Rasterizing primitives: know where to draw the line

Dr Nicolas Holzschuch
University of Cape Town
e-mail: holzschu@cs.uct.ac.za

Rasterization of Primitives

- How to draw primitives?
  - Convert from geometric definition to pixels
  - *rasterization* = selecting the pixels
- Will be done frequently
  - must be fast:
    - use integer arithmetics
    - use addition instead of multiplication

Rasterization Algorithms

- Algorithmics:
  - Line-drawing: Bresenham, 1965
  - Polygons: uses line-drawing
  - Circles: Bresenham, 1977
- Currently implemented in *all* graphics libraries
  - You'll probably never have to implement them yourself

Why should I know them?

- Excellent example of efficiency:
  - no superfluous computations
- Possible extensions:
  - efficient drawing of parabolas, hyperbolas
- Applications to similar areas:
  - robot movement, volume rendering
- The CG equivalent of Euler's algorithm

Map of the lecture

- Line-drawing algorithm
  - *naïve* algorithm
  - Bresenham algorithm
- Circle-drawing algorithm
  - *naïve* algorithm
  - Bresenham algorithm

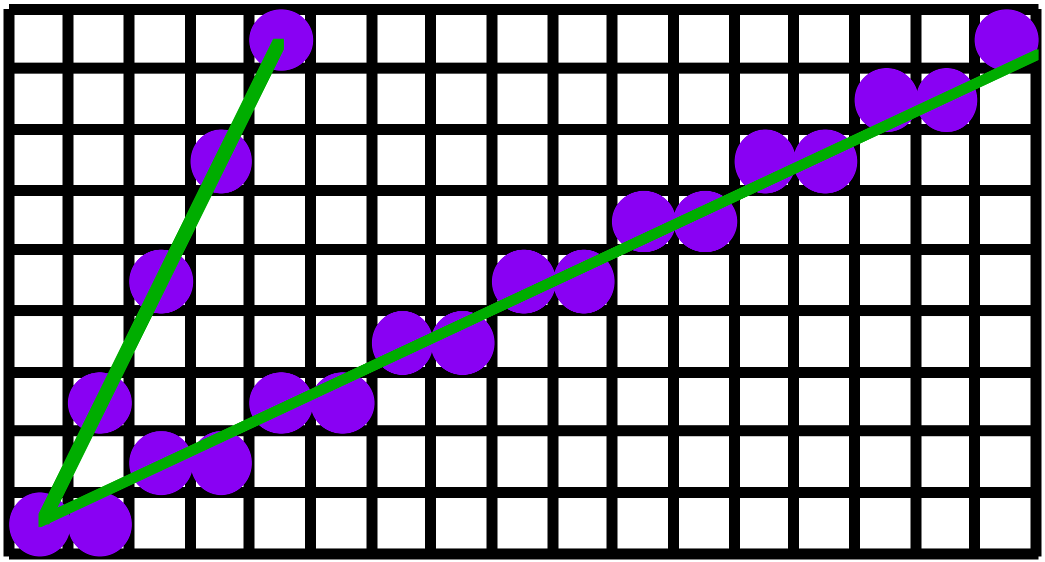
Naïve algorithm for lines

- Line definition: $ax+by+c = 0$
- Also expressed as: $y = mx + d$
  - m = slope
  - d = distance

```
For x=xmin to xmax
   compute y = m*x+d
   light pixel (x,y)
```

Extension by symmetry

- Only works with $-1 \leq m \leq 1$:



Extend by symmetry for $m > 1$

Problems

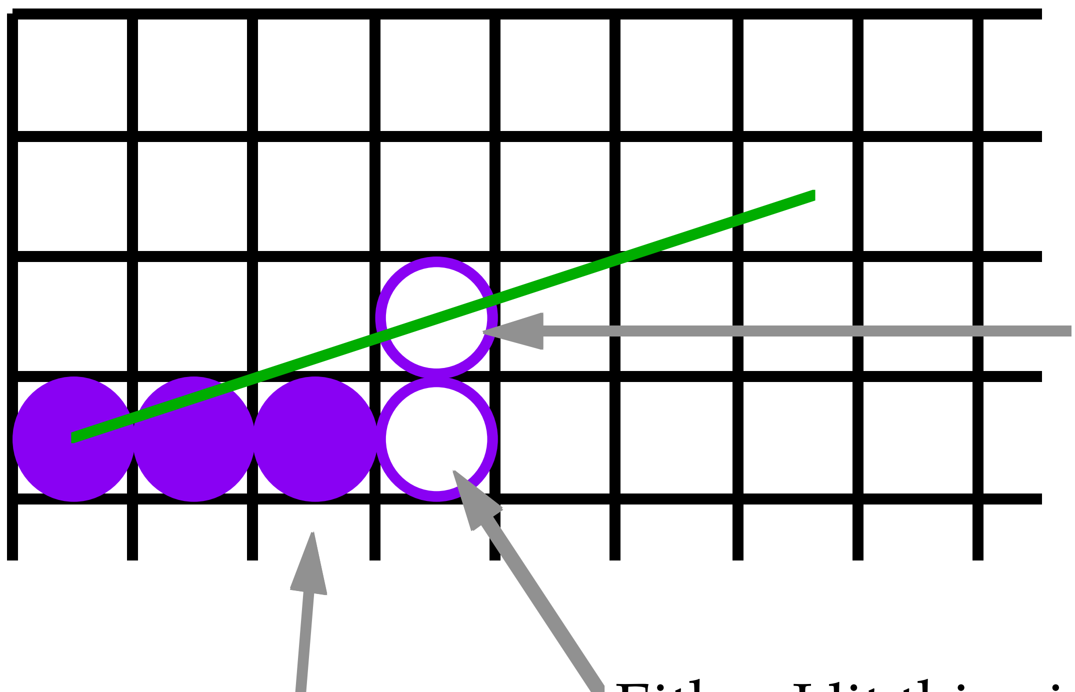
- 2 floating-point operations per pixel
- Improvements:

```
compute y = m*x0+d
For x=xmin to xmax
  y += m
  light pixel (x,y)
```

- Still 1 floating-point operation per pixel
- Compute in floats, pixels in integers

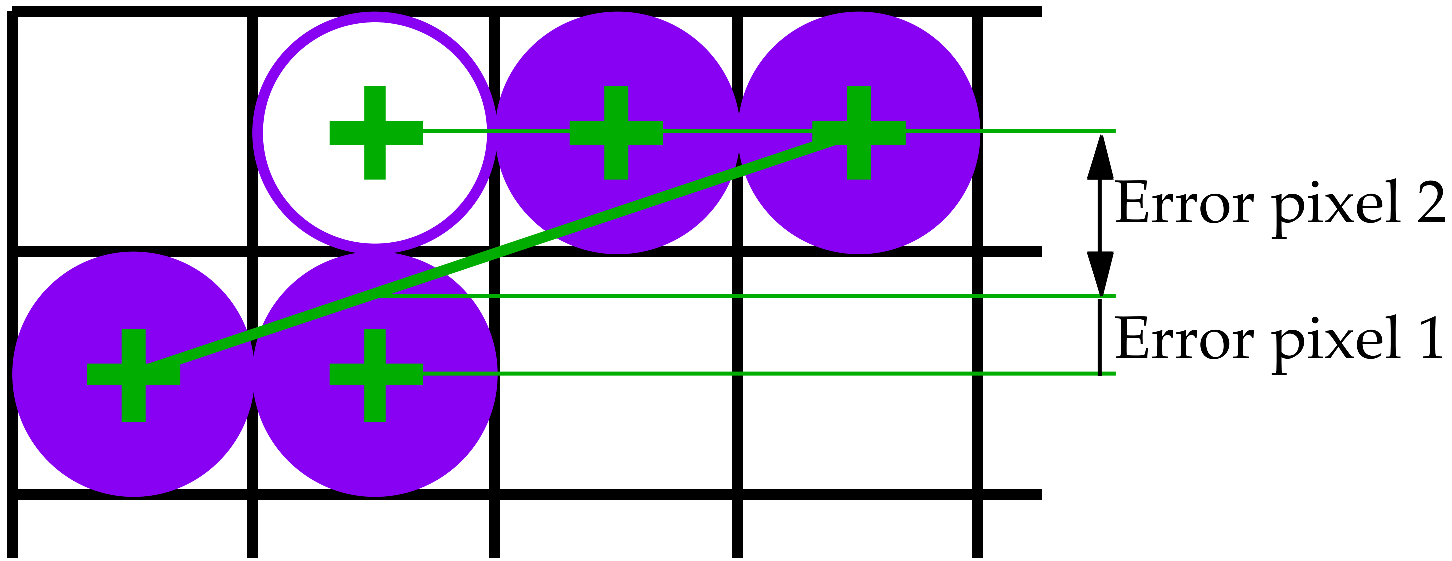
Bresenham algorithm: core idea

- At each step, choice between 2 pixels ($0 \leq m \leq 1$)



Bresenham algorithm

- I need a criterion to pick between them
- Distance between line and center of pixel:
  - the *error* associated with this pixel



Bresenham Algorithm (2)

- The sum of the 2 errors is 1
  - Pick the pixel with error $< 1/2$
- If error of current pixel $< 1/2$,
  - draw this pixel
- Else:
  - draw the other pixel.
    Error of current pixel = 1 - error

How to compute the error?

