Raster Displays and Scan Conversion

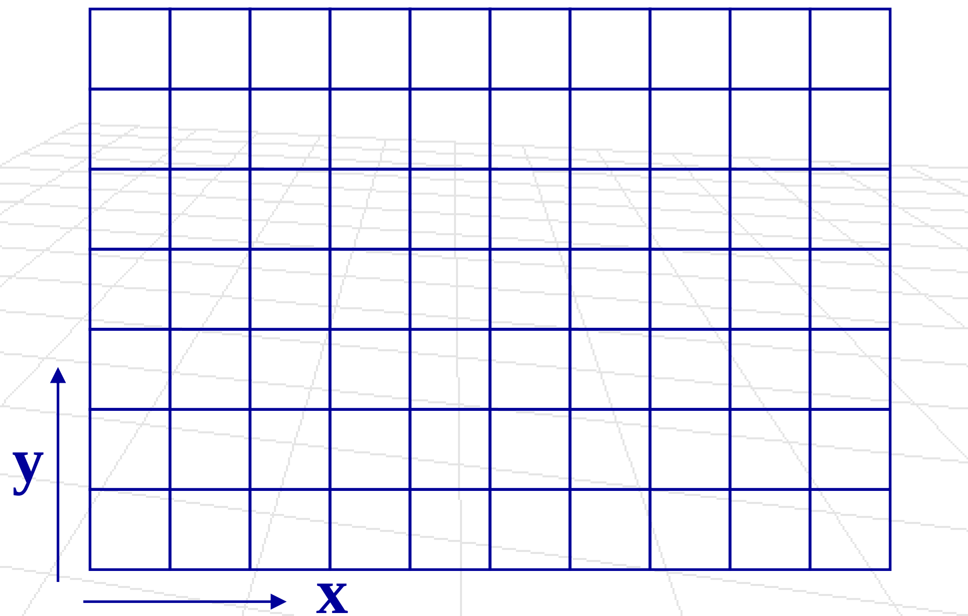
Computer Graphics, CSCD18

Fall 2008

Instructor: Leonid Sigal

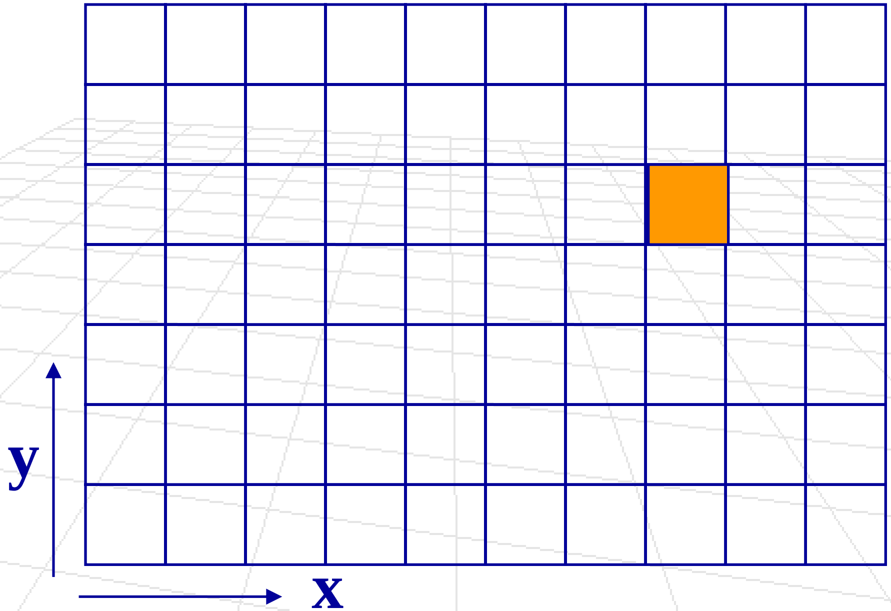
Rater Displays

- Screen is represented by 2D array of locations called *pixels*



Rater Displays

- Screen is represented by 2D array of locations called *pixels*
- At each pixel 2^N intensities/colors can be generated
  - Grayscale $2^8 = 256$
  - Color $(2^8 + 2^8 + 2^8)$

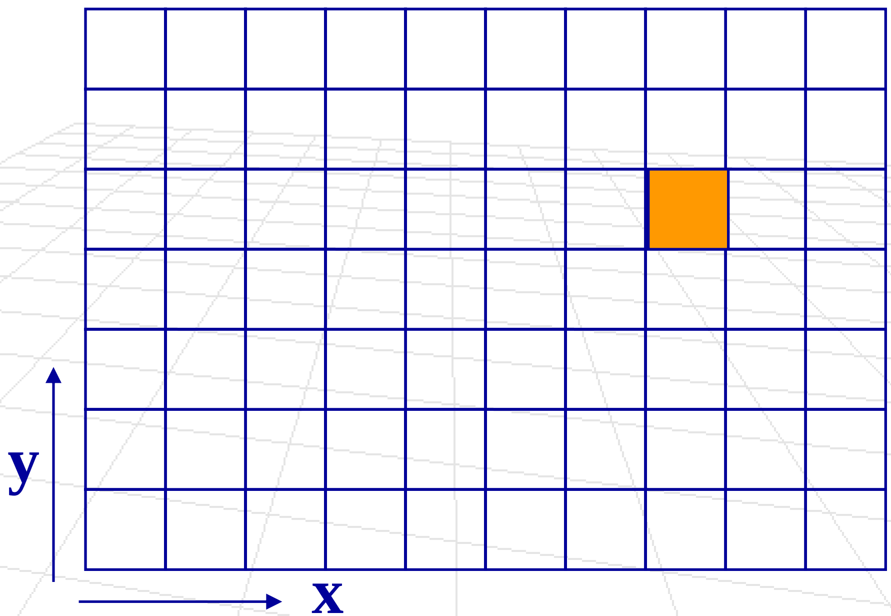


Rater Displays

- Screen is represented by 2D array of locations called *pixels*
- At each pixel 2^N intensities/colors can be generated
  - Grayscale $2^8 = 256$
  - Color $(2^8 + 2^8 + 2^8)$
- Colors are stored in a *frame buffer*
  - physical memory on a graphics card
- Primitive operations

  setpixel (x,y,c)

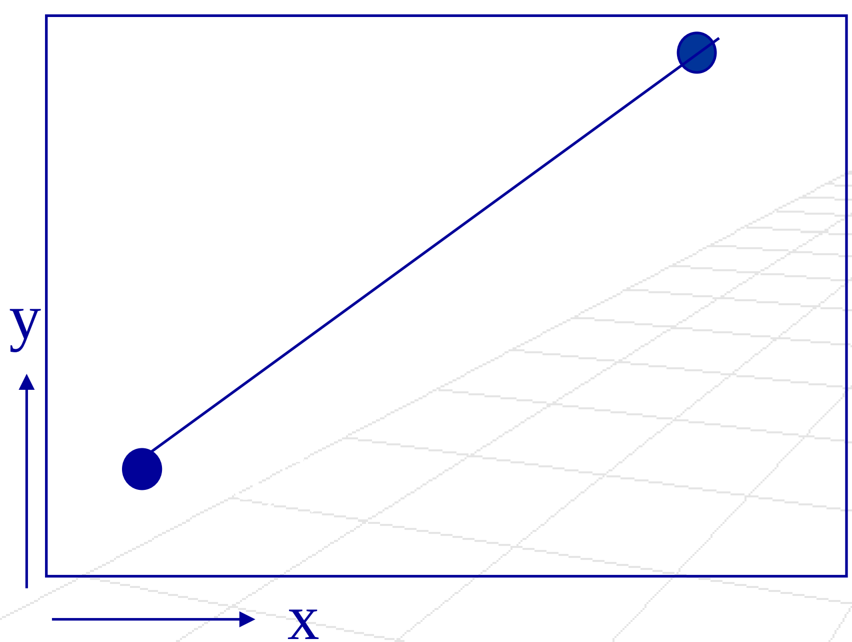
  getpixel (x,y)



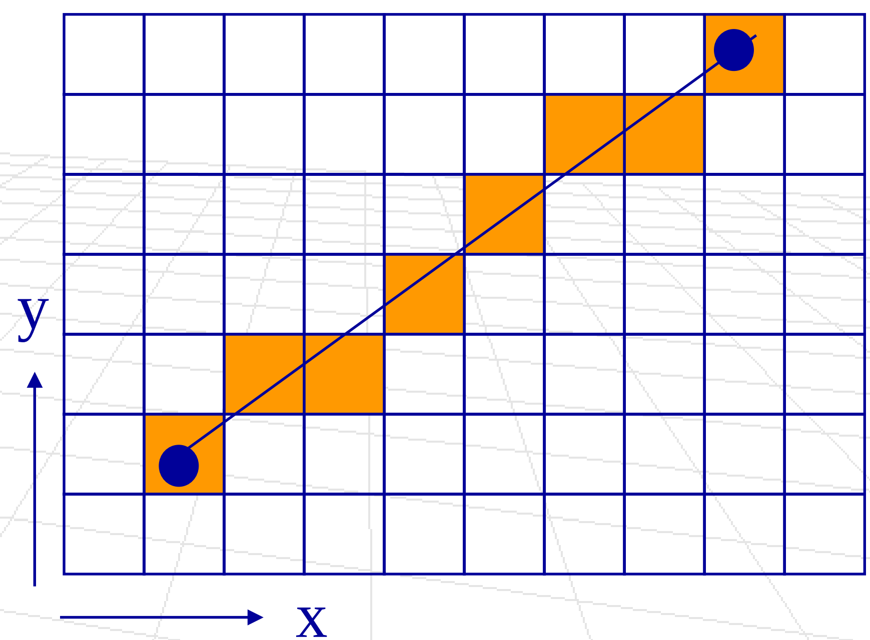
Scan Conversion

- Convert basic CG objects (2D) into corresponding pixelmap representation
- Since objects are often specified using real valued mathematical primitives (e.g. lines, circles, arcs, etc.), often an approximation to object

Continuous line



Digital line



Scan Conversion for Lines

- Set pixels to desired line color to approximate the line from (x_0, y_0) to (x_1, y_1)

- Goals
  - **Accuracy:** pixels should approximate the line as closely as possible
  - **Speed:** line drawing should be as efficient as possible
  - **Visual quality:** uniform brightness
  - **Usability:** independent of point order, independent of the slope

