Lecture #03

Line Rasterization

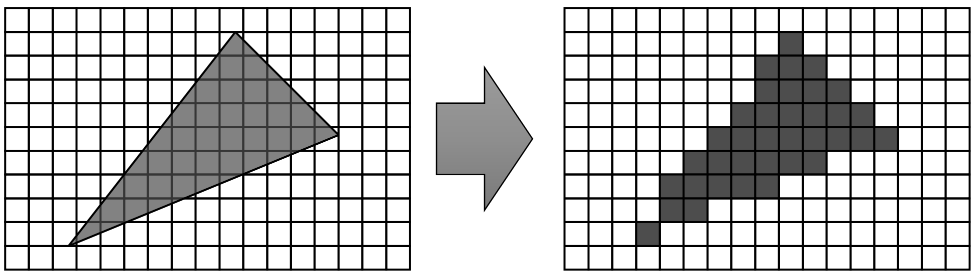
Computer Graphics
Winter Term 2020/21

Marc Stamminger / Roberto Grosso

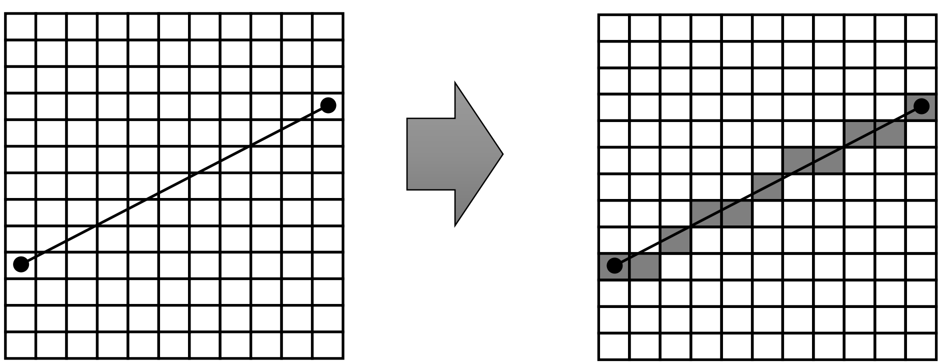
What is Rasterization ?

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

- Triangle primitive:



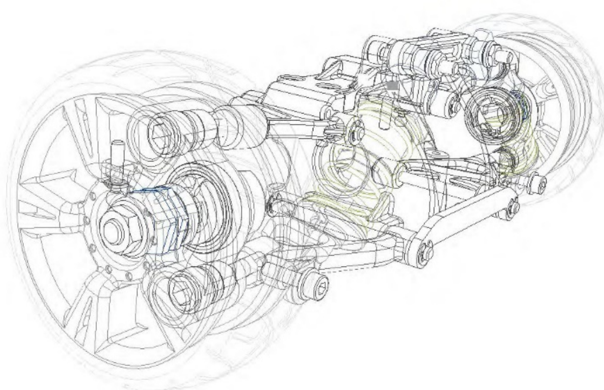
- Line primitive:



Rasterization - Primitives

- Which primitives are of interest ?

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

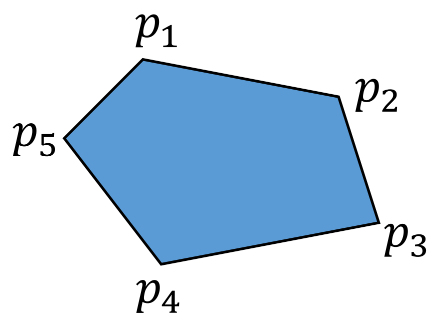


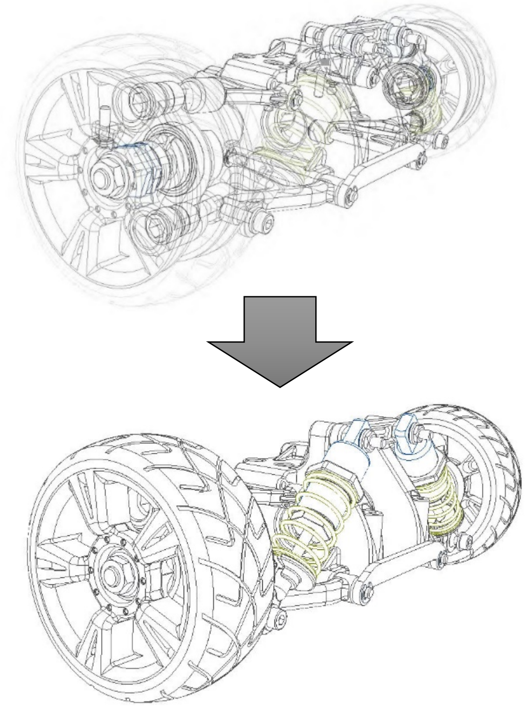
autodesk.com

  - every curve can be approximated by lines

Rasterization - Primitives

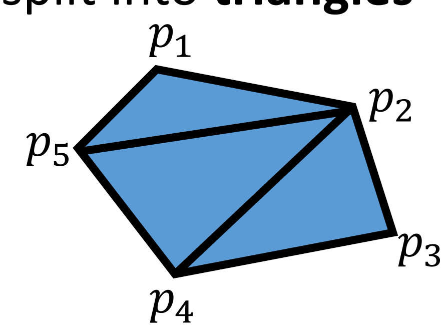
- mostly, we want to **fill** objects → **polygons**
- A **polygon** is defined by an ordered set of points (for now in 2D)

