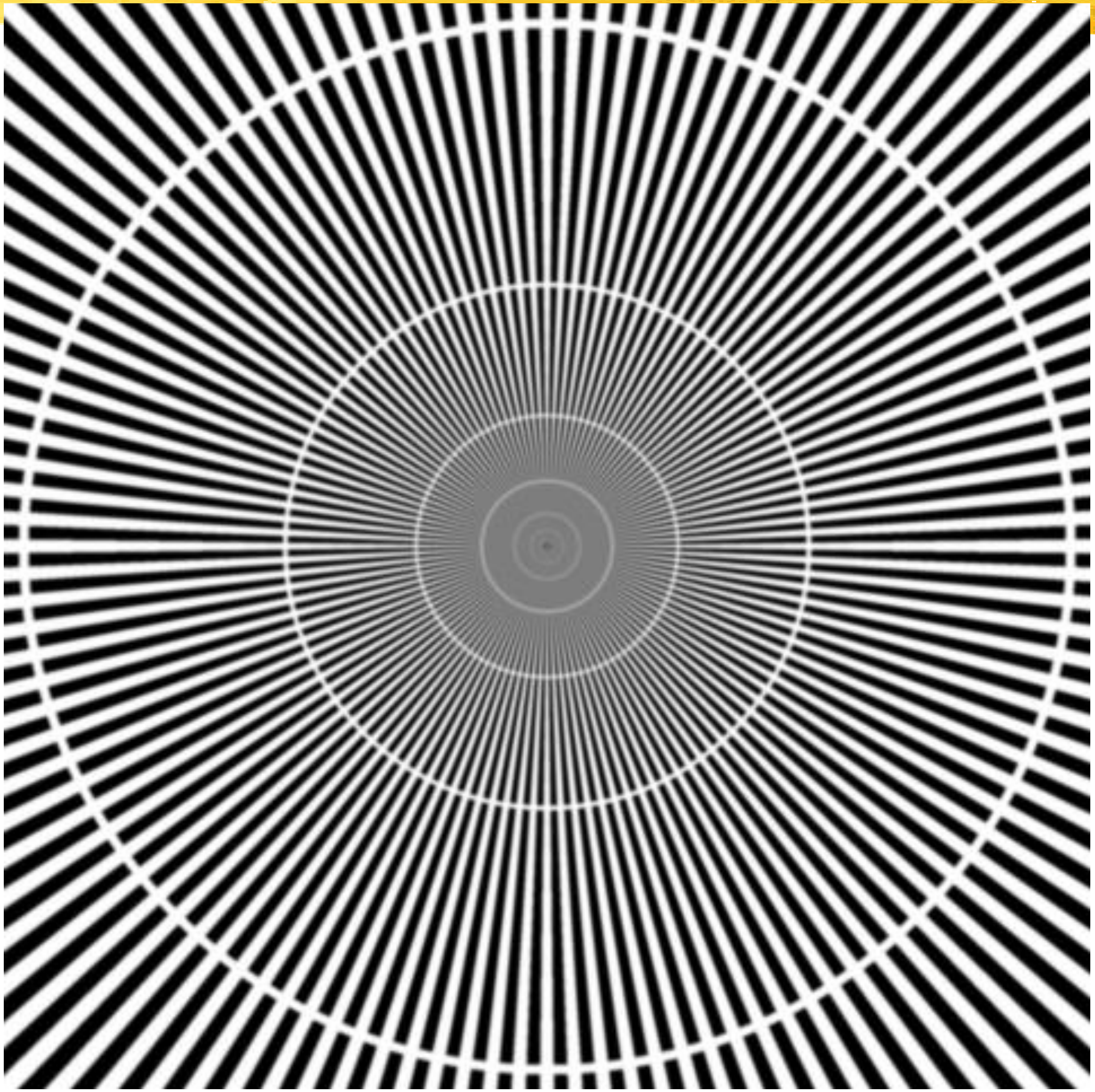
Scan Conversion

- Drawing Lines
- Drawing Circles

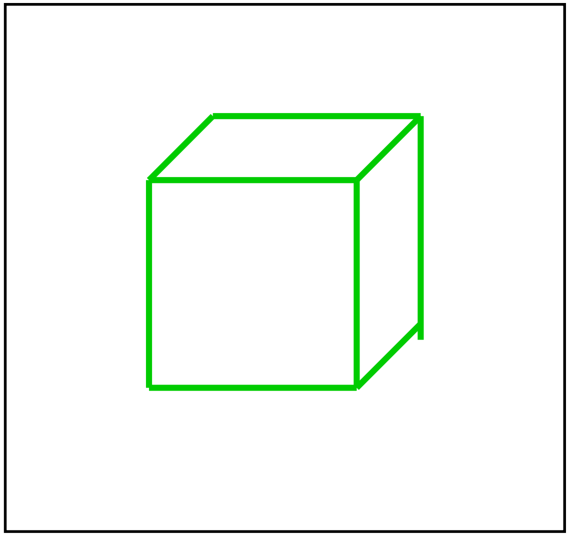
How to Draw This?