Equation of the Line

$$\mathbf{y} = \mathbf{m}\mathbf{x} + \mathbf{b}$$

- Points that are on the line must satisfy equation above (where **m** = slope, **b** = y-intercept)

$$\mathbf{y}_0 = \mathbf{m}\mathbf{x}_0 + \mathbf{b}$$

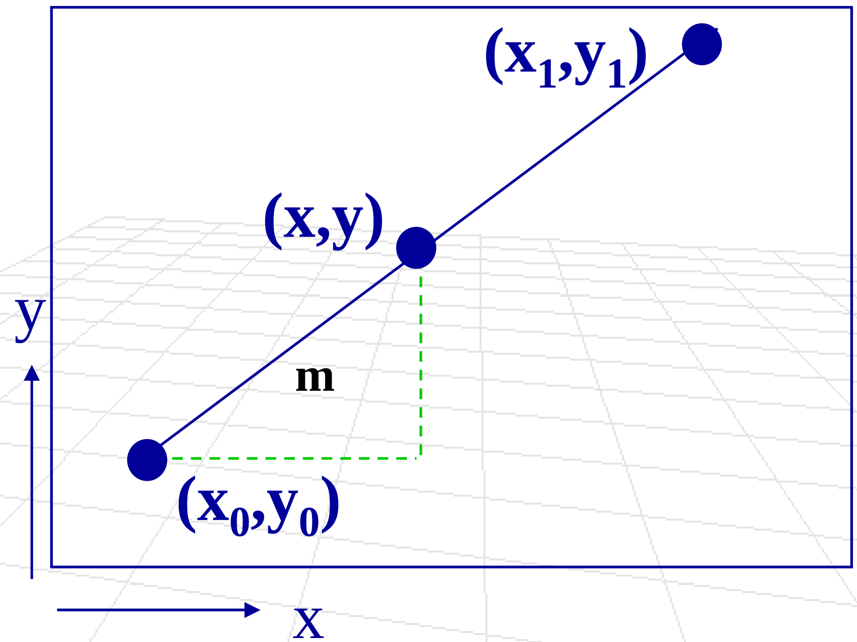
$$\mathbf{y}_1 = \mathbf{m}\mathbf{x}_1 + \mathbf{b}$$

