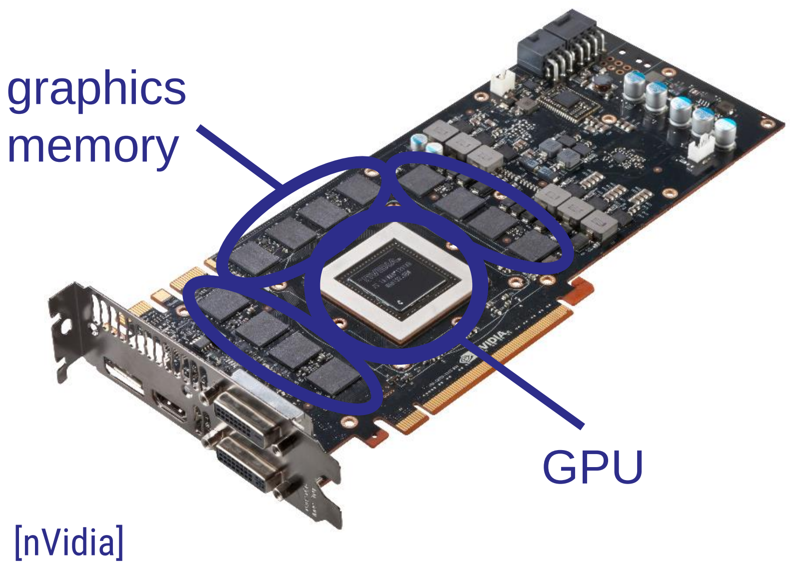
Computer Graphics

Scan Conversion

Computer Graphics Principles

- fastest-possible & most effective technique desired, best use of available resources
  → quality only to the level really wanted
  → often: we trade one thing for another



Scan Conversion Introduction

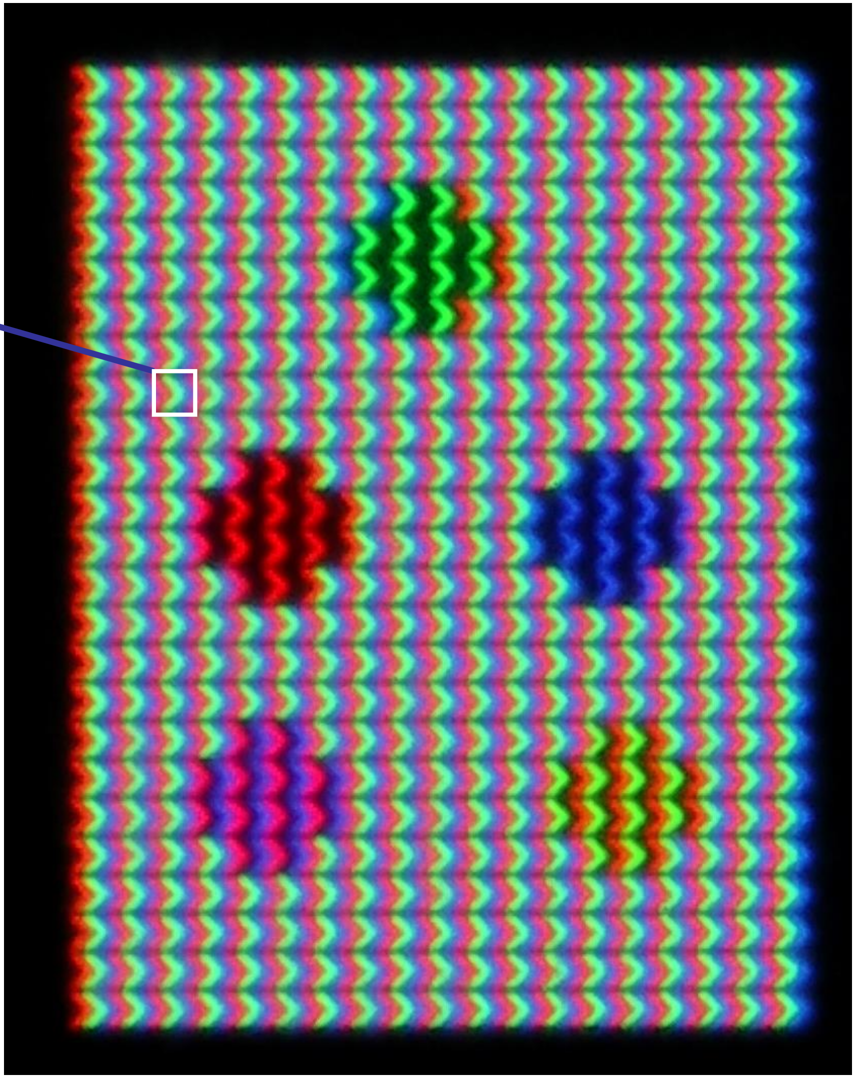
Basic Problem

Line Representations

Naïve Algorithm

Computer Screens: Raster Displays

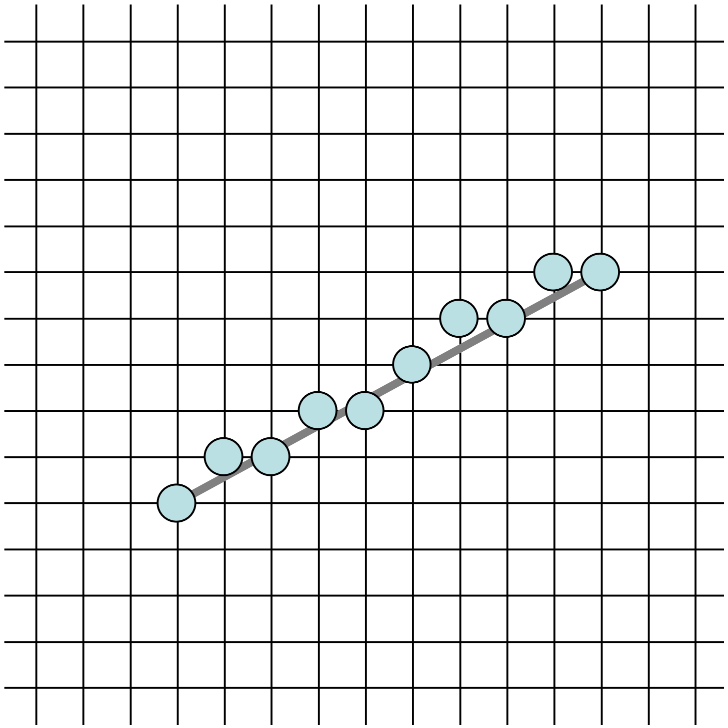
- pixel rasters
  - (usually) square pixels
  - rectangular raster
  - evenly cover the image
  - colors of pixels give impression of shapes (here: circles/dots)



Wikipedia user Rzr999

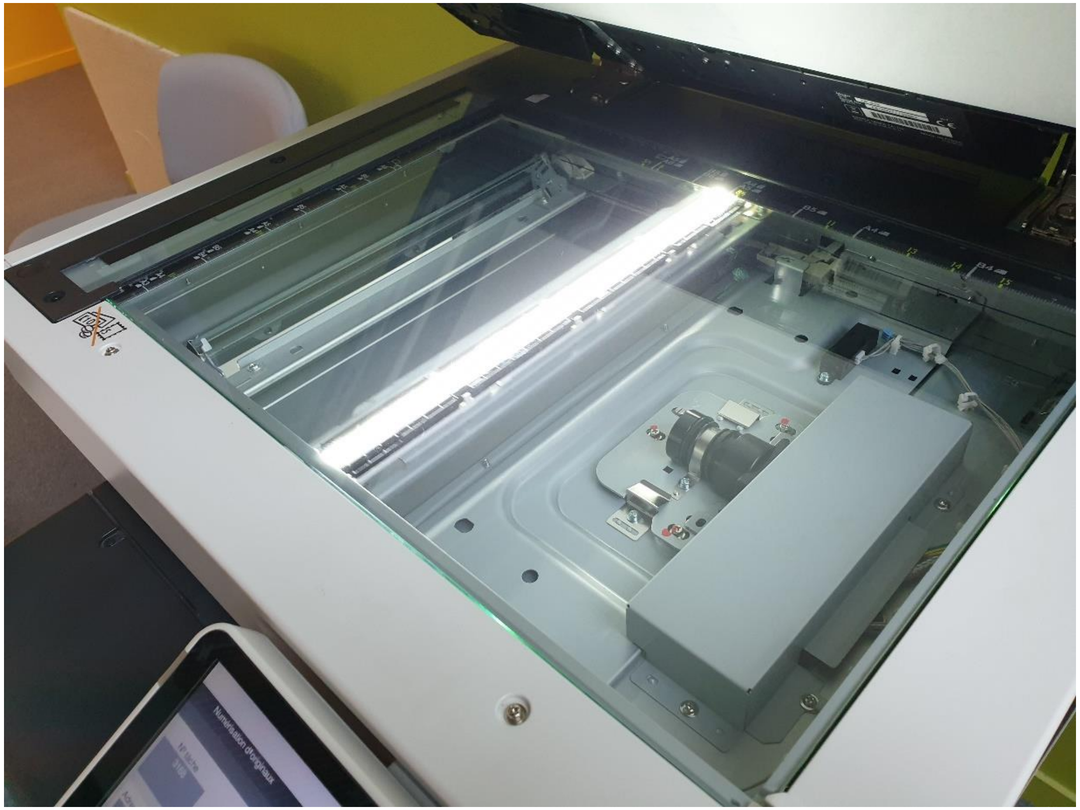
Computer Screens: Raster Displays

- problems
  - – no such things such as “lines”, “circles”, etc.
  - – need scan conversion
- yields pixel graphic
- non-raster display or printing technologies exist as well (plotter)



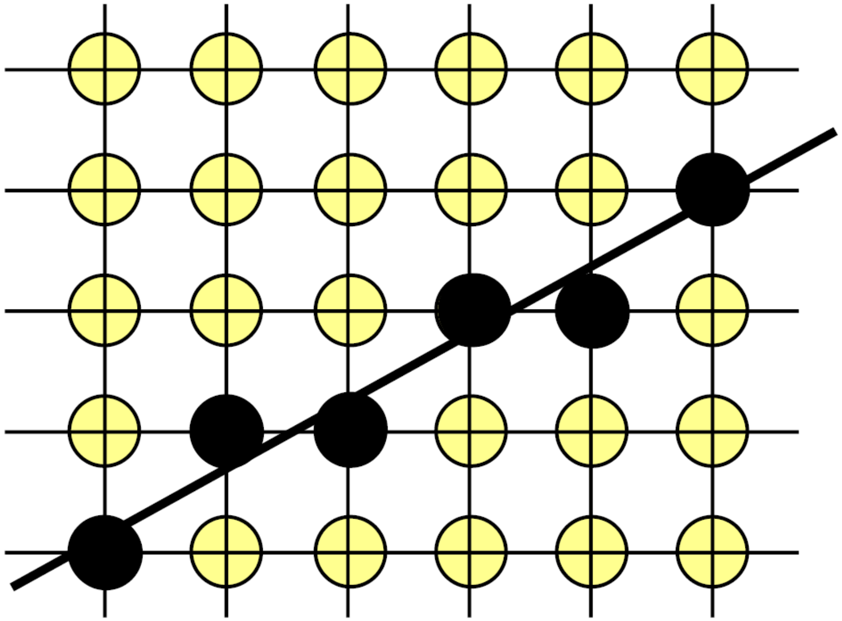
Scan Conversion

- to scan: get the right pixels, line by line
- like an image being scanned by a scanner

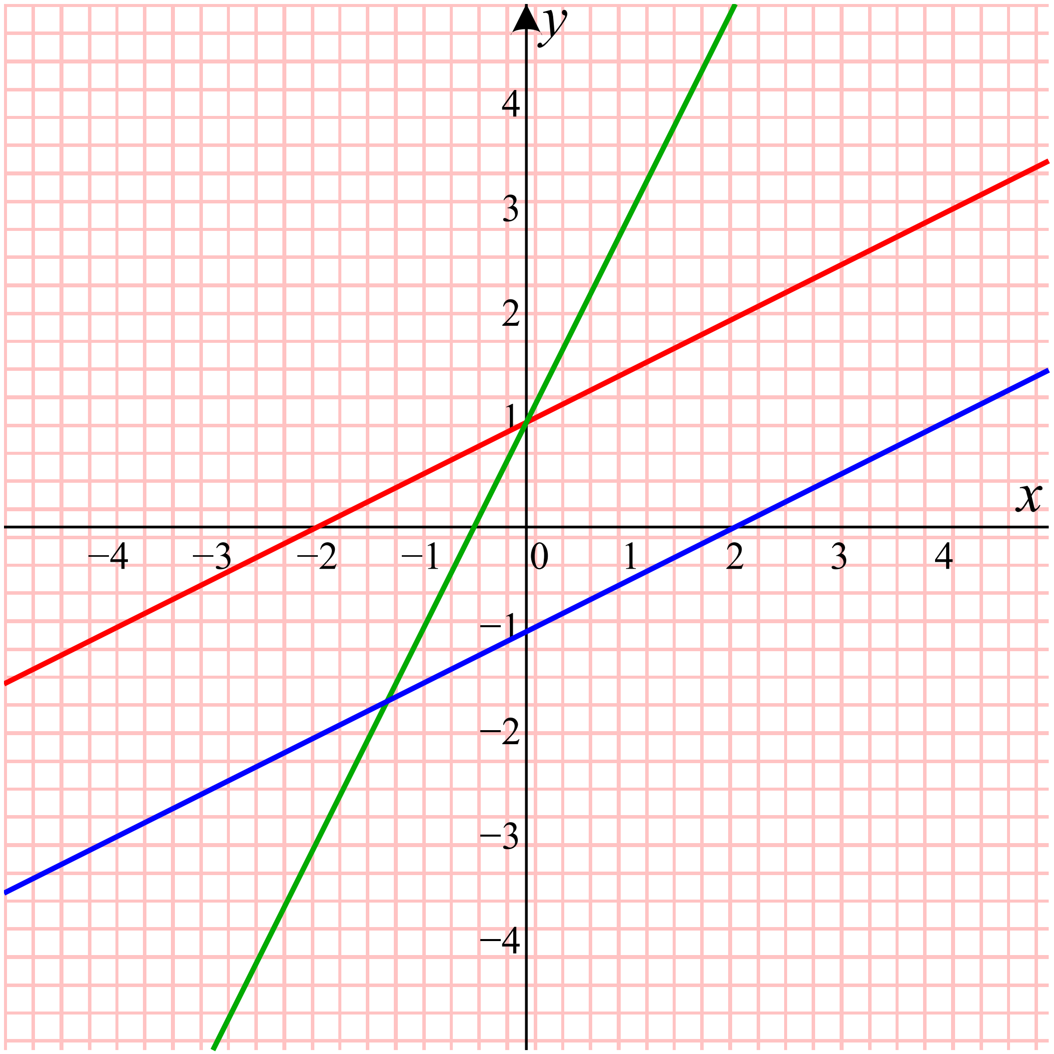


Goal: Draw Graphic Primitives

- graphic primitives: lines, circles, ellipsoids
- requirements:
  - efficiency
  - quality
- problem: how to show lines?
- task: determine the pixels to draw in black
- first: how to draw straight lines