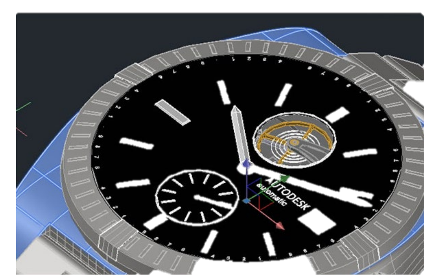




autodesk.com

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





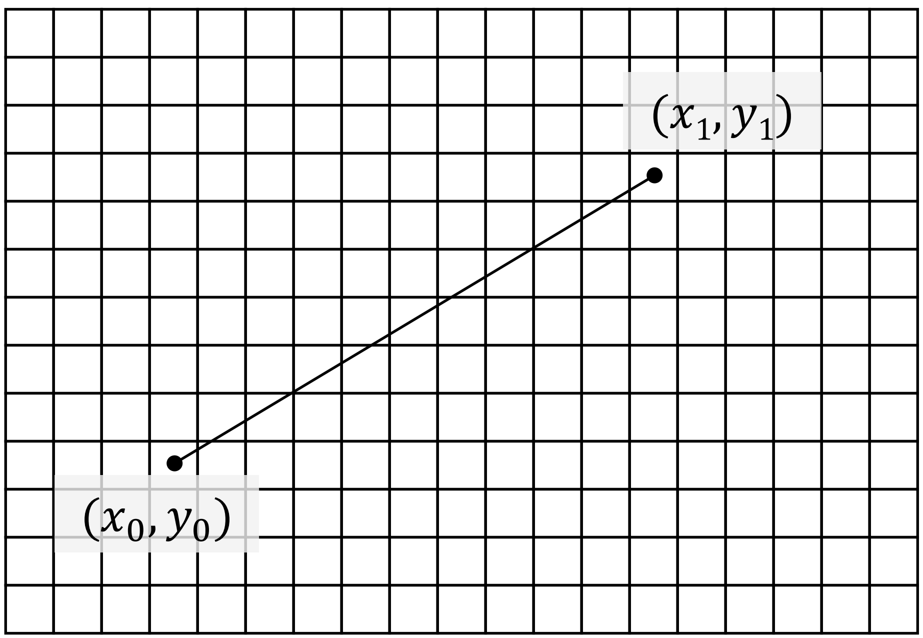
autodesk.com

Rasterization

- This lecture: Rasterization of lines (+ circles)
- Next Lecture: Rasterization of filled objects (Triangles, Polygons)

Line Drawing

- Line Rasterization
  - Given: Segment endpoints (integers $(x_0, y_0), (x_1, y_1)$)
  - Identify: Set of pixels (x, y) that represent the line segment



Line Drawing

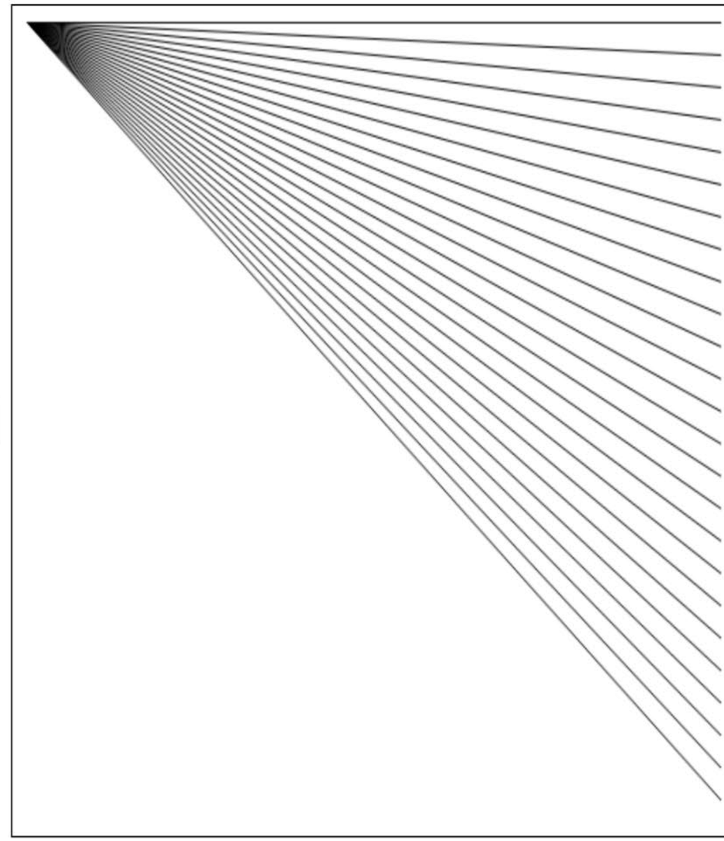
- An iterative version

Code Equation

```
// use setPixel(x,y) to set a pixel (x,y)
function drawLine(x0,y0,x1,y1) {
    var m = (y1 - y0) / (x1 - x0);
    for (var x = x0; x < x1; x++)
        setPixel(x, m*(x-x0) + y0);
}
```