$$\mathbf{m} = \frac{\mathbf{y}_1 - \mathbf{y}_0}{\mathbf{x}_1 - \mathbf{x}_0}$$

$$\mathbf{b} = \mathbf{y}_0 - \mathbf{m}\mathbf{x}_0$$

$$\mathbf{y} = \mathbf{m}(\mathbf{x} - \mathbf{x}_0) + \mathbf{y}_0$$

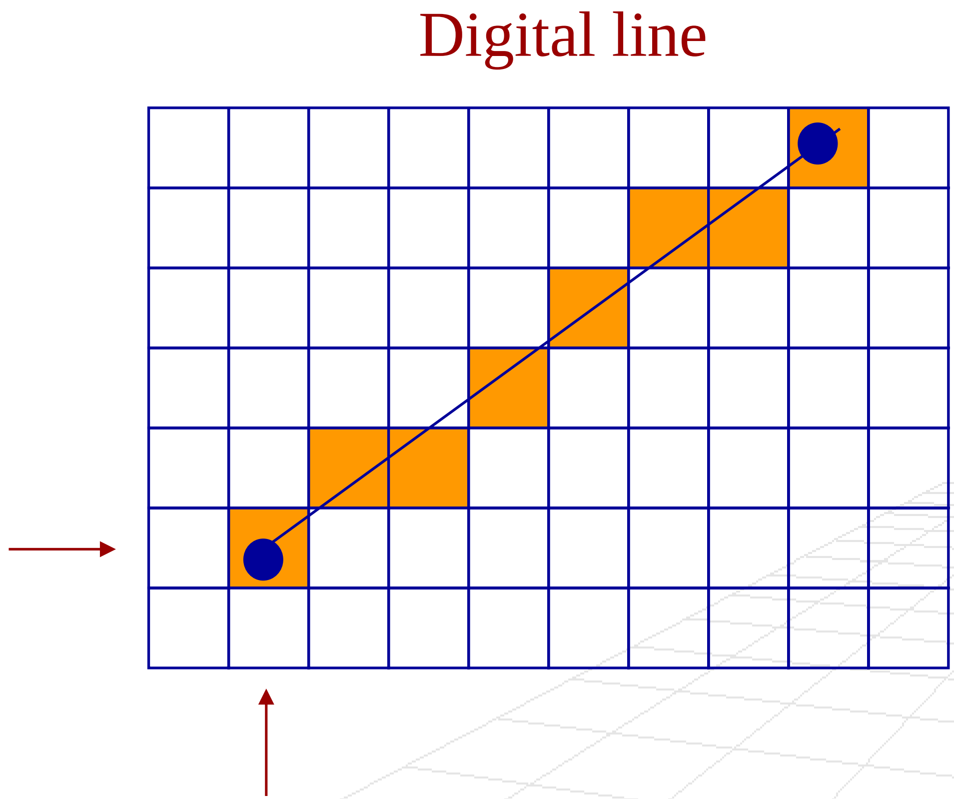
Continuous line



This is the form we'll prefer to use for this lecture

Line Drawing: Basic Idea

- We need to determine the pixels that lie closest to the mathematical line



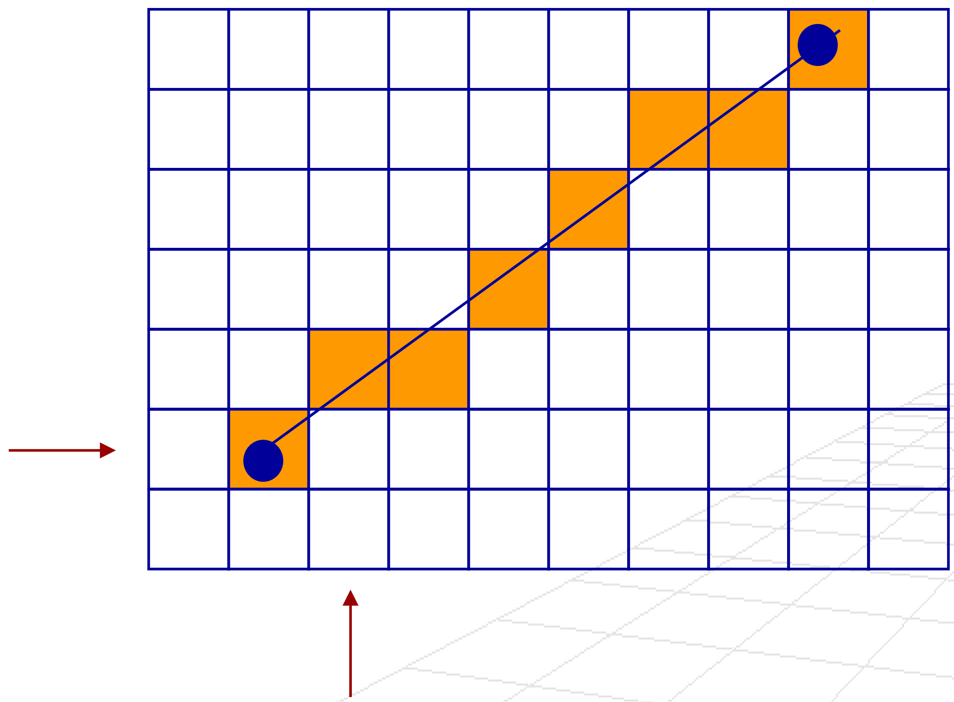
Simple Algorithm

1. Draw pixel at line start

Line Drawing: Basic Idea

- We need to determine the pixels that lie closest to the mathematical line

Digital line

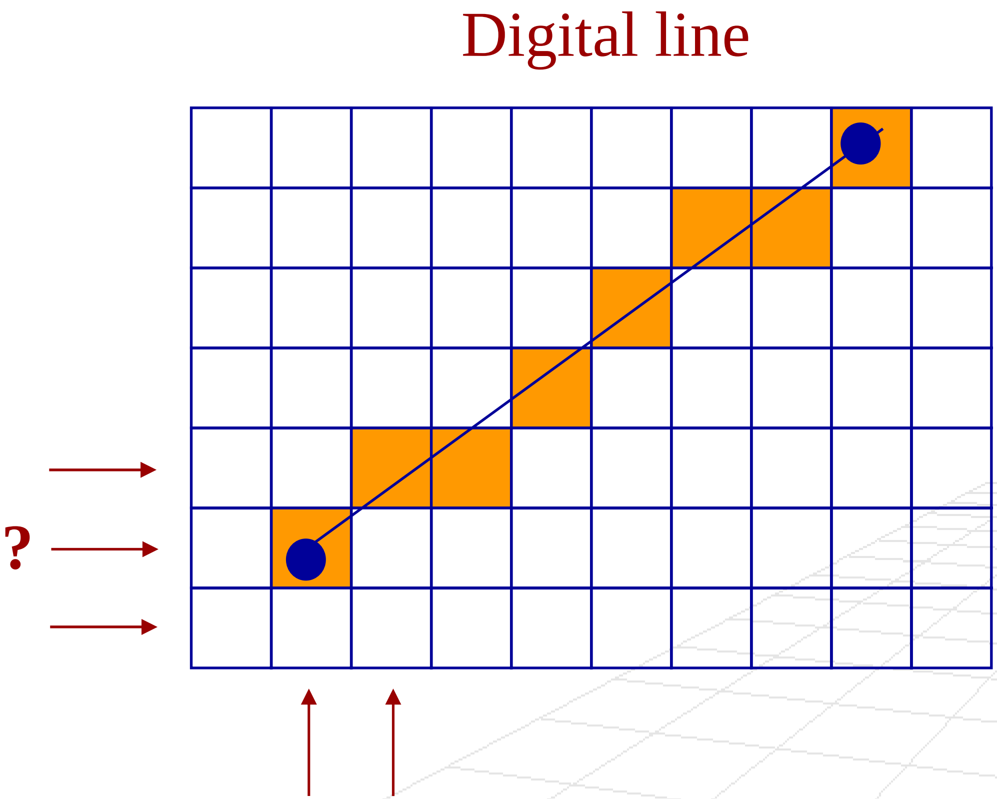


Simple Algorithm

1. Draw pixel at line start
2. Increment x pixel position by 1

Line Drawing: Basic Idea

- We need to determine the pixels that lie closest to the mathematical line



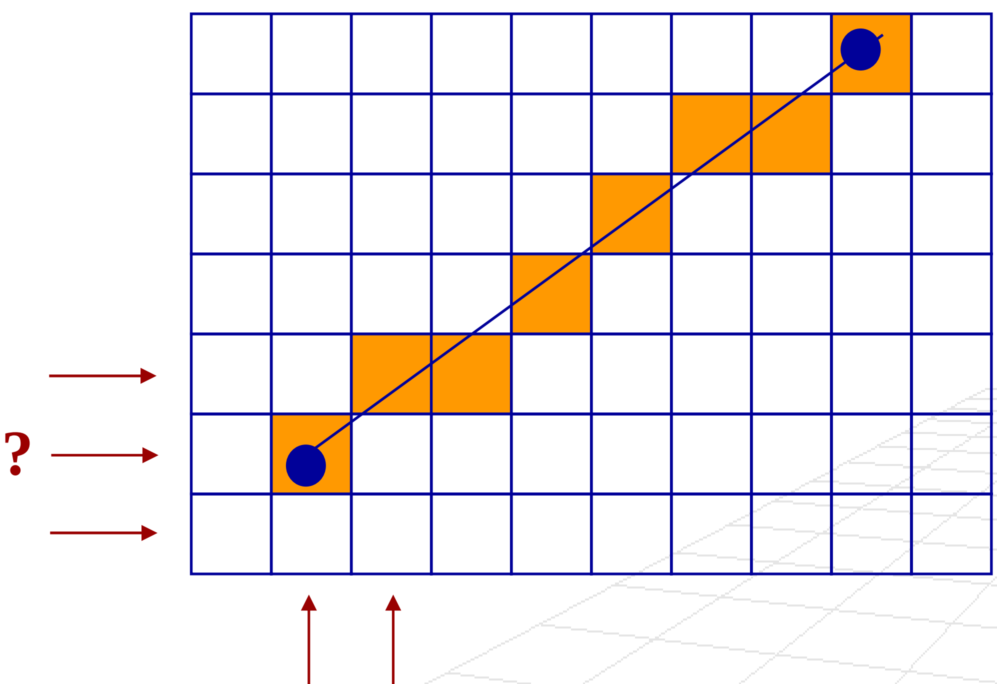
Simple Algorithm

1. Draw pixel at line start
2. Increment x pixel position by 1
3. Determine the y position of the pixel lying closest to the line

Line Drawing: Basic Algoruithm

- We need to determine the pixels that lie closest to the mathematical line

Digital line



Simple Algorithm

compute m
for ($x=x_0$, $x<=x_1$, x++)
    $y = m (x-x_0) + y_0$
    setpixel (x, round(y), c)
end

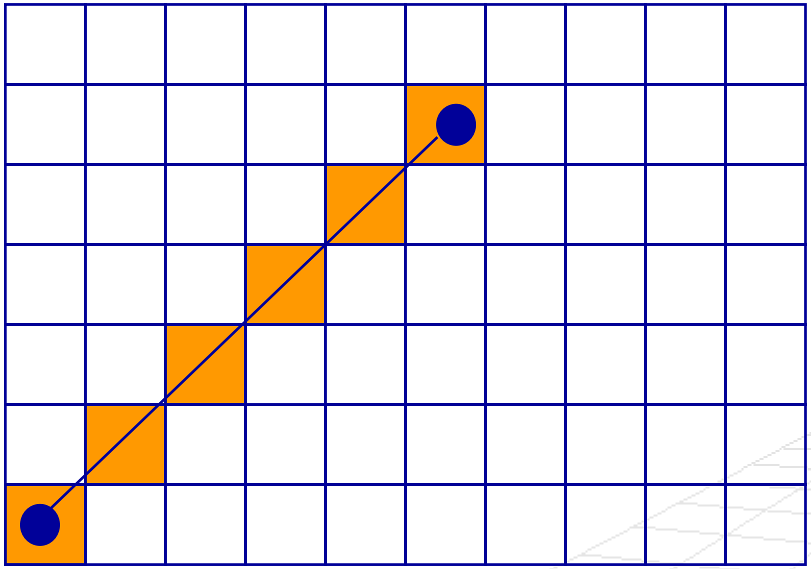
y is real if m is real

Problem: What if points are given in the wrong order?

Solution: Detect ($x_1 < x_0$) and switch order of points

Line Drawing: Basic Algorithm

- Let's test with m = 1

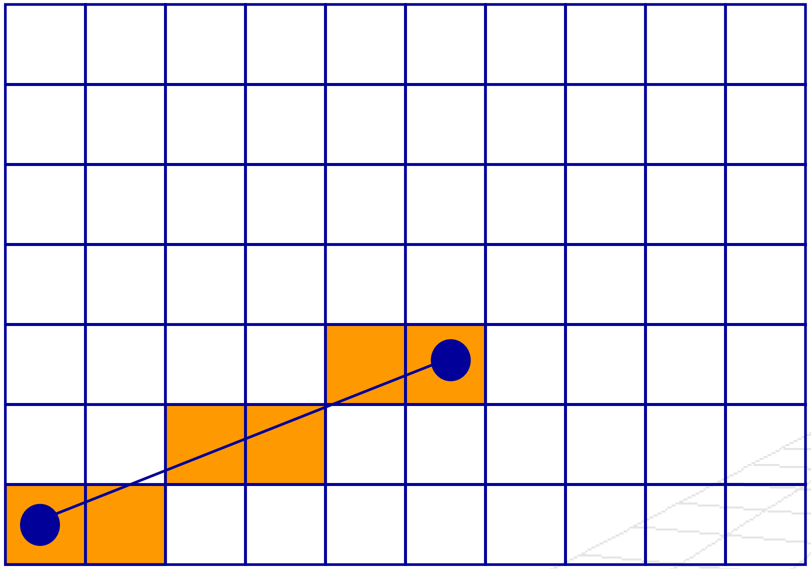


Simple Algorithm

compute m
for $(x=x_0, x<=x_1, x++)$
    $y = m\,(x-x_0) + y_0$
    setpixel (x, round(y), c)
end

Line Drawing: Basic Algorithm

- Let's test with m = 1/2

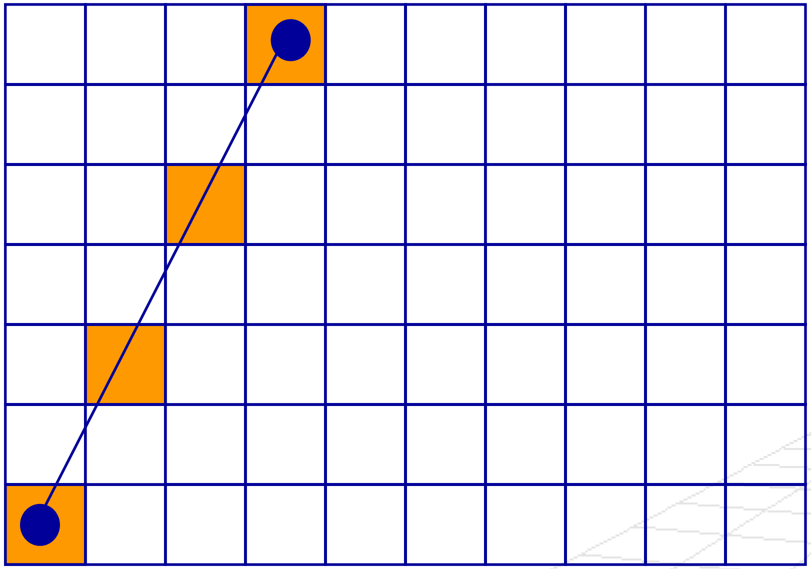


Simple Algorithm

compute m
for ($x=x_0$, $x<=x_1$, x++)
    $y = m\,(x-x_0) + y_0$
    setpixel (x, round(y), c)
end

Line Drawing: Basic Algorithm

- Let's test with $m = 2$



Simple Algorithm

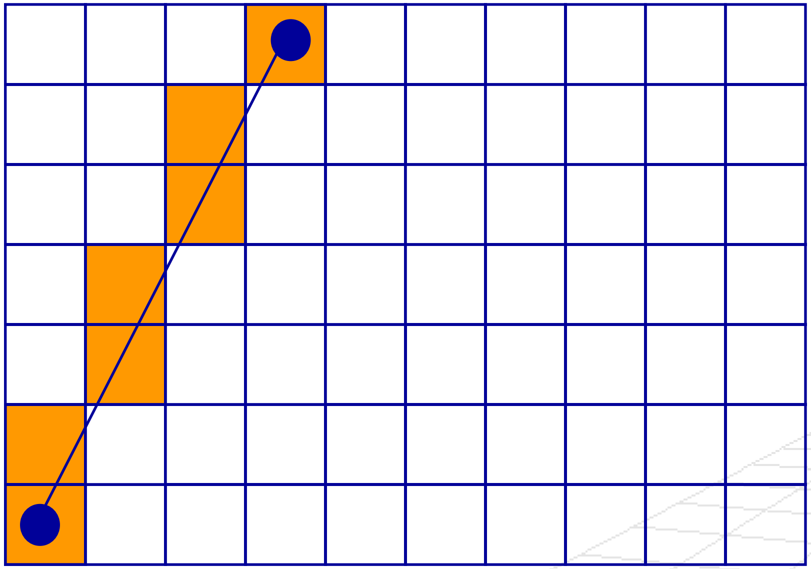
compute m
for $(x=x_0,\ x<=x_1,\ x{+}{+})$
    $y = m\,(x-x_0) + y_0$
    setpixel (x, round(y), c)
end

Problem: **When m > 1**

Solution: **Loop over y instead of x when m > 1**

Line Drawing: Basic Algorithm

- Let's test with $m = 2$



Simple Algorithm (extended)