Scan Conversion: Straight Lines

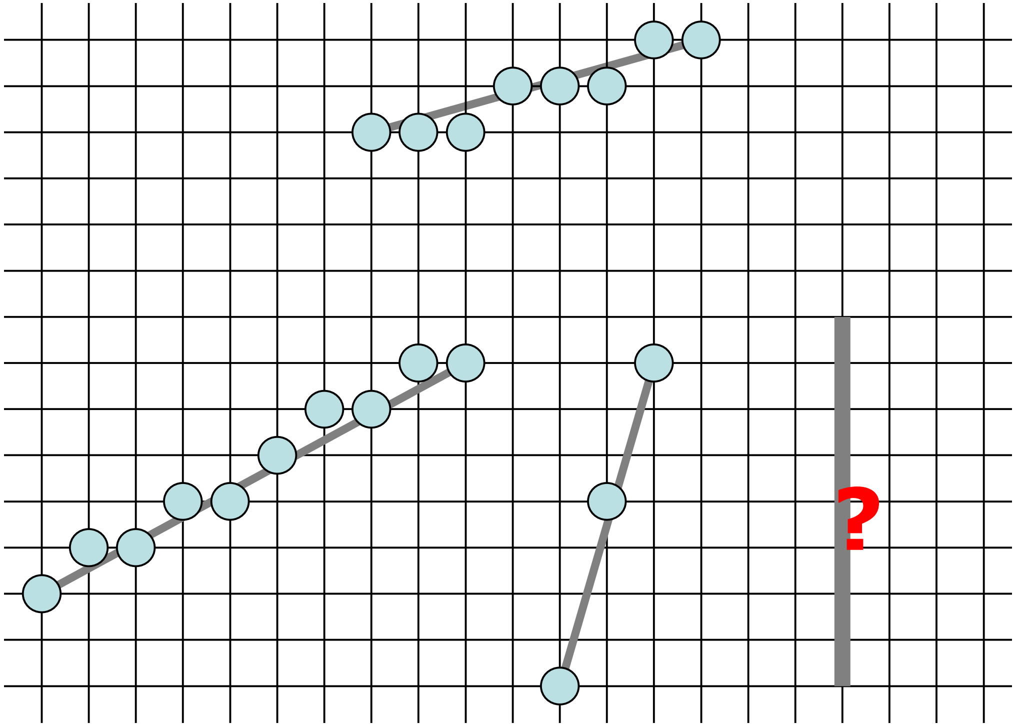


Lines: Mathematical Descriptions

- input: $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$
  $\Delta x = x_2 - x_{1;}\ \Delta y = y_2 - y_{1;}\ m = \Delta y / \Delta x$
- explicit equation: $f(x) = mx + n$
  $m = \Delta y/\Delta x$; n: intersection with *y*-axis
- parametric description: using parameter t
  $x = x_1 + t(x_2\text{-}x_1) = x_1 + t\Delta x$
  $y = y_1 + t(y_2\text{-}y_1) = y_1 + t\Delta y$
- implicit equation : $F(x, y) = ax + by + c = 0$
  $\rightarrow$ advantage for raster conversion

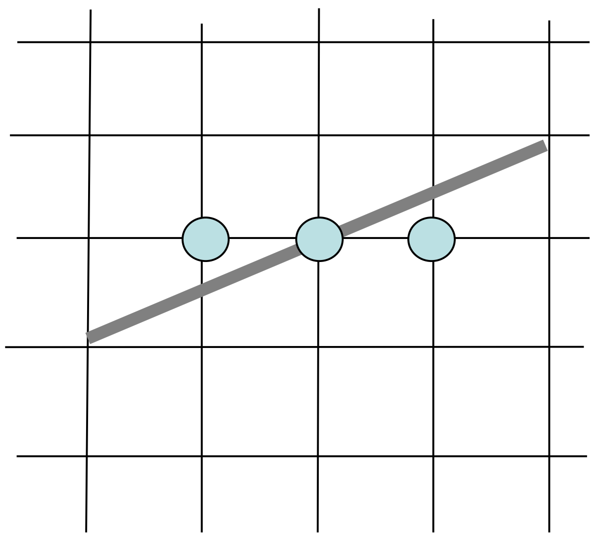
Naïve Algorithm

- use explicit equation $f(x) = mx + n$ and iterate
- problems:
  - accuracy (floating point computations)
  - efficiency (multiplications)
  - rounding
  - sometimes missing pixels or not defined at all



Implicit Line Equation: Advantage

- not only defines the line but tells us if a pixel is on the line or not
- $F(x, y) = ax + by + c$
- $F(x, y) > 0 \rightarrow$ below the line
- $F(x, y) = 0 \rightarrow$ on the line
- $F(x, y) < 0 \rightarrow$ above the line
- we can determine on which side of the (mathematical) line a (discreet) pixel lies!



Implicit Line Equation: Getting there

- $F(x, y) = ax + by + c = 0$
- determining a, b, (and c)

$$f(x) = mx + n;\ m = \Delta y / \Delta x$$

$$y = mx + n$$

$$0 = mx - y + n$$

$$0 = \Delta y\ x - \Delta x\ y + n'$$

$$F(x, y) = \Delta y\ x - \Delta x\ y + n' = 0$$

$$\rightarrow \quad a = \Delta y;\ b = -\Delta x$$

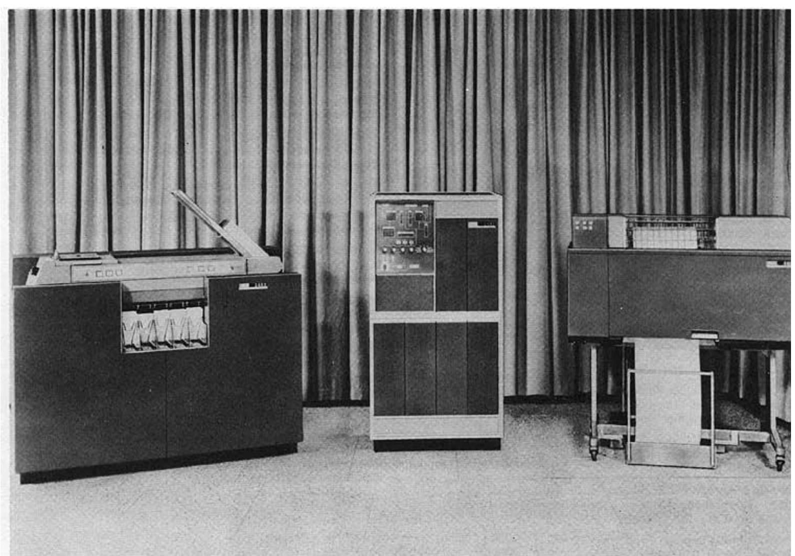
(c = n' can be determined using one point but we won't really need it)

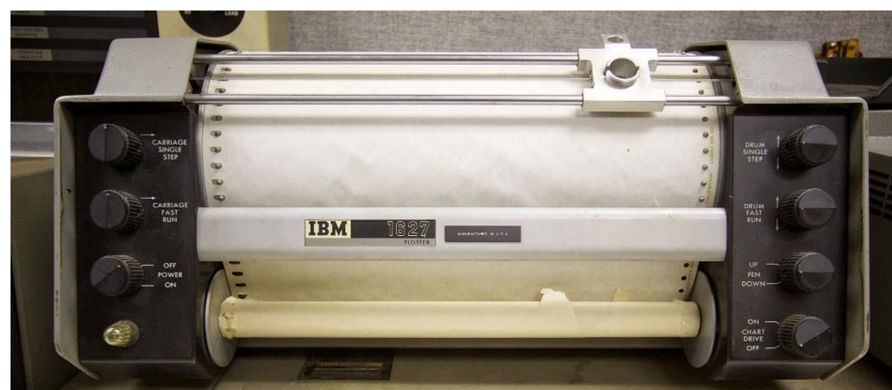
Bresenham’s Midpoint Algorithm

for Lines

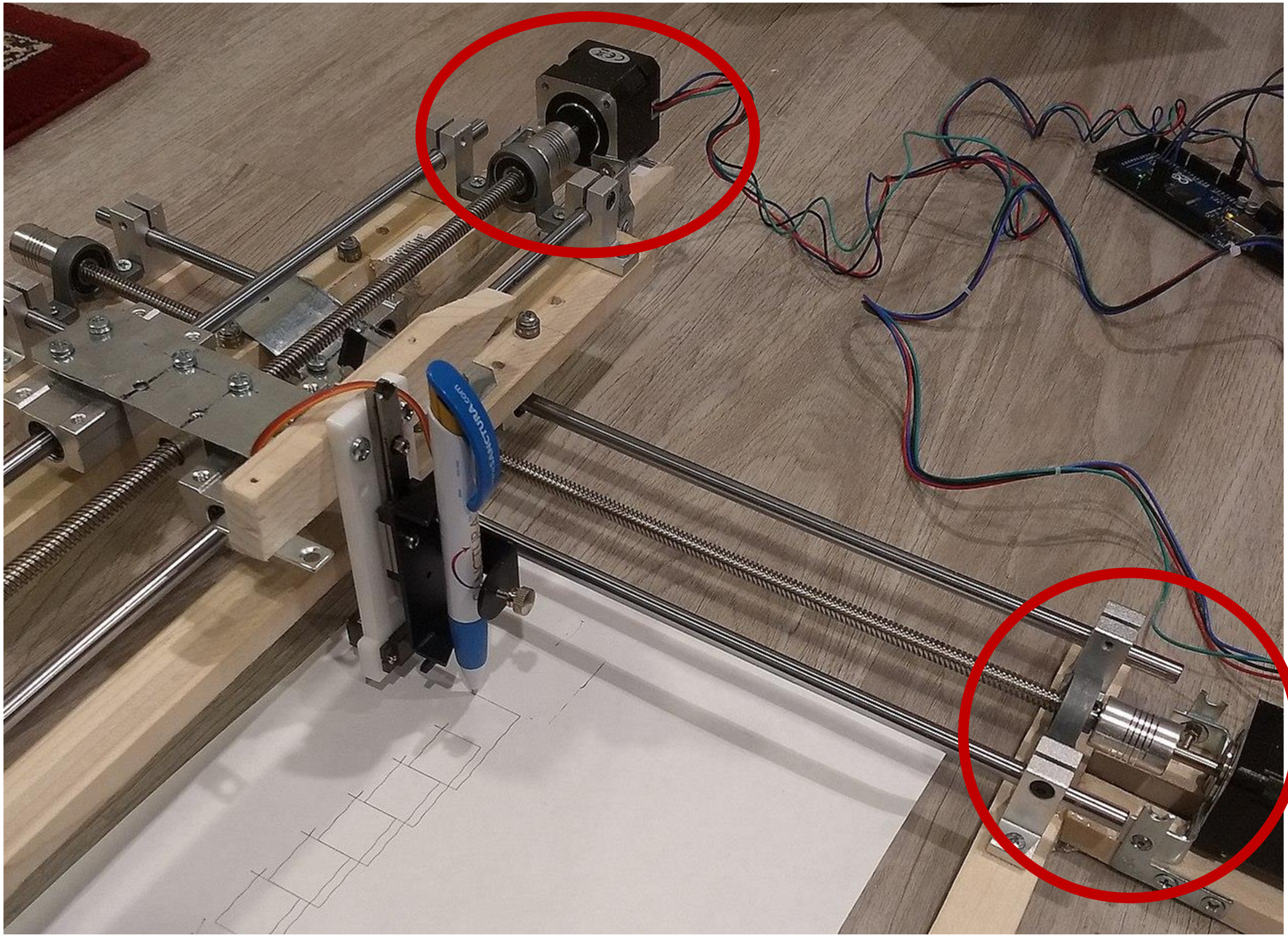
Bresenham Midpoint Algorithm

- by Jack Bresenham (1965) for controlling a plotter:
  - Integer arithmetic (fast, precise)
  - no division, as few multiplications as possible





Pixel Graphics for Vector Plotters?

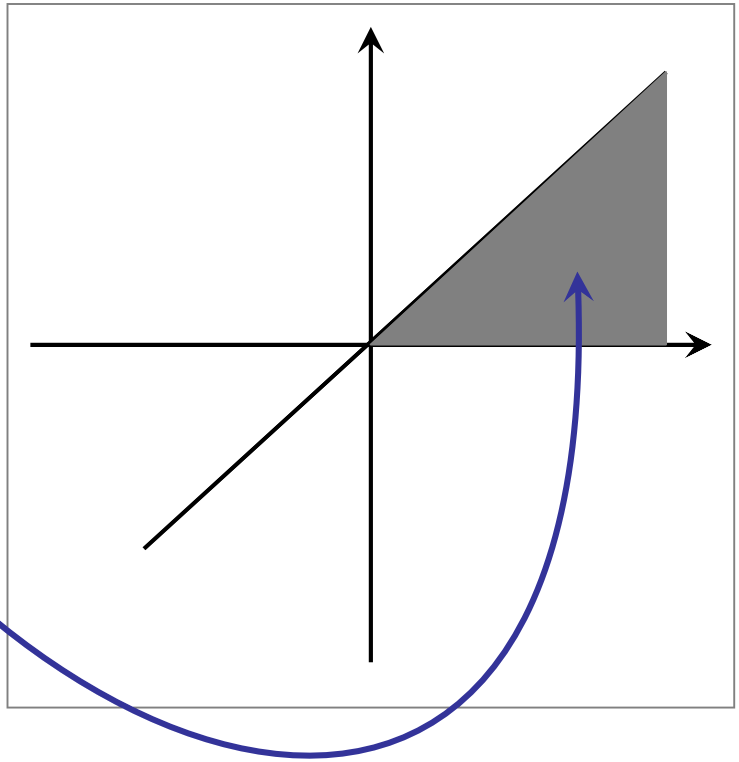


stepper motors essentially driven on a (fine) pixel raster

image by Wikipedia user Chiffre01

Bresenham Midpoint Algorithm

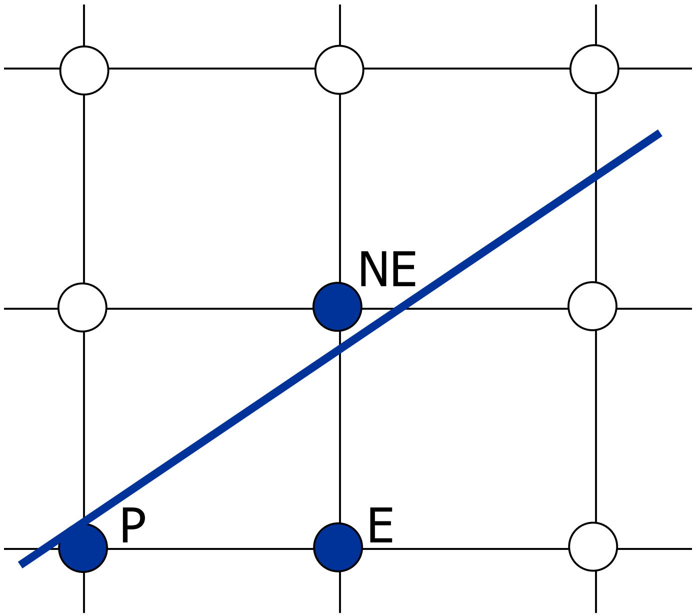
- constraints:
  - slope (m) between 0 and 1 → one octant
  - all pixels on Integer raster
  - this also means rounding start and end point



- later: generalize to other octants

Bresenham Midpoint Algorithm

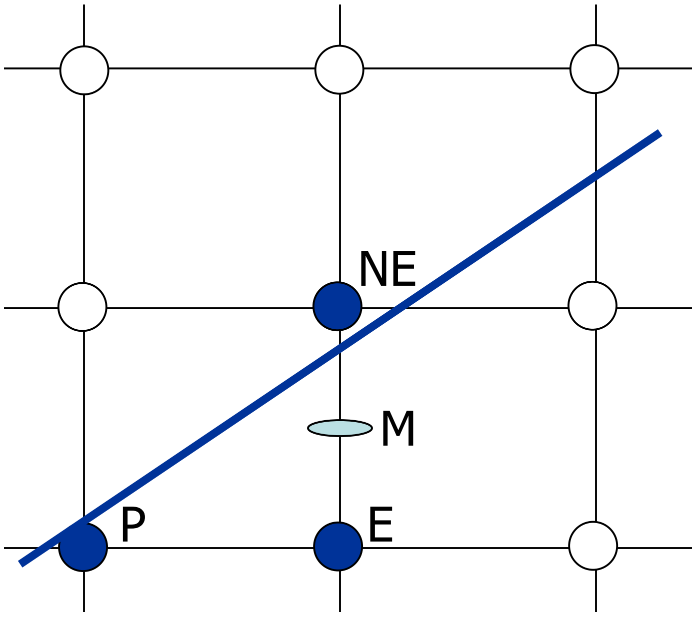
- general idea: iterative positioning of pixels
- previous pixel: P
- next pixels: NE or E
- decision depending on whether line intersection closer to NE or E
- iterate!



