Lecture #3a

Rasterization

Computer Graphics

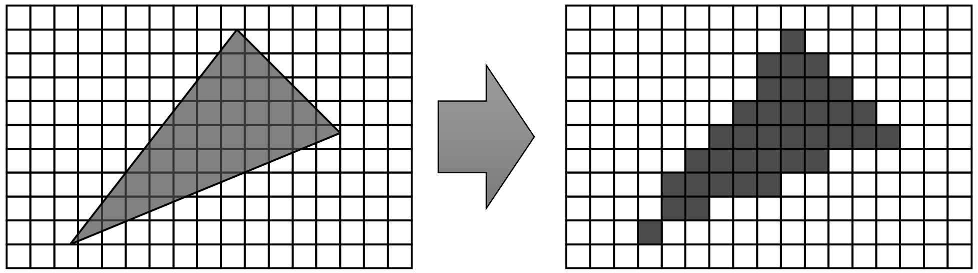
Winter Term 2016/17

Marc Stamminger / Roberto Grosso

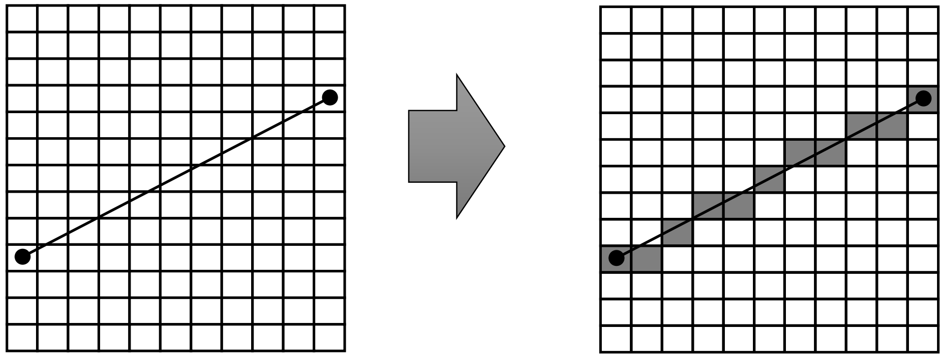
What is Rasterization ?

- Given a primitive, find the pixels that cover this primitive

- Triangle primitive:



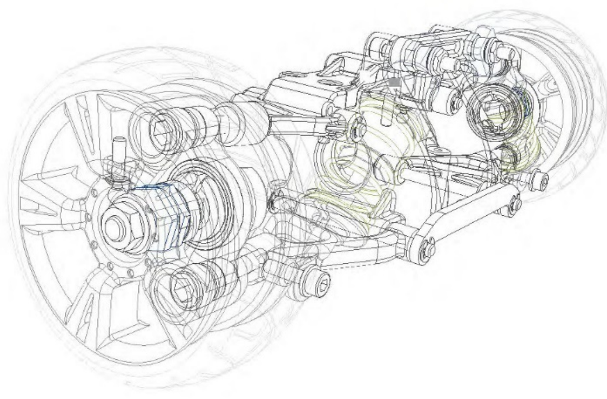
- Line primitive:



Rasterization - Primitives

- Which primitives are of interest ?

- **Lines:**
  - very widely used in CAD (computer aided design) → wireframe models

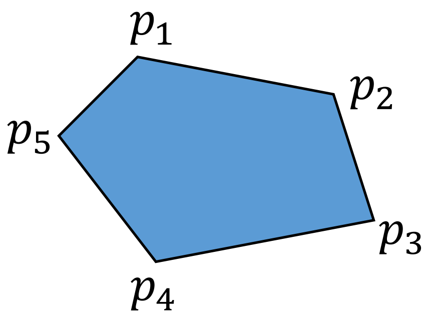


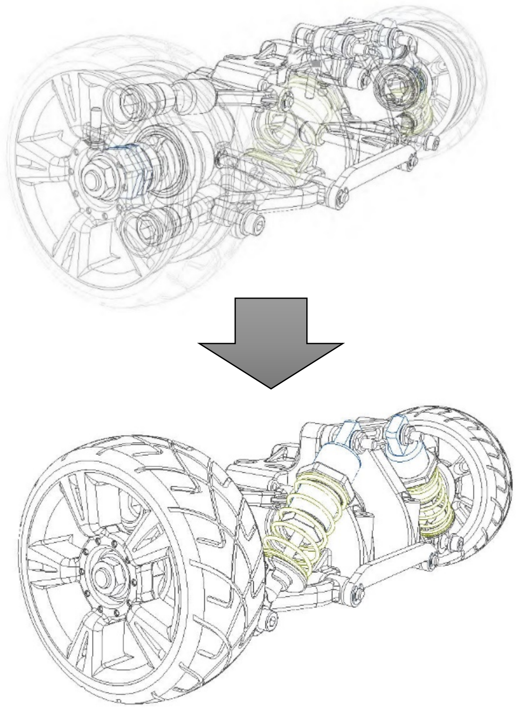
autodesk.com

  - every curve can be approximated by lines

Rasterization - Primitives

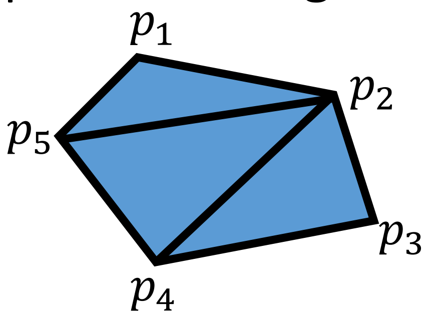
- mostly, we want to **fill** objects → **polygons**
- A **polygon** is defined by an ordered set of points (for now in 2D)





autodesk.com

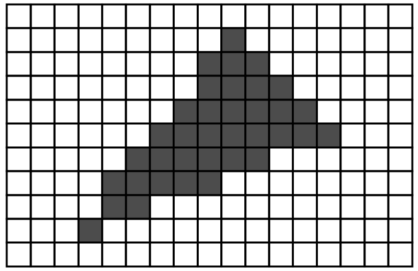
- Every shape can be approximated by a polygon
- Every polygon can be split into **triangles** = **Triangulation**



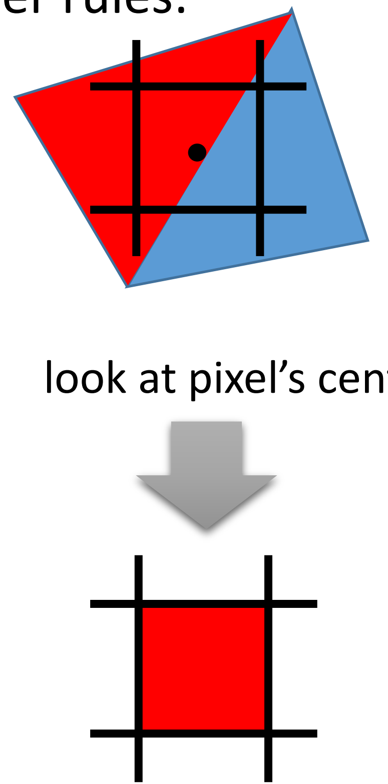
autodesk.com

Rasterization – Aliasing and Antialiasing

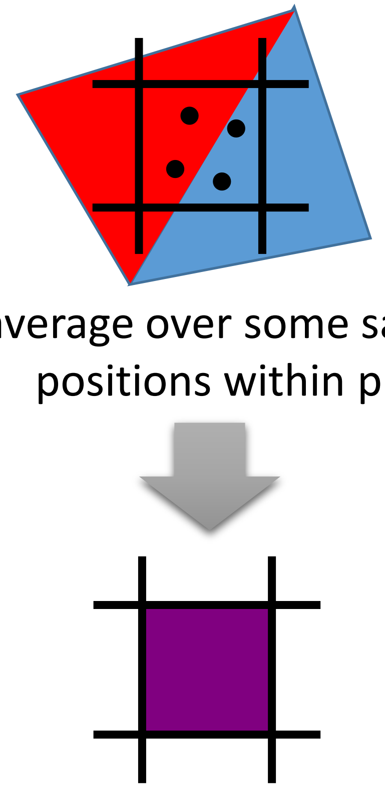
- Simple rasterization rule: set pixel if its center is inside the shape
  → strong jaggies, well visible
  → this is one form of **Aliasing**
  → we will come back to aliasing later



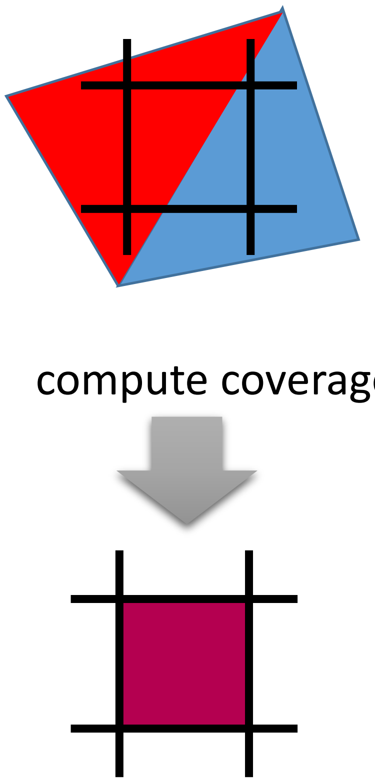
- Other rules:



look at pixel's center



average over some sample positions within pixel



compute coverage

Rasterization – Aliasing and Antialiasing

- Good renderers have more sophisticated rasterization rules
  → look at results of HTML Canvas renderers:

- Even with good renderers aliasing effects remain!
  (see “draw triangles” and “draw stripes” examples)

