Overview of Lecture

- *Initial Problem Statement: Line Drawing*
- Preliminary Solution
- Digression on lines (review of geometry)
- Digression on forward differences
- Revised solution
- Analysis of solution and of problem statement

The Problem

Line Drawing

- Draw a line on a raster screen between two points.
- What's wrong with the statement of the problem?
  - it doesn't say anything about which points are allowed as endpoints
  - it doesn't give a clear meaning to "draw."
  - it doesn't say what constitutes a "line" in the raster world.
  - it doesn't say how to measure the success of a proposed algorithm.

Problem Restatement

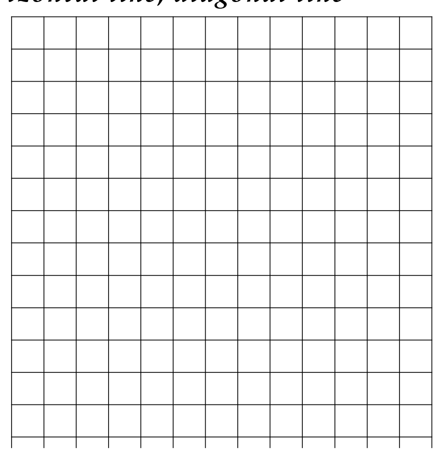
- Given two points P and Q in the plane, determine which pixels on a raster screen should be *on* in order to make a picture of a unit-width line segment starting at P and ending at Q.
  - still fails to address what points are valid endpoints, and what is our criterion for success.

More Precise Statement

- Given two points P and Q in the plane, both with integer coordinates, determine which pixels on a raster screen should be *on* in order to make a picture of a unit-width line segment starting at P and ending at Q.
  - only thing missing is criterion for success, which will prove elusive...

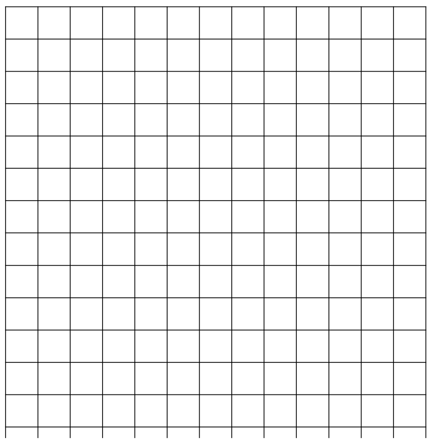
Drawing a Line

- ***Horizontal line, diagonal line***



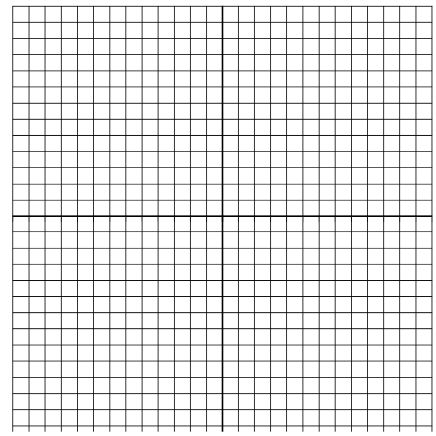
Drawing a Line

- ***Shallow slope line***



Drawing a Line

- ***What if the grid is smaller?***

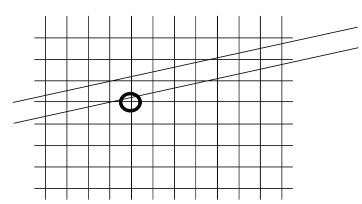


Some Conclusions

- The line should have some *thickness* independent of the grid spacing (i.e., resolution)
- In other words, it's really a thin rectangle
- For horizontal lines, the thinnest rectangle we can hope for is one interpixel-distance
- We should try to draw a "one pixel wide rectangle" instead of a line

Approach:

- For each (a, b) in the integer grid that is also within the thin rectangle, set $pixel(a, b) = 1$; else set $pixel(a, b) = 0$.
- Commentary:
  - we have no justification that this is correct. In this case

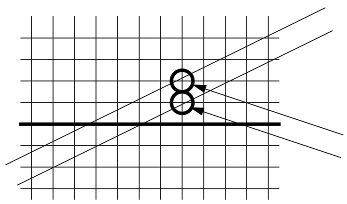


  - who is to say this pixel should not be on? It might make sense......
  - we have said <u>what</u> to do, but not <u>how</u>.