Bresenham: How to Decide?

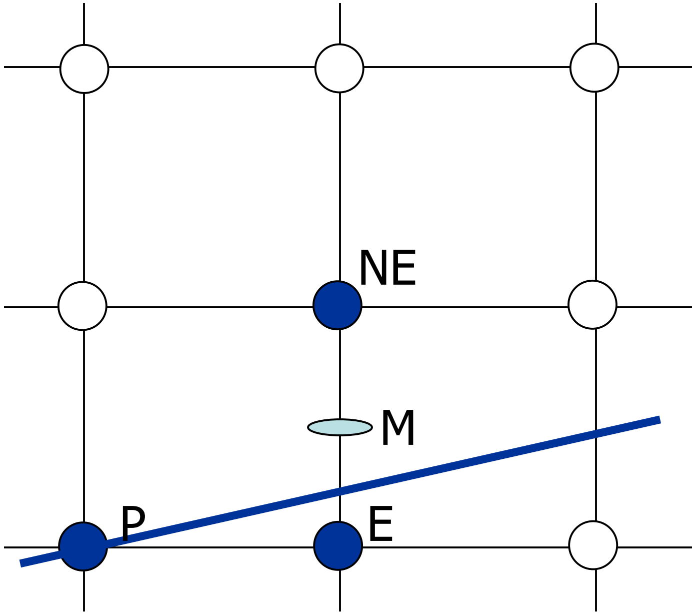
- implicit equation revisited
- easier to determine whether midpoint (M) is above or below the line → F(M) current pixel: P(x, y) midpoint M(x+1, y+½)

$$F(M) = F(x+1, y+\tfrac{1}{2})$$



Bresenham: How to Decide?

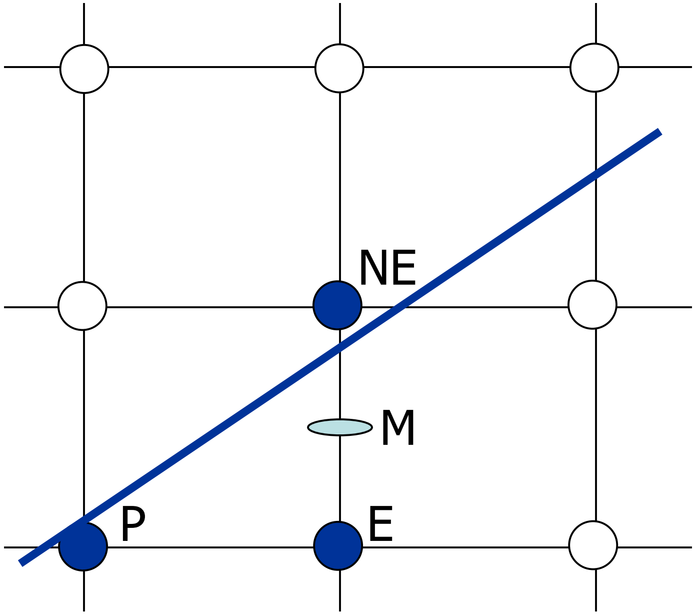
- if $F(M) < 0$
  $\rightarrow$ midpoint above line
  $\rightarrow$ E next pixel



Bresenham: How to Decide?

- if $F(M) < 0$
  $\rightarrow$ midpoint above line
  $\rightarrow$ E next pixel

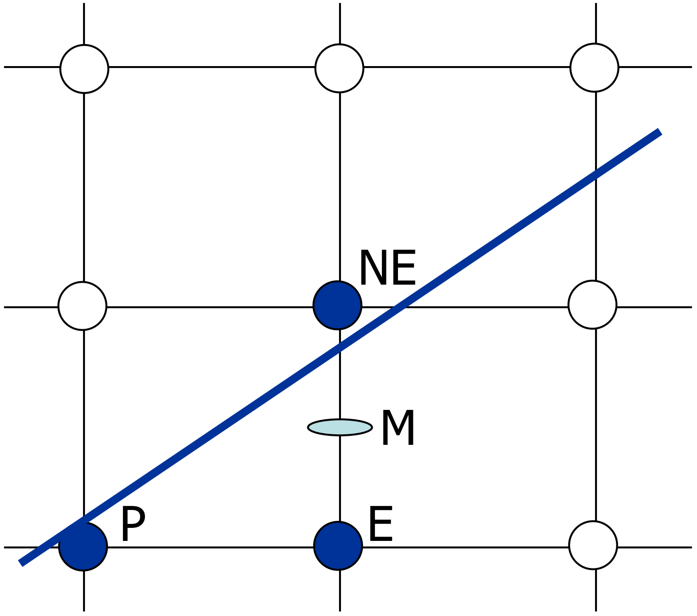
  if $F(M) \geq 0$
  $\rightarrow$ midpoint below line
  $\rightarrow$ NE next pixel



Bresenham: How to Decide?

- if F(M) < 0
  → midpoint above line
  → E next pixel

  if F(M) ≥ 0
  → midpoint below line
  → NE next pixel



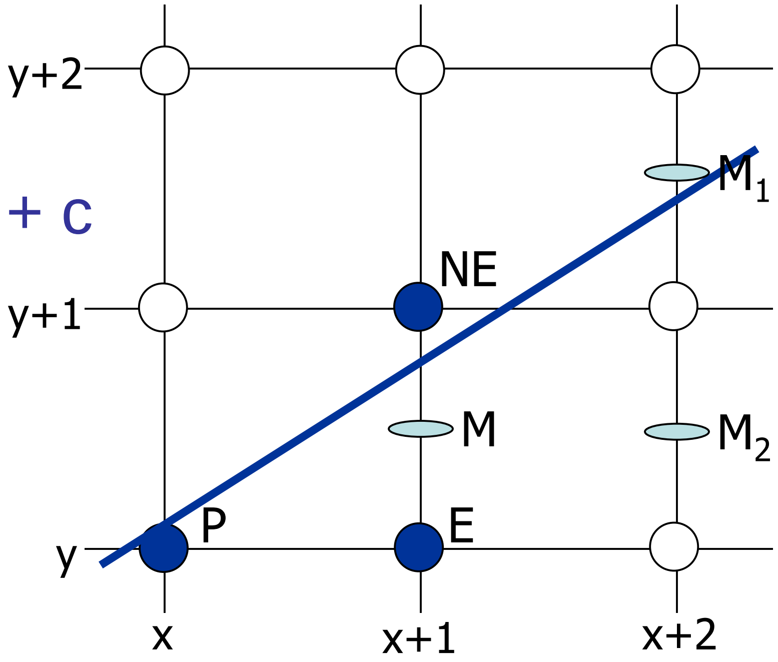
- decision variable:
  $d = F(M) = F(x+1, y+½)$
- BUT: we do not re-compute d each time
- INSTEAD: we compute how it changes!

Bresenham: Iteration, Case 1

NE is next pixel and M_1 next midpoint

- $F(M) = F((x+1), (y+½))$
  $= a(x+1) + b(y+½) + c$
- $F(M_1) = F((x+2), (y+{}^3/_2))$
  $= F((x+1+1), (y+½+1))$
  $= a(x+1+1) + b(y +½+1) + c$
- $F(M_1) - F(M) = a + b$
  $F(M_1) = F(M) + a + b$
- we know: $a = \Delta y$; $b = -\Delta x$
  $F(M_1) = F(M) + \Delta y – \Delta x$
  $d' = d + \Delta y – \Delta x$

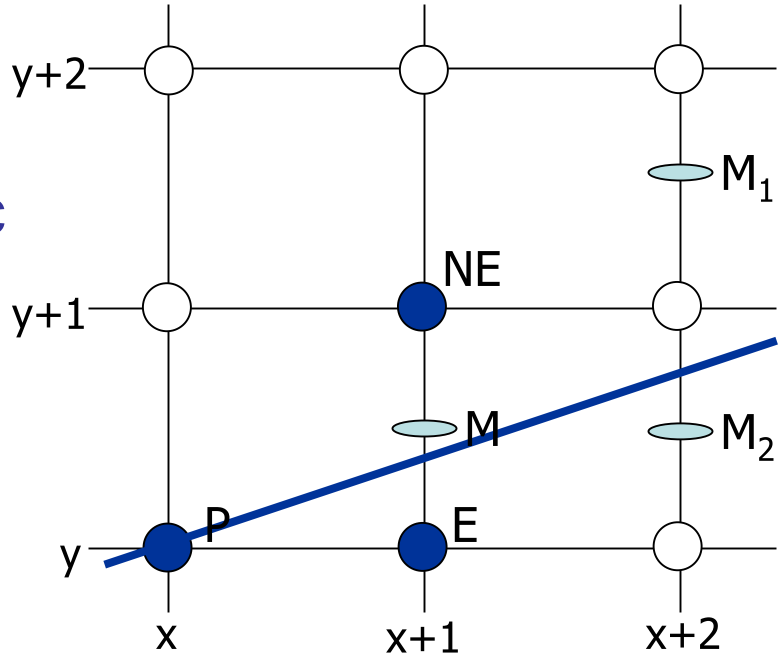
$\Delta d_{NE} = \Delta y – \Delta x$



Bresenham: Iteration, Case 2

E is next pixel and M_2 next midpoint

- $F(M) = F((x+1), (y+½))$
  $= a(x+1) + b(y+½) + c$
- $F(M_2) = F((x+2), (y+½))$
  $= F((x+1+1), (y+½))$
  $= a(x+1+1) + b(y+½) + c$
- $F(M_2) - F(M) = a$
  $F(M_2) = F(M) + a$
- we know: $a = \Delta y$;
  $F(M_2) = F(M) + \Delta y$
  $d' = d + \Delta y$
  $\Delta d_E = \Delta y$



Bresenham: Algorithm

- algorithm overview
  - first pixel = line starting point (rounded)
  - compute d = F(M)
    - select E or NE accordingly
    - set pixel
    - update d according to choice
    - increment x and iterate
  - terminate when x_2 is reached
- how to compute d_0?

Bresenham: Computing d_0

- line starts at $P_1(x_1, y_1)$

  $\rightarrow d_0 = F(M_1)$

  $= F((x_1+1), (y_1+½))$

  $= a(x_1+1) + b(y_1+½) + c$

  $= ax_1 + a + by_1 + ½b + c$

  $= F(x_1,y_1) + a + ½b$
- P_1 lies on the line $\rightarrow F(x_1,y_1) = 0$

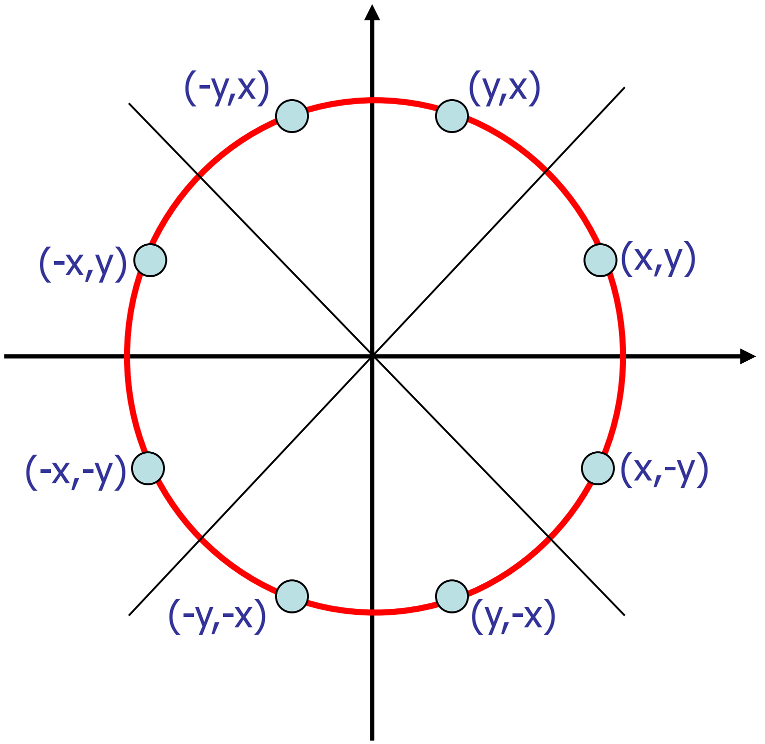
  $\rightarrow d_0 = a + ½b$
- problem: we want Integer values!