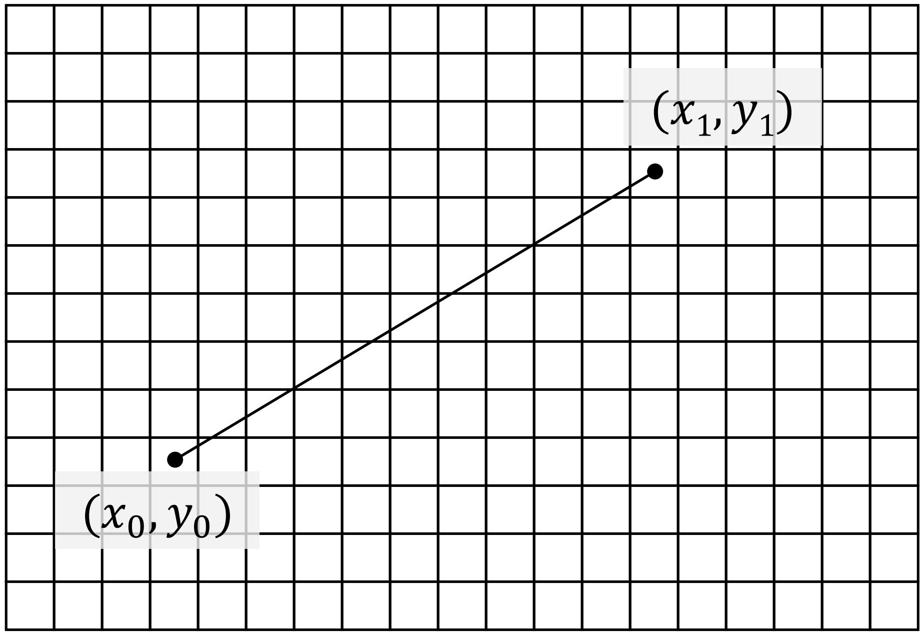
Rasterization

- This lecture:
    - Line Rasterization (+ circles)
    - Filling of boundaries

- Next lecture
    - Direct Polygon / Triangle Rasterization

Line Drawing

- Line Rasterization
  - Given: Segment endpoints (integers $(x_0, y_0), (x_1, y_1)$)
  - Identify: Set of pixels (x, y) to display for segment

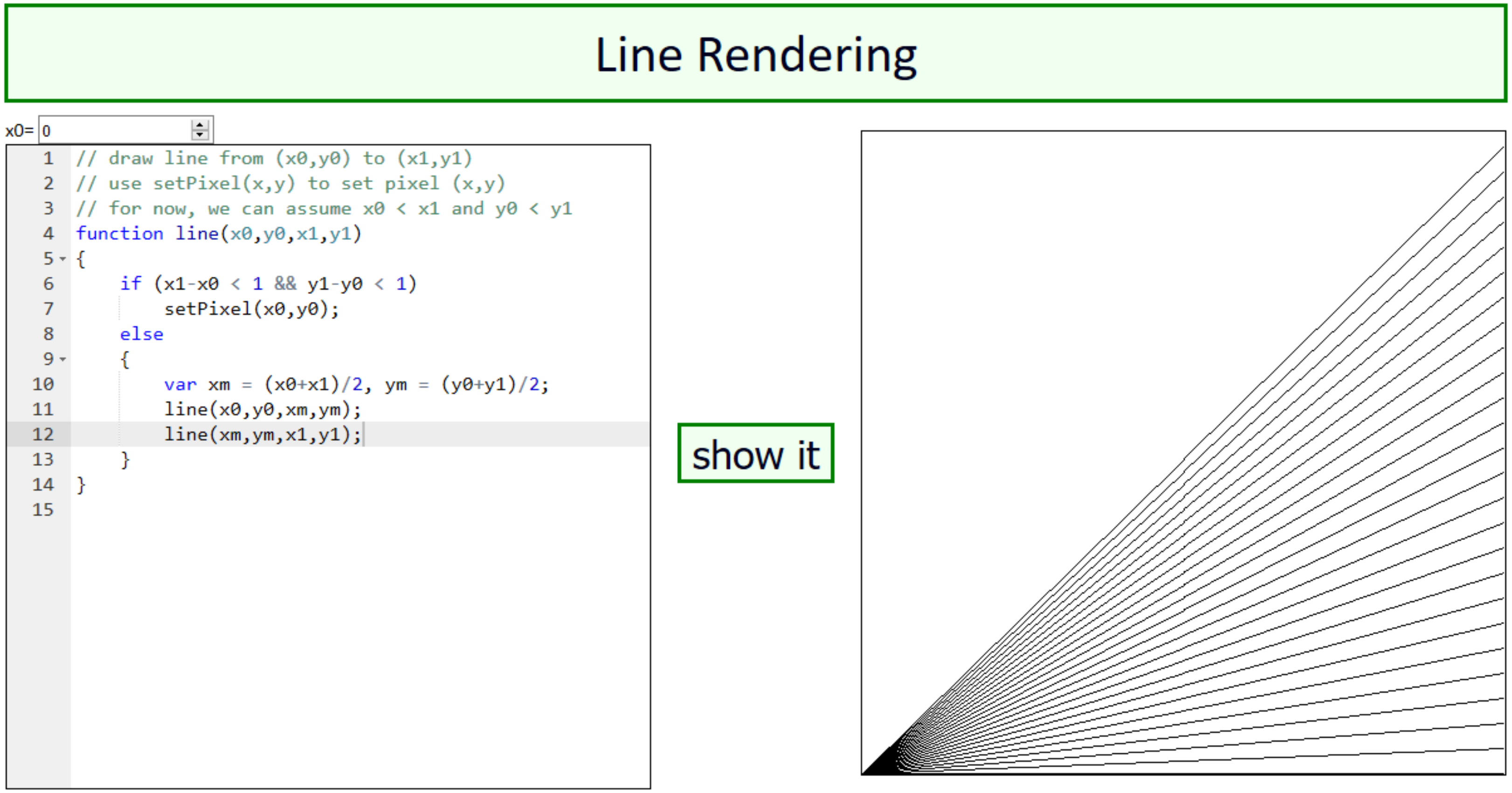


Line Drawing

- Let’s play – line rendering

Line Drawing

- A recursive line rasterizer


Line Rendering

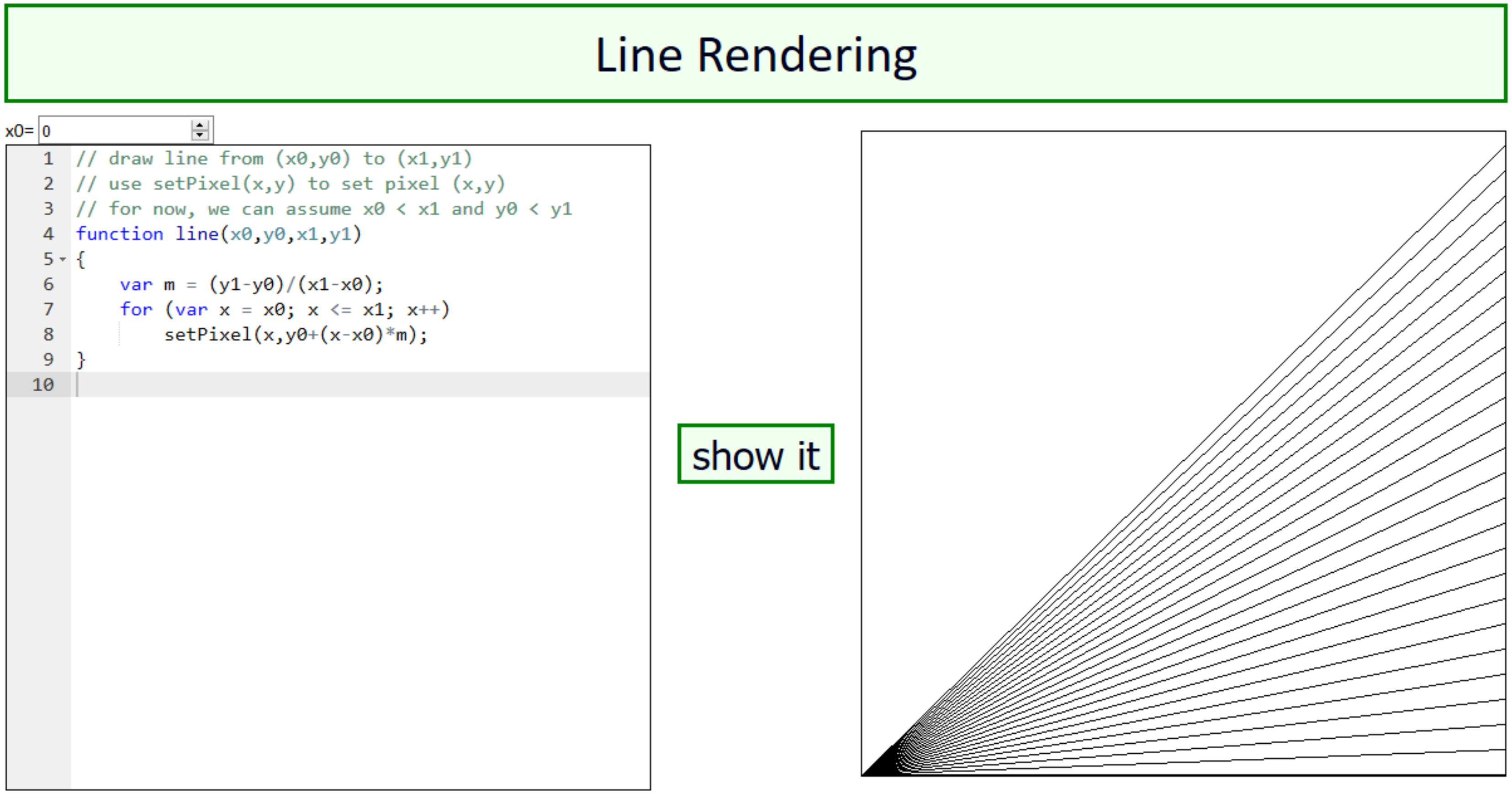
x0= 0

```
// draw line from (x0,y0) to (x1,y1)
// use setPixel(x,y) to set pixel (x,y)
// for now, we can assume x0 < x1 and y0 < y1
function line(x0,y0,x1,y1)
{
    if (x1-x0 < 1 && y1-y0 < 1)
        setPixel(x0,y0);
    else
    {
        var xm = (x0+x1)/2, ym = (y0+y1)/2;
        line(x0,y0,xm,ym);
        line(xm,ym,x1,y1);
    }
}
```