>>>>

Render Fan

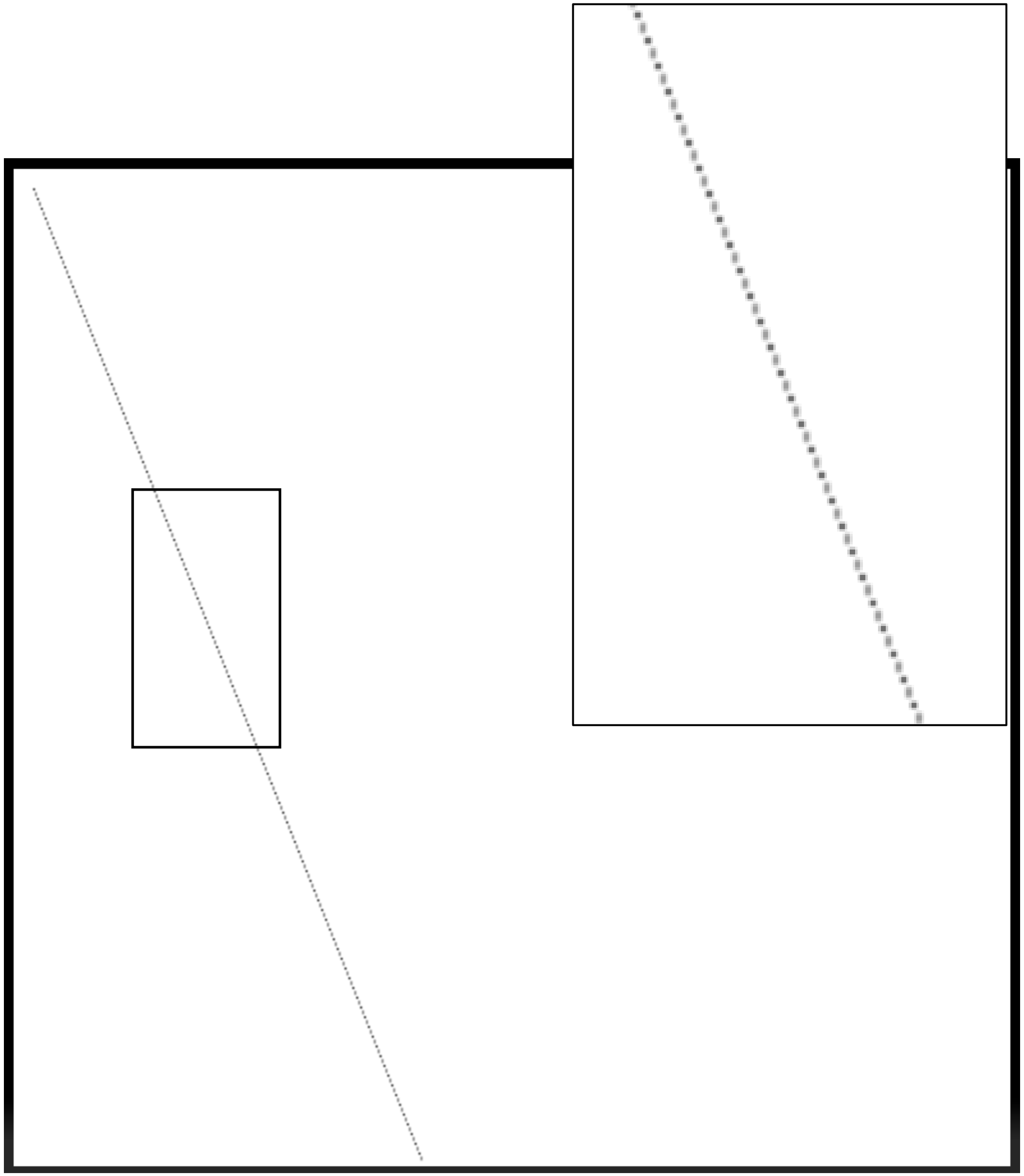


- renders x1-x0 pixels for all lines → but length varies by $\sqrt{2}$

Line Drawing

- Doesn‘t work if slope > 1
- and for x0 > x1, …

- → differentiate cases



Line Drawing

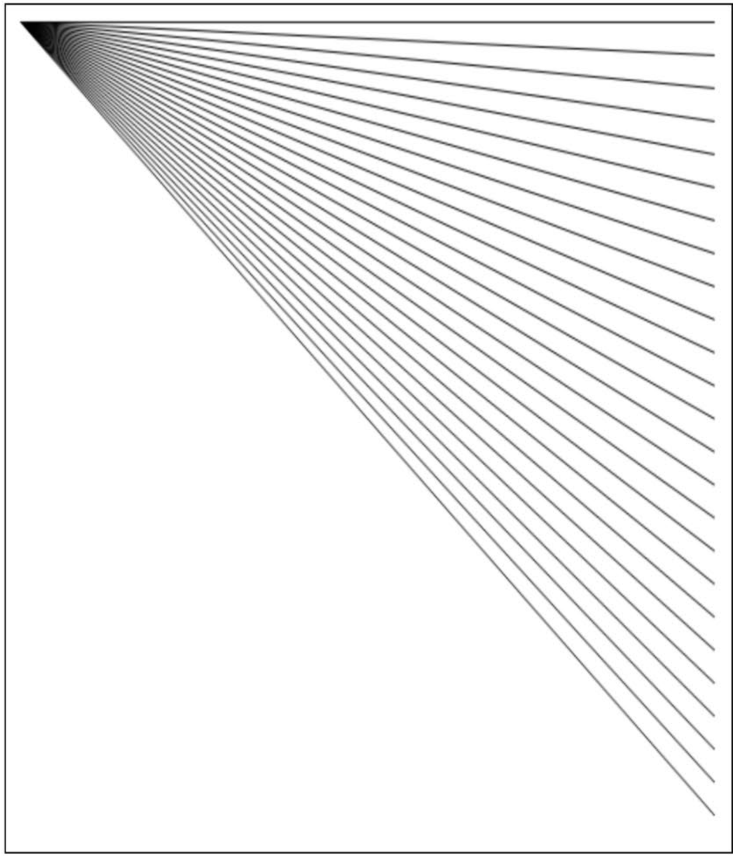
- Incremental version – even simpler

Code Incremental

```
// use setPixel(x,y) to set a pixel (x,y)
function drawLine(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;
    }
}
```