- Line defined as: $ax + by + c = 0$
- Distance from pixel (x0,y0) to line:

  $$d = ax0 + by0 + c$$
- Draw this pixel iff:

  $$ax0 + by0 + c < 1/2$$
- Update for next pixel:

  $$x \mathrel{+}= 1, \;\; d \mathrel{+}= a$$

We're still in floating point!

- Yes, but now we can get back to integer:
  $e = 2ax0 + 2by0 + 2c - 1 < 0$
- If $e<0$, stay horizontal, if $e>0$, move up.
- Update for next pixel:
  - If I stay horizontal: $e \mathrel{+}= 2a$
  - If I move up: $e \mathrel{+}= 2a + 2b$

Bresenham algorithm: summary

- Several good ideas:
  - use of symmetry to reduce complexity
  - choice limited to two pixels
  - error function for choice criterion
  - stay in integer arithmetics
- Very straightforward:
  - good for hardware implementation
  - good for assembly language

Circle: *naïve* algorithm

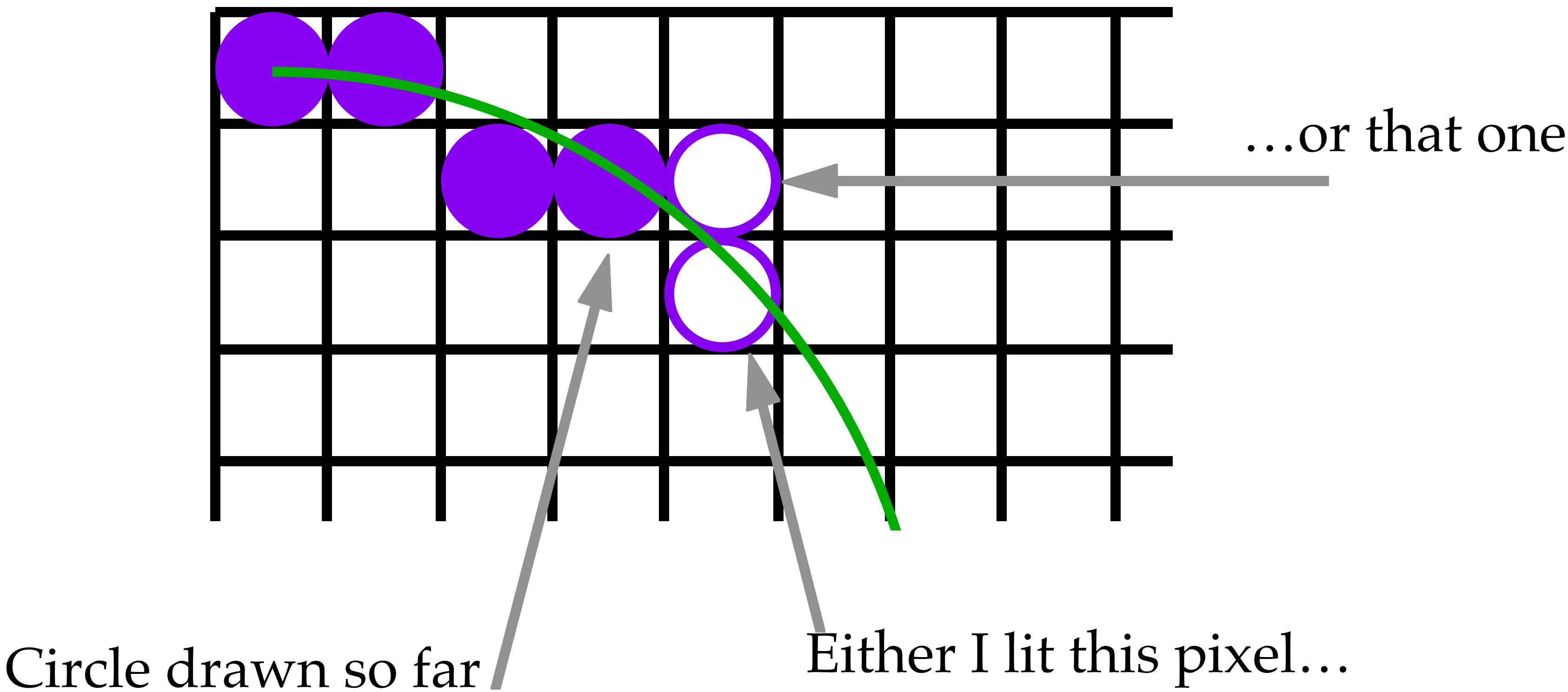
- Circle equation: $x^2+y^2-r^2 = 0$
- Simple algorithm:

```
for x = xmin to xmax
    y = sqrt(r*r - x*x)
     draw pixel(x,y)
```