- → for our purpose: slow, pixels may be set multiple times…

Line Drawing

- An iterative version

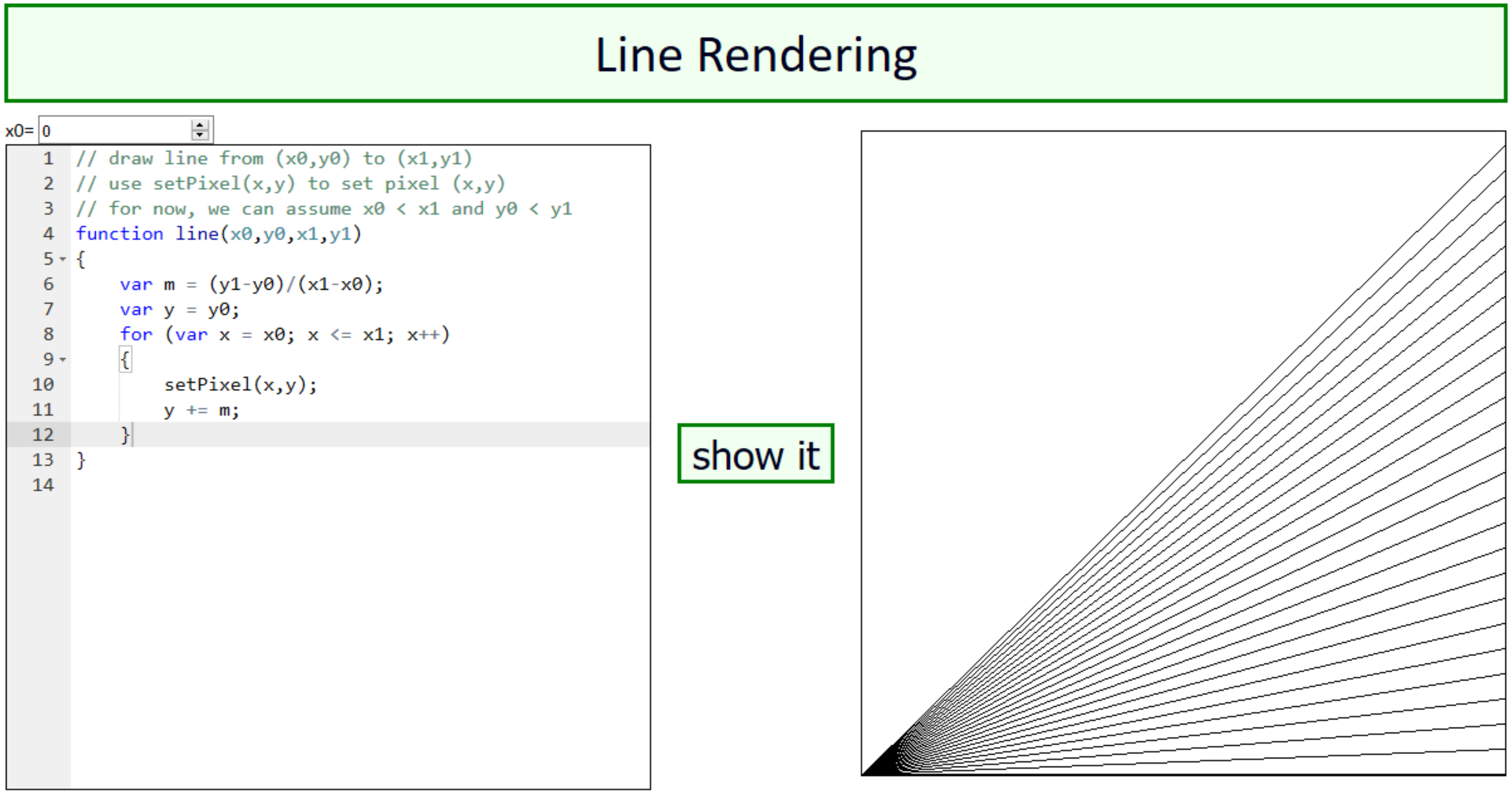


```
// draw line from (x0,y0) to (x1,y1)
// use setPixel(x,y) to set pixel (x,y)
// for now, we can assume x0 < x1 and y0 < y1
function line(x0,y0,x1,y1)
{
    var m = (y1-y0)/(x1-x0);
    for (var x = x0; x <= x1; x++)
        setPixel(x,y0+(x-x0)*m);
}

```

- renders x1-x0 pixels for all lines → but length varies by $\sqrt{2}$

Line Drawing

- Iterative version 2 – even simpler

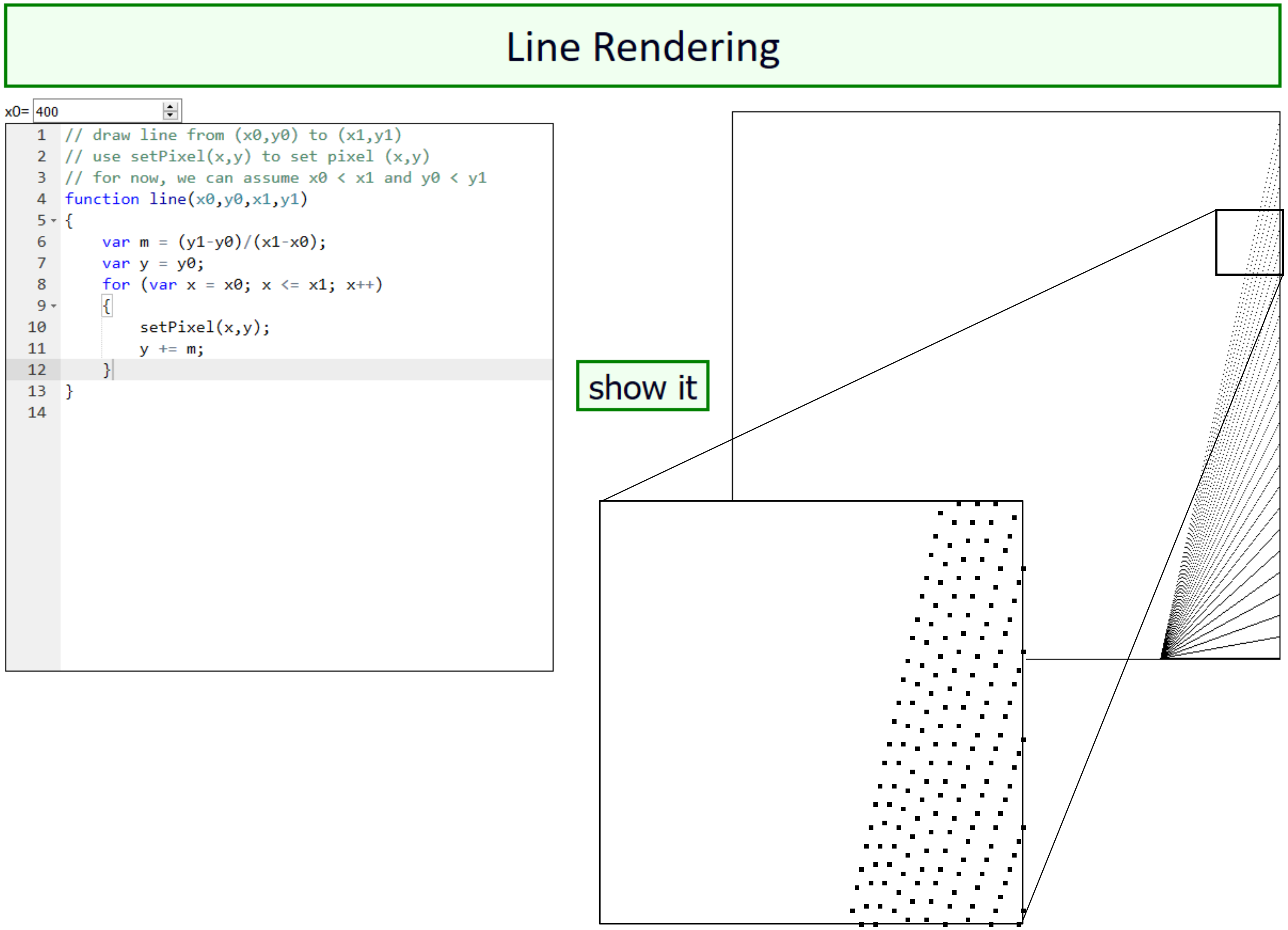


```
// draw line from (x0,y0) to (x1,y1)
// use setPixel(x,y) to set pixel (x,y)
// for now, we can assume x0 < x1 and y0 < y1
function line(x0,y0,x1,y1)
{
    var m = (y1-y0)/(x1-x0);
    var y = y0;
    for (var x = x0; x <= x1; x++)
    {
        setPixel(x,y);
        y += m;
    }
}
```