Bresenham: Computing d_0

- we are only interested in sign of d!
  $\rightarrow$ multiply everything by 2!
- multiplication without effect on the sign
  $\rightarrow \quad d_0 = 2a + b$
  $\quad\quad\quad = 2\Delta y - \Delta x \; (a = \Delta y;\ b = -\Delta x)$
  $\rightarrow \Delta d_E = 2\Delta y$
  $\rightarrow \Delta d_{NE} = 2\Delta y - 2\Delta x$

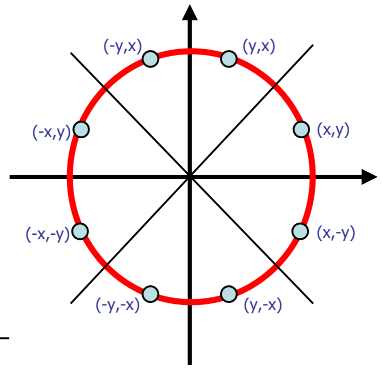
Bresenham: Extension to All Slopes

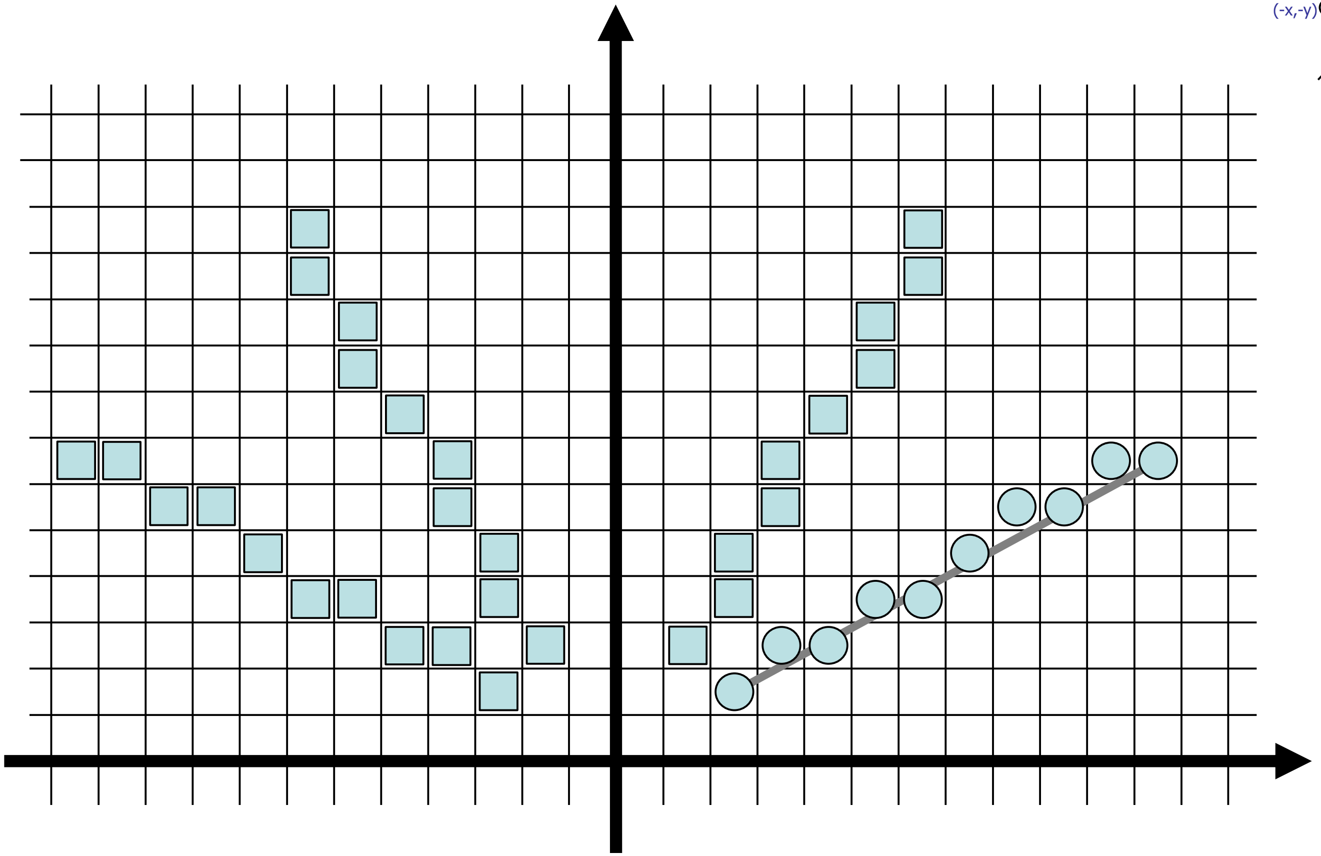
- use of symmetry:
  - compute line as before
  - changing signs of x and/or y before drawing a pixel
  - switching x and y
  - combinations of these



Bresenham: Extension to All Slopes

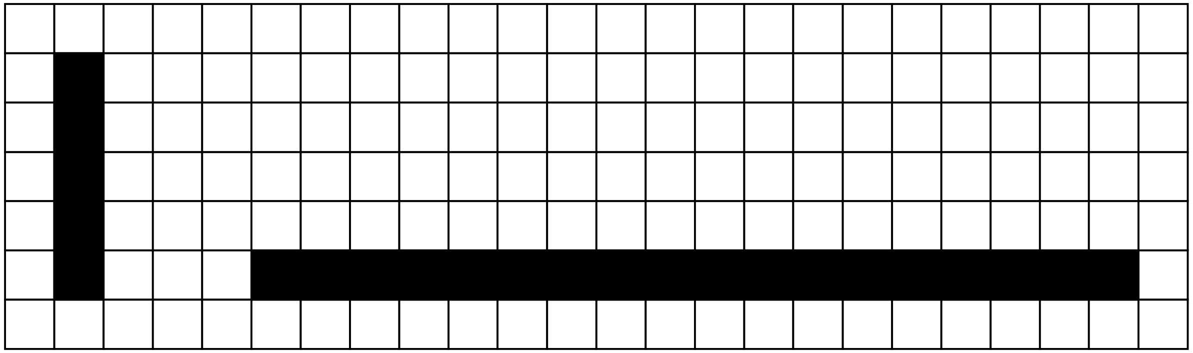
- examples for using symmetry to draw lines with other slopes



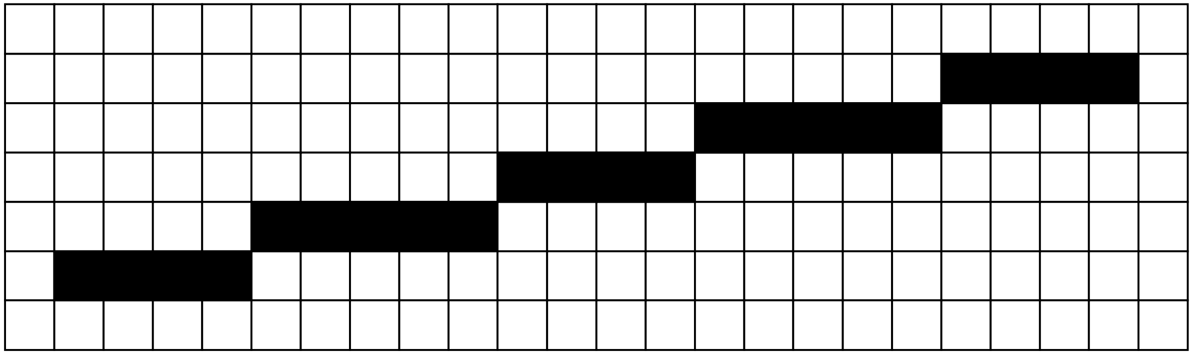


Bresenham: Possible Extensions

- special cases for lines along axes



- looking for patterns in lines (when/why?)



$\rightarrow$ slope **m** is always a **rational number**

Bresenham-Lines: Summary

- incremental algorithm
- using only Integer arithmetic
- using only additions during iterations
- multiplications only for setup
- using symmetry to extend to all octants
- FAST!!!

Bresenham's Midpoint Algorithm

for Circles

Let’s Have More Fun: Circles!

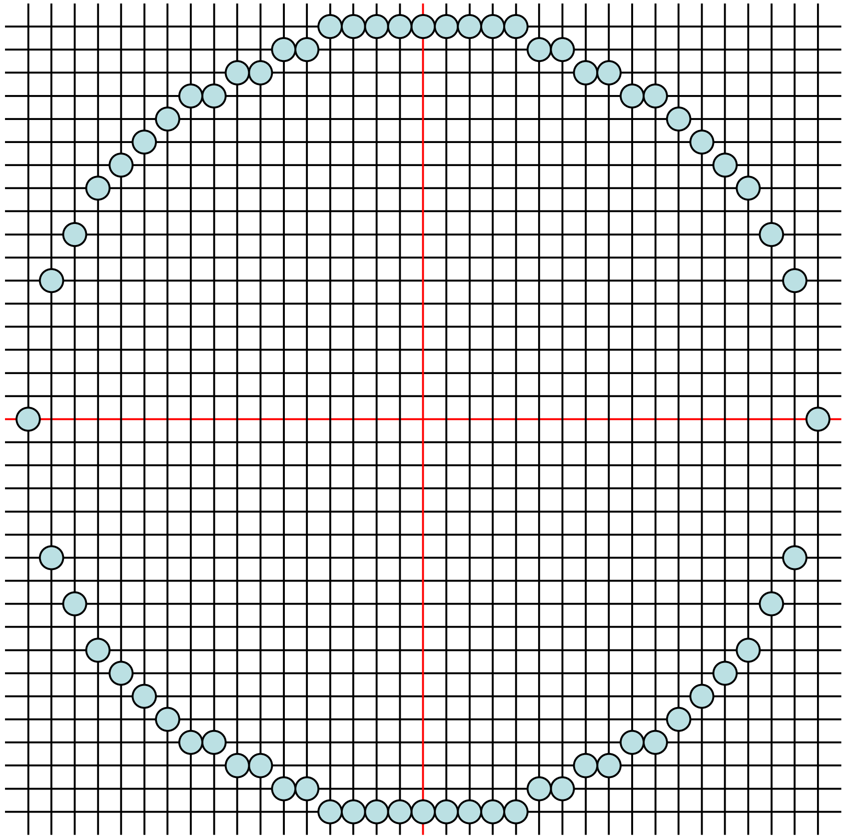
- input: center point $C(x_c, y_c)$ and radius r
- circle equation: $F(x,y) = x^2 + y^2 = r^2$
  if $C = (0, 0)$
- general: $F(x,y) = (x-x_c)^2 + (y-y_c)^2 = r^2$
- naïve approach to draw circle:
  solve for y

$$y = \pm\sqrt{r^2 - x^2}$$

  and iterate over x

Problems with Naïve Algorithm

- expensive computations
  - square roots
  - powers of 2
  - inaccurate!
  - sloooow!
- incomplete pixels where $|x| \approx r$
- we need something better!



Parametric Approach for Circles

- use parametric equation:
  $x = r \cos \varphi$ and $y = r \sin \varphi$
- iterate over φ
- no problems with holes at $|x| \approx r$ anymore
- but: trigonometric functions expensive to compute
- still not efficient enough!

Bresenham Midpoint for Circles

- use same idea as with lines:
  implicit function and decision point
- $F(x,y) = x^2 + y^2 - r^2$

  $= 0$ for points on the circle

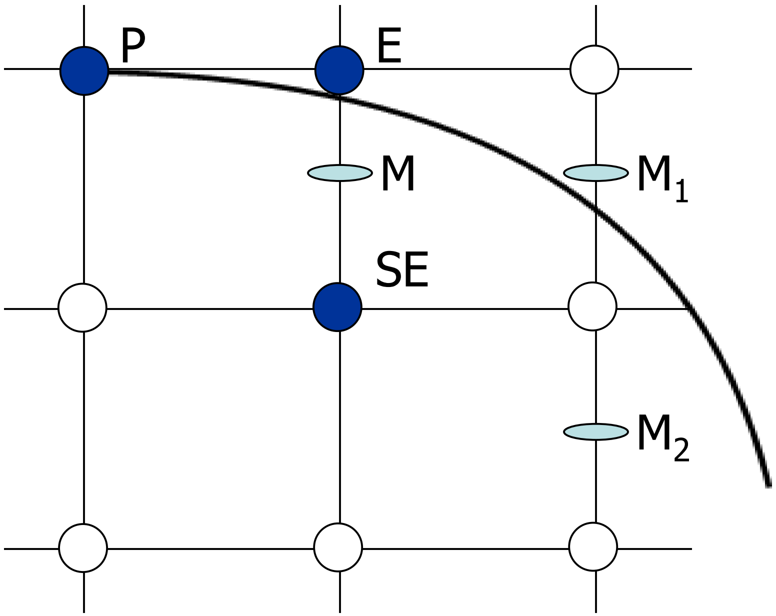
  < 0 for points within the circle

  > 0 for points outside of the circle

  (assuming the circle centered at 0,0)

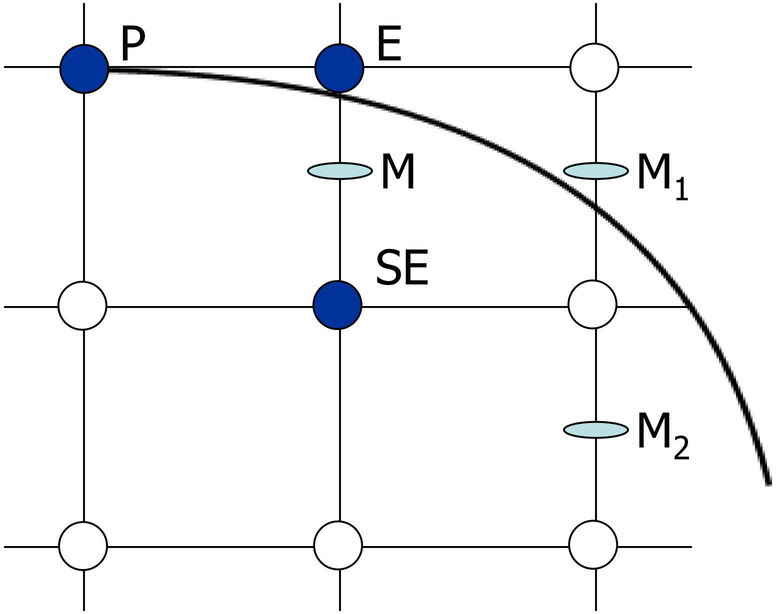
Bresenham Midpoint for Circles

- use only one octant again
- from pixel P decide between E and SE
- based on midpoint's position to circle
- goals (again):
  - use incremental algorithm
  - avoid divisions and multiplications



Bresenham Midpoint for Circles

- midpoint $M(x+1, y-½)$
- decision variable
  $d = F(M)$
  $= (x+1)^2 + (y-½)^2 – r^2$
- select E if $d < 0$
  (circle is above M)
- select SE if $d \geq 0$
  (circle on or below M)
- we now need
  to compute the increments of d again



Bresenham Midpoint for Circles

Case 1: $d < 0$; select E

- $d = F(M)$

  $= F((x+1), (y-½))$

  $= (x+1)^2 + (y-½)^2 – r^2$

- $d' = F(M_1)$

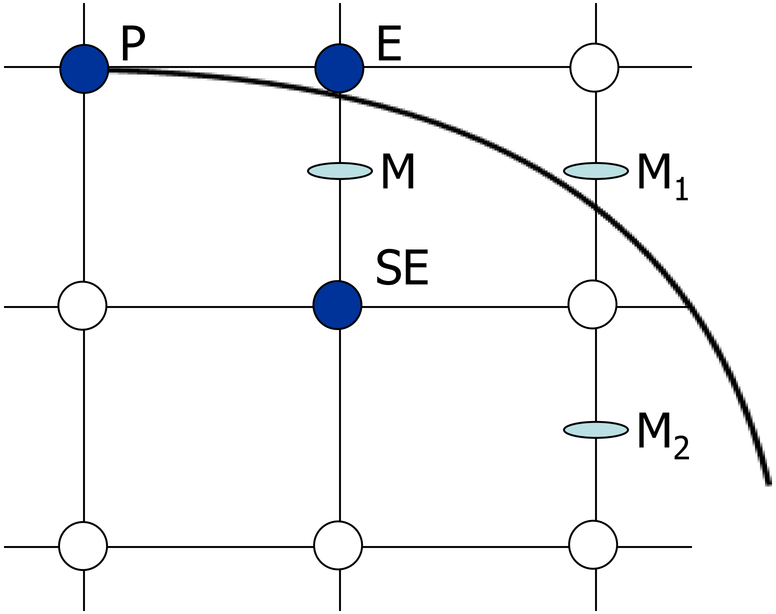
  $= F((x+2), (y-½))$

  $= (x+2)^2 + (y-½)^2 – r^2$

  $= (x+1)^2 + (y-½)^2 – r^2$

  $+ 2x + 3$

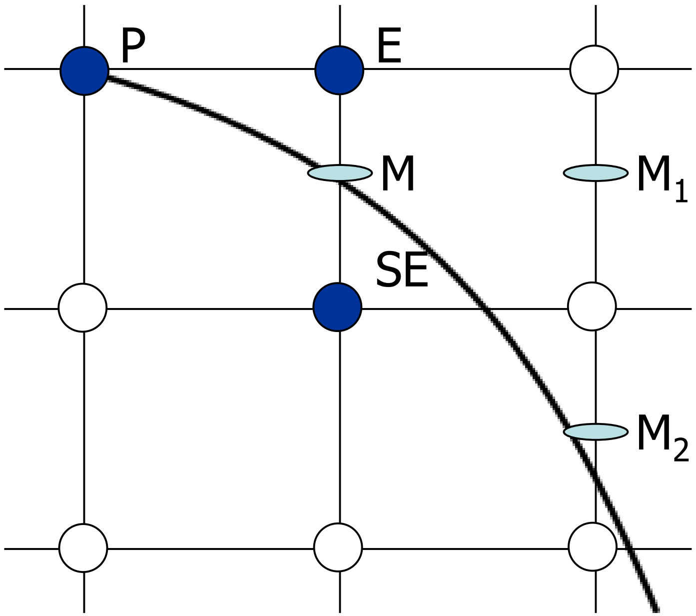
  $= F(M) + 2x + 3 \quad \rightarrow \quad \Delta d_E = 2x + 3$