A Reconsideration (1/2)

(Hack city)

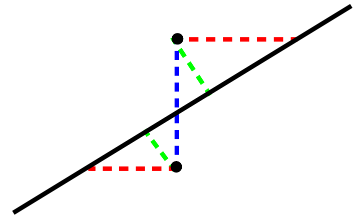
- Looking for points that lie within the thin rectangle
- For horizontal and diagonal line, just drawing one pixel in each column looked pretty good
- *Revise our goal*: seek to draw, in each column of the image, exactly one pixel that is within the thin rectangle.



- Sometimes there are *two!*

A Reconsideration (2/2)

- We pick the one “closer” to mid-line. How to measure?



Further Simplification: Look for Symmetry

- If we could “draw” lines with shallow positive slope ($0 \leq$ slope ≤ 1), we would be done.
- To draw lines with shallow negative slope ($0 \geq$ slope ≥ -1), we just negate all y-values.
- For lines with slope > 1 or slope < -1, we swap x- and y-axes.
- Code looks like this:


```
(void) swap_y_draw(int x, int y)
{
   draw_pixel(x, -y)
}

(void) swap_xy_draw(int x, int y)
{
   draw_pixel(y, x)
}

(void) swap_xy_y_draw(int x, int y)
{
   draw_pixel(y, -x)
}
... if (0 <= slope <= 1)
   draw_fn = draw_pixel
   draw_line(Px, Py, Qx, Qy, draw_fn)
else if (-1 <= slope <= 0)
   draw_fn = swap_y_draw
   draw_line(Px, -Py, Qx, -Qy, draw_fn)
else if (1 < slope)
   draw_fn = swap_xy_draw
   draw_line(Py, Px, Qy, Qx, draw_fn)
else
   draw_fn = swap_xy_y_draw
   draw_line(-Py, Px, -Qy, Qx, draw_fn)
...
```

Revised Problem Statement

- We have simplified problem using symmetry: need only deal with lines with 0 <= slope <= 1
- Simplified problem using assumption about close pixels being relevant ones...

Given $P = (x_0, y_0)$ and $Q = (x_1, y_1)$, points of the integer lattice, with

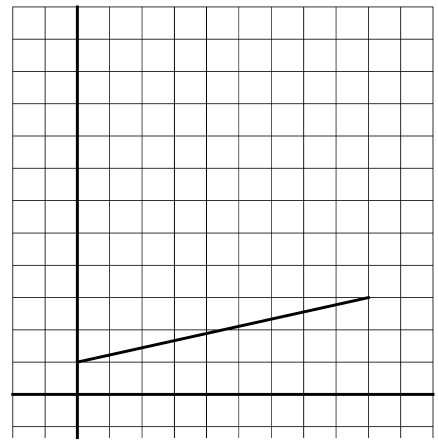
$$x_0 < x_1$$
$$y_0 < y_1$$
$$(y_1 - y_0) \le (x_1 - x_0)$$

find, for each x between x_0 and x_1 the point (x, y) that is closest to the line between P and Q.

- The "closest" will give us troubles later.

An Example

Line from $(0, 1)$ to $(9, 3)$
Equation is $-2x + 9y - 9 = 0$



Let distance from any point (a, b) to L be d_L.

Then $d_L(a, b) = \frac{|-2a + 9b - 9|}{\sqrt{2^2 + 9^2}} = \frac{|-2a + 9b - 9|}{\sqrt{85}}$

- Let's derive this formula

Implicit Formula for Lines (1/2)

- 2 point formula: infinite line between (x_0, y_0) and (x_1, y_1) is

$\Delta y = m\Delta x$ (explicit form)

$$(y - y_0) = \frac{(y_1 - y_0)}{(x_1 - x_0)}(x - x_0)$$

$$(y - y_0)(x_1 - x_0) = (y_1 - y_0)(x - x_0)$$

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

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

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

Implicit Formula for Lines (2/2)

- Formula has the implicit form $Ax + By + C = 0$.
- If $L = \{(x, y): Ax + By + C = 0\}$ (implicit set definition), and c is not 0, and $L_1 = \{(x, y): (cA)x + (cB)y + (cC) = 0\}$ ...
- then $L = L_1$, i.e., the coefficients are adjustable, up to a scale factor.
- This adjustability up to a scale factor will appear later in the guise of "homogeneous coordinates," which are one of the triumphs of art and geometry.
- Later we'll use parametric form
  $x = (1 - t)x_0 + tx_1$
  $y = (1 - t)y_0 + ty_1$

(handles infinite slopes, finite line segments).