- only one addition within loop

Line Drawing

- only works for lines with slope < 1

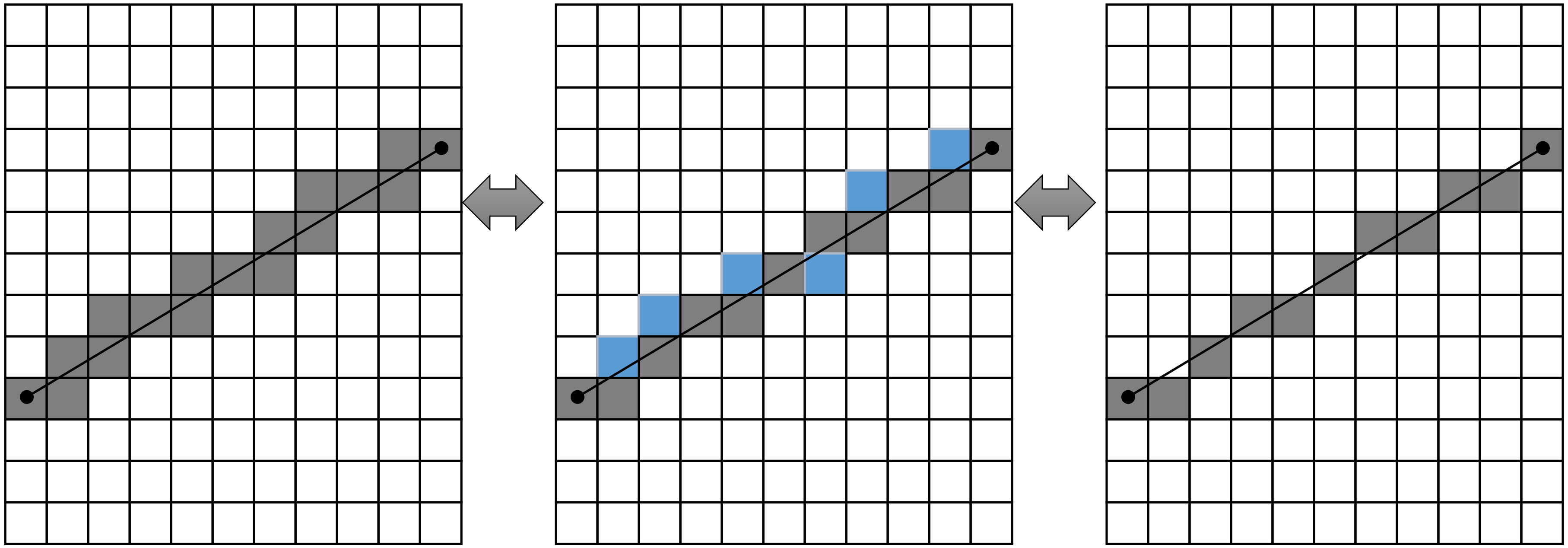

Line Rendering

x0= 400

```
// draw line from (x0,y0) to (x1,y1)
// use setPixel(x,y) to set pixel (x,y)
// for now, we can assume x0 < x1 and y0 < y1
function line(x0,y0,x1,y1)
{
    var m = (y1-y0)/(x1-x0);
    var y = y0;
    for (var x = x0; x <= x1; x++)
    {
        setPixel(x,y);
        y += m;
    }
}
```

Line Drawing

- Line Rasterization: Problem statement (without anti-aliasing)



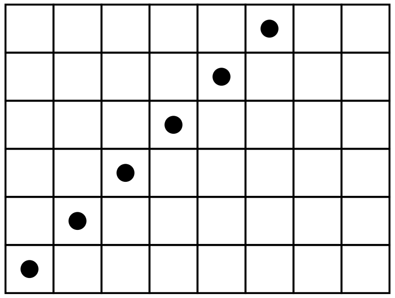
Mark all pixels *touched* by the line. Line appears to be thicker

blue pixels should not be considered

Better (thinnest) approximation of the line

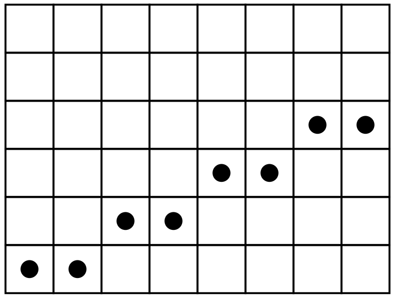
Line Drawing

- Problem Statement
  - How to draw a line from $P_0 = (x_0, y_0)$ to $P_1 = (x_1, y_1)$
  - Examples
    - $(0{,}0)$ to $(6{,}6)$



Slope = $6/6$

    - $(0{,}0)$ to $(8{,}4)$



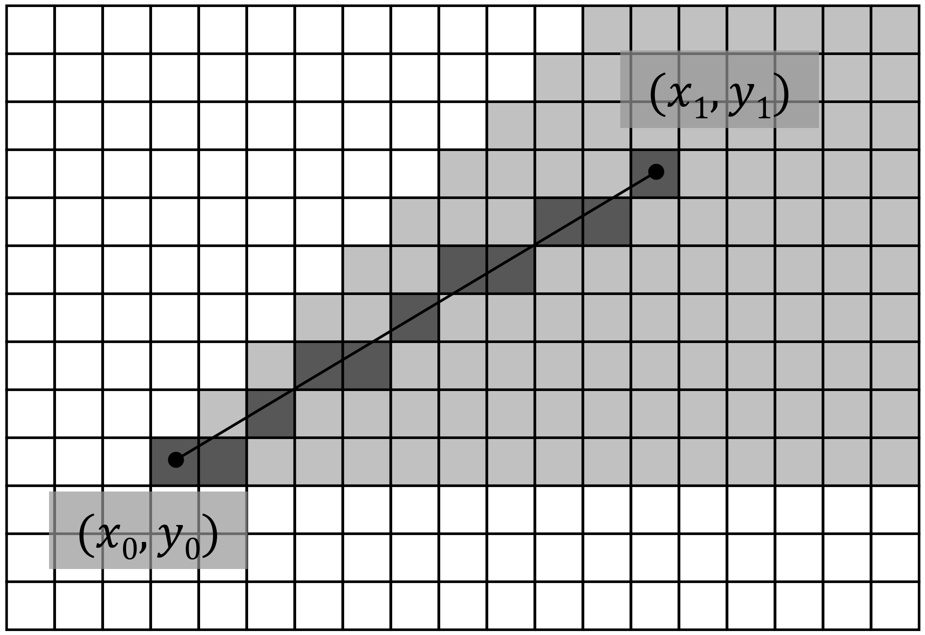
Slope = $4/8$

Line Drawing

- For now, we
- Simplification
  - Slope m: $0 < m < 1$ where $m = \Delta y/\Delta x = (y_1 - y_0)/(x_1 - x_0)$
  - $x_0 < x < x_1$: $y = y_0 + m(x - x_0)$
  - all other cases can be treated similarly

Line Drawing

- Slope m: $0 < m < 1$ where $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$



Line Drawing

- Brute force algorithm
  - x_0, x_1, y_0, y_1 are integers
  - Direct version

```
float m = (float)(y1 – y0) / (x1 – x0)

for int x = x0 to x1
   float y = y0 + m(x – x0)
   draw_pixel (x, round(y))
```

Line Drawing

- Simple algorithm, incremental version
- Remark:

$$y_n = y_0 + m(x_n - x_0)$$
$$y_{n+1} = y_0 + m(x_n + 1 - x_0) = y_n + m$$

```
float m = (float)(y1 - y0)/(x1 - x0)
float y = y0
int x = x0

while (x <= x1)
    draw_pixel(x, round(y))
    x = x + 1
    y = y + m
```

Line Drawing: Bresenham

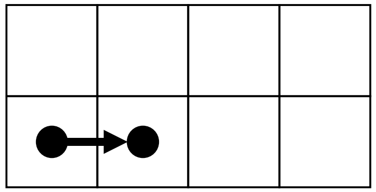
- Bresenham-Algorithm based on incremental version (see right)
- goal
    - avoid float-operations
    - use integer only
- if $0 < m < 1$ and $x_0 < x_1$:
    - y remains either the same
    - or is increased by one

```
// incremental line drawing
float m = (float)(y1 – y0)/(x1 – x0)
float y = y0
int x = x0

while (x <= x1)
    draw_pixel(x, round(y))
    x = x + 1
    y = y + m
```

```
// Bresenham line drawing
int y = y0
int x = x0

while (x <= x1)
    draw_pixel(x,y)
    x = x + 1
    if (some condition)
        y = y + 1
```

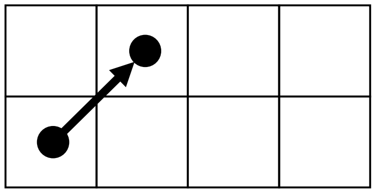
- Two cases:

Case 1:



East

Case 2:



North-East

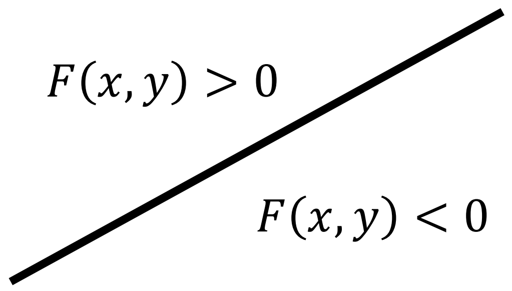
- How to decide between **E** and **NE** ?