>>>>

Render Fan



- only one addition within loop

Line Drawing

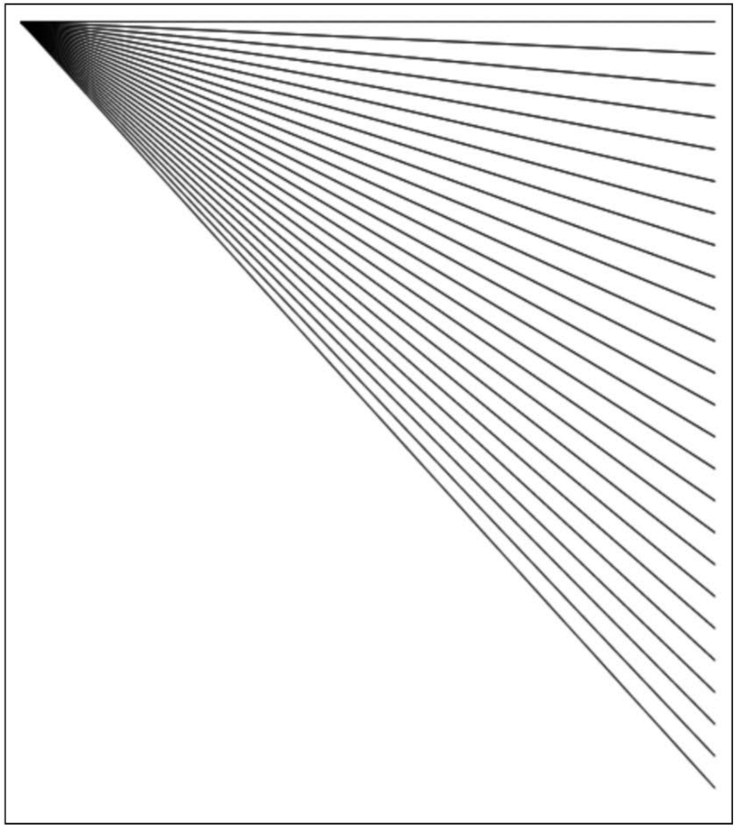
- A recursive line rasterizer

Code Recursive

```
// use setPixel(x,y) to set a pixel (x,y)
function drawLine(x0,y0,x1,y1) {
    var xm = (x0 + x1) / 2;
    var ym = (y0 + y1) / 2;
    setPixel(xm,ym);
    if (x1-x0 > 1 || y1-y0 > 1) {
        drawLine(x0,y0,xm,ym);
        drawLine(xm,ym,x1,y1);
    }
}
```

>>>>

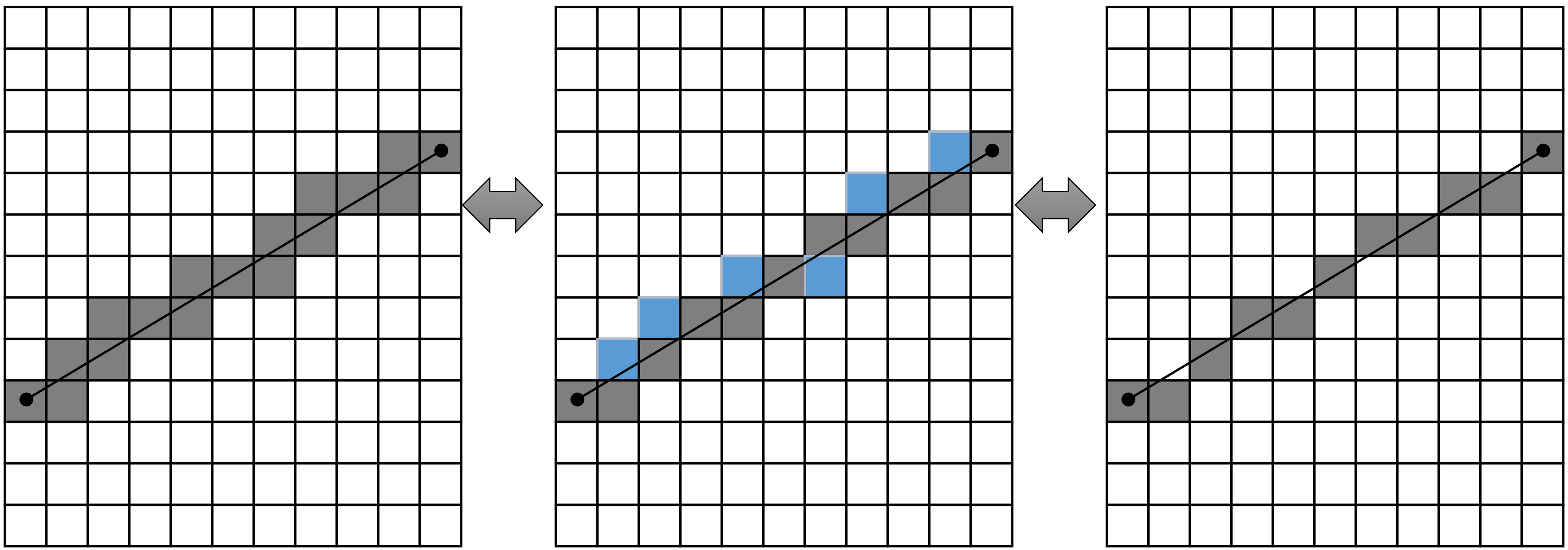
Render Fan



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

Line Drawing

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



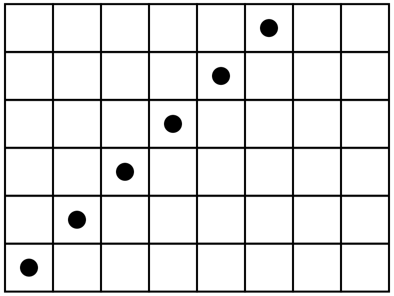
Mark all pixels *touched* by the line. Line appears to be thicker