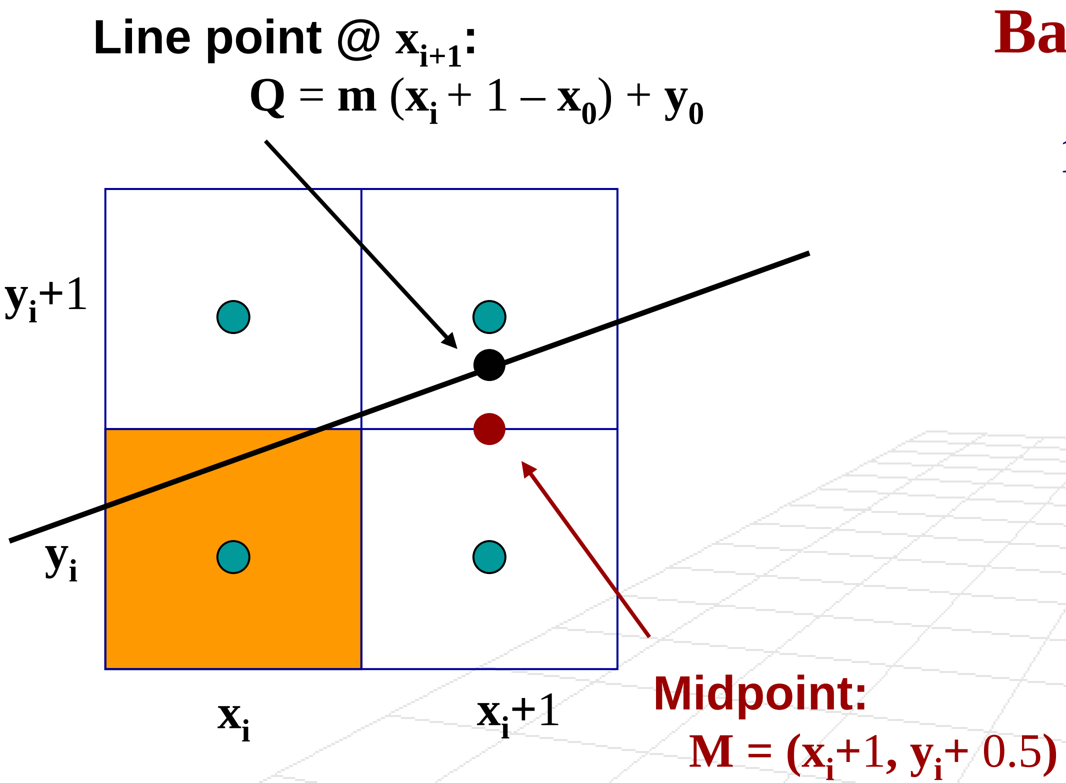
compute m
if (m <= 1)
    for ($x = x_0$, $x <= x_1$, x++)
        $y = m (x-x_0) + y_0$
        setpixel (x, round(y), c)
    end
else
    for ($y = y_0$, $y <= y_1$, y++)
        $x = (y-y_0)/m + x_0$
        setpixel (round(x), y, c)
    end
end

Line Drawing: Basic Algorithm

- **Key disadvantage: inefficiency**
  - relies on floating-point operations to compute pixel positions
  - floating-point operations are slow
- **Alternative: Bresnham's Algorithm**
  - Incremental integer approach

Bresenham's Algorithm

- **Incremental approach**: assume pixel (x_i, y_i) is on how do we tell which pixel to turn on next?



Basic Idea

1. Test if **Q** < is above or bellow **M**

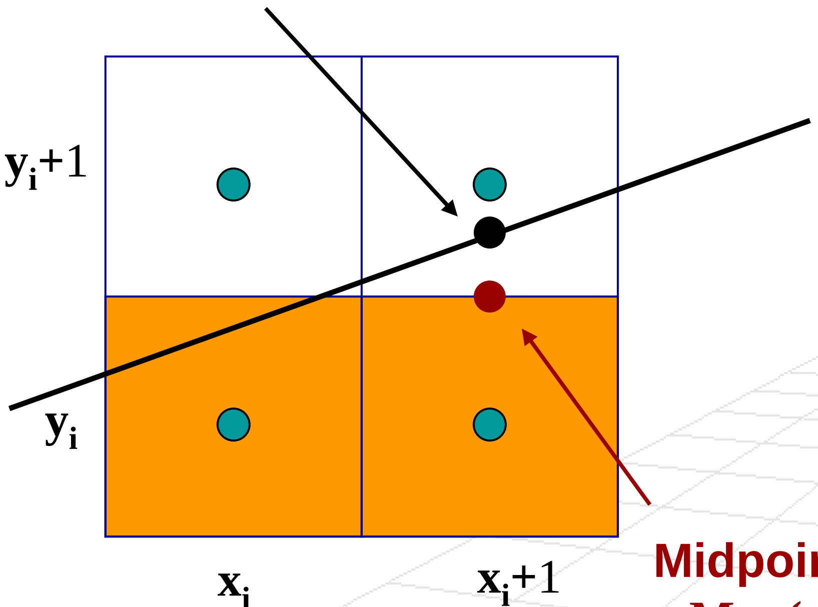
(also known as Midpoint Algorithm)

Bresenham's Algorithm

- **Incremental approach**: assume pixel $(\mathbf{x_i}, \mathbf{y_i})$ is on how do we tell which pixel to turn on next?

Line point @ $\mathbf{x_{i+1}}$:

$$\mathbf{Q} = \mathbf{m}\,(\mathbf{x_i} + 1 - \mathbf{x_0}) + \mathbf{y_0}$$



Midpoint:

$$\mathbf{M} = (\mathbf{x_i}+1, \mathbf{y_i}+ 0.5)$$

Basic Idea

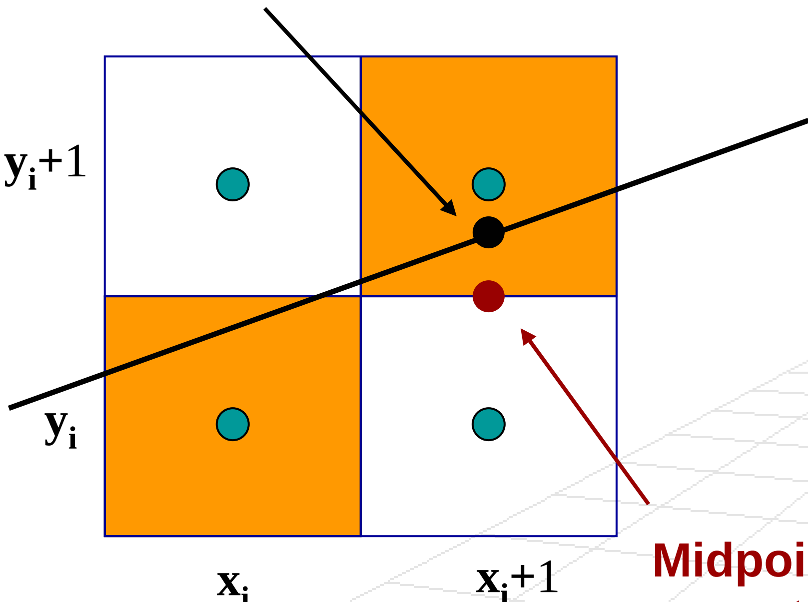
1. Test if **Q** < is above or bellow **M**
2. If **Q** bellow **M** turn on $(\mathbf{x_{i+1}}, \mathbf{y_i})$

(also known as Midpoint Algorithm)

Bresenham's Algorithm

- **Incremental approach**: assume pixel (x_i, y_i) is on how do we tell which pixel to turn on next?

Line point @ x_{i+1}:
$Q = m(x_i + 1 - x_0) + y_0$



Midpoint:
$M = (x_i+1, y_i+ 0.5)$

Basic Idea

1. Test if **Q** < is above or bellow **M**
2. If **Q** bellow **M** turn on (x_{i+1}, y_i)
3. If **Q** above **M** turn on (x_{i+1}, y_{i+1})

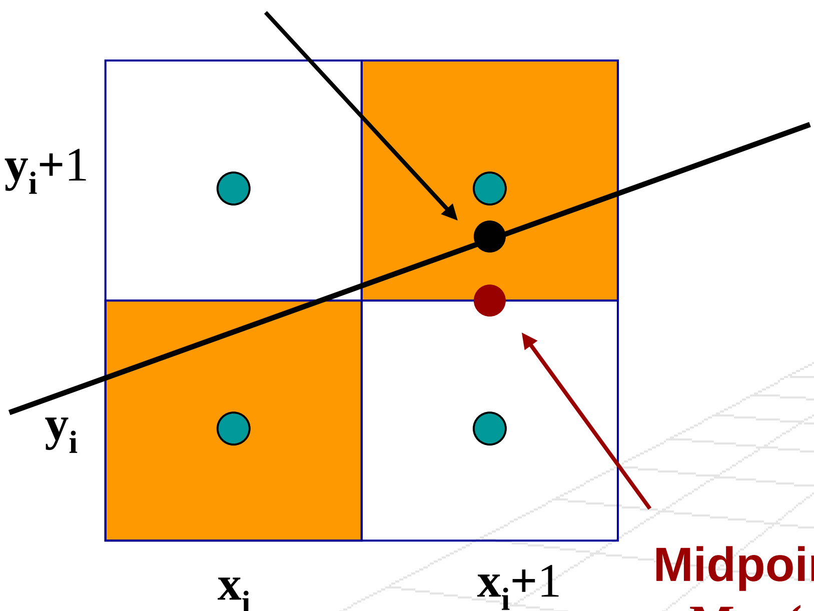
(also known as Midpoint Algorithm)

Bresenham's Algorithm

- **Incremental approach**: assume pixel (x_i, y_i) is on how do we tell which pixel to turn on next?