Bresenham Midpoint for Circles

Case 2: $d \geq 0$; select SE

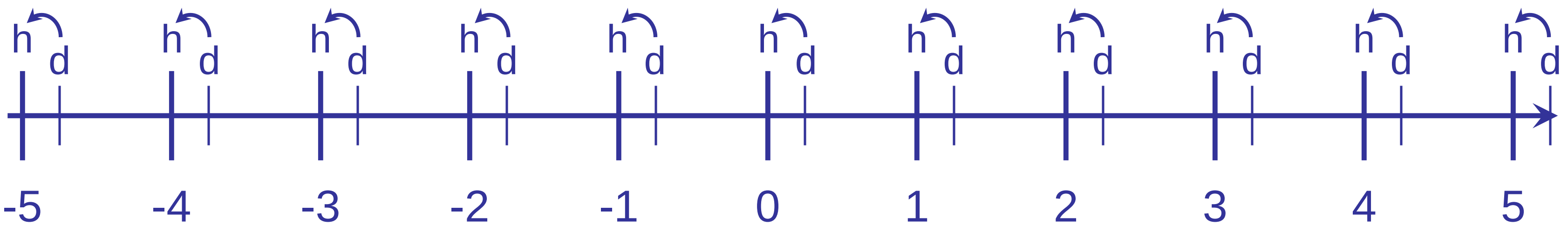
- $d = F(M)$
  $= F((x+1), (y-½))$
  $= (x+1)^2 + (y-½)^2 – r^2$
- $d' = F(M_2)$
  $= F((x+2), (y-{}^3/_2))$
  $= (x+2)^2 + (y-{}^3/_2)^2 – r^2$
  $= (x+1)^2 + (y-½)^2 – r^2$
  $+ 2x + 3 - 2y + 2$
  $= F(M) + 2x – 2y + 5 \rightarrow \Delta d_{SE} = 2(x-y) + 5$



Bresenham Midpoint for Circles

Initial value of d

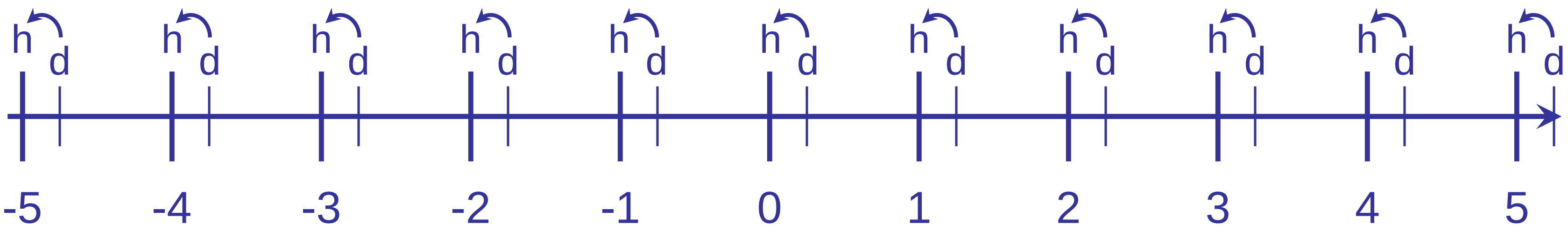
- first pixel $P(0, r) \rightarrow$ first midpoint $M(1, r-½)$
  $d_0 = F(1, r-½) = 1^2 + (r-½)^2 - r^2 = {}^5/_4 - r$
- d_0 is not Integer!
- but any d is only ¼ away from an Integer
- mathematical trick: new decision variable
  $h ::= d - {}^1/_4$ such that $h_0 = 1 - r$



Bresenham Midpoint for Circles

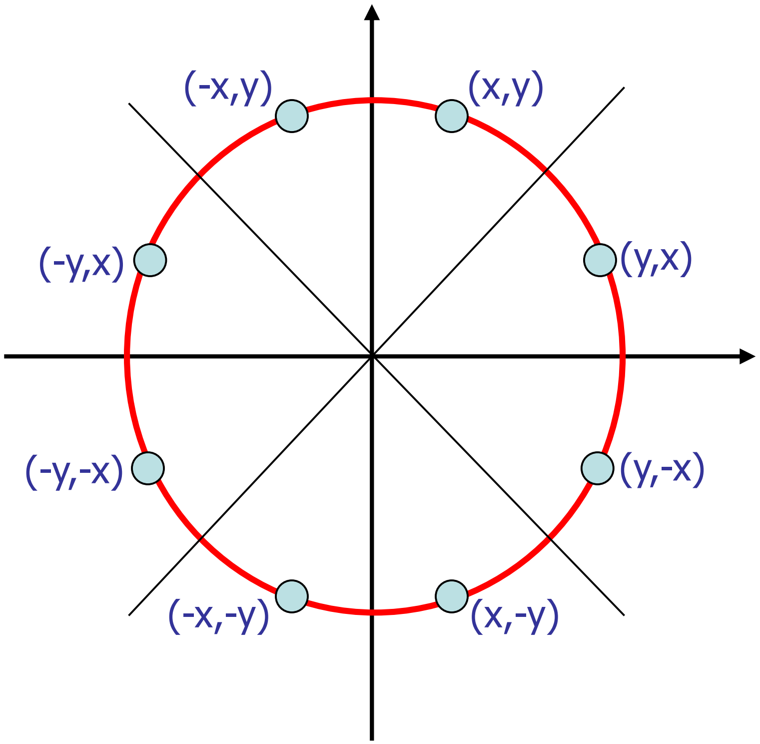
$h ::= d - {}^1/_4$ such that $h_0 = 1 - r$

- decision $\mathbf{d < 0}$ turns into $\mathbf{h < -{}^1/_4}$
- **but**: computation of h uses only Integers
  – we don't care about actual values, we **only** care about positive or negative
  $\rightarrow$ we can test $h < 0$
  – computationally equivalent!



Bresenham Circle Symmetry

- similar to line octants:
  - changing signs of *x* and/or *y* before drawing pixel
  - switching *x* and *y*
  - combinations of these
- set all eight pixels at the same time
- for circles not centered at (0, 0): offset pixels



Bresenham Circle Summary

- efficient algorithm
  - incremental and Integer arithmetic
  - $^1/_8$ of the circle only
- still multiplications needed for increments
  - $\Delta d_E = 2x + 3$ and $\Delta d_{SE} = 2(x-y) + 5$
  - line algorithm had constant increments
  - can we do this here, too?
- the fun goes on: second order differences
  - idea: compute increments of increments
  - how: consider two steps in advance

Bresenham's Midpoint Algorithm

for Circles
(and other quadratic curves):
Second Order Differences

Second Order Differences

Case 1: E was selected

- pixel was (x, y) and becomes (x+1, y)
- increments change as well

old	new
$\Delta d_E \ = 2x + 3$	$\Delta d_E \ = 2(x+1) + 3$
$\Delta d_{SE} = 2x - 2y + 5$	$\Delta d_{SE} = 2(x+1) - 2y + 5$

- differences of the increments

$\Delta^E \Delta d_E \ = 2$

$\Delta^E \Delta d_{SE} = 2$

Second Order Differences

Case 2: SE was selected

- pixel was (x, y) and becomes (x+1, y-1)
- increments change as well

old	new
$\Delta d_E = 2x + 3$	$\Delta d_E = 2(x+1) + 3$
$\Delta d_{SE} = 2x - 2y + 5$	$\Delta d_{SE} = 2(x+1) - 2(y-1) + 5$

- differences of the increments

$\Delta^{SE}\Delta d_E = 2$

$\Delta^{SE}\Delta d_{SE} = 4$

Application of 2nd Order Differences

slightly adjusted algorithm:

- setup $h_0 ::= d_0 - {}^1/_4 = 1 - r$
- setup $\Delta d_{E\_0} = 2x_0 + 3$ and $\Delta d_{SE\_0} = 2(x_0 - y_0) + 5$
- iterate until we reach $x = x_1$:
  - if $h \leq 0$ (i.e., $h \leq -{}^1/_4$ or $d \leq 0$): select E as next pixel
    - update h with Δd_E
    - update Δd_E with $\Delta^E \Delta d_E$ (i.e., Δd_E += 2)
    - update Δd_{SE} with $\Delta^E \Delta d_{SE}$ (i.e., Δd_{SE} += 2)
  - else ($h > 0$; i.e., $h > -{}^1/_4$ or $d > 0$): select SE as next pixel
    - update h with Δd_{SE}
    - update Δd_E with $\Delta^{SE} \Delta d_E$ (i.e., Δd_E += 2)
    - update Δd_{SE} with $\Delta^{SE} \Delta d_{SE}$ (i.e., Δd_{SE} += 4)

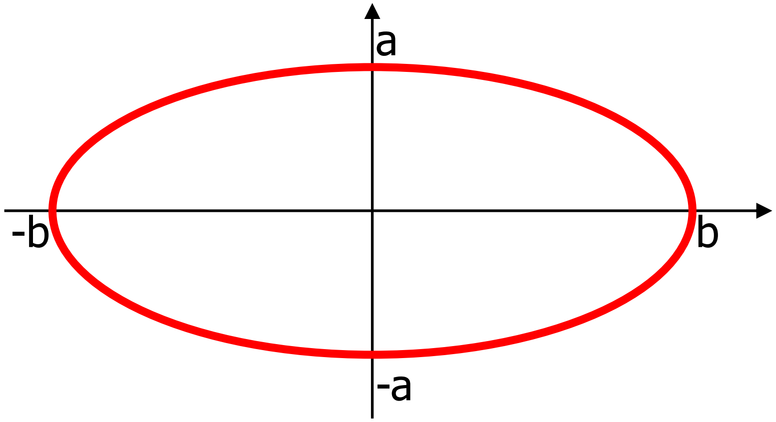
Bresenham's Midpoint Algorithm

for other curves

What about ellipses ...?

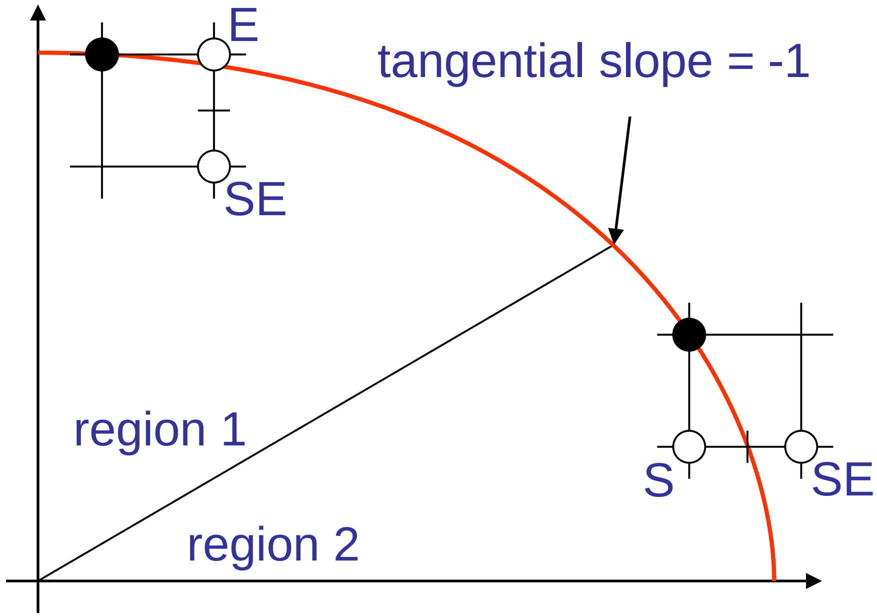
... or any other curves – the fun never stops

- similar as before:
  use simple case of implicit equation
  $F(x,y) = a^2x^2 + b^2y^2 - a^2b^2 = 0$
- only consider
  axis-aligned ellipses
- consider 1st quadrant
  (not octant this time)
- goal: incremental algorithm



Bresenham: Ellipses

- additional difficulty: 2 regions per quadrant
- change of selection mode during rastering
  - first: E or SE
  - then: S or SE
- change when slope changes from > -1 to < -1
- for first pixel where $a^2(x+1) \geq b^2(y-½)$



Bresenham: Ellipses

Region 1: selection of E or SE $M(x+1, y-½)$

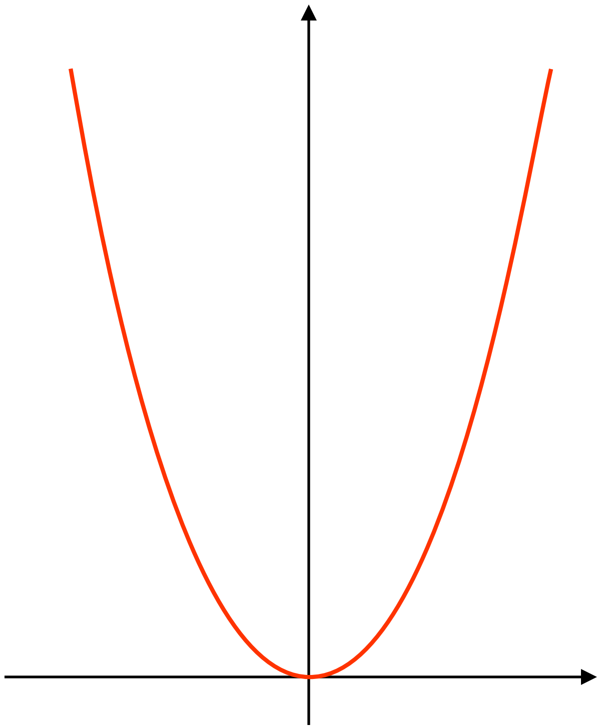
- $d = F(M) = a^2(x+1)^2 + b^2(y-½)^2 – a^2b^2$
- selecting E, new midpoint $M_1(x+2, y-½)$
  $d' = a^2(x+2)^2 + b^2(y-½)^2 – a^2b^2$
  $= a^2(x+1)^2 + b^2(y-½)^2 – a^2b^2 + a^2(2x + 3)$
- selecting SE, new midpoint $M_2(x+2, y-{}^3/_2)$
  $d' = a^2(x+2)^2 + b^2(y-{}^3/_2)^2 – a^2b^2$
  $= a^2(x+1)^2 + b^2(y-½)^2 – a^2b^2$
  $+ a^2(2x + 3) + b^2(-2y + 2)$
  $= d + a^2(2x + 3) + b^2(-2y + 2)$
- region 2 analogously