Line Drawing: Bresenham

- The implicit equation for a line

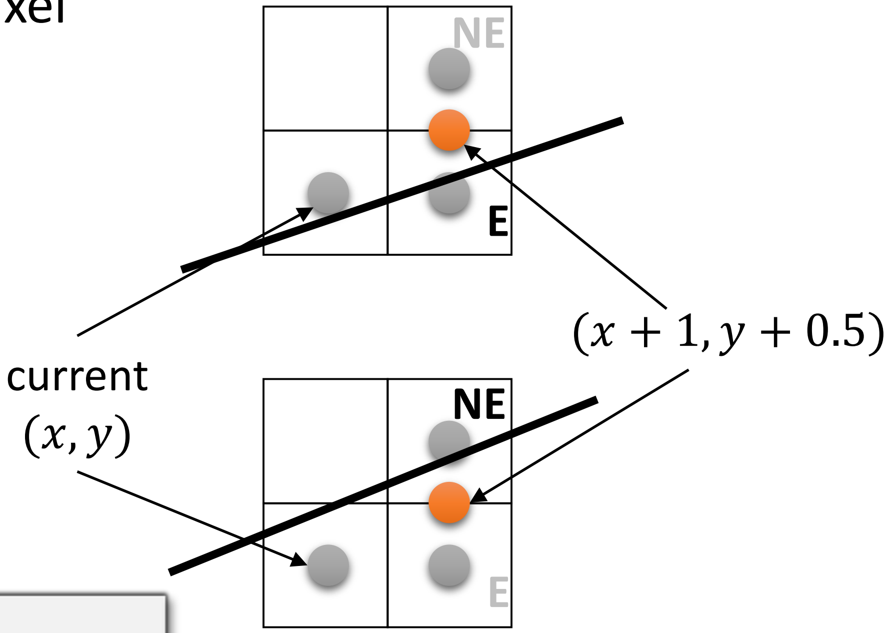
$$F(x, y) = (y - y_0) - m(x - x_0)$$

- $F(x, y) = 0$: (x, y) is **on** the line
- $F(x, y) < 0$: (x, y) is **below** the line
- $F(x, y) > 0$: (x, y) is **above** the line



Line Drawing: Bresenham

- Midpoint decider
  → look at midpoint between E and NE pixel
  - if line below midpoint **GO EAST**
  - otherwise, **GO NORTH-EAST**



- That is:

```
// Bresenham line drawing
int y = y0
int x = x0

while (x <= x1)
    draw_pixel(x,y)
    x = x + 1
    if (F(x,y+0.5) < 0)
        y = y + 1
```

Line Drawing: Bresenham

- Performance considerations:
  Making the evaluation of the decider faster
  - Incremental
  - Integer operation only

- But F is rational value (m is rational)…

- But we can multiply F with arbitrary positive value
  $\rightarrow$ get rid of denominator of m
  - $F(x,y) = y(x_1 - x_0) + x(y_0 - y_1) + y_1x_0 - y_0x_1 =$
    $\Delta x(y - y_0) - \Delta y(x - x_0)$

Line Drawing: Bresenham

- Incremental algorithm: Compute F incrementally in variable d
  →First step in loop

$$d = F(x_0 + 1, y0 + {}^{1}/_{2})$$

- Within loop, if $d < 0$
  → **NE**: $(x_0, y_0) \rightarrow (x_0 + 1, y_0 + 1)$
  - Next test will be at $(x_0 + 2, y_0 + 1 + {}^{1}/_{2})$
  - $F\left(x_0 + 2, y_0 + \frac{3}{2}\right) = \ \ldots = F(x_0 + 1, y_0 + {}^{1}/_{2}) + \Delta x - \Delta y$
  - → Incremental update of d: $\quad d_{new} = d_{old} + \Delta x - \Delta y$

- Analog, if $d > 0$
  → E: $(x_0, y_0) \rightarrow (x_0 + 1, y_0)$
  - Next test will be at $\left(x_0 + 2, y_0 + \frac{1}{2}\right)$
  - $F\left(x_0 + 2, y_0 + \frac{1}{2}\right) = \cdots = F\left(x_0 + 1, y + \frac{1}{2}\right) + (y_0 - y_1)$
  - Incremental update of *d:* $\quad d_{new} = d_{old} - \Delta y$

Line Drawing: Bresenham

- Algorithm

```
int y = y0
int x
float d = F(x0+1,y0+0.5) // decider
for x = x0 to x = x1
    draw_pixel(x,y)
    if (d < 0) then // go NE
       y = y + 1
       d = d + (x1 – x0) + (y0 – y1)
    else // go E
       d = d + (y0 – y1)
```

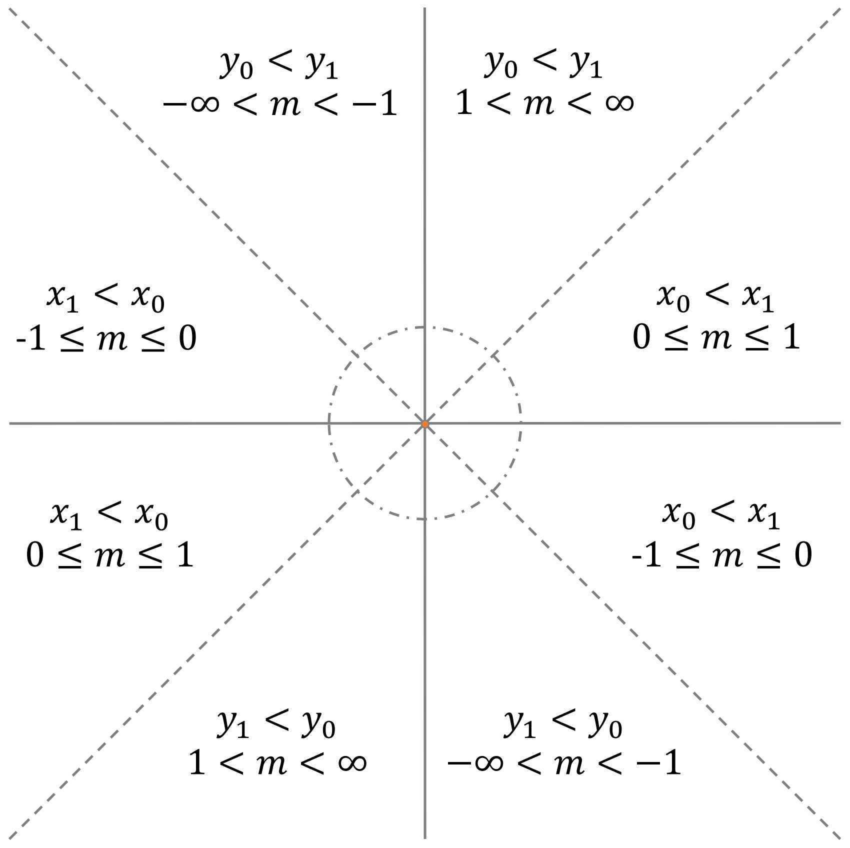
Line Drawing: Bresenham

- Initialization of D has a 0.5-parameter → initial value multiple of 0.5
- All other increments are integer
- → multiple with 2 → integer only

```
int x = x0
int y = y0
int Δx = x1 – x0
int Δy = y1 – y0
int D = Δx – 2Δy , ΔDE = -2Δy , ΔDNE = 2(Δx - Δy)

while (x <= x1)
    draw_pixel(x,y)
    x = x + 1
    if(D < 0) {
        y = y + 1
        D = D + ΔDNE
    }
    else
        D = D + ΔDE
```

Line Drawing: Bresenham

- handling multiple slopes: consider eight regions: octants



Line Drawing: Bresenham

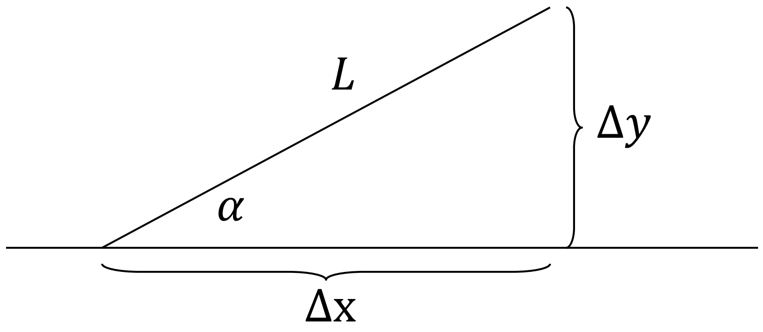
- Remark: negative slopes
  - update on y is different
    - if line above midpoint update to $(x+1, y)$
    - otherwise update to $(x+1, y-1)$
  - update on decision variable is subtly different:
    $F\left(x+1, y+\frac{1}{2}\right) > 0 \Rightarrow$ goto $(x+1, y-1)$ and next test at $\left(x+2, y-\frac{3}{2}\right)$
    $F\left(x+1, y+\frac{1}{2}\right) \leq 0 \Rightarrow$ goto $(x+1, y)$ and next test at $(x+2, y-\frac{1}{2})$

Line Drawing: Bresenham

- One possible strategy
  - If $|m| > 1$: swap coordinates, i.e. $x \leftrightarrow y$
  - if $x_0 > x_1$: swap start and end points
  - if $m < 0$: set step in y to be -1
  - use $\Delta x = x_1 - x_0$ and $\Delta y = |y_1 - y_0|$

