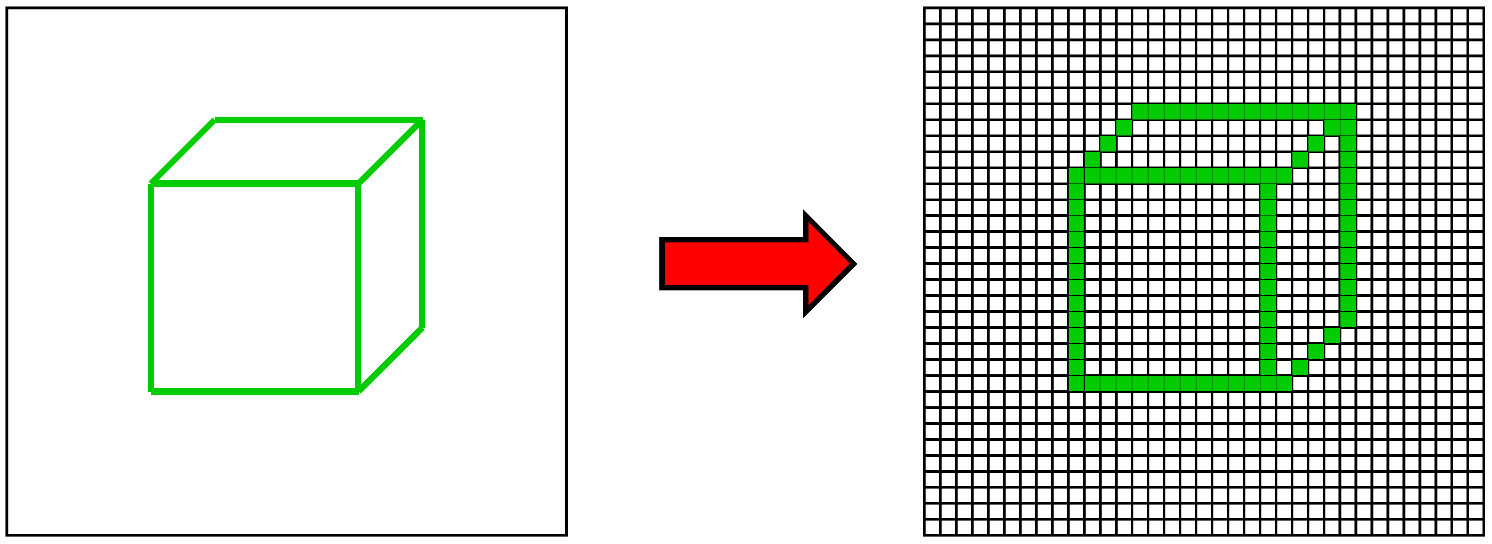
Rasterization

Lin Lu (吕琳)

Shandong University

http://irc.cs.sdu.edu.cn/~lulin/

Rasterization, a.k.a. Scan Conversion



Scan Conversion: Process of converting ideal to raster

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 arithmetic
    - 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