Bresenham: Ellipses

- d_0 based on 1^{st} pixel $(0,a) \rightarrow M(1, a\text{-}½)$
  $d_0 = F(M) = a^2 1^2 + b^2(a\text{-}½)^2 – a^2b^2$
  $= a^2 + a^2b^2 – ab^2 + {}^1/_4 b^2 – a^2b^2$
  $= a^2 – b^2({}^1/_4 – a)$
- when changing regions compute new d_0!
- rest similar to circle
- second order differences possible
- use symmetry, draw four pixels at a time
- move ellipse by offsetting drawn pixels

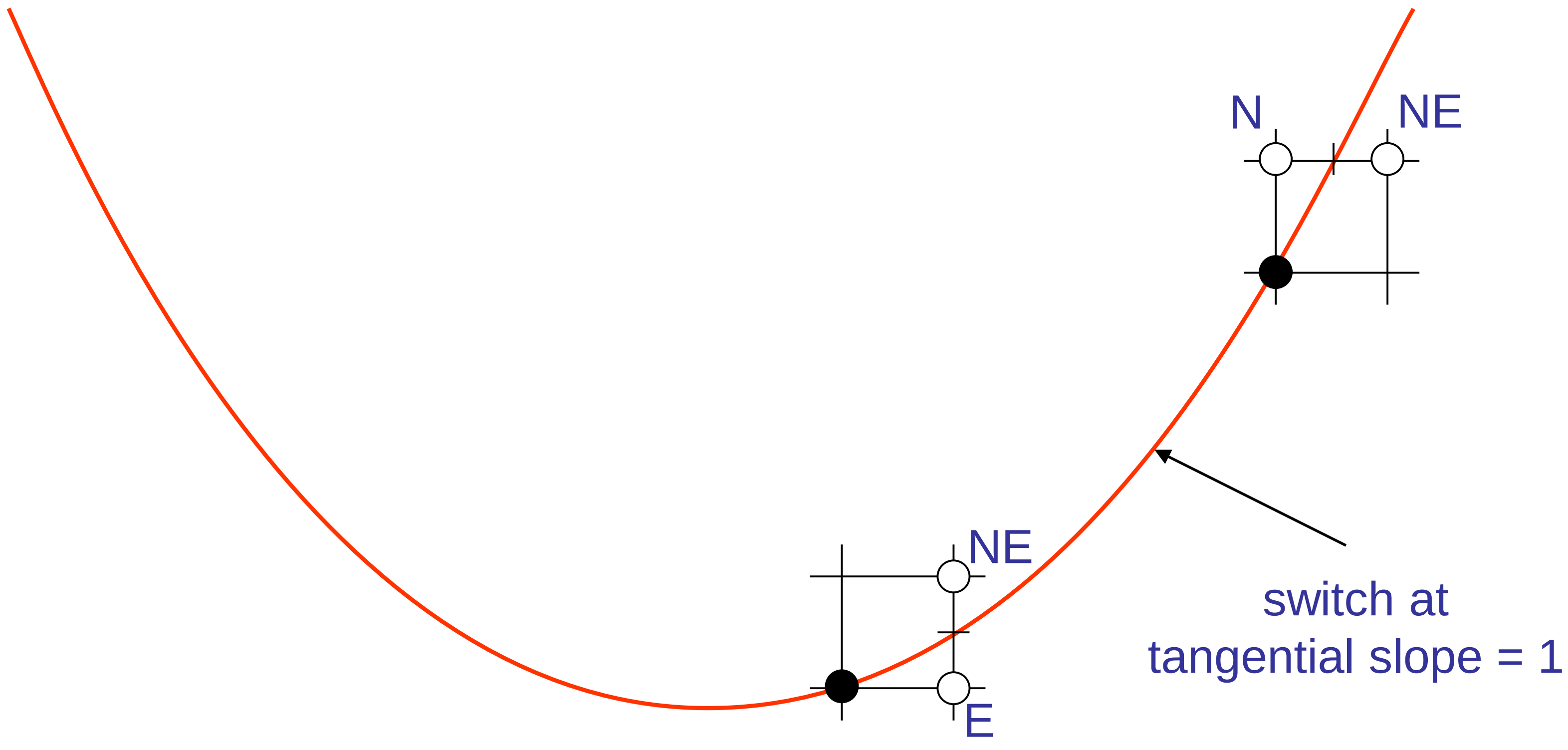
Bresenham: Yet Other Curves

- similar to circle and ellipse
- e.g., parabola $y = x^2$
- derive implicit form
  $F(x, y) = x^2 – y = 0$
- compute d and d’ and
  derive increments
- use n^{th} order differences
  for curves of degree n
- e.g., 2^{nd} order difference for $y = x^2$



Bresenham: Regions for Parabolas

- change regions if slope crosses 1 or -1



Bresenham Midpoint Algorithms

Summary

- fast and simple because
  - incremental technique using Integer arithmetic
  - avoiding multiplications/divisions
- possible extensions
  - many other curves (implicit equations needed)
  - curves not axis-aligned

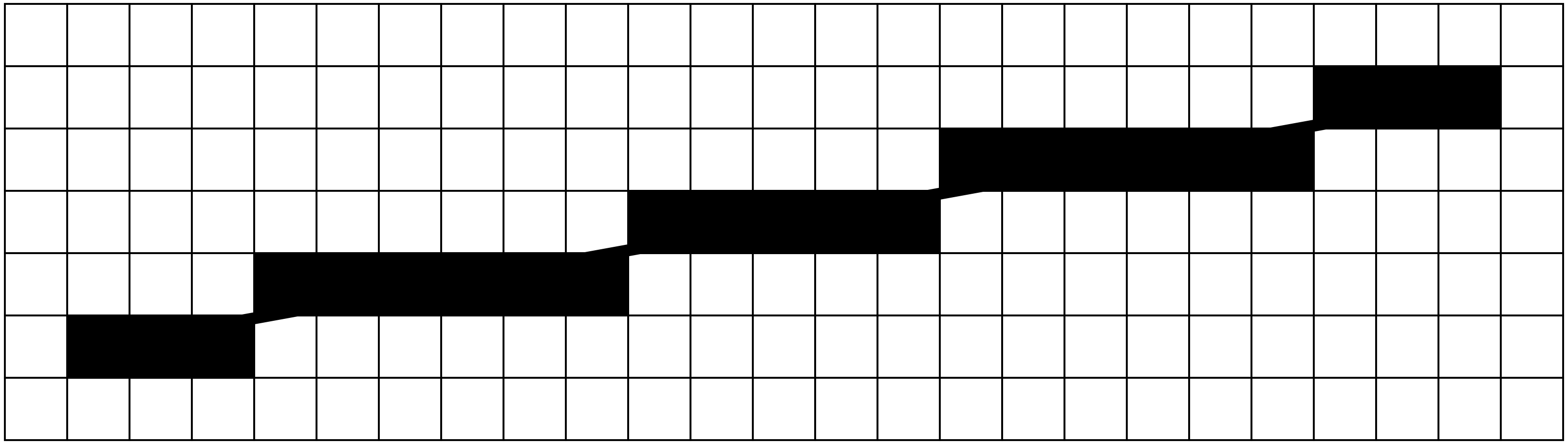
Anti-Aliasing

For Lines

And other Techniques

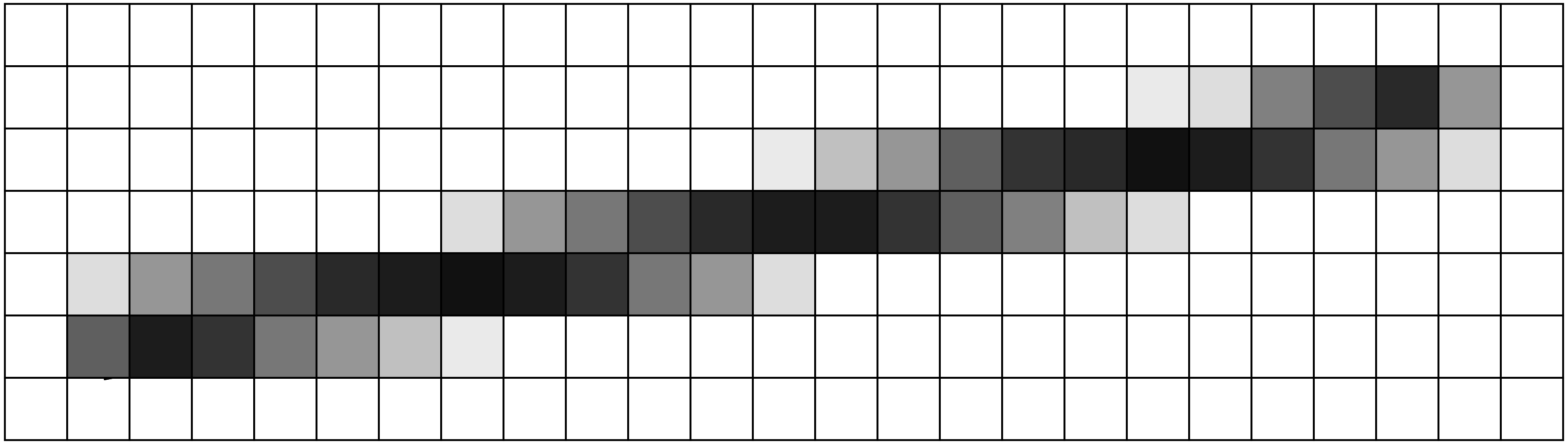
Anti-Aliasing for Lines

- Bresenham's midpoint algorithm:
  - jaggy shape due to discrete pixels
  - perceived width varies
    (e.g., diagonal vs. horizontal or vertical)



Anti-Aliasing for Lines

- 1 pixel wide line assumed
  - gray values from pixel coverage
  - derived from midpoint value
- OpenGL: **`glEnable(GL_LINE_SMOOTH)`** plus a **`glHint()`** call for quality control



Other Techniques for Anti-Aliasing

- on the 2D pixel image level (FSAA)

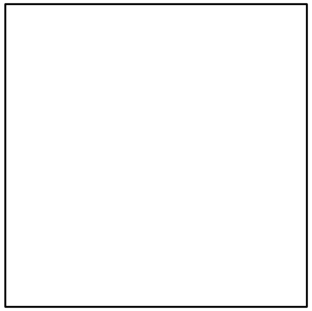
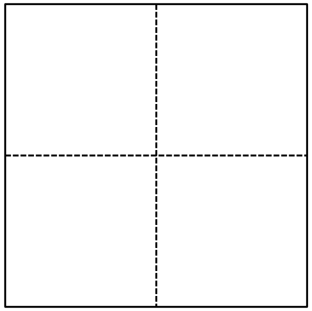
=

pixel

=

Other Techniques for Anti-Aliasing

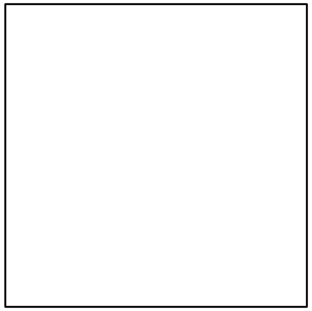
- on the 2D pixel image level (FSAA)
  - super-sampling (2×2, 4×4, etc.)

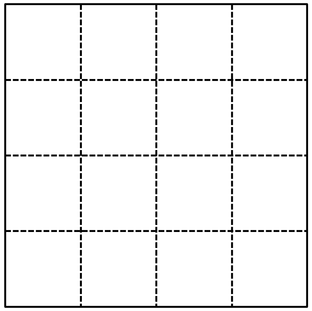





  - pixel color = (Σ subpixel colors) / N
  - memory & rendering time grow exponentially!

Other Techniques for Anti-Aliasing

- on the 2D pixel image level (FSAA)
  – super-sampling (2×2, 4×4, etc.)

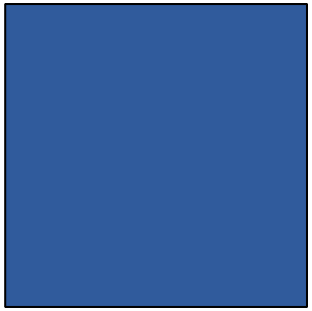
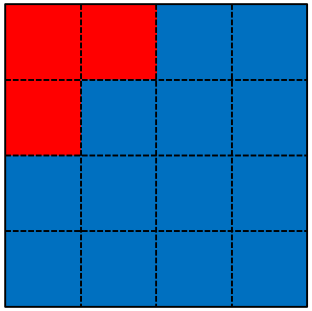
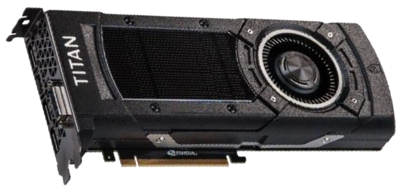




  – pixel color = (Σ subpixel colors) / N
  – memory & rendering time grow exponentially!

Other Techniques for Anti-Aliasing

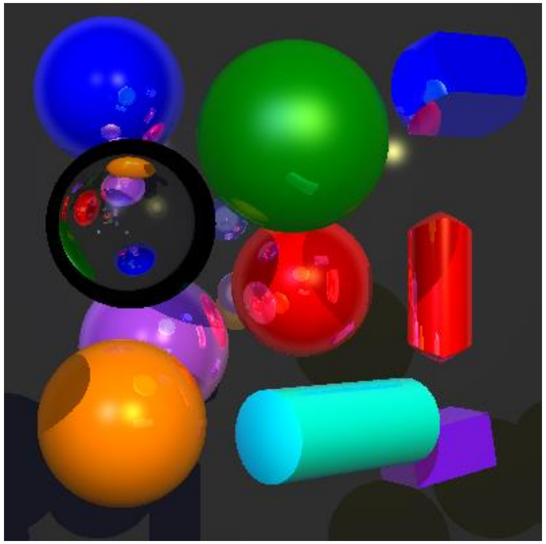
- on the 2D pixel image level (FSAA)
  - super-sampling (2×2, 4×4, etc.)

  - pixel color = (Σ subpixel colors) / N
  - memory & rendering time grow exponentially!