blue pixels should not be considered

Better (thinnest) approximation of the line

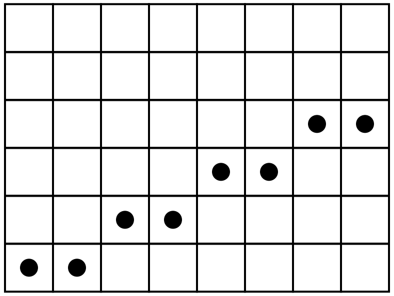
Line Drawing

- 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 $(8{,}4)$



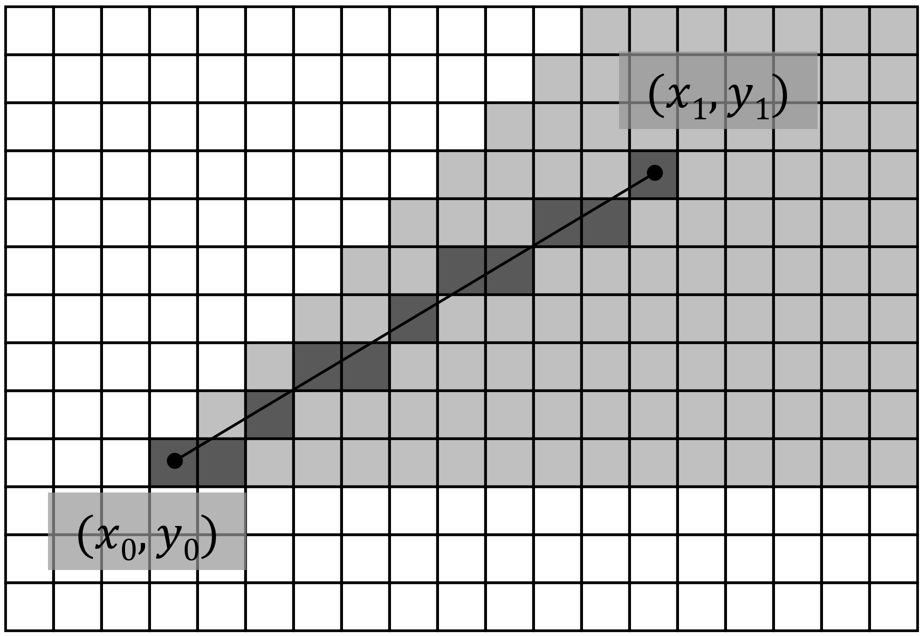
Slope $= 4/8$

Line Drawing

- 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)
   setPixel (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)
    setPixel(x, round(y))
    x = x + 1
    y = y + m
```

Line Drawing: Bresenham

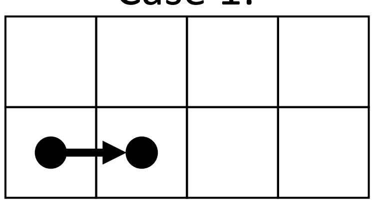
- Bresenham-Algorithm based on incremental version (see right)
- goal
  - avoid float-operations
  - use integer only
- if $0 < m < 1$ and $x_0 < x_1$:
  - y remains either the same
  - or is increased by one

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

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

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

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

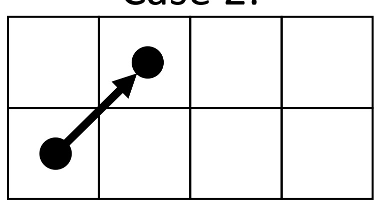
- Two cases:

Case 1:



East

Case 2:



North-East

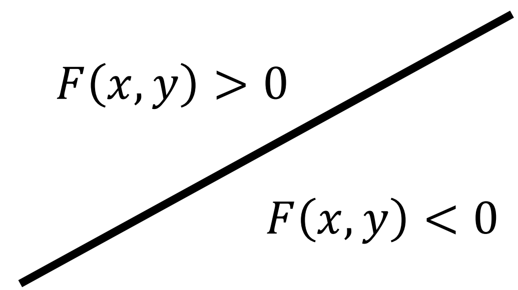
- How to decide between **E** and **NE** ?

Line Drawing: Bresenham

- The implicit equation for a line

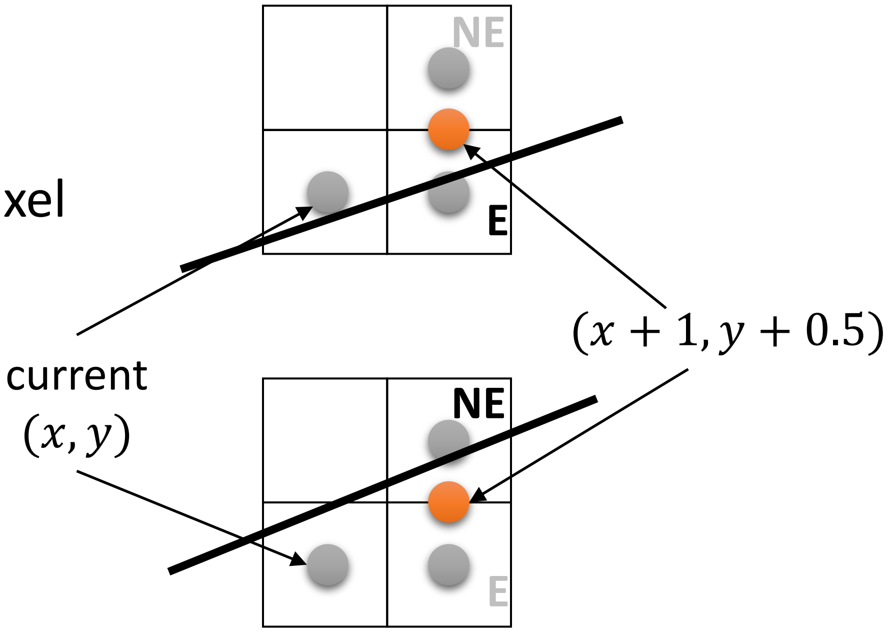
$$F(x, y) = (y - y_0) - m(x - x_0)$$

- $F(x, y) = 0$: (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)
    setPixel(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
  - $F(x, y) = y(x_1 - x_0) + x(y_0 - y_1) + y_1 x_0 - y_0 x_1 =$
    $\Delta x(y - y_0) - \Delta y(x - x_0)$

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: $\qquad d_{new} = d_{old} + \Delta x - \Delta y$

- 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:* $\qquad d_{new} = d_{old} - \Delta y$

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)
```