Map of the lecture

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

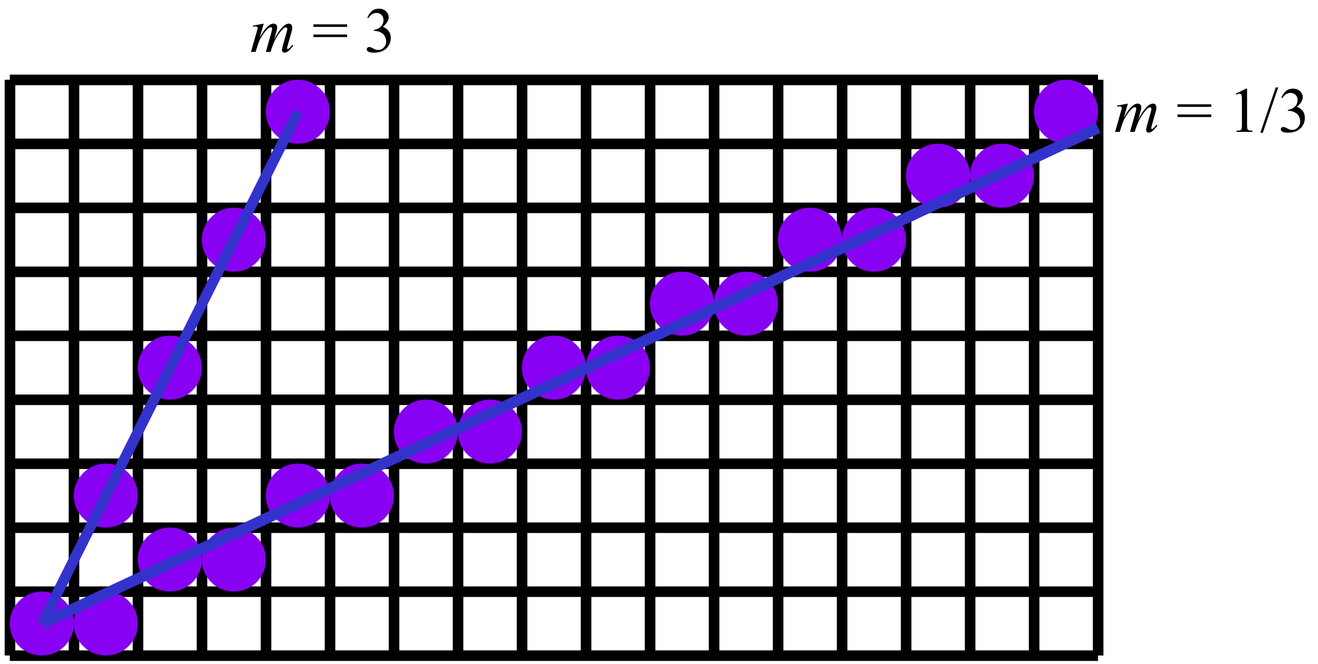
Naïve algorithm for lines

- 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 $m > 1$

A Really Simple Line Algorithm

- Equation for a line: $y(x) = mx + b$ $(0 \le x < 1)$
- Step along one pixel at a time in the "fast" direction, here x direction, fill in one pixel per column
- So, just evaluate for each x

```
void line (int x0, int y0, int x1, int y1){
    float m = whatever;
    float b = whatever;
    int x;
    for(x=x0;x<=x1;x++) {
        float y= m*x + b;
        draw_pixel(x,Round(y));
    }
}
```

- **Certainly correct, but slow:**
  - **integer add, cast to float, floating multiply and add, plus round every step.**

Lines: DDA Algorithm

- **Optimize the previous to remove multiply from inner loop.**
- **If we know y(x), we can calculate y(x+1):**