Line point @ x_{i+1}:

$Q = m(x_i + 1 - x_0) + y_0$



Midpoint:

$M = (x_i+1, y_i+ 0.5)$

Basic Idea

1. Test if **Q** < is above or bellow **M**
2. If **Q** bellow **M** turn on (x_{i+1}, y_i)
3. If **Q** above **M** turn on (x_{i+1}, y_{i+1})
4. Repeat above steps for the newly draw point

Steps guarantee that closest pixel to line is always chosen

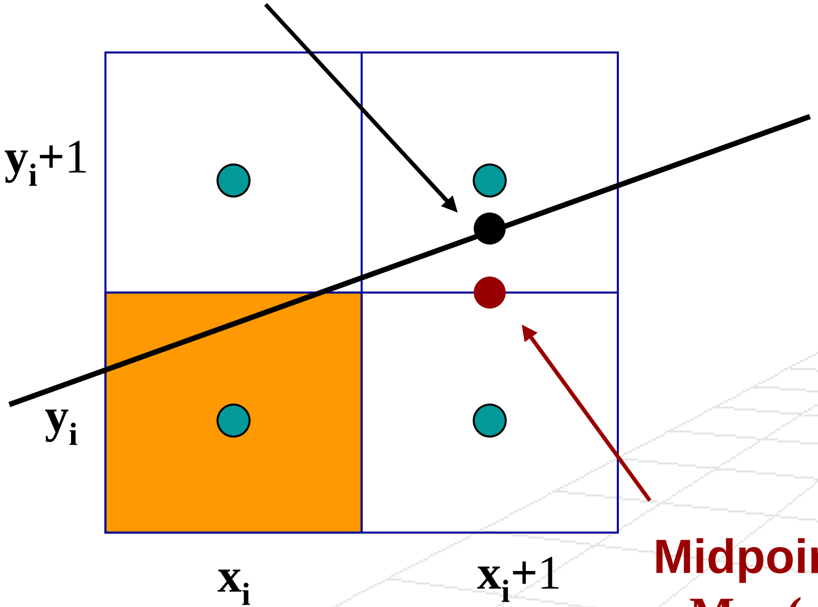
(also known as Midpoint Algorithm)

Bresenham's Algorithm

- **Incremental approach**: assume pixel (x_i, y_i) is on how do we tell which pixel to turn on next?

Line point @ x_{i+1}:

$Q = m\,(x_i + 1 - x_0) + y_0$



Midpoint:

$M = (x_i+1,\ y_i+ 0.5)$

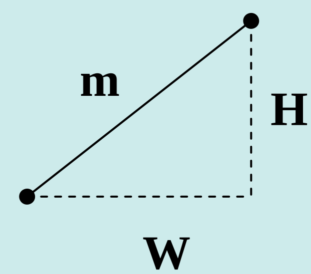
Basic Idea

How do we decide if **Q** is above or bellow **M** ?

Look at the implicit function of the line f(x,y)

Implicit function of the line

$$H = y_1 - y_0$$
$$W = x_1 - x_0$$



$$y = m(x - x_0) + y_0$$

$$y = \frac{H}{W}(x - x_0) + y_0$$

$$Wy = H(x - x_0) + Wy_0$$

$$f(x, y) = 0 = H(x - x_0) + W(y_0 - y)$$

$$f(x, y) = 0 = 2H(x - x_0) - 2W(y - y_0)$$

If $f(x,y) = 0$ then (x,y) on the line

If $f(x,y) < 0$ then (x,y) above line

If $f(x,y) > 0$ then (x,y) bellow line

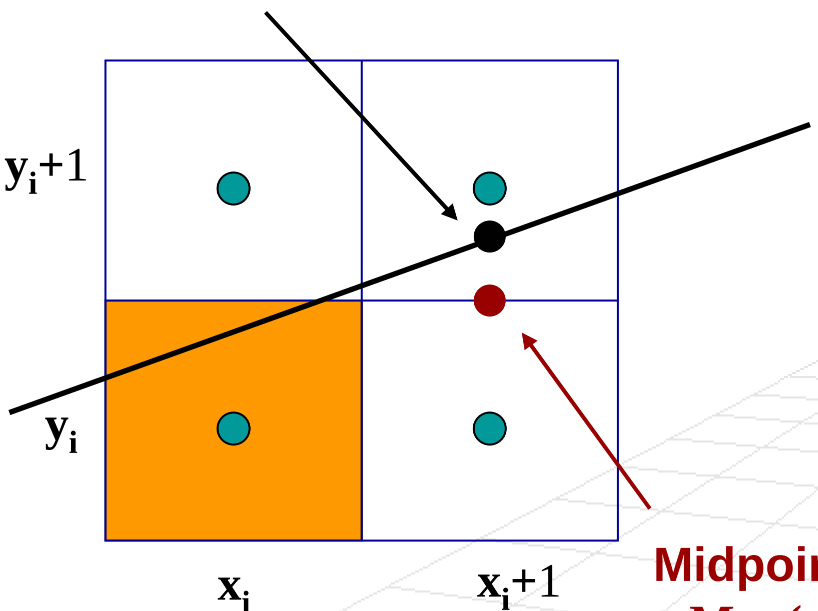
Exercise for home, show that indeed

Bresenham's Algorithm

- **Incremental approach**: assume pixel (x_i, y_i) is on how do we tell which pixel to turn on next?

Line point @ x_{i+1}:

$Q = y = m(x_i + 1 - x_0) + y_0$



Midpoint:

$M = (x_i+1, y_i+ 0.5)$

Basic Idea

How do we decide if **Q** is above or bellow **M** ?

Look at the implicit function of the line **f(x,y)** @ the midpoint

If $f(x_i+1, y_i+0.5) < 0$ then $x_j = x_i+1$, $y_j = y_i$

If $f(x_i+1, y_i+0.5) < 0$ then $x_j = x_i+1$, $y_j = y_i+1$

Now, why did we multiply by 2?

$$\mathbf{f}(\mathbf{x},\mathbf{y}) = 0 = 2\mathbf{H}(\mathbf{x}-\mathbf{x}_0) - 2\mathbf{W}(\mathbf{y}-\mathbf{y}_0)$$

$$\mathbf{f}(\mathbf{x_i}+1,\mathbf{y_i}+0.5) = 0 = 2\mathbf{H}(\mathbf{x_i}+1-\mathbf{x}_0) - 2\mathbf{W}(\mathbf{y_i}+0.5-\mathbf{y}_0)$$

Now this computation can be done in terms of integers

Now, why did we multiply by 2?

$$\mathbf{f}(\mathbf{x},\mathbf{y}) = 0 = 2\mathbf{H}(\mathbf{x}-\mathbf{x}_0) - 2\mathbf{W}(\mathbf{y}-\mathbf{y}_0)$$

$$\mathbf{f}(\mathbf{x}_\mathbf{i}+1,\mathbf{y}_\mathbf{i}+0.5) = 0 = 2\mathbf{H}(\mathbf{x}_\mathbf{i}+1-\mathbf{x}_0) - 2\mathbf{W}(\mathbf{y}_\mathbf{i}+0.5-\mathbf{y}_0)$$

- Note, that we only need to keep track of **f(x,y)** at the mid points, which can be done efficiently incrementally

$$\mathbf{f}(\mathbf{x}+1,\mathbf{y}) = \mathbf{f}(\mathbf{x},\mathbf{y}) + 2\mathbf{H}$$

$$\mathbf{f}(\mathbf{x}+1,\mathbf{y}+1) = \mathbf{f}(\mathbf{x},\mathbf{y}) + 2(\mathbf{H}-\mathbf{W})$$

Now, why did we multiply by 2?

$$\mathbf{f}(\mathbf{x},\mathbf{y}) = 0 = 2\mathbf{H}(\mathbf{x}-\mathbf{x}_0) - 2\mathbf{W}(\mathbf{y}-\mathbf{y}_0)$$

$$\mathbf{f}(\mathbf{x}_\mathbf{i}+1,\mathbf{y}_\mathbf{i}+0.5) = 0 = 2\mathbf{H}(\mathbf{x}_\mathbf{i}+1-\mathbf{x}_0) - 2\mathbf{W}(\mathbf{y}_\mathbf{i}+0.5-\mathbf{y}_0)$$

- Note, that we only need to keep track of **f(x,y)** at the mid points, which can be done efficiently incrementally

Very Efficient

$$\mathbf{f}(\mathbf{x}+1,\mathbf{y}) = \mathbf{f}(\mathbf{x},\mathbf{y}) + 2\mathbf{H}$$

$$\mathbf{f}(\mathbf{x}+1,\mathbf{y}+1) = \mathbf{f}(\mathbf{x},\mathbf{y}) + 2(\mathbf{H}-\mathbf{W})$$

Bresenham's Algorithm

Note, initially $f(x_0,y_0)=0$, so first test is @ $f(x_0+1, y_0+0.5)$

$$\mathbf{f}(\mathbf{x}_0 + 1, \mathbf{y}_0 + 0.5) = 2\mathbf{H}(\mathbf{x}_0 + 1 - \mathbf{x}_0) - 2\mathbf{W}(\mathbf{y}_0 + 0.5 - \mathbf{y}_0) = 2\mathbf{H} - \mathbf{W}$$

```
y = y0
H = y1 – y0
W = x1 – x0
f =  2H – W

for (x = x0, x <= x1, x++)
     setpixel (x, y, c)
     if (f <0)
          f += 2H      // y stays the same
     else
          y++          // y increases
          f += 2(H-W)
     end
end
```

Bresenham's Algorithm

- **Limitations:** same as the basic line drawing, the Bresenham's algorithm in the last slide only works for $\mathbf{m} < \mathbf{1}$ and has to be altered for a more general cases

- To make it general need to (as in the basic line drawing algorithm)
  - Switch order of points if necessary
  - Iterate over y if $\mathbf{m} > \mathbf{1}$

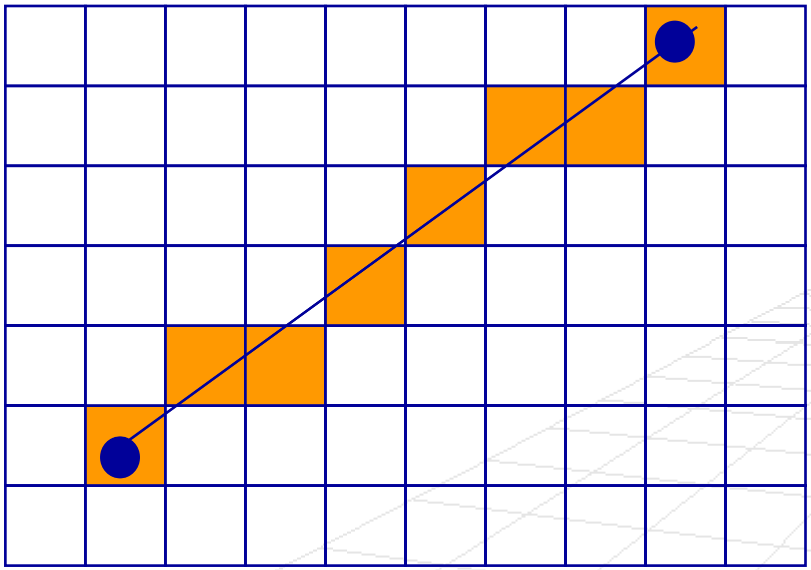
Aliasing

- An unfortunate artifact of the line scan conversion discussed is that lines have “jaggy” appearance
- This phenomenon is called aliasing