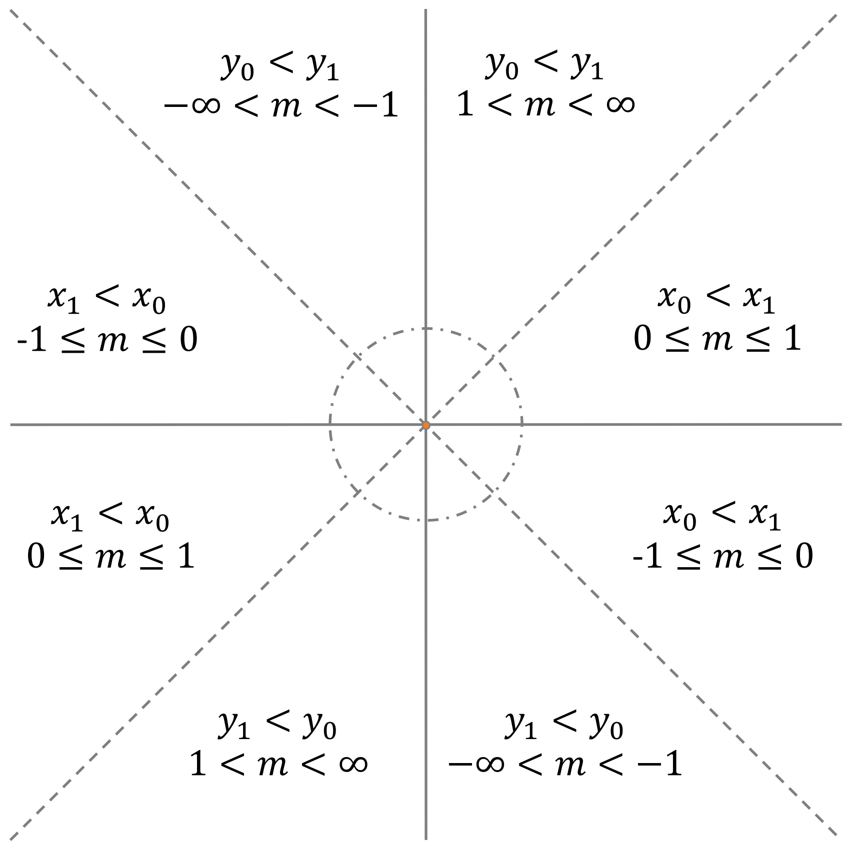
Line Drawing: Bresenham

- 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 $\left(x+2, y-\frac{3}{2}\right)$

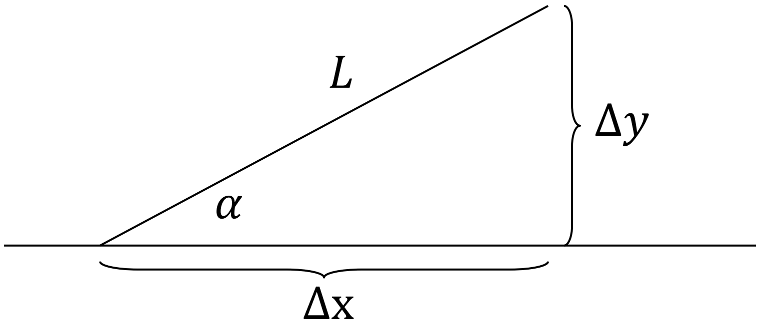
    $F\left(x+1, y+\frac{1}{2}\right) \leq 0 \Rightarrow$ goto $(x+1, y)$ and next test at $(x+2, y-\frac{1}{2})$

Line Drawing: Bresenham

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

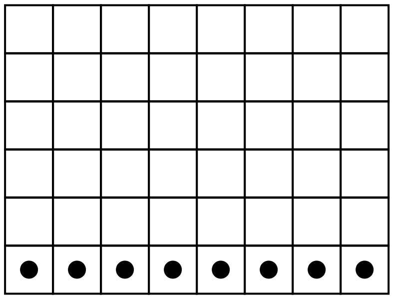
Line Drawing

- 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 $= \max(\Delta x, \Delta y)$
  - Assume $|m| < 1$
    number of pixels $= L \cos \alpha$
    where L length of line