Digression on Dot Products (1/2)

- Dot products (inner products) between vectors are defined simply as sum of pairwise multiplications (NOT vector multiplications!)

$$\begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \bullet \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} = x_1 y_1 + \ldots + x_n y_n$$

- Can also use row vectors $[x_1 + \ldots + x_n] \bullet [y_1 + \ldots + y_n]$
- The dot product of x with itself is therefore

$$x_1^2 + \ldots + x_n^2$$

which (for $n = 2$ or $n = 3$) looks like the square of the distance from x to the origin.

- Define the "norm"

$$\|x\| = \sqrt{x_1^2 + \ldots + x_n^2}$$

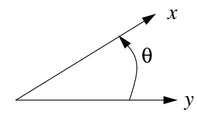
as the *length* of x.

Digression on Dot Products (2/2)

Dot products and angles

- A *unit vector* is one with $\|x\| = 1$.
- If x and y are unit vectors, then there's a relation between the dot product and the angle θ between x and y, namely

$$x \bullet y = \cos\theta$$



- In general (even for non-unit vectors)

$$x \bullet y = \|x\|\|y\|\cos\theta$$

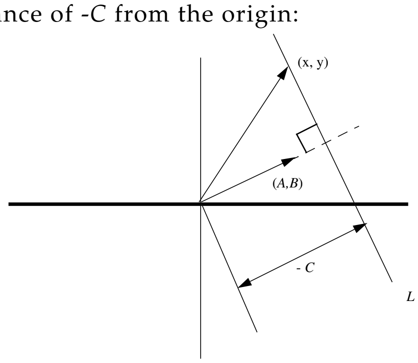
- The rule of thumb is that $x \bullet y$ measures "how much x points in the same direction as y". Positive means "same direction".

Distance to Line Using Dot Product (1/6)

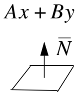
- A geometric interpretation of $Ax + By + C = 0$. Rewrite it as

$$\begin{bmatrix} A \\ B \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = -C$$

- Then we see (assuming (A,B) is a unit vector) that the line L is perpendicular to (A,B) at a distance of $-C$ from the origin:

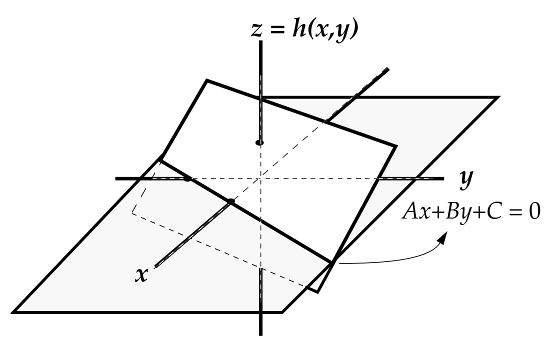


- Generalizes to plane equation $Ax + By + Cz + D = 0$ with normal $\bar{N} = [A, B, C]$



A Different Interpretation of A Line

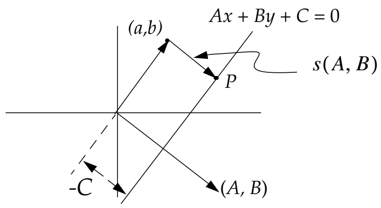
- If we let $z = h(x, y) = Ax + By + C$, then the graph of h looks like this:



- Our line is the intersection of two planes where $h = 0$.

Distance to Line *(2/6)*

- Find distance from a $R = (a, b)$ to the line $L = \{(x, y)\colon\ Ax + By + C = 0\}$.