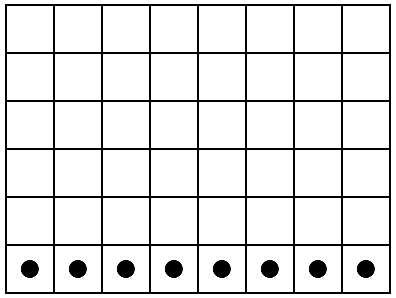
Line Drawing

- Problems:
  - The length of a line is measured in screen units = pixels
  - Ideally: number of pixels of scan-converted line equal length
  - If line longer than no. of pixels, it looks fragmented
  - Bresenham algorithm generates number of pixels $= \max(\square x, \Delta y)$
  - Assume $|m| < 1$
    number of pixels $= L \cos \alpha$
    where L length of line

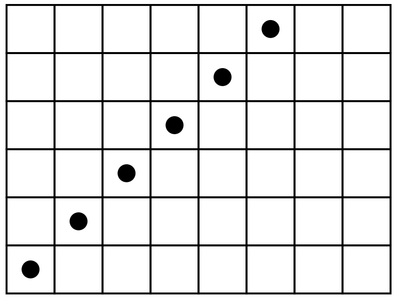


Line Drawing: antialiasing

- Problems
  - Line intensity varies with slope



Horizontal line:
1 pixel / unit length



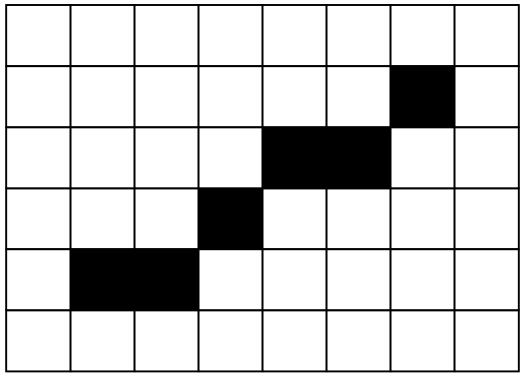
Diagonal line:
$1/\sqrt{2}$ pixel / unit length

→ on grey scale screen: modify intensity by $\frac{1}{\sqrt{2}\cos\alpha}$

  - “Jaggies” → typical **aliasing** artifact

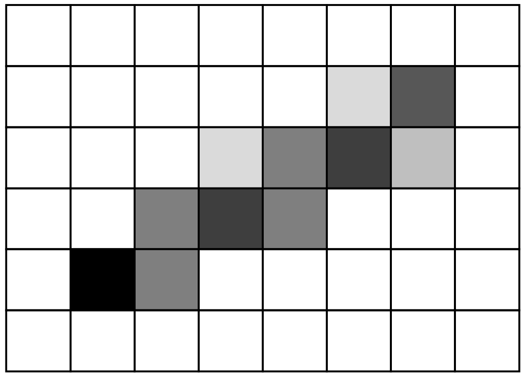
Line Drawing: antialiasing

- Antialiased Bresenham
  - In the original Bresenham, only one pixel is drawn per incremental step. The desired intensity (here: black) is entirely assigned to that pixel.



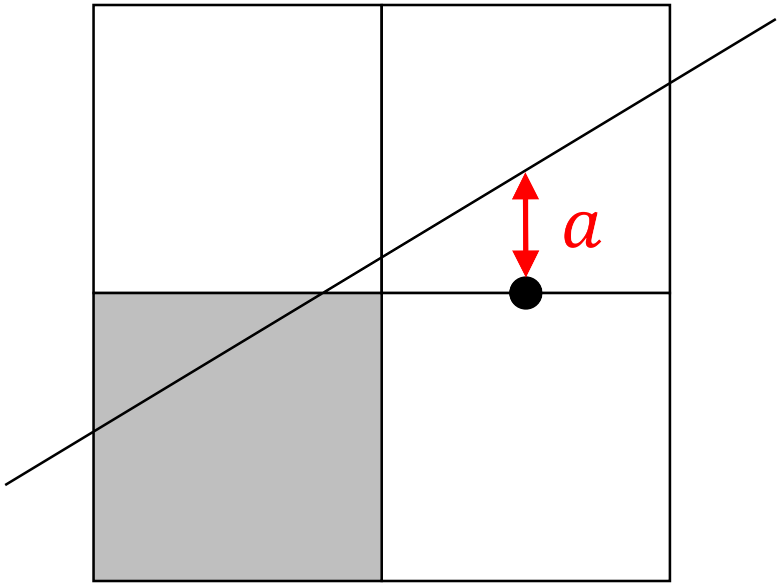
Line Drawing: antialiasing

- Antialiased Bresenham
  - With antialiasing, (up to) two pixels are drawn per incremental step (and column). The intensity of these pixels sums up to the desired intensity.



Line Drawing: antialiasing

- In order to decide which pixels we should draw and how to choose the weighting factors, we need the signed distance a between the true line and the midpoint between the E- and the NE-pixel.

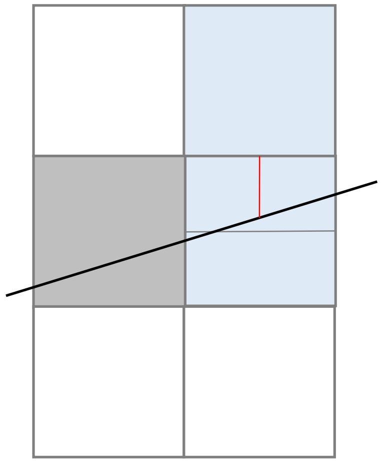


The distance can be computed from the decision variable d:

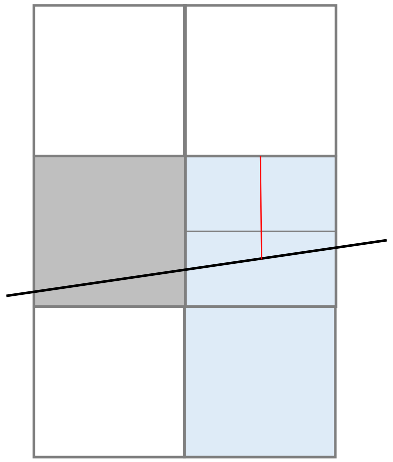
$$a = \frac{d}{2\Delta x}$$

Line Drawing: antialiasing

- Which pixels should be drawn?
- Case $d \geq 0$ (choose E)


$a < 0.5$

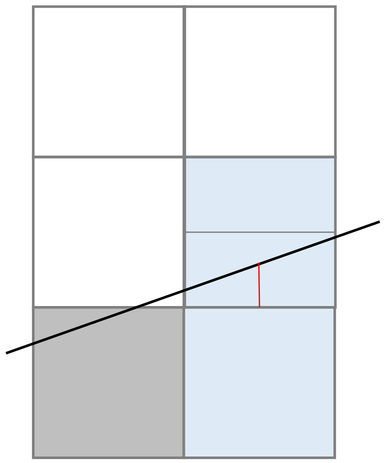
draw pixels:
$(x+1, y)$ with intensity factor $1 - |a + 0.5|$
$(x+1, y+1)$ with intensity factor $|a + 0.5|$


$a > 0.5$

draw pixels:
$(x+1, y)$ with intensity factor $1 - |a + 0.5|$
$(x+1, y-1)$ with intensity factor $|a + 0.5|$

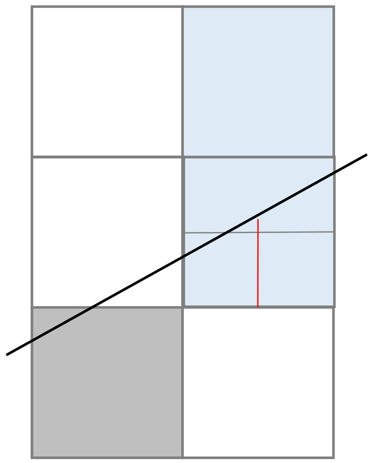
Line Drawing: antialiasing

- Case $d < 0$ (choose NE)



$a > -0.5$

draw pixels:
$(x + 1, y + 1)$ with intensity $1 - |a - 0.5|$
$(x + 1, y)$ with intensity $|a - 0.5|$



$a < -0.5$

draw pixels:
$(x + 1, y + 1)$ with intensity $1 - |a - 0.5|$
$(x + 1, y + 2)$ with intensity $|a - 0.5|$

Circle Drawing

- Circle
  - Center $c = (x_c, y_c)$
  - Circle of radius r

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

- For now:
  - Center at $(0,0)$

- Eight-fold symmetry
  - 1st octant: $0 \leq y < x$
  - 2nd octant: $0 <= x < y$
  - 3rd octant: $0 <= -x < y$
  - 4th octant: $0 <= y < -x$
  - 5th octant: $0 <= -y < -x$
  - 6th octant: $0 <= -x < -y$
  - 7th octant: $0 <= x < -y$
  - 8th octant: $0 <= -y < x$

Circle Drawing

- Draw pixels using the 8-fold symmetry
  add offset $c = (x_c, y_c)$ to center circle at (x_c, y_c)

```
// The pixel (x,y) is in the 2nd octant
void draw8pixel(xc,yc,x,y)
{
    draw_pixel(xc+x,yc+y); // (x,y) 2nd octant
    draw_pixel(xc+y,yc+x); // 1st octant
    draw_pixel(xc-x,yc+y); // 3rd octant
    draw_pixel(xc-y,yc+x); // 4th octant
    ...
}
```