- Work by octants and use symmetry

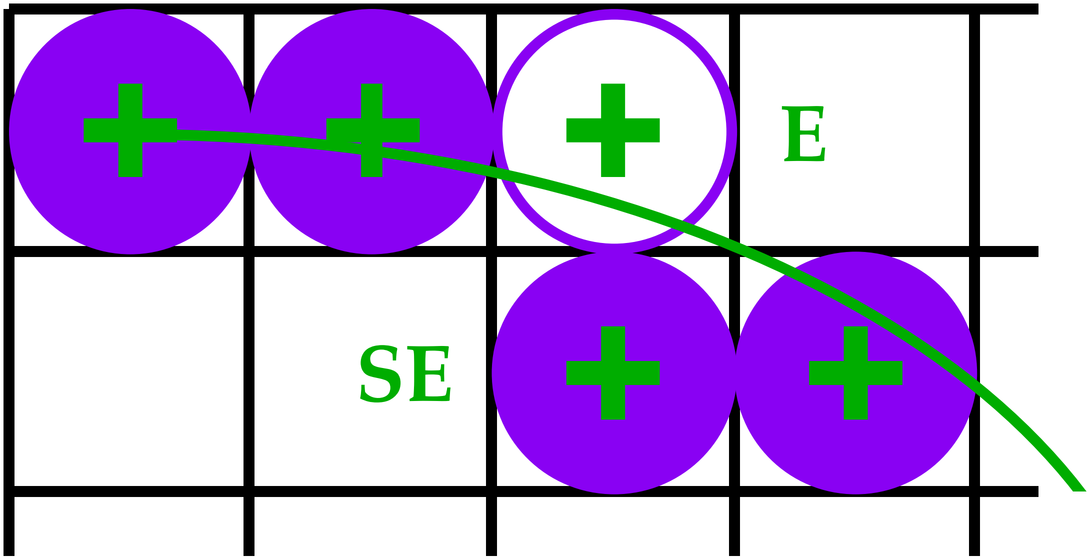
Circle: Bresenham algorithm

- Choice between two pixels:



Bresenham for circles

- Mid-point algorithm:



If the midpoint between pixels is inside the circle, E is closer
If the midpoint is outside, SE is closer.

Bresenham for circles (2)

- Error function: $d = x^2+y^2 - r^2$
- Compute d at the midpoint:
  - last pixel drawn: (x,y)
  - $d = (x+1)^2 + (y - 1/2)^2 - r^2$
  - $d < 0$: draw SE
  - $d \geq 0$: draw E

Updating the error

- If I increment x:
  - d += $2x$ +3
- If I decrement y:
  - d += $-2y$ + 2
- Two mult, two add per pixel
- Can you do better?

Doing even better

- The error is not linear
- However, what I add to the error is
- Keep Δx and Δy:
  - At each step:
  - Δx += 2, Δy -= 2
  - d += Δx
  - If I decrement y, d += Δy
- 4 additions per pixel

Midpoint algorithm: summary

- Extension of line drawing algorithm
- Test based on midpoint position
- Position checked using function:
  - sign of $(x^2+y^2-r^2)$
- With two steps, uses only additions

Extension to other functions

- Midpoint algorithm easy to extend to any curve defined by: $f(x,y) = 0$
- If the curve is polynomial, can be reduced to only additions using n-order differences

Conclusion

- The basics of Computer Graphics:
  - drawing lines and circles
- Simple algorithms, easy to implement with low-level languages
- So far, a one-task world:
  - our primitives extend indefinitely
  - Windows = boundaries = clipping