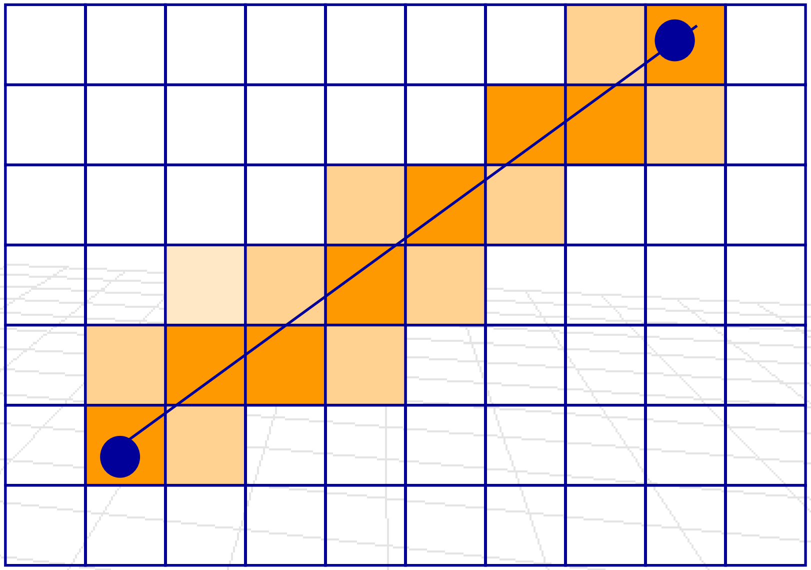
Anti-aliasing

- **Main idea:** rather than just drawing in 0’s and 1’s, use “in-between” values in neighborhood of the mathematical line

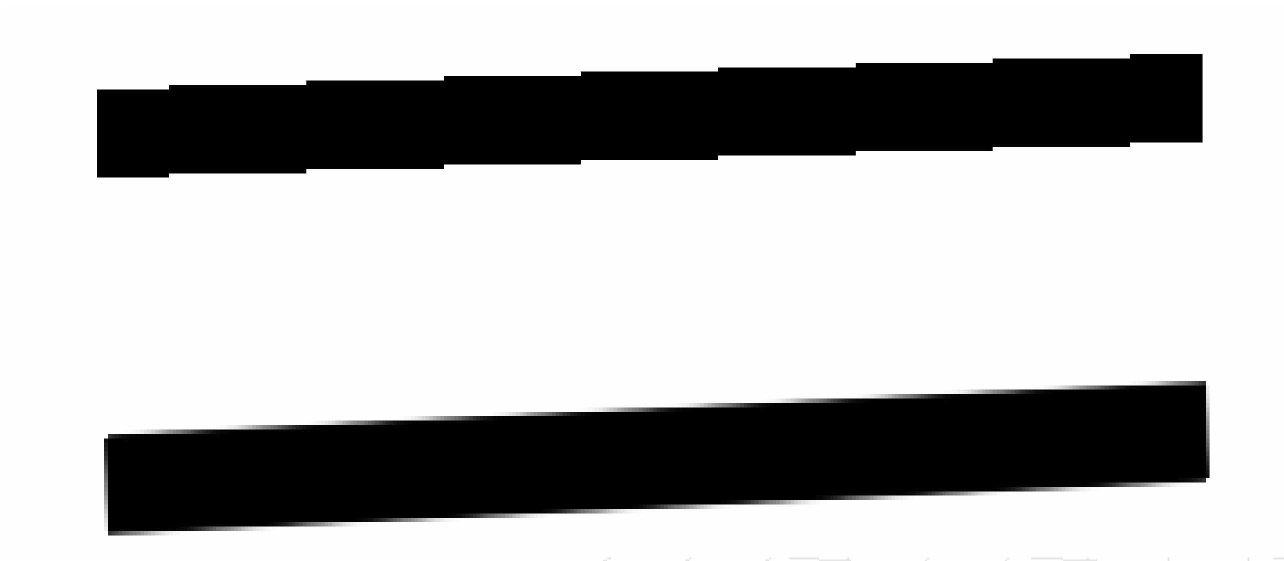
Aliased line



Anti-aliased line



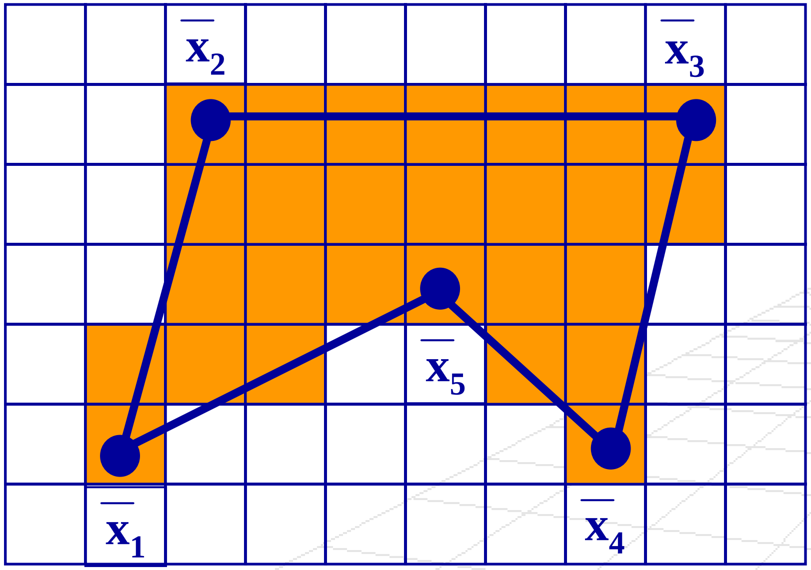
Anti-aliasing Comparison



Polygon Filling – Scan Conversion

- **Goal:** find pixels that occupy inside of the polygon and fill them with a given color

Polygon Filling

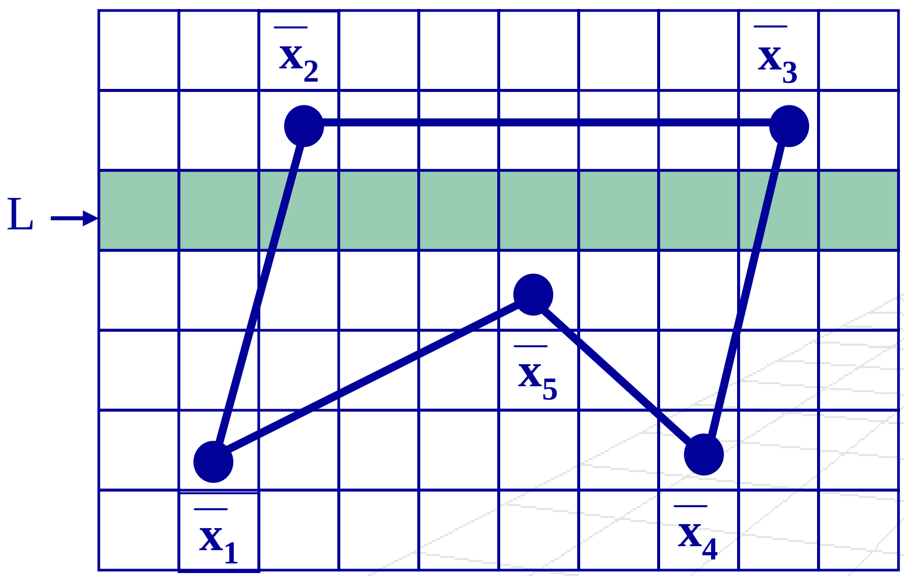


$$\mathbf{P} = (\bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2, \bar{\mathbf{x}}_3, \bar{\mathbf{x}}_4, \bar{\mathbf{x}}_5)$$

Polygon Filling Idea

- **Goal:** find pixels that occupy inside of the polygon and fill them with a given color

Polygon Filling



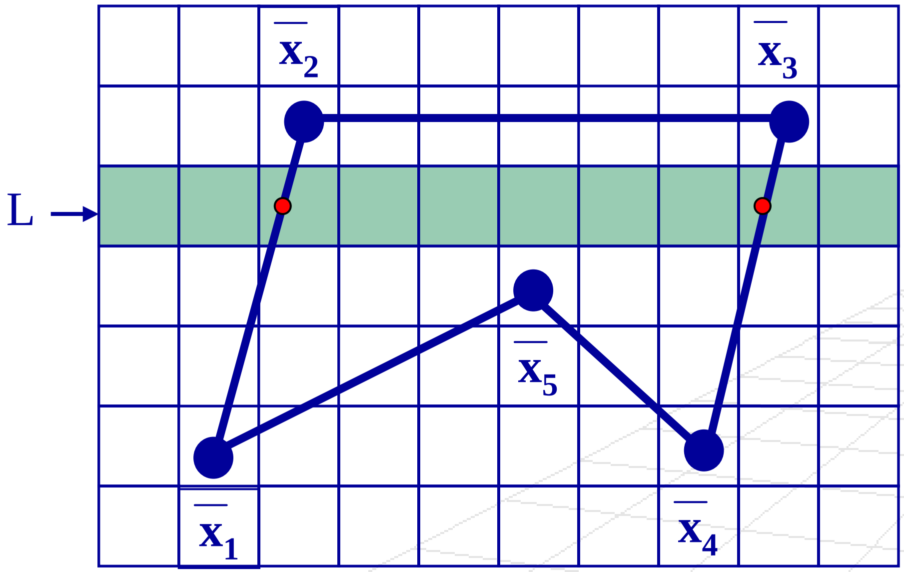
Simple Idea

For each horizontal scanline L

Polygon Filling Idea

- **Goal:** find pixels that occupy inside of the polygon and fill them with a given color