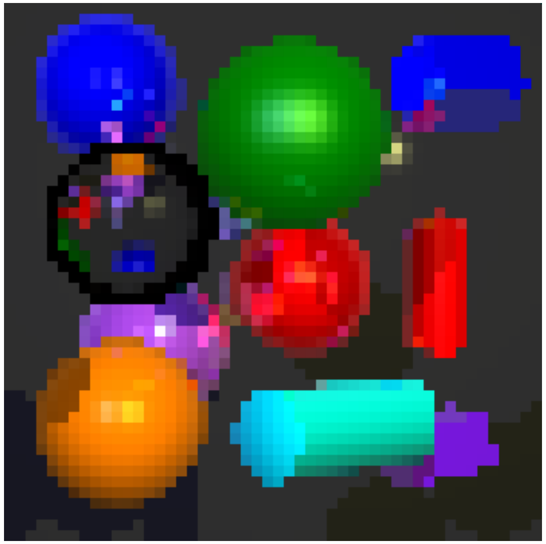
Other Techniques for Anti-Aliasing

- on the 2D pixel image level (FSAA)
  - example for 2×2 and 4×4:



Other Techniques for Anti-Aliasing

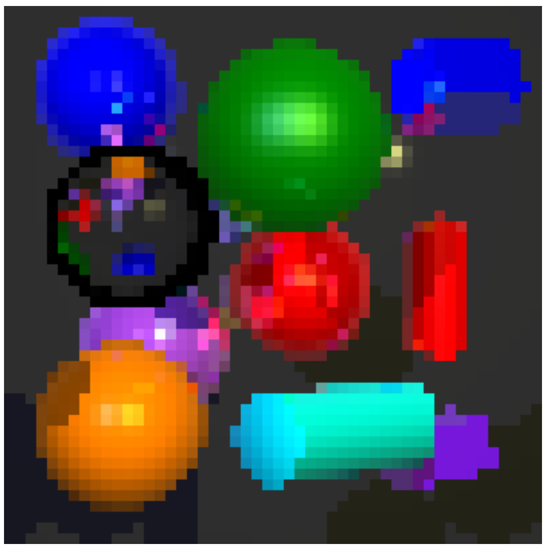
- on the 2D pixel image level (FSAA)
  - example for 2×2 and 4×4:



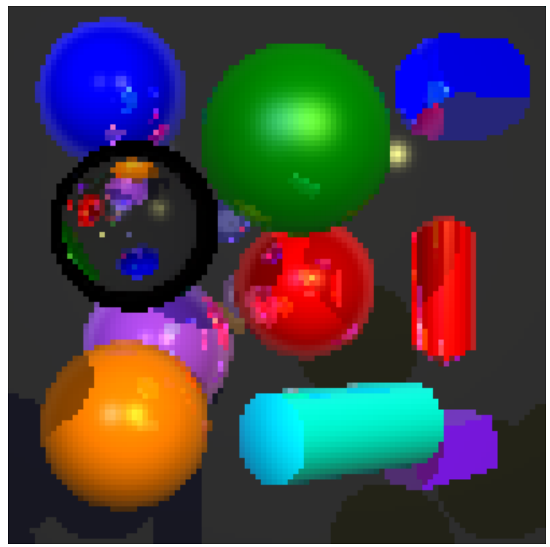
50×50 pixel grid

Other Techniques for Anti-Aliasing

- on the 2D pixel image level (FSAA)
  - example for 2×2 and 4×4:



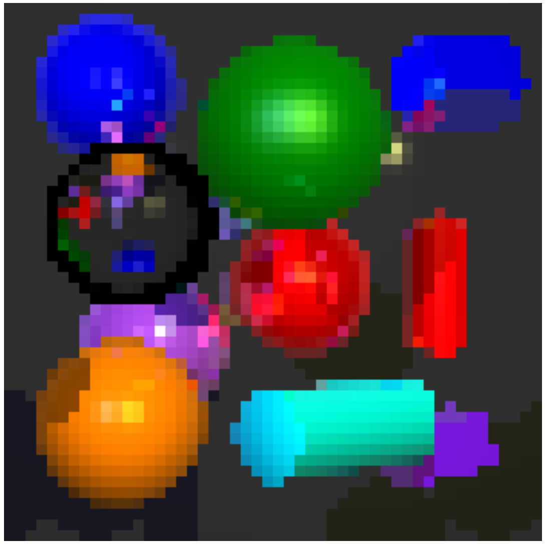
50×50 pixel grid



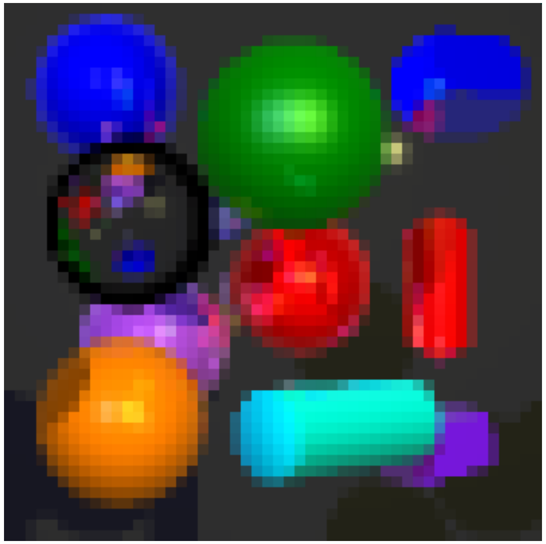
2×2 super-sampling

Other Techniques for Anti-Aliasing

- on the 2D pixel image level (FSAA)
  - example for 2×2 and 4×4:



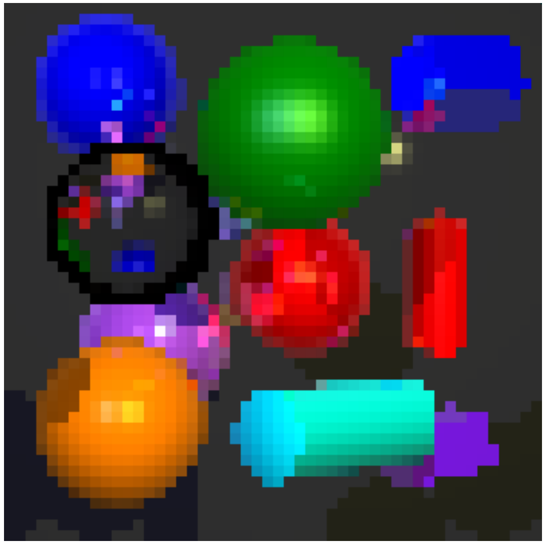
50×50 pixel grid



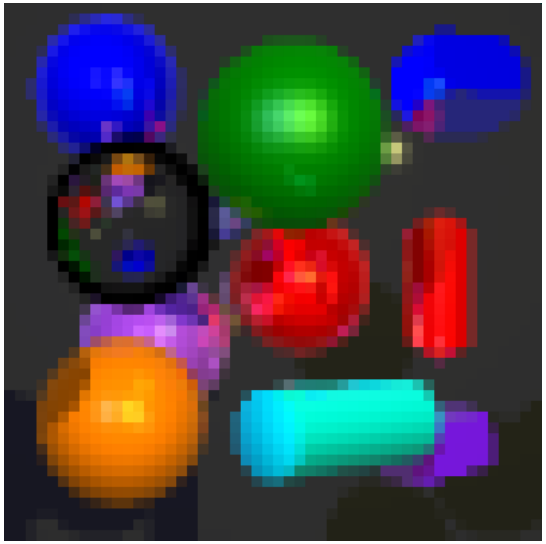
2×2 super-sampling

Other Techniques for Anti-Aliasing

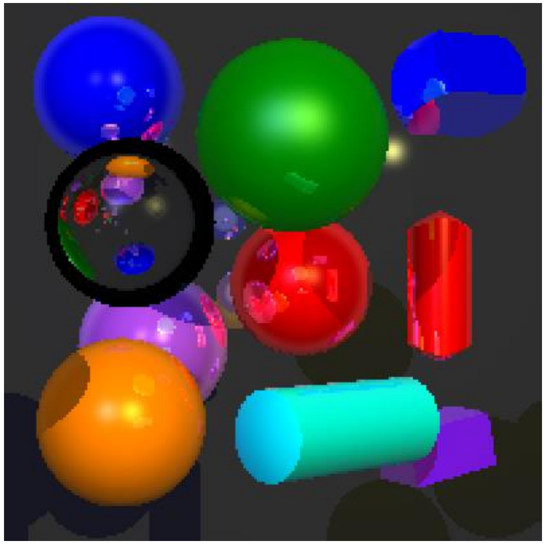
- on the 2D pixel image level (FSAA)
  - example for 2×2 and 4×4:



50×50 pixel grid



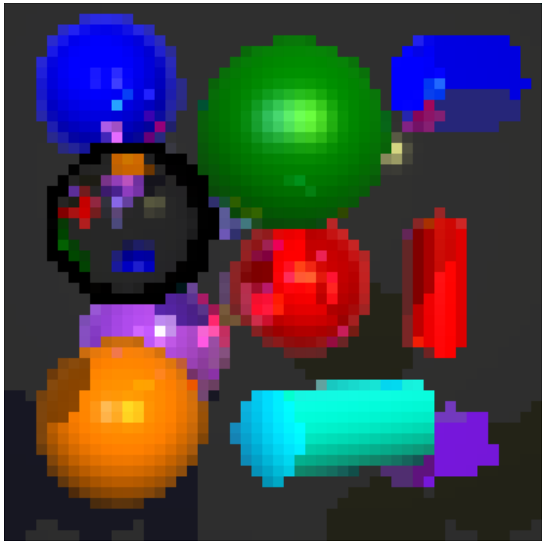
2×2 super-sampling

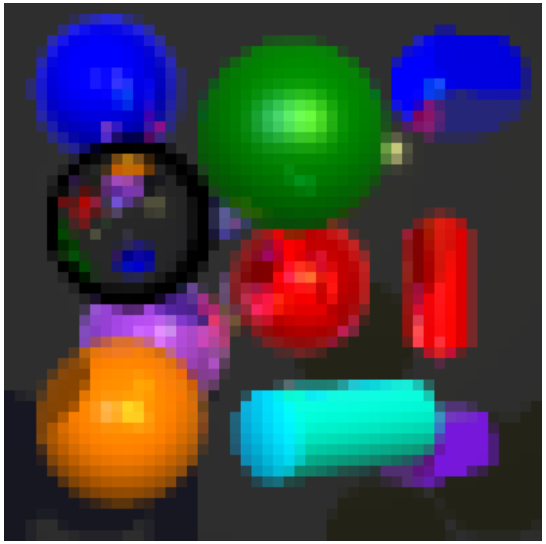


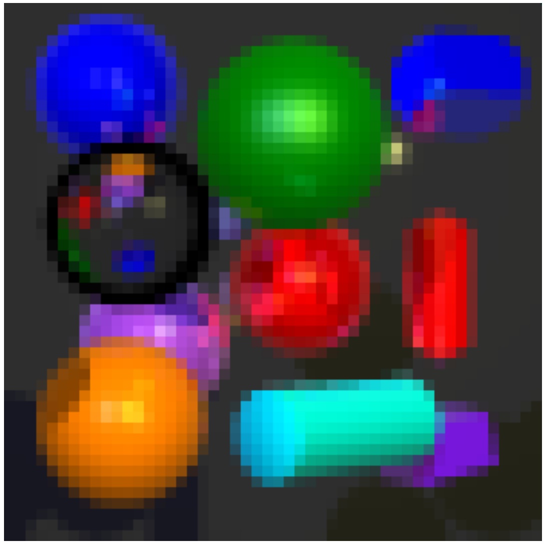
4×4 super-sampling

Other Techniques for Anti-Aliasing

- on the 2D pixel image level (FSAA)
  - example for 2×2 and 4×4:

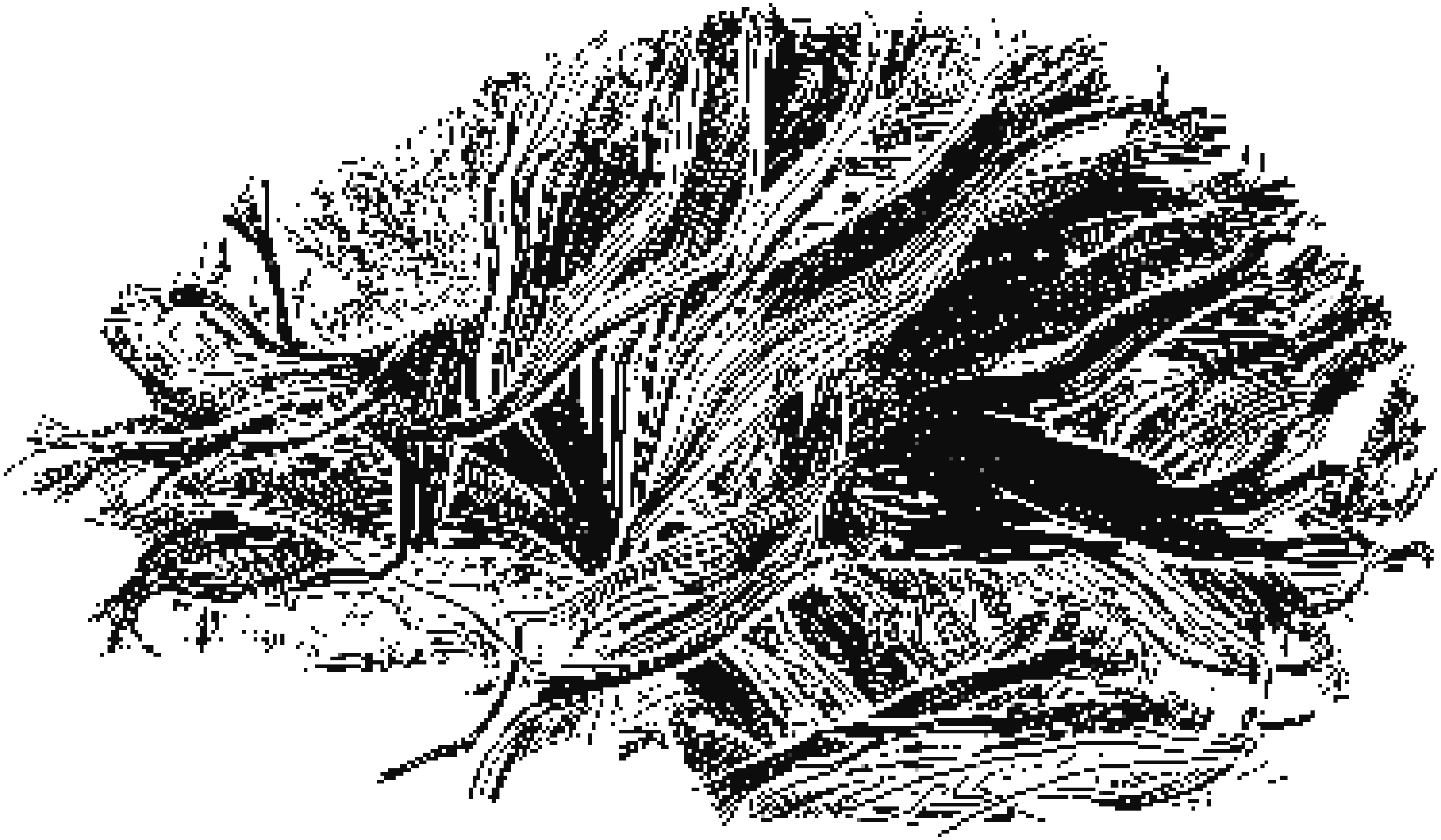

50×50 pixel grid


2×2 super-sampling


4×4 super-sampling

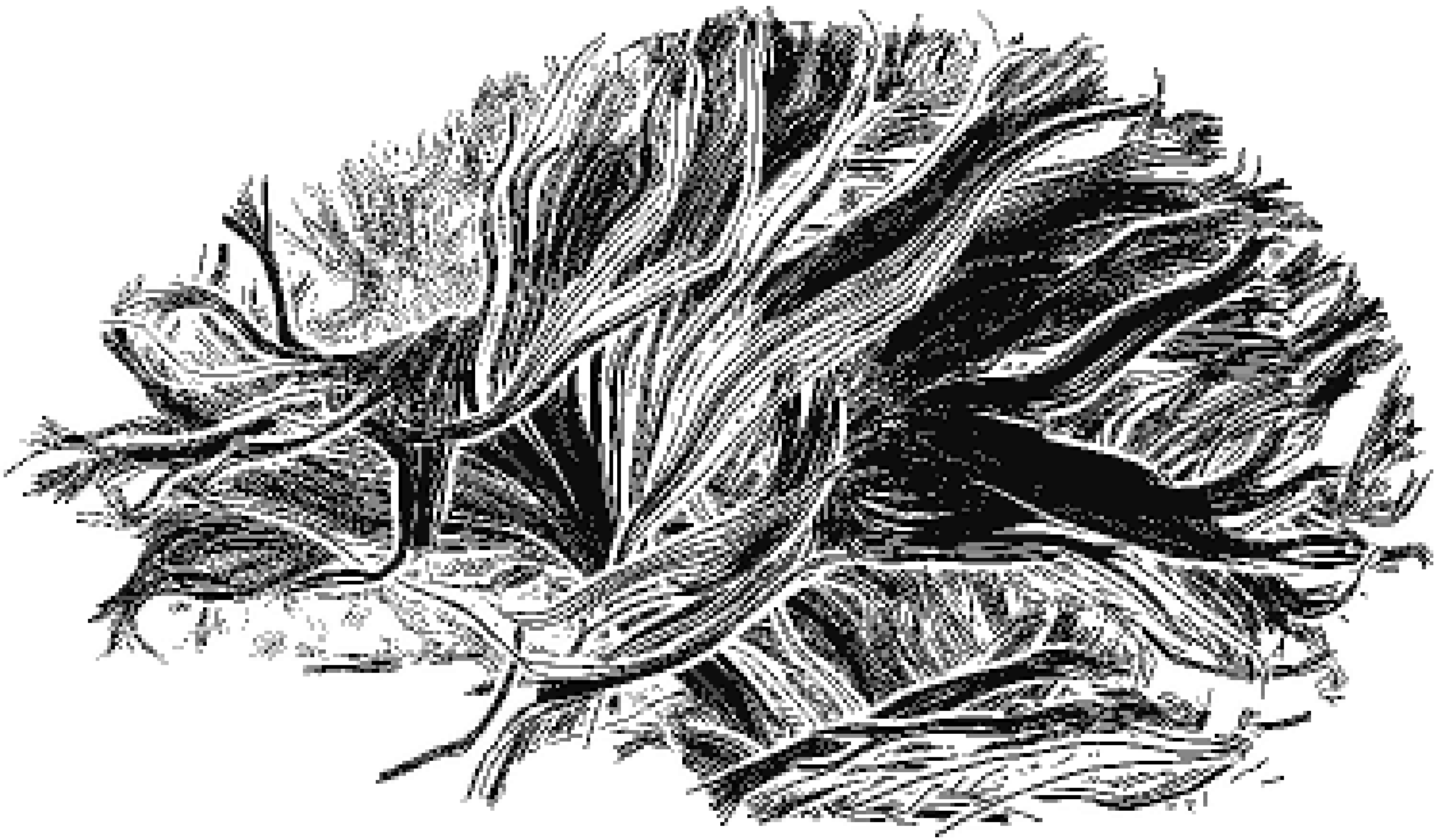
Other Techniques for Anti-Aliasing

- example: no AA, 2×2, and 4×4



Other Techniques for Anti-Aliasing

- example: no AA, 2×2, and 4×4

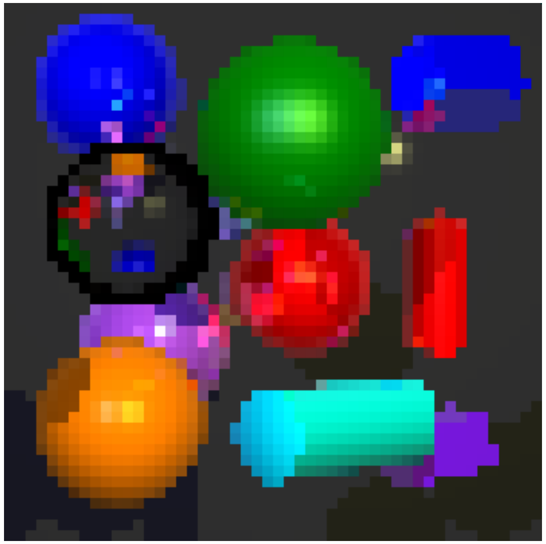


Other Techniques for Anti-Aliasing

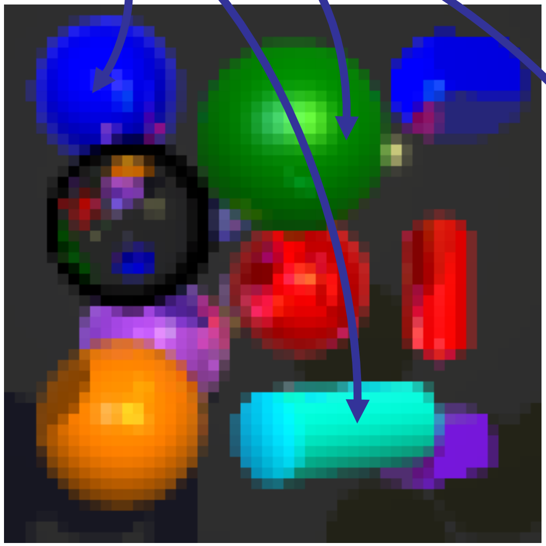
- example: no AA, 2×2, and $\boxed{4\times4}$

Other Techniques for Anti-Aliasing

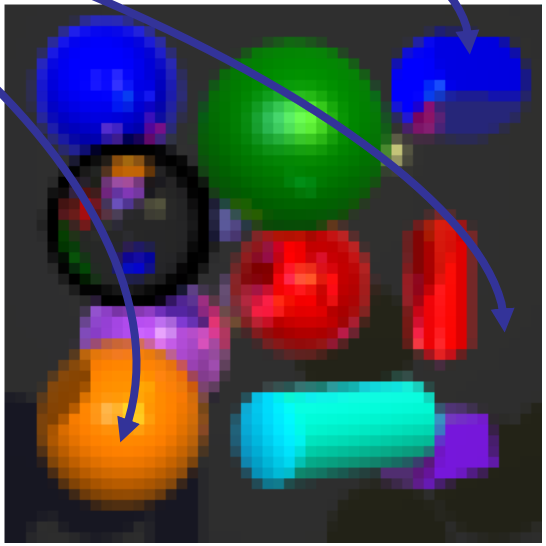
- super-sampling is **very** expensive
- lots of computations where more detail is not needed



50×50 pixel grid



2×2 super-sampling



4×4 super-sampling

Other Techniques for Anti-Aliasing

- less expensive technique: **multi-sampling**
  - special case (optimization) of super-sampling
  - only *z*-value (from *z*-buffer) is truly super-sampled
  - HSR aliasing removed, but not other aliasing

4×4 super-sampling

4×4 multi-sampling

A-Buffer

- Loren Carpenter (1984)
- first used in Star Trek II’s Genesis effect
- goals:
  - similarly effective and simple as z-buffer
  - anti-aliasing of image
  - correct handling of transparency
  - only modest performance decrease
- idea:
  - subdivide each pixel using a bit mask

A-Buffer

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

- 8x4 bit mask to store sub-pixel fragments
- each regular pixel stores a list of fragments that it comprises
- each fragment contains its parameters (area, color, opacity, z_{min} and z_{max}) and a bit mask for its coverage
- final pixel value by considering coverage area and color/transparency values of all fragments using the bit mask

A-Buffer

- first: computing bit mask for one fragment:
  - polygon clipped to pixel borders → fragment
  - bits right of each fragment's edge are set to 1
  - both bit masks are XORed to obtain final bit mask of fragment:

0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1

XOR

0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	0	1	1	1

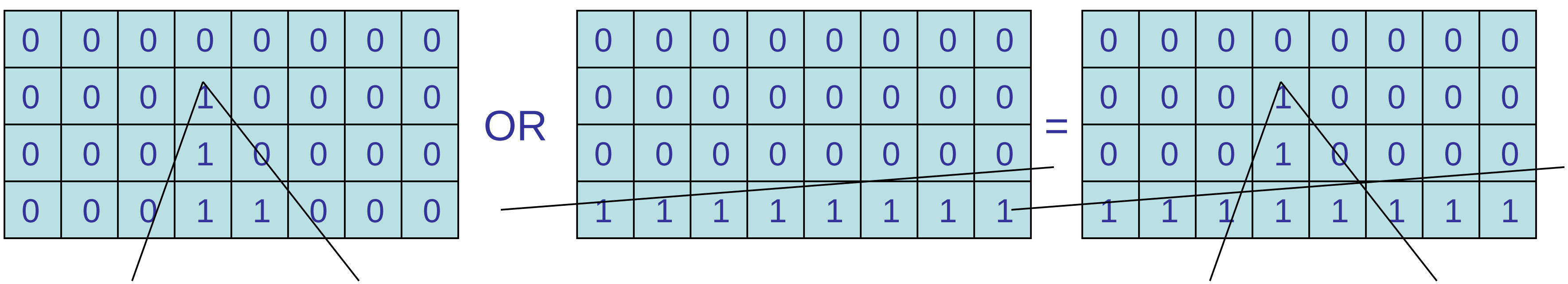
=

0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	1	0	0	0

- done for all polygons to obtain all fragments

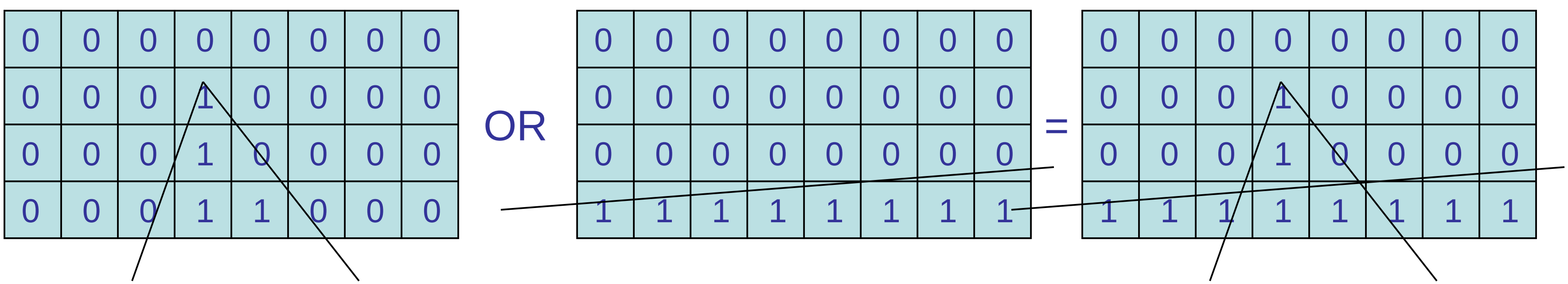
A-Buffer

- second: computing a pixel’s color by traversing fragment list
  - inside and outside regions
  - processing of fragments front-to-back
  - successive computation of inside mask until pixel covered or fragment list processed
  - pixel color computed w.r.t. to covered region

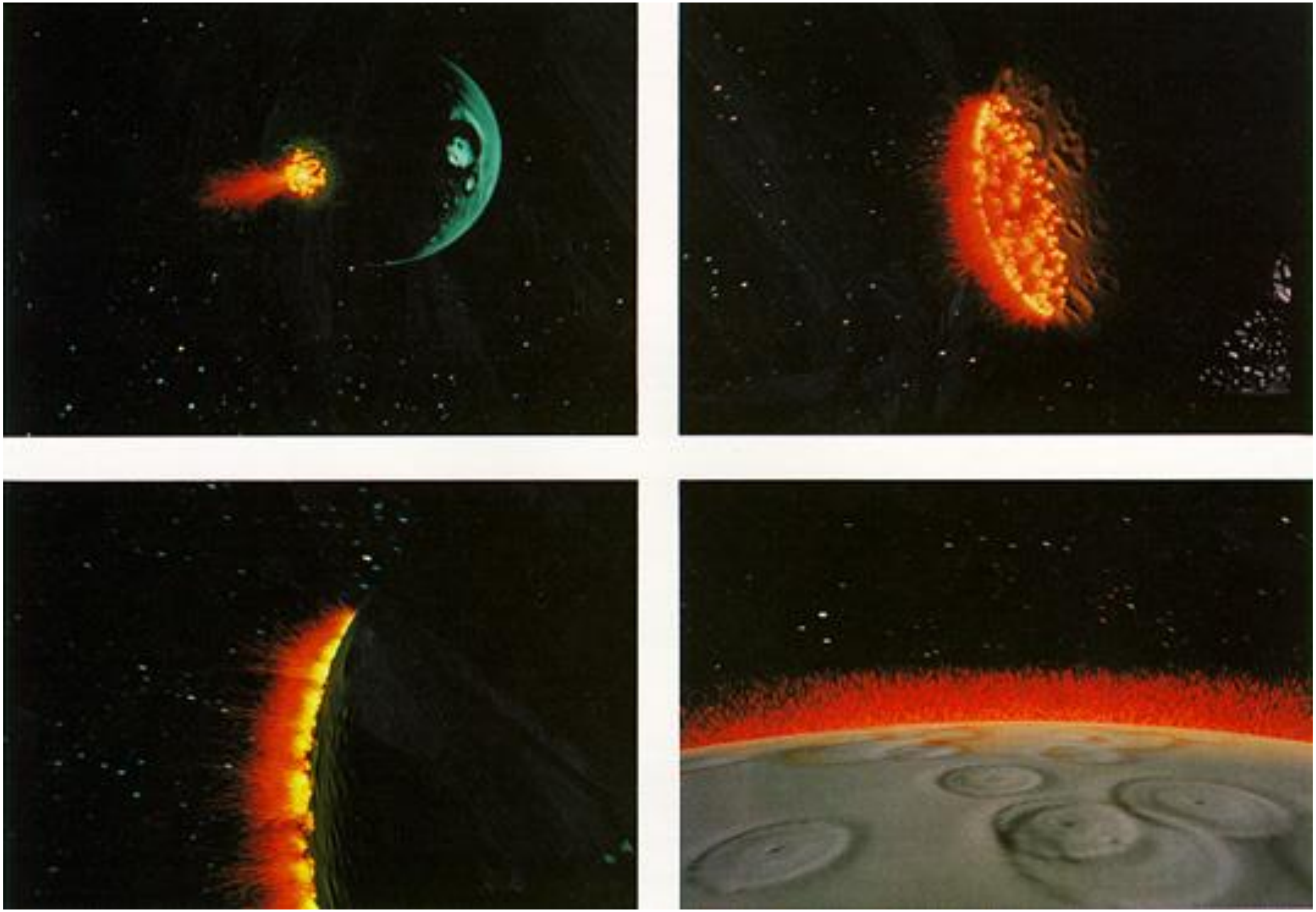


A-Buffer

- second: computing a pixel's color by traversing fragment list
  – fragments are considered only if overlap mask
  – only contribution from the part that is different from current inside mask (AND operation of new fragment's mask with outside mask)
  – transparency: recursion with transparent part



Genesis Effect by ILM/Lucasfilm ’82



Summary Scan Conversion

- most display are pixel-based
- need pixel representations for mathematical elements: lines, curves, …
- need to carry out many times
- fastest-possible realization needed, even for today’s fast rendering hardware
- Bresenham’s midpoint algorithm for line primitives

Summary Scan Conversion

- later also other shapes:
  triangles, polygons, etc.
- also need to understand perception:
  aliasing effects and anti-aliasing methods
- dedicated anti-aliasing of lines
- general anti-aliasing through
  super-sampling
  (i.e., computation of sub-pixels)
- balance of speed and quality

Summary CG Principles

- fastest & most effective technique desired
  - avoid expensive operations
  - avoid unnecessary operations
  - avoid numerical problems
  - mathematical tricks to get needed information
- quality only to the level really wanted
  - never compute more than needed
- often: we trade one thing for another
  - more complex math for fewer final operations
  - more computation for better quality