Circle Drawing

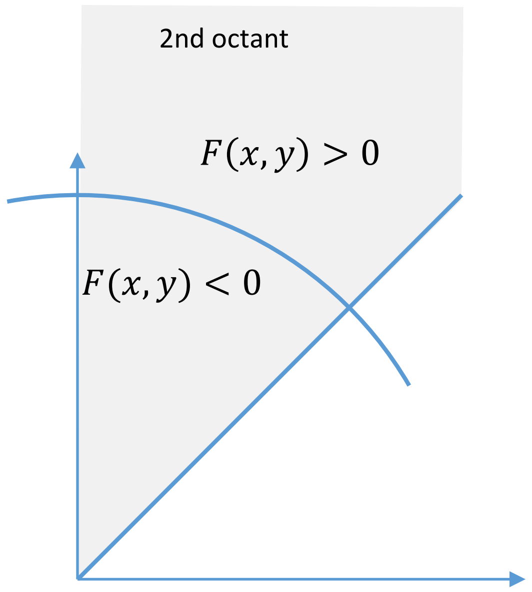
- The 2nd octant: $m < 0;\ |m| < 1;\ 0 < x < y$
- The implicit function

$$F(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2$$

- The circle

$$\{x \in \mathbb{R}^2 : F(x, y) = 0\}$$

- Properties
  - $F(x, y) > 0 \rightarrow (x, y)$ is outside/above the circle
  - $F(x, y) \leq 0 \rightarrow (x, y)$ is inside/below the circle



Circle Drawing

- The decider variable
  - $d = F(x+1, y-1/2)$
- The increment
  - $d > 0$ ((x, y) outside the circle)
    - $(x, y) \rightarrow (x+1, y-1)$
  - $d < 0$ ((x, y) inside the circle)
    - $(x, y) \rightarrow (x+1, y)$

Circle Drawing

- The increment of the decider variable
  - Set $d = F(x+1, y-1/2)$
  - Case $d < 0$; next test at $(x+2, y-1/2)$
    - $F\left(x+2, y-\frac{1}{2}\right) - F\left(x+1, y-\frac{1}{2}\right) = \cdots = 2x+3$
    - $\Rightarrow d = d + 2x + 3$
  - Case $d > 0$; next test at $\left(x+2, y-\frac{3}{2}\right)$
    - $F\left(x+2, y-\frac{3}{2}\right) - F\left(x+1, y-\frac{1}{2}\right) = \cdots = 2(x-y)+5$
    - $\Rightarrow d = d + 2(x - y) + 5$

Circle Drawing

- The increment of the decider variable
  - The increment of d depends on the position (x, y)
  - Introduce new variables E and SE (E: east, SE: south east)

$$E = 2x + 3; \; SE = 2(x - y) + 5$$

  - E and SE can be computed incrementally
    $\rightarrow$ incrementally compute the increment
    - If $d < 0$: $d = d + E$; $E = E + 2$; $SE = SE + 2$
    - If $d > 0$: $d = d + SE$; $E = E + 2$; $SE = SE + 4$

Circle Drawing

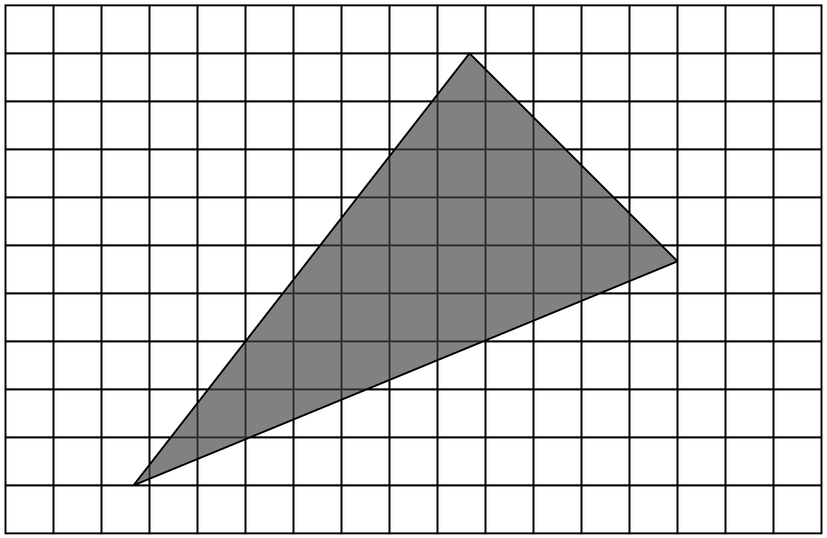
- Remarks
  - Use $d = F(x + 1, y - ½) - ¼$
  - Use only integer precision, x, y and r are taken to be ints

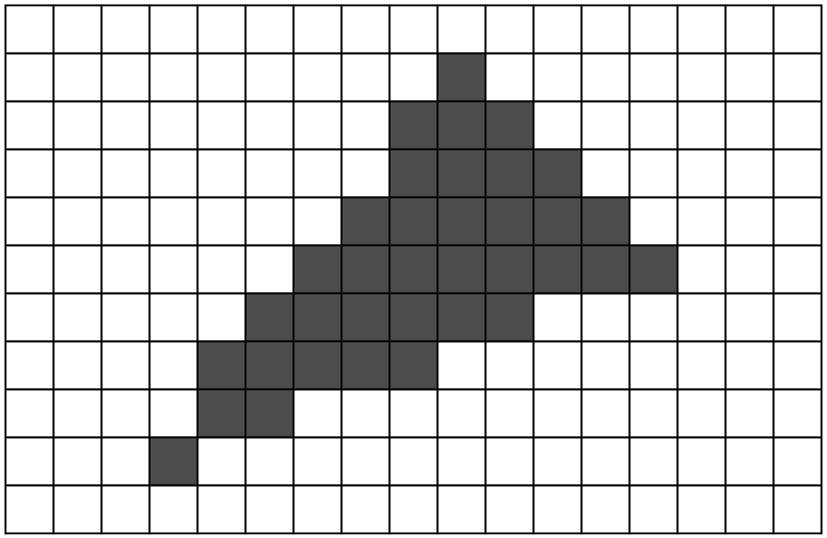
Circle Drawing

```
// Bresenham. 77
void Bresenham_Circle(xc,yc,r)
{
        x = 0; y = r;
        d = 1 – r; e = 3; se = 5 – 2*r;
        do {
                draw8pixel(xc,yc,x,y);
                if d < 0 then
                        d = d + e;
                        e = e + 2;
                        se = se + 2;
                        x = x + 1;
                else
                        d = d + se;
                        e = e + 2;
                        se = se + 4;
                        x = x + 1;
                        y = y - 1;
        } while (x <= y)
}
```

Polygon Rasterization

- Problem statement
  - Given a 2D-polygon with n vertices $P_1, \ldots, P_n$
  - Color all pixels inside the polygon





- Idea: rasterize boundery, fill interior → **seed fill algorithm**

Seed-Fill Algorithm

- Start at one point (seed)
  - Set it to fill color
  - look at neighbor pixels:
    if not set, call seed fill for these pixels recursively

- Recursive algorithm → BAD
- please don’t tell Prof. Philippsen

Seed-Fill Algorithm

- Recursive algorithm

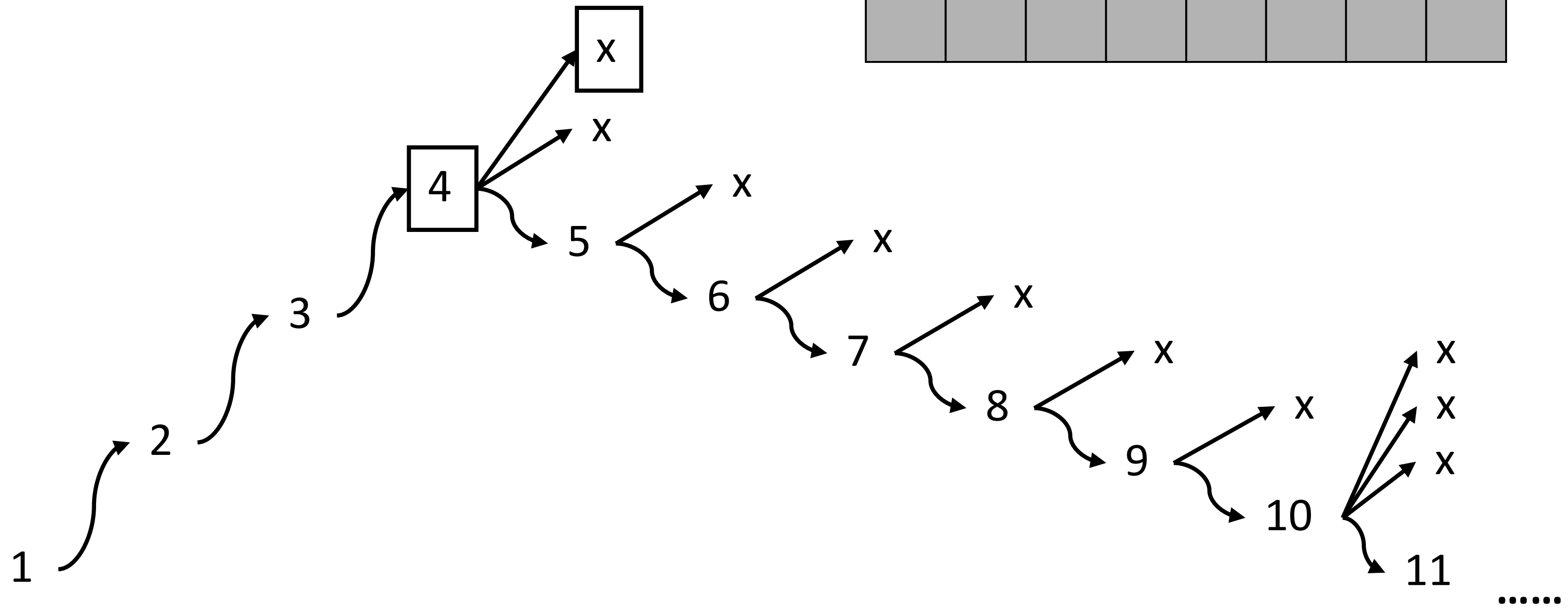
```
seedfill (x,y,fillcolor)
    if (color(x,y) == fillcolor)
        return; //boundary reached or fillcolor already set
    color(x,y) = fillcolor;
    seedfill(x+1,y);     //right
    seedfill(x-1,y);     //left
    seedfill(x,y+1);     //up
    seedfill(x,y-1);     //down
```

