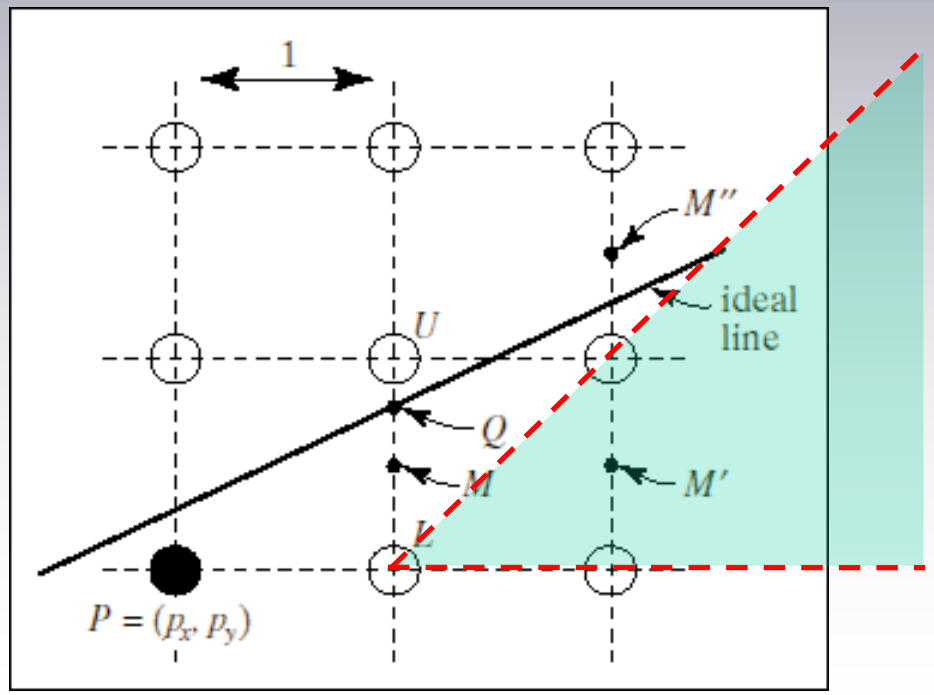
- Key efficiency insight: **F** does not have to be fully evaluated every step
- Suppose we do the full evaluation once and get **F(x+1, y+0.5)**
  - If we choose **L**, next midpoint to evaluate **M'** is at **F(x+2, y+0.5)**
  - If we choose **U**, next midpoint **M''** would be at **F(x+2, y+1.5)**



from Hill

Midpoint line drawing: Implementation

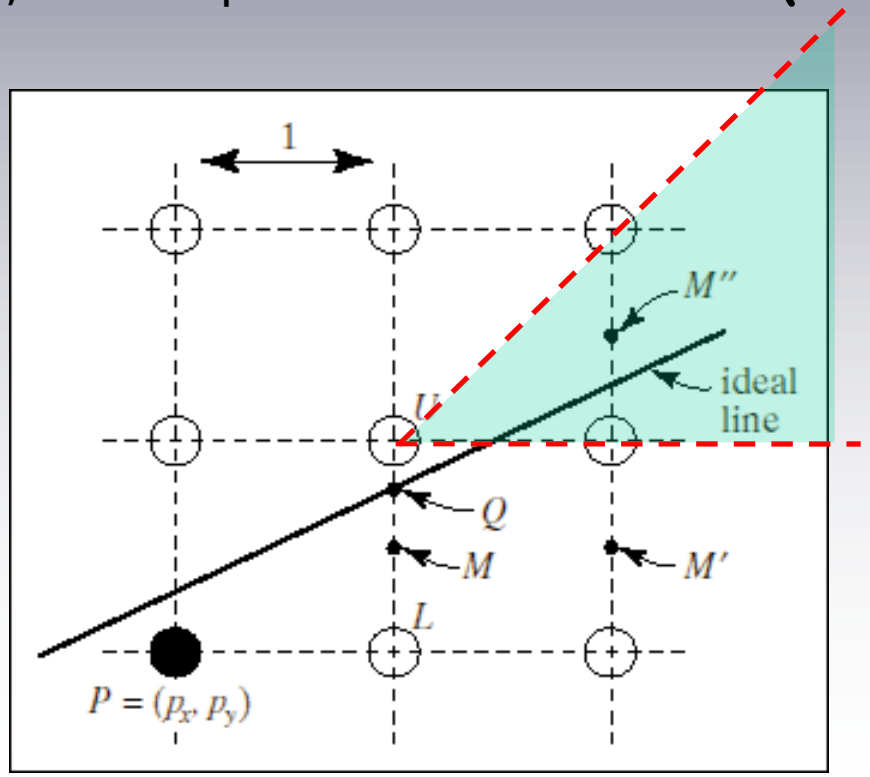
- Key efficiency insight: **F** does not have to be fully evaluated every step
- Suppose we do the full evaluation once and get **F(x+1, y+0.5)**
  - If we choose **L**, next midpoint to evaluate **M'** is at **F(x+2, y+0.5)**
  - If we choose **U**, next midpoint **M''** would be at **F(x+2, y+1.5)**