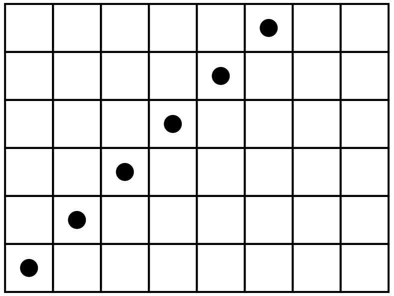


Line Drawing: Anti-Aliasing

- Problems
  - Line intensity varies with slope



Horizontal line:
1 pixel / unit length

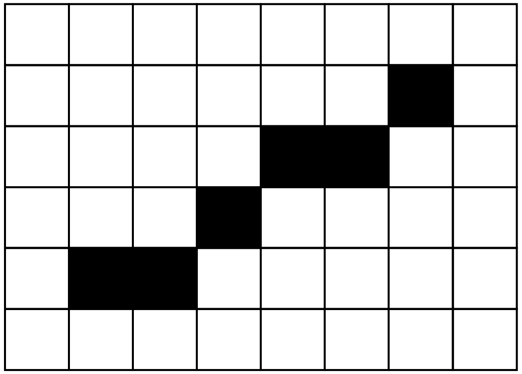


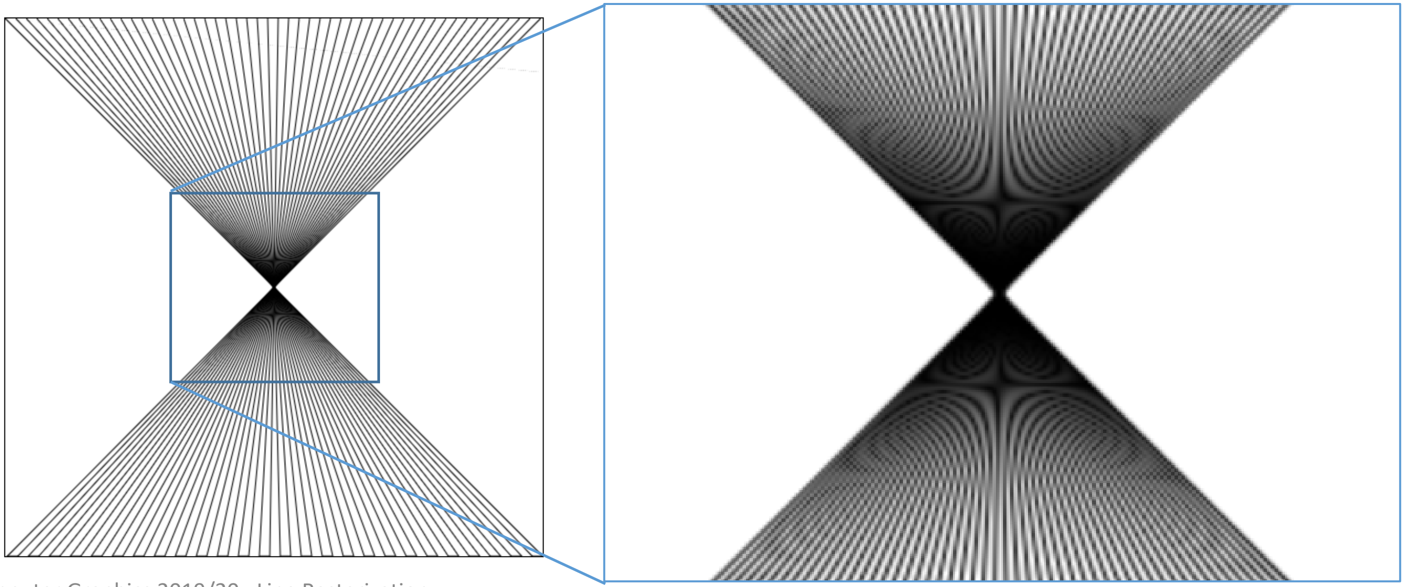
Diagonal line:
$1/\sqrt{2}$ pixel / unit length

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

Line Drawing: Anti-Aliasing

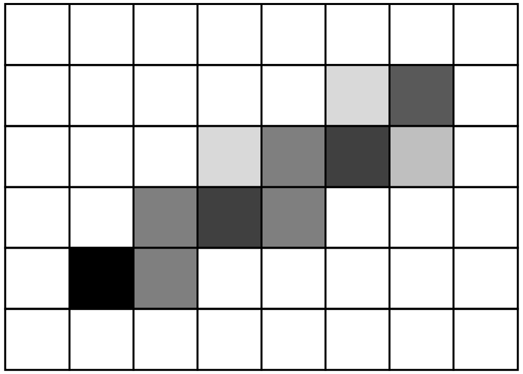
- “Jaggies” → typical **aliasing** artifact
  - In the original Bresenham, only one pixel is drawn per incremental step. The desired intensity (here: black) is entirely assigned to that pixel.
  - Can also result in patterns → **Moire effect** (Wikipedia)