- For some s, $P = (a, b) + s(A, B)$ is a point P of L (vector addition).
- The length of $s(A, B)$ is the distance to L.
  - $(a, b) + s(A, B) = (a + sA, b + sB)$.
  - point P is on L if and only if

$$A(a + sA) + B(b + sB) + C =$$

$$Aa + sA^2 + Bb + sB^2 + C = 0$$

  - solving for s, we get

$$s = \frac{-C - Aa - Bb}{A^2 + B^2}$$

Distance to Line *(3/6)*

- The length of $s(A, B)$ is $\|s(A, B)\|$

$$= |s|\ \|(A, B)\| = |s|\sqrt{A^2 + B^2} = \frac{|-C - Aa - Bb|}{A^2 + B^2} \cdot \sqrt{A^2 + B^2}$$

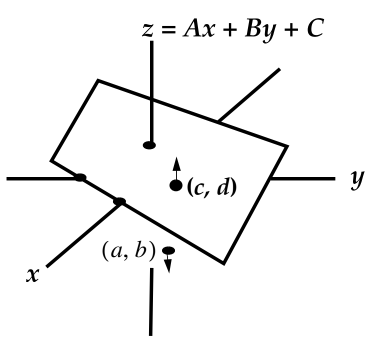
- In absolute value, we can remove the negatives from $|-C-Aa-Bb|$ to get

$$d_L(R) = \frac{Aa + Bb + C}{\sqrt{A^2 + B^2}}$$

- *For a fixed representation of the line L, the distance to a variable point (a, b) is proportional to $|Aa + Bb + C|$.*

Distance to Line *(4/6)*

- OBS I: If the line passes between (a, b) and (c, d) then exactly <u>one</u> of $h(a, b) = Aa + Bb + C$ and $h(c, d) = Ac + Bd + C$ will be negative.



Distance to Line *(5/6)*

- OBS II: The average of any two numbers will be positive if greater of the numbers is positive, and similarly for negative values.
- OBS III: The average,

$$\frac{h(a, b) + h(c, d)}{2}$$

is the same as the value of h midway between the points:

$$h\left(\frac{a + c}{2}, \frac{b + d}{2}\right)$$

- Check:

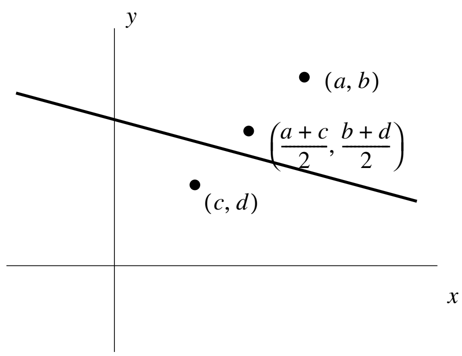
$$\frac{h(a, b) + h(c, d)}{2} = \frac{Aa + Bb + C + Ac + Bd + C}{2} =$$

$$\frac{A(a + c) + C + B(b + d) + C}{2} = A\left(\frac{a + c}{2}\right) + B\left(\frac{b + d}{2}\right) + C$$

$$h\left(\frac{a + c}{2}, \frac{b + d}{2}\right)$$

Distance to Line (6/6)

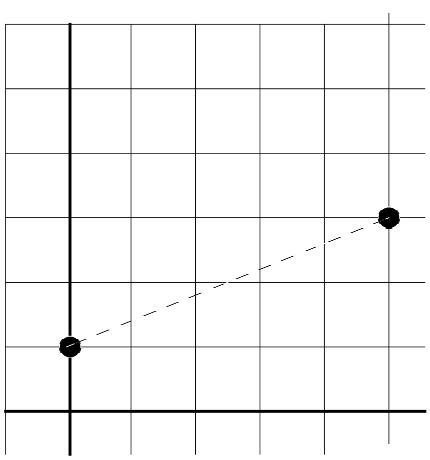
- So ... this midpoint value determines which of the two points is closer to the line: pick the point with opposite sign
- We can use either the midpoint value or the average of the values (they're equal) to make the determination. We'll call this value the decision variable for picking the closer pixel.



An Emerging Algorithm:

- Walk from x_0 to x_1.
- Start by turning on $(x, y) = (x_0, y_0)$
- For each subsequent step:
- Increment x.
- Look at $h\left(x, y+\frac{1}{2}\right)$ to choose between two possible points: (x, y) and $(x, y+1)$.
  - if positive, turn on one pixel
  - if negative turn on the other
  - if you chose $(x, y+1)$, increment y
  - otherwise do nothing
- Remember h=0 on the line

Example



An ambiguity:

- Q: When we evaluate $h\left(x, y+\frac{1}{2}\right)$ and find it positive, do we turn on (x, y) or $(x, y+1)$?
- A: It all depends.
- If increasing the value of y increases the value of h, then we should turn on (x, y) and conversely.
- Unfortunately, an increase in y may increase h or decrease it, depending on the value of B
- If B is positive, then increasing y also increases h
- If B is not positive...we can multiply all three of A, B, C by -1...
- Recall the form of the line:

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

- Note that in our restricted case, B is already positive!