from Hill

Midpoint line drawing: Implementation

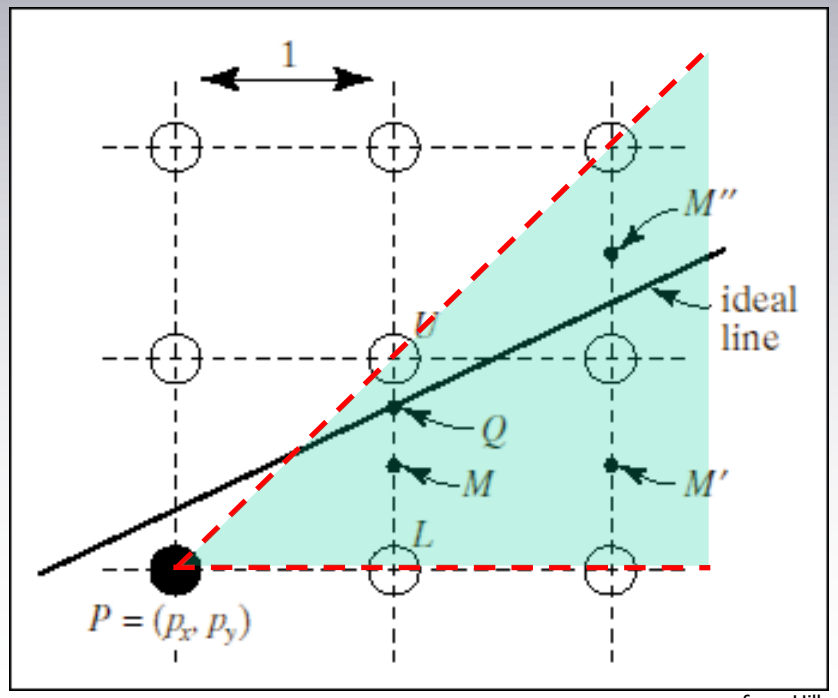
- Key efficiency insight: **F** does not have to be fully evaluated every step
- Suppose we do the full evaluation once and get **F(x+1, y+0.5)**
  - If we choose **L**, next midpoint to evaluate **M'** is at **F(x+2, y+0.5)**
  - If we choose **U**, next midpoint **M''** would be at **F(x+2, y+1.5)**



from Hill

Midpoint line drawing: Implementation

- Expanding these out using **`F(x, y) = dy x - dx y + dx b`**:
  - F_M **`= F(x + 1, y + 0.5) = dy(x + 1) - dx(y + 0.5) + dx b`**
  - $F_{M'}$ **`= F(x + 2, y + 0.5) = dy(x + 2) - dx(y + 0.5) + dx b`**
  - $F_{M''}$ **`= F(x + 2, y + 1.5) = dy(x + 2) - dx(y + 1.5) + dx b`**
- Note that $F_{M'} - F_M = dy$ **and** $F_{M''} - F_M = dy - dx$
- So depending on whether we choose **L** or **U**, we just have to add **dy** or **dy - dx**, respectively, to the old value of **F** in order to get the new value



from Hill

Midpoint line drawing: Algorithm

- To initialize, we do a full calculation of **F** at the first midpoint next to the left line endpoint:

$$
\begin{aligned}
F(x_0 + 1,\ y_0 + 0.5) &= dy(x_0 + 1) - dx(y_0 + 0.5) + dx\ b \\
&= dy\ x_0 - dx\ y_0 + dx\ b + dy - 0.5\ dx \\
&= F(x_0,\ y_0) + dy - 0.5\ dx
\end{aligned}
$$

  - But $F(x_0, y_0) = 0$ since it's on the line, so our first $F = dy - 0.5\ dx$
  - Only the sign matters for the decision, so to make it an **integer value** we multiply by 2 to get $2F = 2\ dy - dx$
- To update, keep current values for **x** and **y** and a running total for **F**:
  - When **L** is chosen: **F += 2dy** and **x++**
  - When **U** is chosen: **F += 2(dy - dx)** and **x++**, **y++**

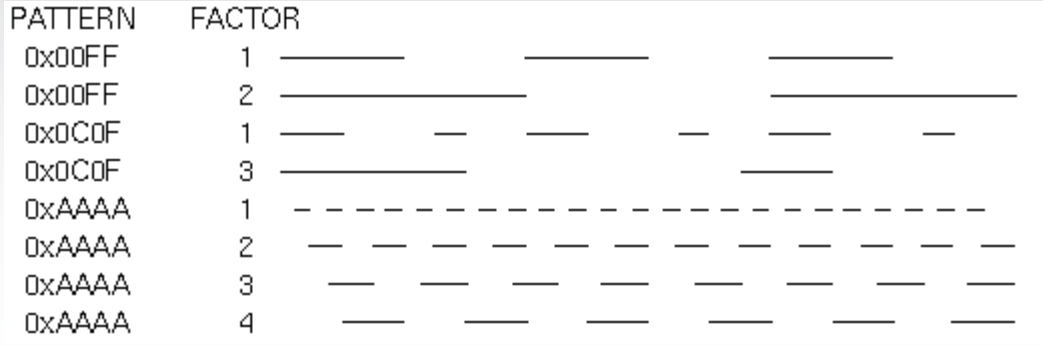
Line drawing speed

- 100,000 random lines in 500 x 500 window (average of 5 runs)