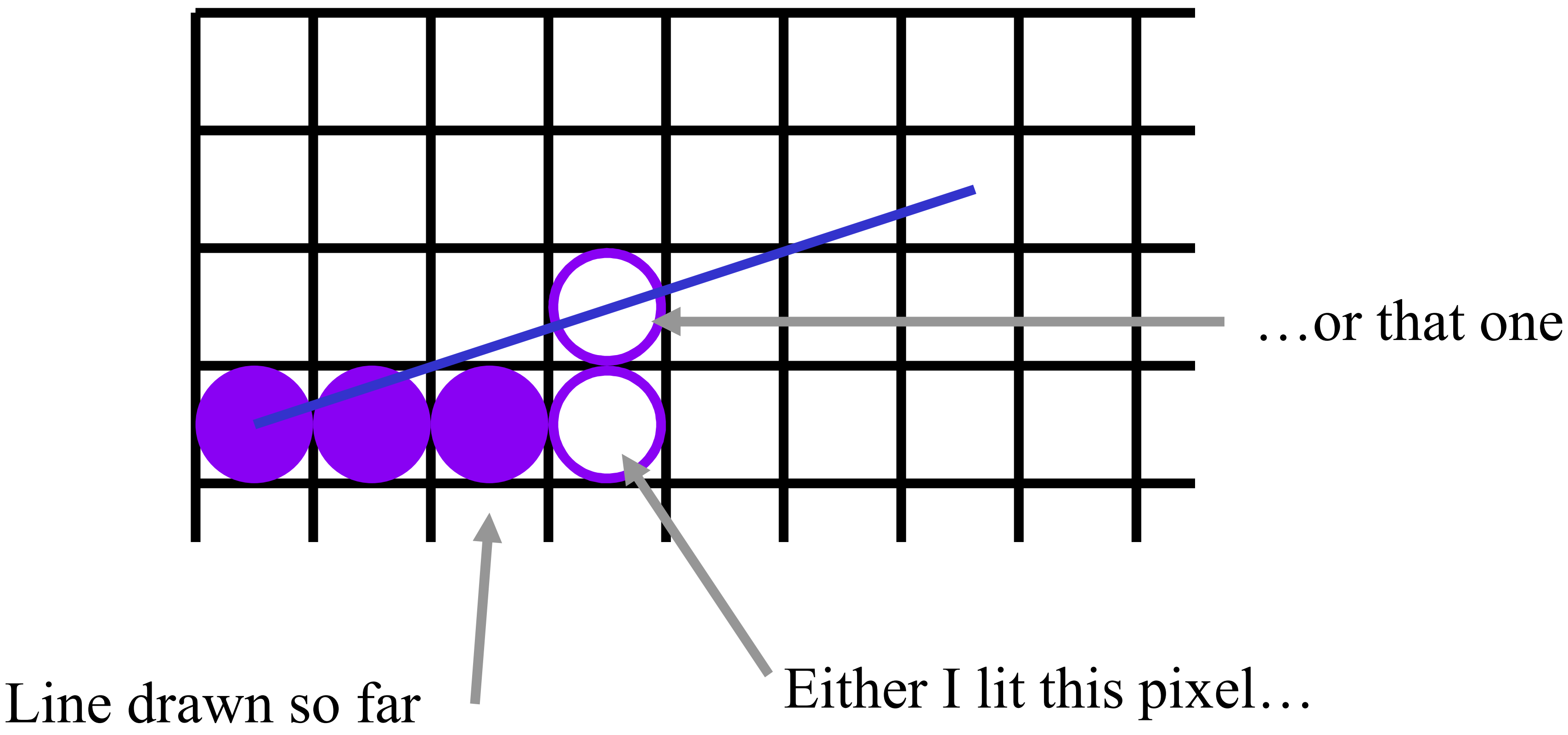
  $y(x+1) = mx + m + b = y(x) + m$

```
void line (int x0, int y0, int x1, int y1){
    float y = y0;
    float m = (y1 - y0)/ (float) (x1 - x0);
    int x;
    for(x=x0;x<=x1;x++) {
        draw_pixel(x,Round(y));
        y += m;
    }
}
```

- **This is called Differential Digital Analyzer (DDA)**
- **Problem: Floating-point add and rounds are expensive**