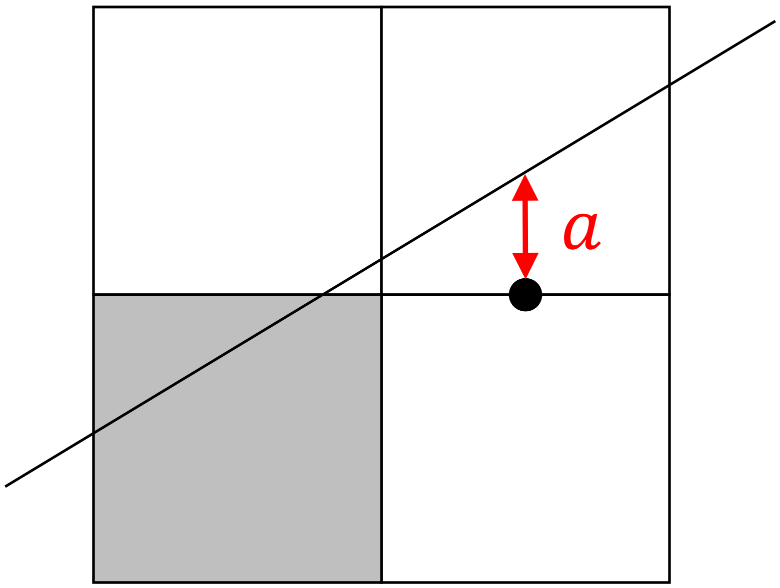
Line Drawing: Anti-Aliasing

- 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: Anti-Aliasing

- 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.

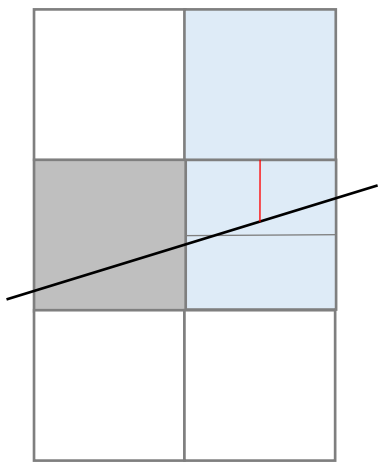


The distance can be computed from the decision variable d:

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

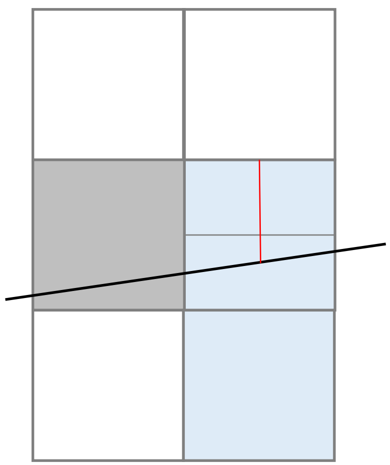
Line Drawing: Anti-Aliasing

- 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 $|a + 0.5|$

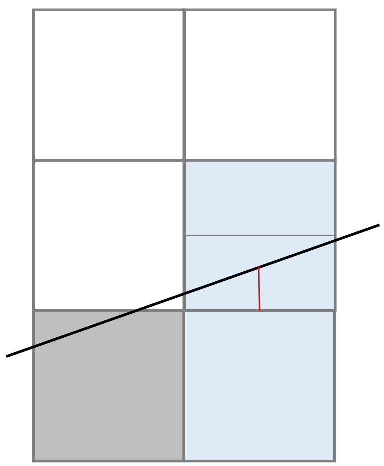


$a > 0.5$

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

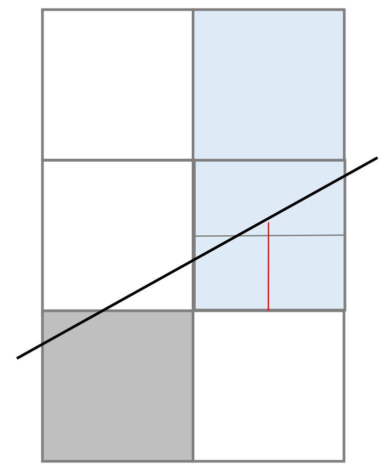
Line Drawing: Anti-Aliasing

- 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 $|a - 0.5|$

Further Reading - 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: $0 <= -y < x$

Further Reading - 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
    ...
}
```

Further Reading - Circle Drawing

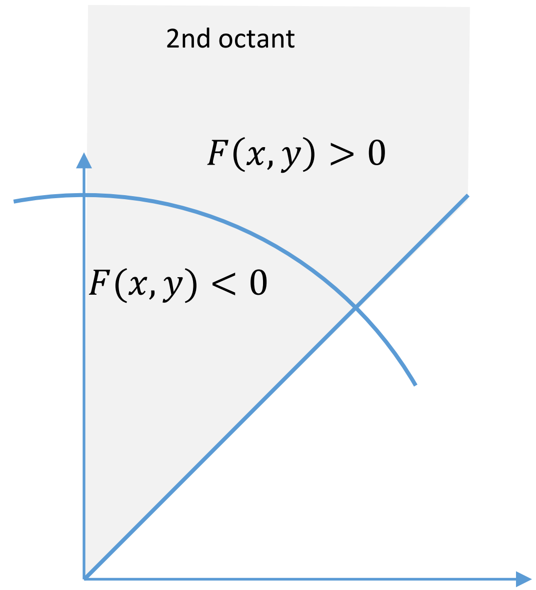
- The 2nd octant: $m < 0$; $|m| < 1$; $0 < x < y$
- The implicit function

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

- 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



Further Reading - 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)
    - $(x, y) \rightarrow (x+1, y)$

Further Reading - 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 $\left(x+2, y-\frac{3}{2}\right)$
    - $F\left(x+2, y-\frac{3}{2}\right) - F\left(x+1, y-\frac{1}{2}\right) = \cdots = 2(x-y)+5$
    - $\Rightarrow d = d + 2(x - y) + 5$

Further Reading - 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
    → incrementally compute the increment
    - If $d < 0$: $d = d + E$; $E = E + 2$; $SE = SE + 2$
    - If $d > 0$: $d = d + SE$; $E = E + 2$; $SE = SE + 4$

Further Reading - Circle Drawing

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

Further Reading - 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)
}
```

Next Lecture

- Polygon Rasterization