Polygon Filling



Simple Idea

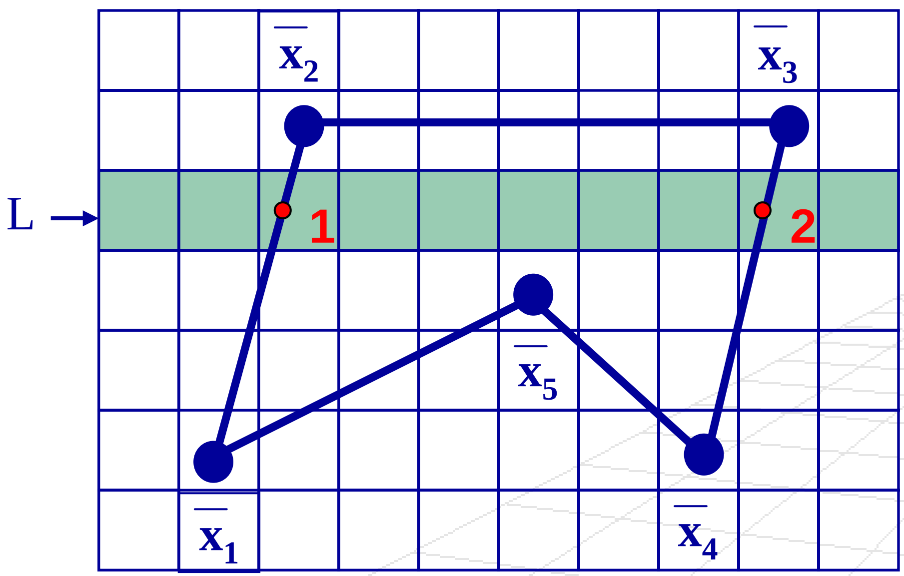
For each horizontal scanline L

1. Find intersection of L with P (store in active edge list AEL)

Polygon Filling Idea

- **Goal:** find pixels that occupy inside of the polygon and fill them with a given color

Polygon Filling



Simple Idea

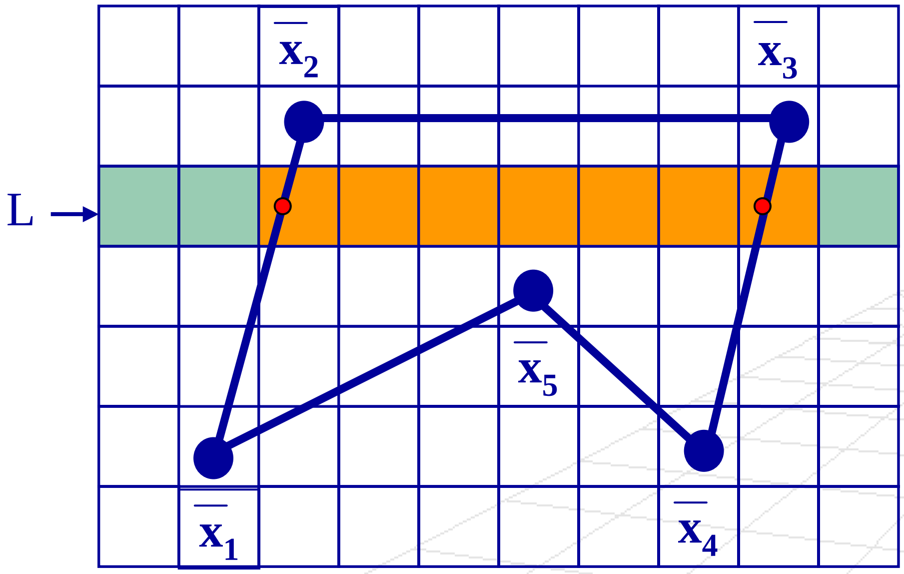
For each horizontal scanline L

1. Find intersection of L with P (store in active edge list AEL)
2. Sort intersections by increasing value of x

Polygon Filling Idea

- **Goal:** find pixels that occupy inside of the polygon and fill them with a given color

Polygon Filling



Simple Idea

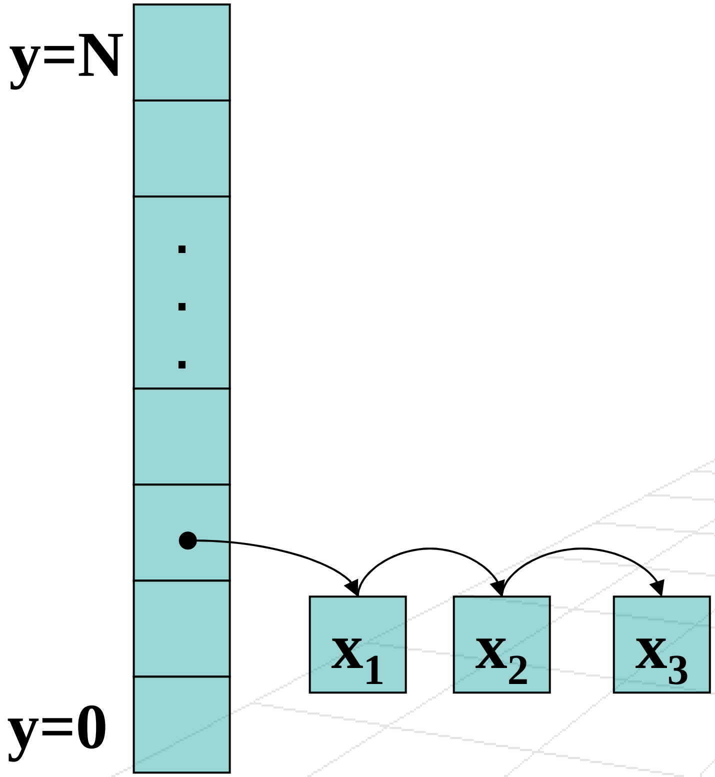
For each horizontal scanline L

1. Find intersection of L with P (store in active edge list AEL)
2. Sort intersections by increasing value of x
3. Fill pixels between pairs of intersections

Polygon Filling Algorithm

Algorithm

Active Edge List (AEL)



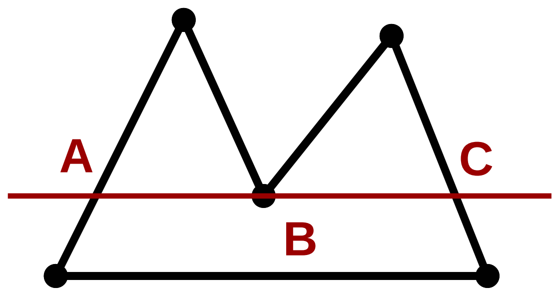
for each edge $[(x_0, y_0), (x_1,y_1)]$ in P

    x=x0;
    compute m
    for (y=y0, y<=y1, y++)
        x = x + 1/m
        place (round(x), y) in AEL
    end

end

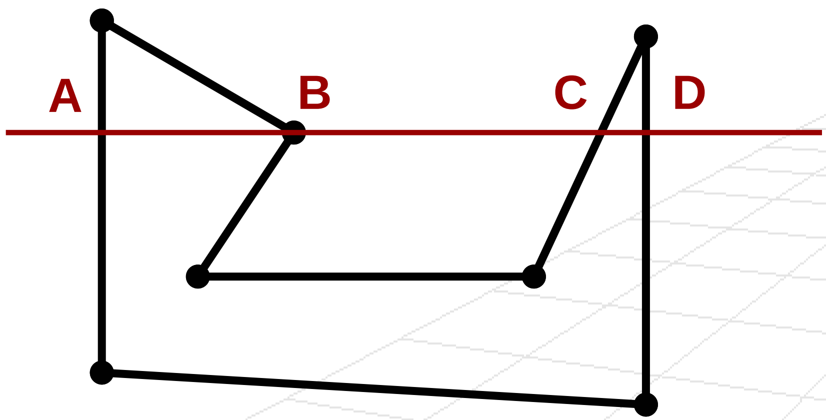
Does this remind you of anything?

Polygon Filling – Special Cases



Intersections:

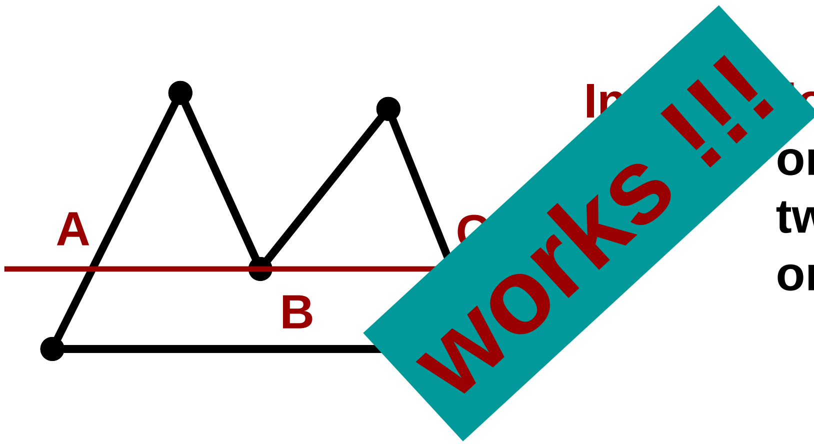
one @ A
two @ B
one @ C



Intersections:

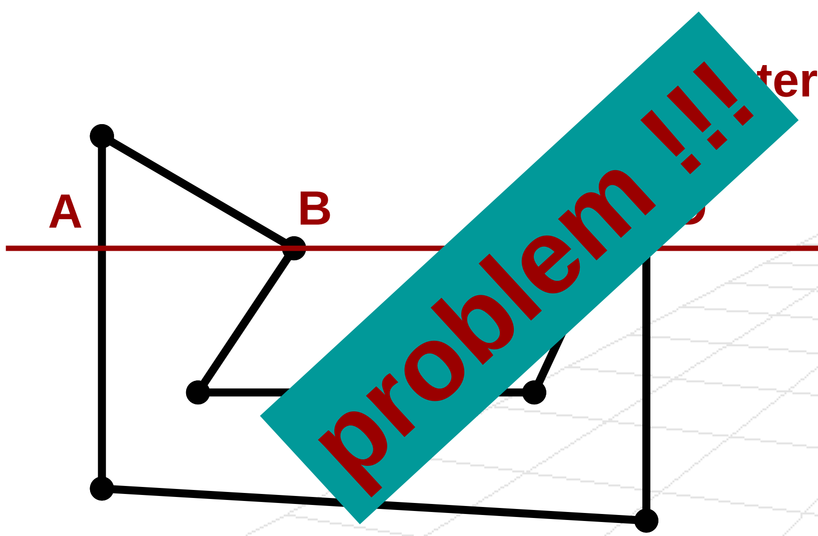
one @ A
two @ B
one @ C
one @ D

Polygon Filling – Special Cases



Intersections:

one @ A
two @ B
one @ C



Intersections:

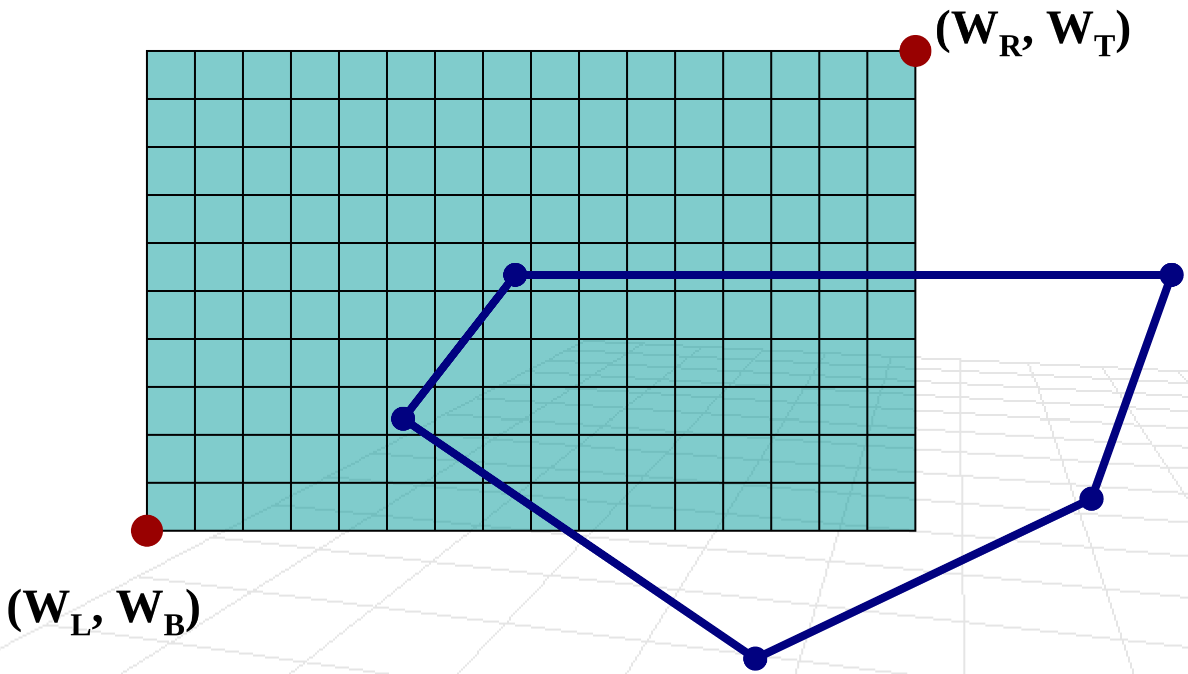
one @ A
two @ B
one @ C
one @ D

Polygon Filling – Handling Special Cases

- All problems can be handled by 2 simple rules
  - only rasterize edges (not intersections)
  - Ignore horizontal edges

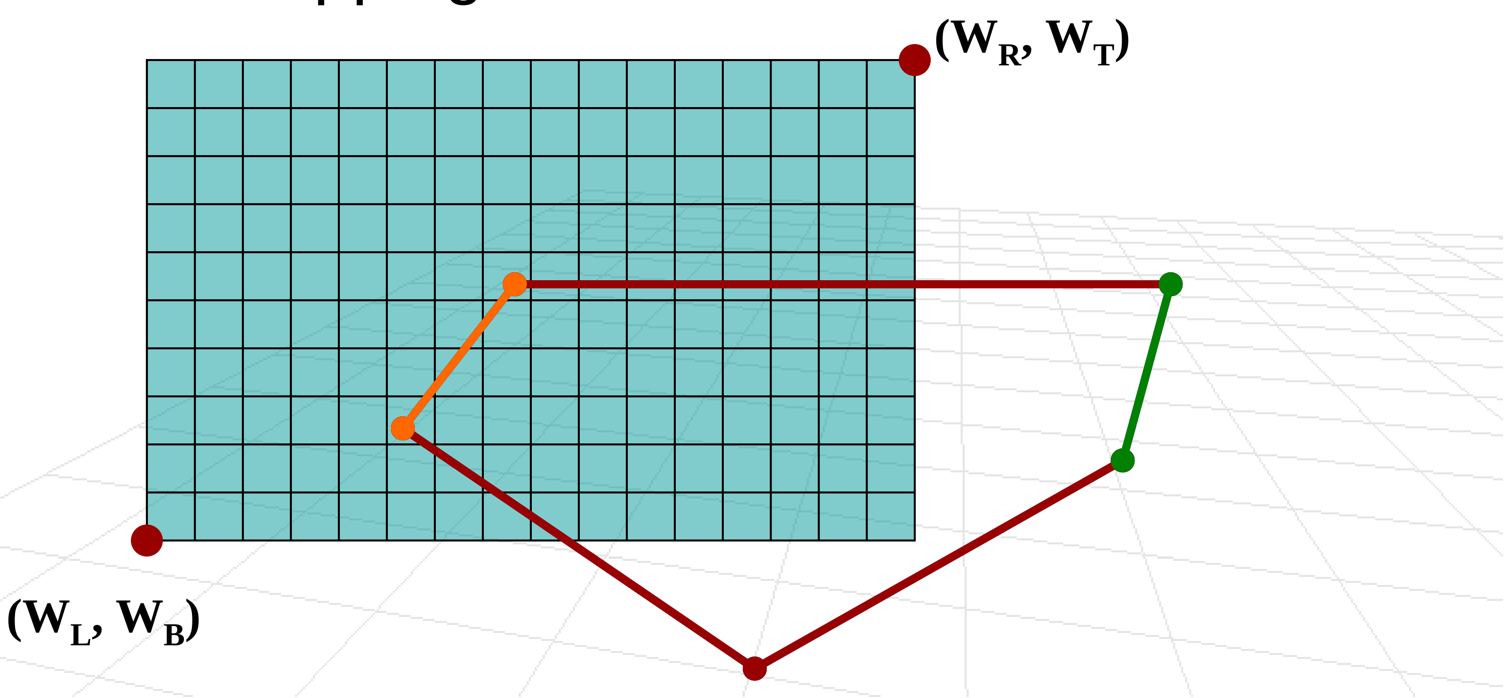
Clipping

- **Clipping:** used to determine which parts of the line/polygon lie inside viewing window
  - allows efficient rendering and rastering



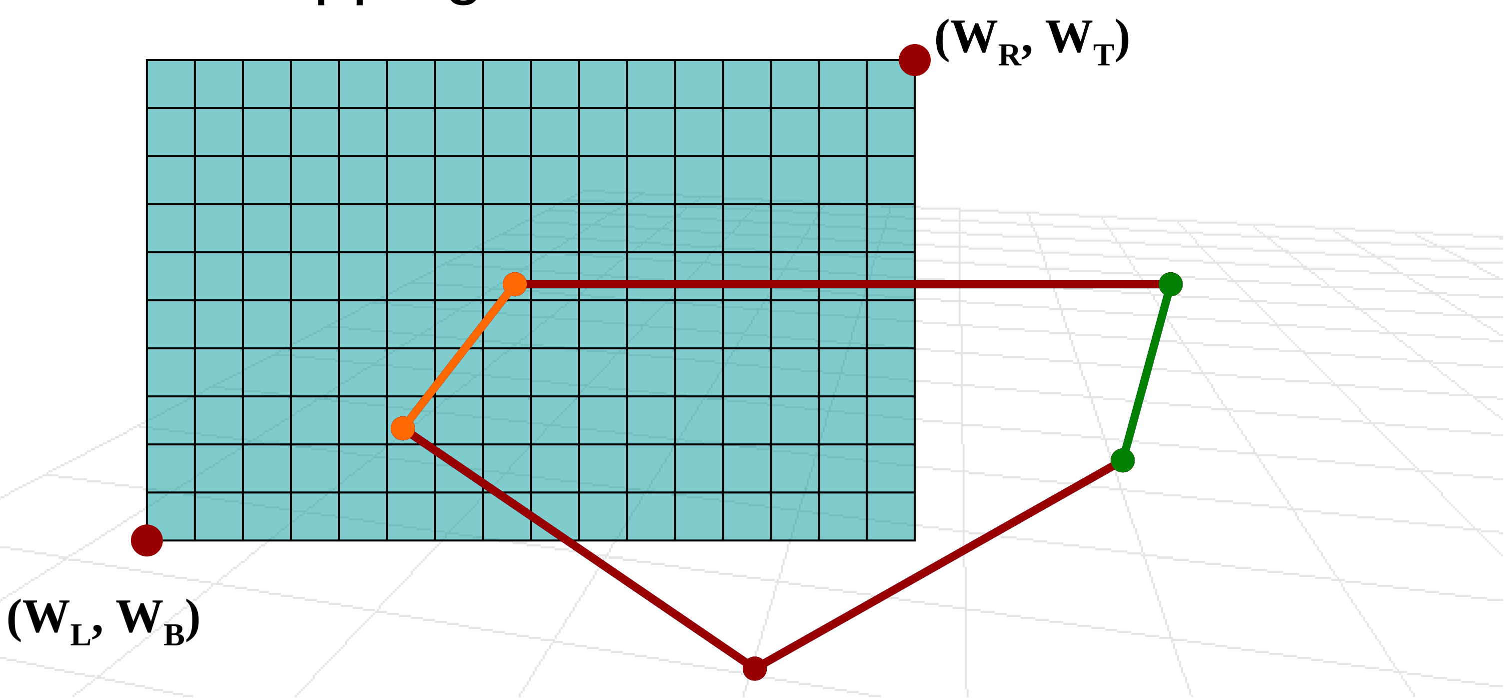
Clipping Algorithm

- Every line segment of the polygon is either
  - trivially inside (both endpoints lie inside the window)
  - trivially outside (both endpoints lie outside of one of the half-spaces that define the window)
  - candidate for clipping



Clipping Algorithm

- Every line segment of the polygon is either
  - trivially inside (both endpoints lie inside the window) ✓ **Keep**
  - trivially outside (both endpoints lie outside of one of the half-spaces that define the window) ✗ **Remove**
  - candidate for clipping



Clipping Algorithm

- Every line segment of the polygon is either
  - trivially inside (both endpoints lie inside the window) ✓ **Keep**
  - trivially outside (both endpoints lie outside of one of the half-spaces that define the window) ✗ **Remove**
  - candidate for clipping
    - Find the intersection with the window (if exists)
    - Disregard irrelevant part of the segment