The Midpoint Algorithm(v.I)

```
/* for slopes between 0 and 1 */
midpoint_special_case(
   int x0,
   int y0, /* left end of line */
   int x1,
   int y1 /* right end of line */
)
{
   int x, y;
   int A, B, C;
   A = (y0 - y1); B = (x1 - x0);
   C = x0*y1 - x1*y0;
   Pixel(x0, y0);
   for (x = x0 + 1; x <= x1; x++) {
      if ((A*x + B*(y+0.5)+ C) < 0){
         /* draw top pixel */
         y++;
         Pixel(x, y);
      }
      else {
         /* draw bottom pixel */
         Pixel(x, y);
      }
   }
}
```


Analysis

- We’ve ignored the < 0 vs <= 0 issue.
- Need to check boundary conditions: slope = 0, $x0 = x1$, etc.
- Uses 2 multiplies, 3 adds, 2 compares per pixel drawn. Uses floating point values....
- Claim: Can do much better.
- Note: since we only care about signs, instead of using

$$h\left(x, y + \frac{1}{2}\right)$$

or

$$\frac{h(x, y) + h(x, y+1)}{2}$$

we can use *twice* either one, getting rid of floating-point numbers.

Brief Review

- We want to draw a line segment S between two integer points P and Q; the slope of S can be assumed, using symmetry, to be between 0 and 1.
- We decided that S is best represented by a 1 unit wide rectangle consisting of all points whose perpendicular distance to S is less than or equal to 1/2.
- In an attempt to do this, we decided to turn on one pixel in each vertical column, hoping this will yield a good result.
- In each column, we pick the pixel closest to the line.

Continuation of Review

- We notice that if we write an equation of the line in the form $Ax + By + C = 0$, and multiply through by -1 (if necessary) to make sure B is positive, then the function $h(x, y) = Ax + By + C$ is positive for points above the line, and negative for points below it.
- We notice that the graph of $z = h(x, y)$ is planar, and its intersection with the xy-plane is just our line.
- If (a, b) and (c, d) are on opposite sides of the line, then $h(a, b)$ and $h(c, d)$ have opposite signs.
- If $h(a, b) + h(c, d)$ is positive, and $h(a, b)$ is negative, then (a, b) is closer to the line, and vice versa.
- $h(a, b) + h(c, d)$ is exactly twice the value of h midway between (a, b) and (c, d) .in particular, $h(x, y) + h(x, y+1) = 2h(x, y+0.5)$

Continuation of Review

- Leads us to algorithm:

```
start at x = x0, y = y0;
for each x from x0+1 to x1 ...
   compute 2h(x, y+0.5)
   if positive, plot (x,y)
   if negative, plot (x,y+1) and
       increment y.
```

- To continue improving the algorithm, let's look for computational efficiency: can we re-use our former work?
- The answer is "yes," but let's arrive at it by playing a game...

A Boring Game

- I think of a constant function on the real line.

  You have to guess $f(8), f(9), f(10), f(11)$, ...

- f(8) =
- f(9) =
- f(10) =
- f(11) =

A Slightly Less Boring Game

I think of an affine function $f(x) = ax + b$ on the real line, and you have to guess the values at a sequence of points as before.

- f(3) =
- f(4) =
- f(5) =
- f(6) =
- f(7) =

How you did it...

- You observed that...
- $f(x+1) - f(x) = [a(x+1)+b] - [ax+b] = a$
- This is independent of x. Once two successive points have let you determine the value of "a", you just keep adding "a" to get subsequent answers.