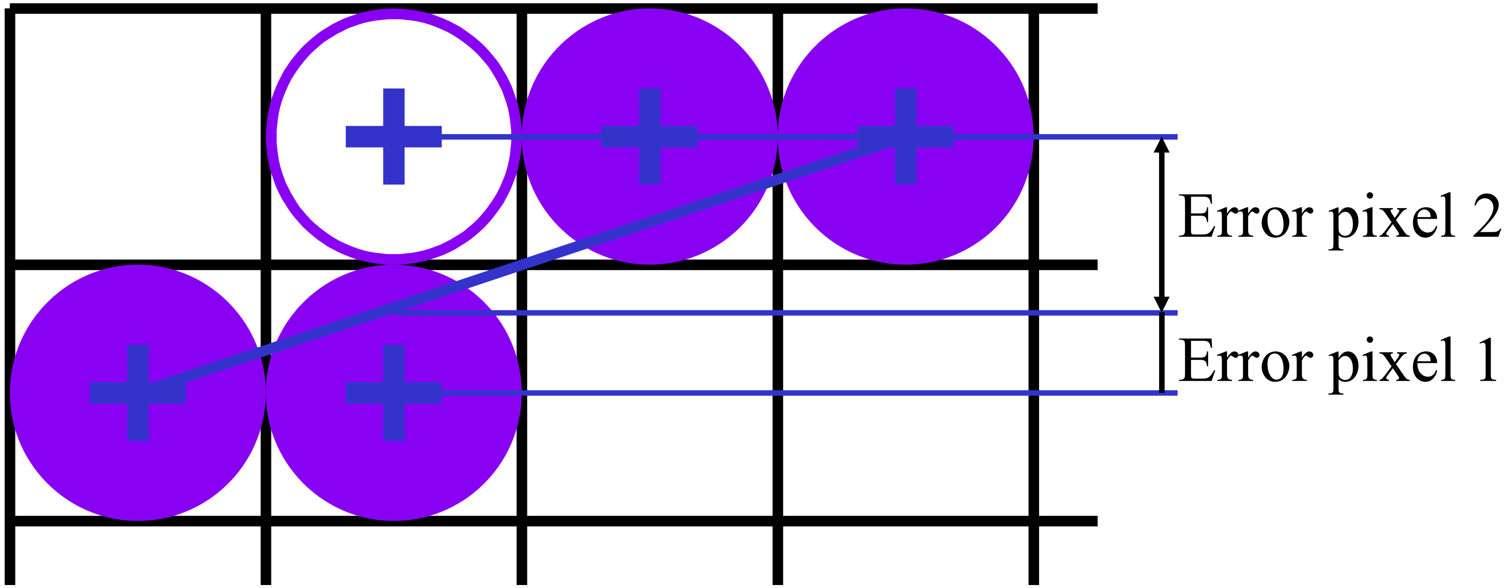
Bresenham algorithm: core idea

- 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?

- Origin (0,0) is the lower left corner
- Line defined as: $y = ax + c$
- Distance from pixel (p,q) to line:

  $$d = y(p) - q = ap - q + c$$
- Draw this pixel iff:

  $$ap - q + c < 1/2$$
- 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:
  $e = 2ap - 2q + 2c - 1 < 0$
- If $e<0$, stay horizontal, if $e>0$, move up.
- Update for next pixel:
  - If I stay horizontal: x+=1, e += $2a$
  - If I move up: x+=1, y+=1, e += $2a - 2$

Bresenham Algorithm

```
void draw_line(int x0, int y0, int x1, int y1)
{
    int x, y = y0;
    int dx = 2*(x1-x0), dy = 2*(y1-y0);
    int dydx = dy-dx, F = dy-dx/2;

    for (x=x0 ; x<=x1 ; x++) {
        draw_pixel(x, y);
        if (F<0) F += dy;
        else {y++; F += dydx;}
    }
}
```

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

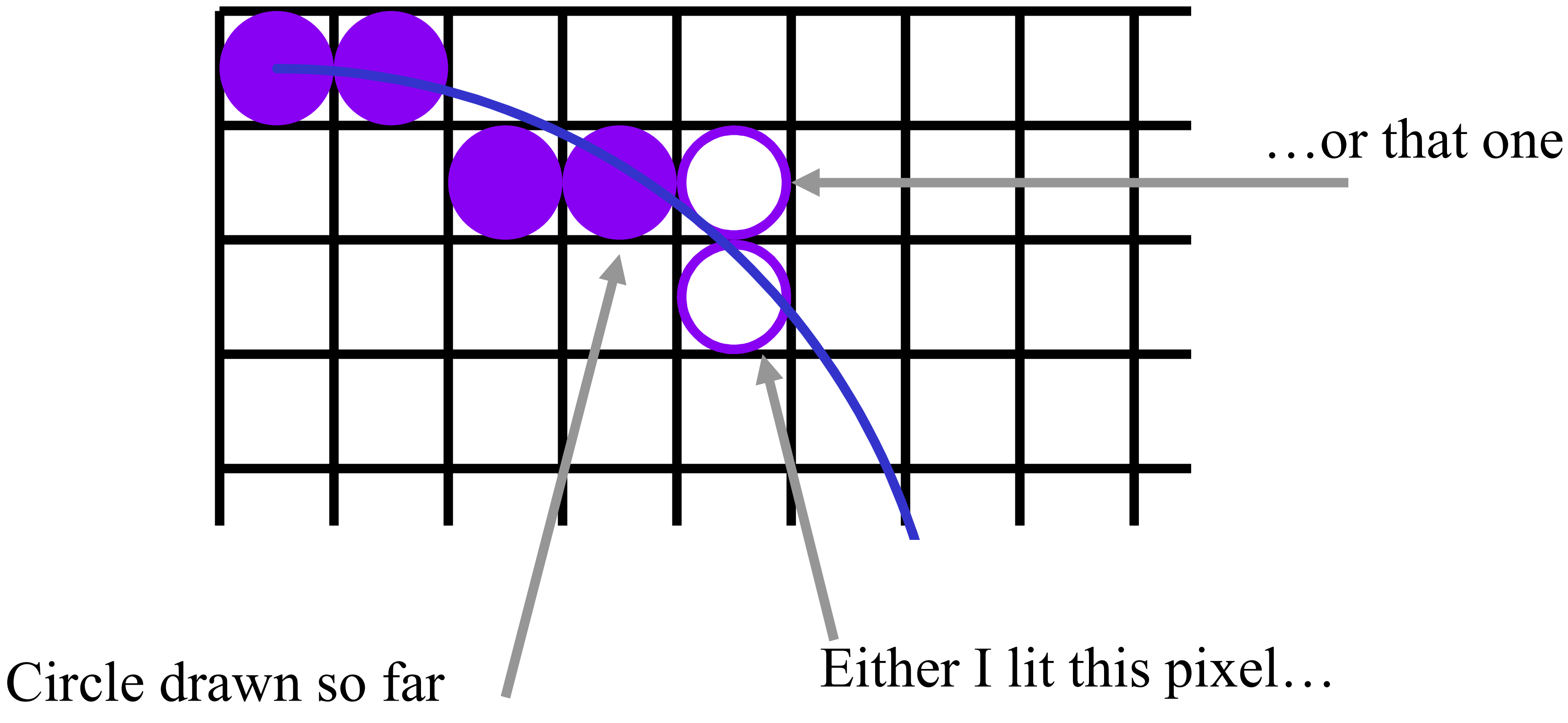
- Circle equation: $x^2+y^2-r^2 = 0$
- Simple algorithm:

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

- Work by octants and use symmetry

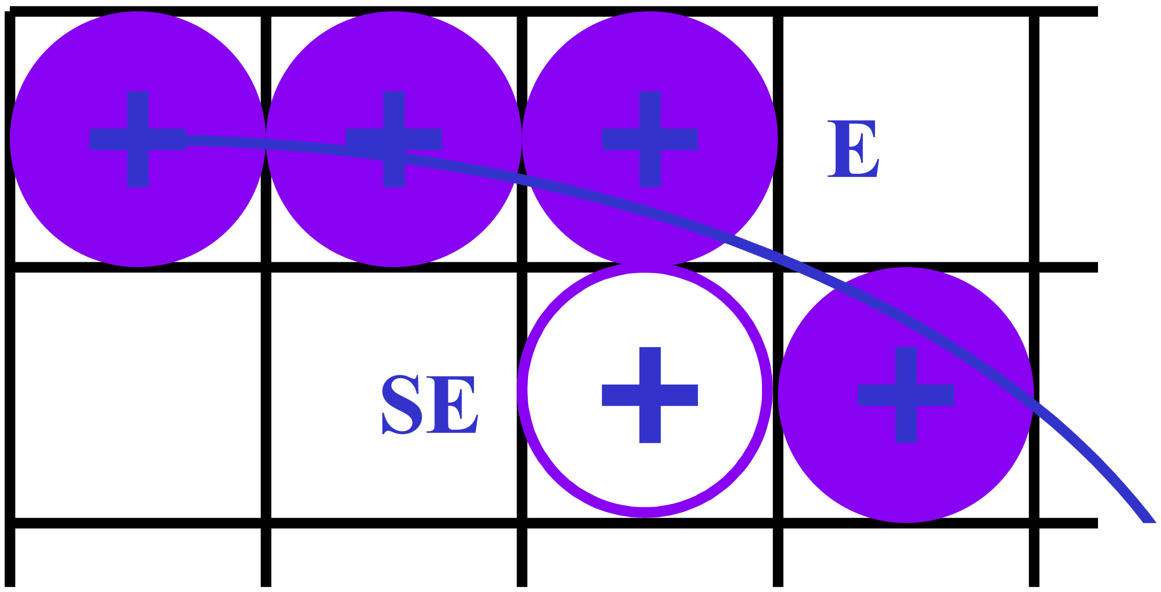
Circle: Bresenham algorithm

- Choice between two pixels:



Bresenham for circles

- Mid-point algorithm:



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

Bresenham for circles (2)

- Error function: $d = x^2+y^2 - r^2$
- Compute d at the midpoint:

  If the last pixel drawn is (x,y),

  then E = $(x+1,y)$, and SE = $(x+1,y-1)$.

  Hence, the midpoint = $(x+1,y-1/2)$.

  - $d(x,y) = (x+1)^2 + (y - 1/2)^2 - r^2$
  - $d < 0$: draw E
  - $d \geq 0$: draw SE

Updating the error

- In each step (go to E or SE), i.e., increment x: x+=1:
  - d += $2x$ +3
- If I go to SE, i.e., x+=1, y+=-1:
  - 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