- DDA: 6.8 seconds
- Midpoint: 2.5 seconds
- OpenGL using **`GL_LINES`** (in software): 1.6 seconds

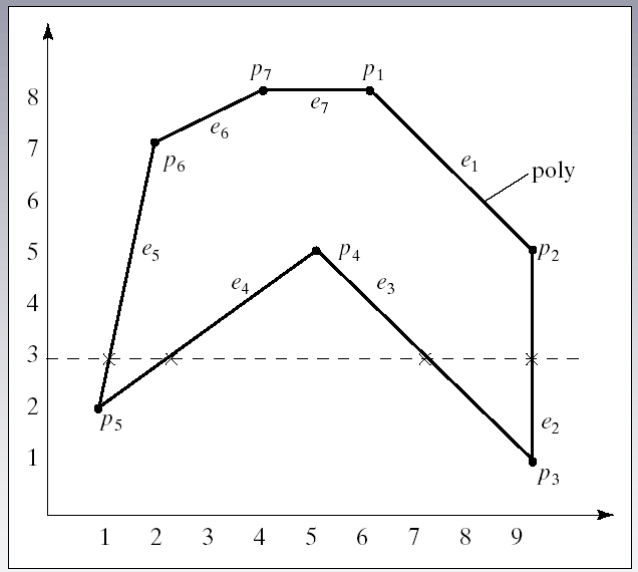
Extensions

- How to draw thick (>1 pixel wide) lines?
  - Nested Bresenham's (perpendicular to main line at each step, or series of parallel lines)
- Stippled/dashed lines
  - Add state (pen up/down) inside loop



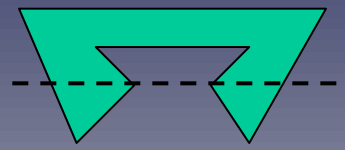
Polygon Rasterization

- Given a set of vertices, want to fill the interior
- Basic procedure:
  - Iterate over **scan lines** between and bottom vertex
  - For each scan line, find all intersections with polygon edges
  - Sort intersections by X-value and fill pixel runs between even pairs of intersections

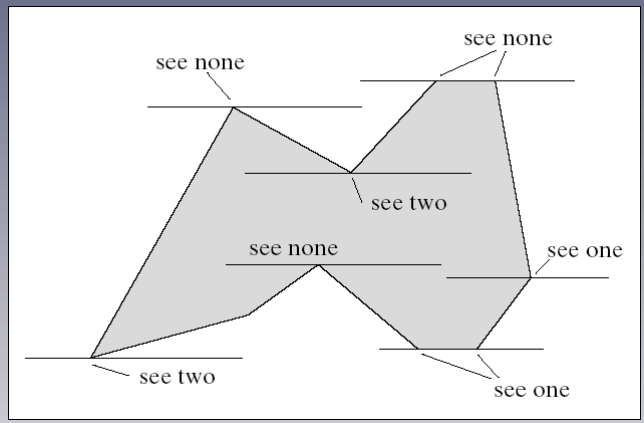


from Hill

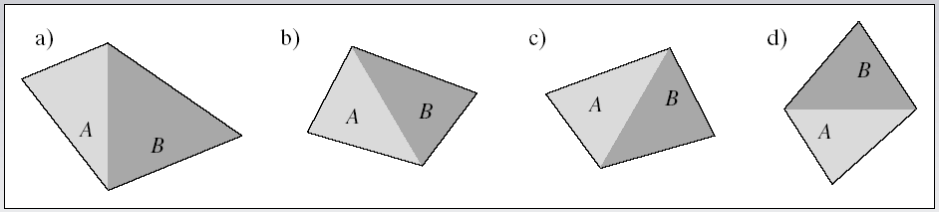
Polygon Rasterization: Notes



- Refinements
  - Maintain correct parity by discarding intersections with upper endpoint of any edge
  - Avoid double-drawing at border of abutting polygons (bad if blending/ XOR'ing): **Left/bottom ownership**
    - Draw pixel on left edge of run but not right
    - Discard intersections with any horizontal edges



from Hill



- Efficiency
  - Avoid checking intersections with all poly edges for each scanline by keeping an **active edge list**

Rasterizing triangles

- Special case of polygon rasterization
  - Exactly two **active** edges at all times
- One method:
  - Fill scanline table between top and bottom vertex with leftmost and rightmost side by using DDA or midpoint algorithm to follow edges
  - Traverse table scanline by scanline, fill run from left to right