- Cons: Very deep recursion possible (requires large stack), rather inefficient

Seed-Fill Algorithm

- Example
  - 1: seed point
  - Recursion tree

▨	▨	▨	▨	▨	▨	▨	▨
▨	10	9	8	7	6	5	▨
▨	11	12	1	2	3	4	▨
▨	18	13	14	15	16	17	▨
▨	▨	▨	▨	▨	▨	▨	▨



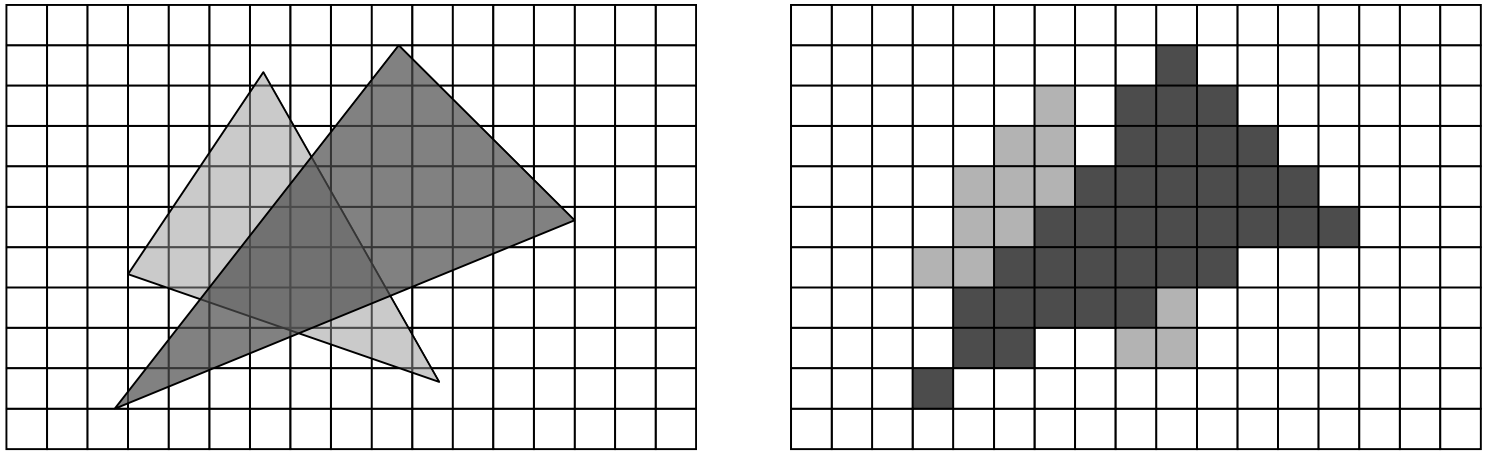
Seed-Fill Algorithm

- Apply for Polygon Rasterization:
  - Draw boundary of polygon using Bresenham **in unique color**
  - Pick a point inside
  - Do seed fill from this point
  - Replace unique color by desired one

- Evaluation for rasterization of polygons
  - Unique color only (no shading, see later)
    - How to correctly define boundary…
    - and not interfere with previously drawn objects
  - How to find seed position?
  - Not very efficient !

Polygon Rasterization

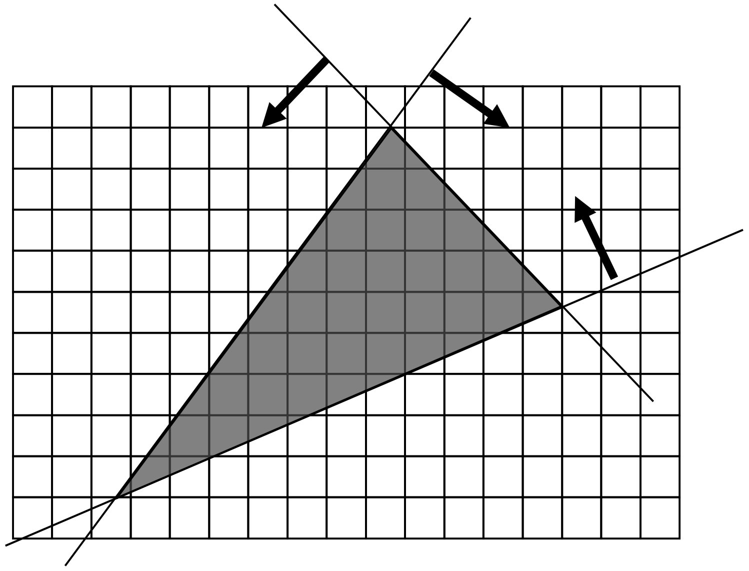
- Better: 2D Scan Conversion
  - We directly find the pixels within a polygon



Polygon Rasterization

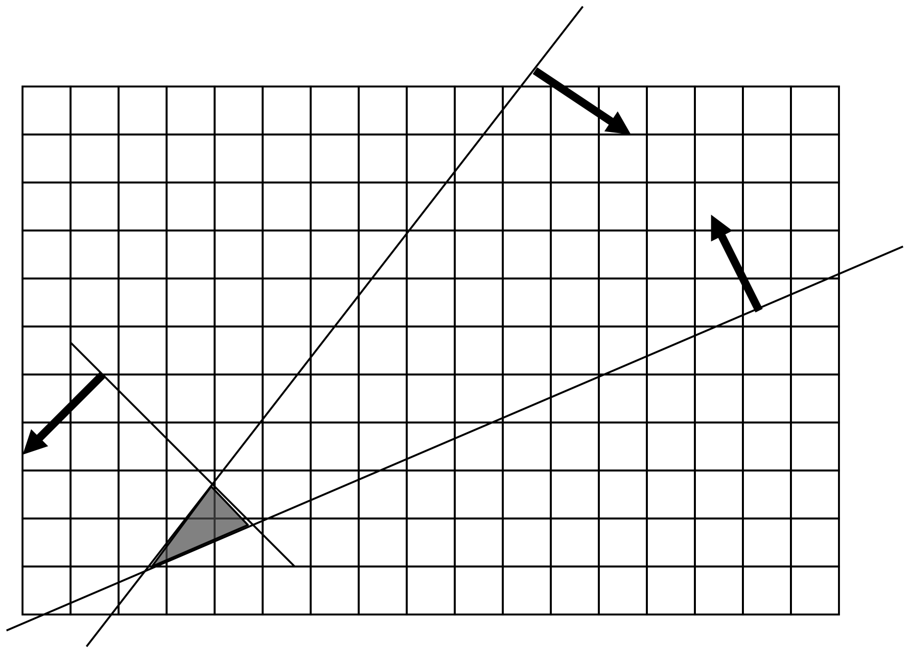
- Brute force solution for triangles

```
foreach pixel (x,y)
        Foreach edge E
                if (x,y) on wrong side of E
                        continue with next pixel
        set pixel (x,y)
```



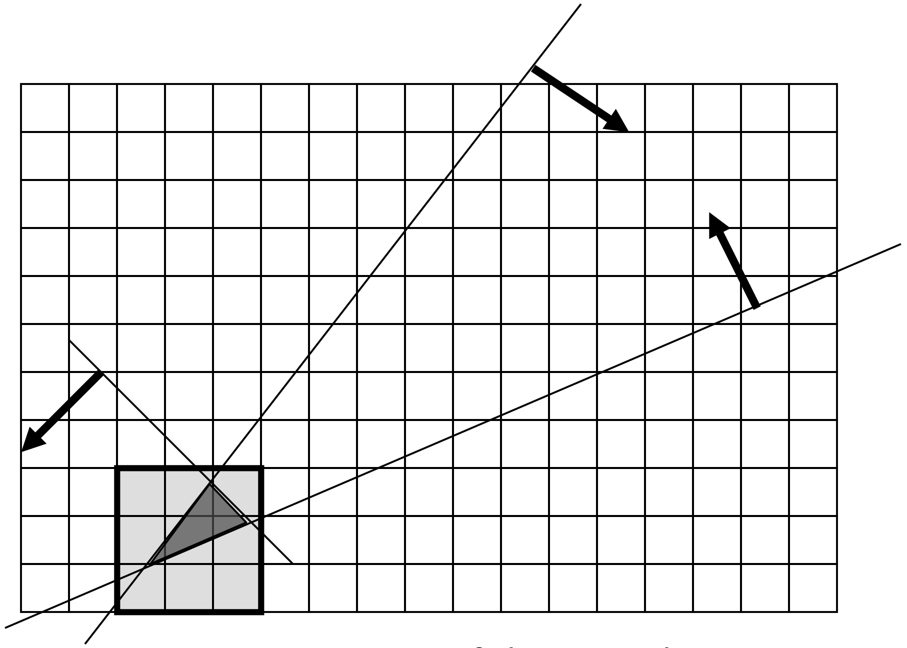
Polygon Rasterization

- Brute force solution for triangles
    - If the triangle is small, a lot of useless computation



Polygon Rasterization

- Brute force solution for triangles
  - Improvement: Compute only for the screen bounding box of the triangle
    → see programming exercise



$X_{min}, X_{max}, Y_{min}, Y_{max}$ of the triangle vertices

Polygon Rasterization

- Can we do better? – Yes!
  - Using line rasterization

- → **next lecture**