Yet Another Game

- I think of a function $f(x, y) = Ax + By + C$, and you have to guess its values at a connected sequence of points, such as f(2, 3), f(3, 3), f(3, 4), f(4, 4), f(4, 5), f(5, 5), f(6, 5)...
- f(2, 3) =
- f(3, 3) =
- f(3, 4) =
- f(4, 4) =
- f(4, 5) =
- f(5, 5) =
- f(6, 5) =

How you did it

- You observed that $f(x+1, y) - f(x, y) = A$, and
- $f(x, y+1) - f(x, y) = B$, so that
- if you know $f(x, y)$, and you have determined A and B, you can determine the next value...
- General definition: $f(x+1) - f(x)$ is called a (forward) difference of f, and is denoted $\Delta f(x)$.
- For 2-variable functions we define

  $\Delta_1 f(x, y) = f(x+1, y) - f(x, y)$, and

  $\Delta_2 f(x, y) = f(x, y+1) - f(x, y)$
- These are sometimes called *partial differences*

More General Computations

- If you know $f(x, y)$ and want to find $f(x+1, y+1)$, here's a way to do it (assuming you know the difference functions as well):

  (i) find $f(x+1, y) = f(x, y) + \Delta_1 f(x, y)$

  (ii) find $f(x+1, y+1) = f(x+1, y) + \Delta_2 f(x+1, y)$
- Note: if the partial differences are easier to compute than the function f itself, then this is a good thing to do for efficiency
- Note: if the partial differences are constant, then the point at which they are computed is irrelevant, and we might just write $\Delta_1 f = 4$, and leave off the (x, y), being informal about it.
- Exercise: if

$$f(x, y) = Ax^2 + Bx + Cy^2 + D$$

then what are the partial differences of f ?

An Improved Midpoint Algorithm

- We need to compute, repeatedly, the value of our *decision variable*:
- $D = (2A)x + (2B)y + (2C + B)$ (This is just $2h\left(x, y + \frac{1}{2}\right)$, which we used for deciding which pixel to choose).
- We typically have computed it at some point (x, y), and use it to choose between $(x+1, y)$, moving East, or $(x+1, y+1)$, moving North East.
- The choice of pixel tells us which next midpoint is so we can update D appropriately
- $D(x+1, y) = 2A(x+1) + 2By + (2C + B)$
  $\Delta_E = D(x+1, y) - D(x) = 2A$
- $D(x+1, y+1) = 2A(x+1) + 2B(y+1) + (2C + B)$
  $\Delta_{NE} = D(x+1, y+1) - D(x) = 2A + 2B$

The Improved Code

```
midpoint_special_case(int x0, int y0,
        int x1, int y1)
{
   int   x, y, A, B, C, D, delta_E,
         delta_NE;
   A = y0 - y1; B = x1 - x0;
   C = x0 * y1 - x1 * y0;
   D = 2*A*x0 + 2*B*y0 + 2*C + B;
   delta_E = 2*A;
   delta_NE = delta_E + 2*B;
   Pixel(x0, y0);
   x = x0 + 1; y = y0;

   for(; x <= x1; x++) {
      if (D <0) { /*Draw top pixel*/
         y++;
         D += delta_NE;
      }
      else { /* D>0 draw lower pixel */
         D += delta_E;
      }
      Pixel(x, y);
   }
}
```

• This is equivalent to Bresenham's algorithm.

Review of Problem and Solution

- Problem was to render a unit-thickness line segment between two integer points in the plane.
- Revised problem was to find all pixels that lay within this line segment.
- Revised problem was to find all pixels that lay within vertical distance 1/2 of the centerline.
- Solution is fast, but does not entirely solve original problem.
- Midpoint(P, Q) finds all pixels within vertical distance 1/2 of the line segment from P to Q, assuming
  (a) P is to the left of Q and no higher than Q
  (b) Line from P to Q has slope no more than 1
- Question: How does the algorithm depend on $(x0, y0)$ and $(x1, y1)$ being integer points?
- Note: we will return to original problem later in lectures on antialiasing...
- Note: in effect we use vertical distance: the average of 2 numbers = h(midpoint). Why is this okay?

Alternate line drawing algorithm

```
for x from x0 to x1
   compute y = mx + b
   Pixel(x, round(y))
```

• Analysis of algorithm:
  – one pixel per column---reasonable only if slope is small
  – uses floats and rounding---slow
  – generates the same pixels as the fast algorithm (except, perhaps, in the marginal cases).
  – generalizes nicely to non-rational slopes and non-integer endpoints

Extra Graph Paper