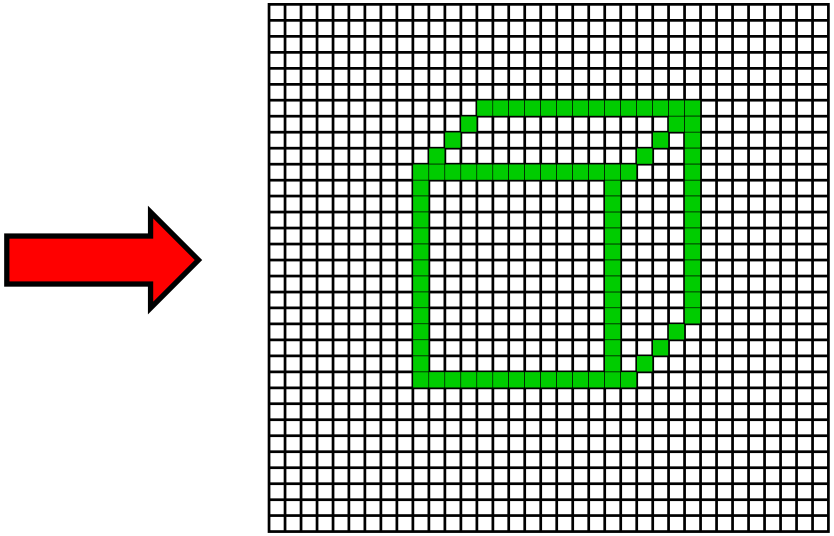
Start From Simple

How to draw a line: $y(x) = mx + b$?

Scan Conversion, a.k.a. Rasterization



Ideal Picture



Raster Representation

Scan Conversion: Process of converting shapes to raster

Scan Conversion Algorithms

- A discrete set of pixels can only *approximate* a continuous geometric object
- This means that scan conversion usually introduces error
- Properties of good scan conversion algorithms:
  - *Accuracy*
  - *Efficiency*
- Challenges
  - Modify all *the right* pixels
  - Modify *only* the right pixels
  - Calculate their values correctly
  - Do it quickly
- ***So, start with a correct algorithm and optimize it***

A Really Simple Line Algorithm

- Equation for a line: *$y(x) = mx + b$ ($0 <= x < 1$)*
- 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's Algorithm

This does the right thing (same as DDA) at a **cost of only 2 or 3 integer adds per point**. (assumes sorted endpoints, 0<slope<1)

```
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;}
    }
}
```

why does this work?

Implicit Function for a Line

Line $\mathbf{L}$ from $[x_0, y_0]$ *to* $[x_1, y_1]$.

$\mathbf{P}_0 = [x_0, y_0]$,

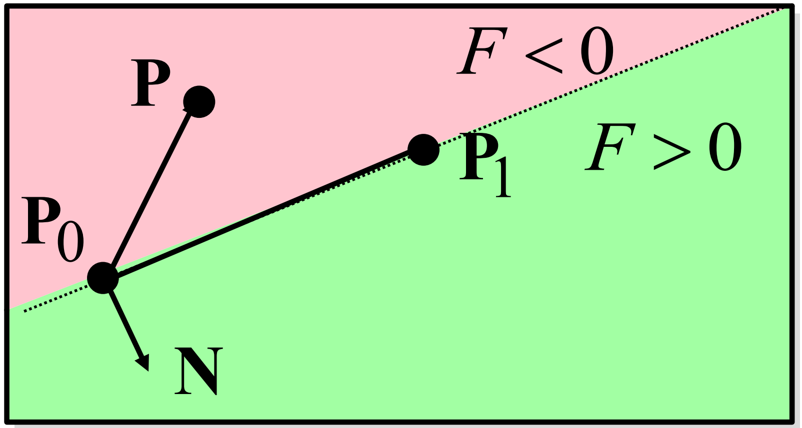
$\mathbf{P}_1 = [x_1, y_1]$.

$dx = x_1 - x_0, \; dy = y_1 - y_0$

$\mathbf{N} = [dy, -dx]$

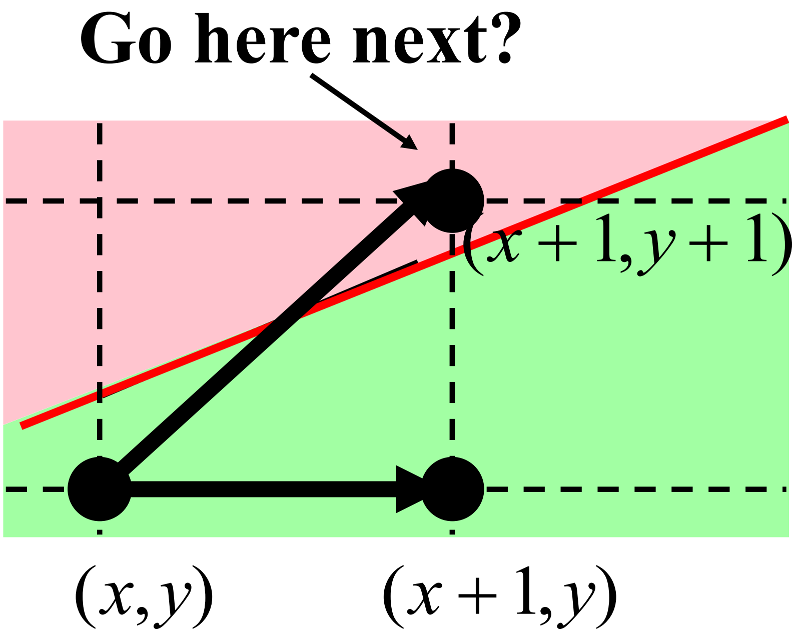
implicit function : $F(\mathbf{P}) = 2\mathbf{N} \cdot (\mathbf{P} - \mathbf{P}_0)$

$F = 0 \rightarrow \mathbf{P}$ is on $\mathbf{L}$



Why the factor of 2? Because we're going to divide by 2 later.

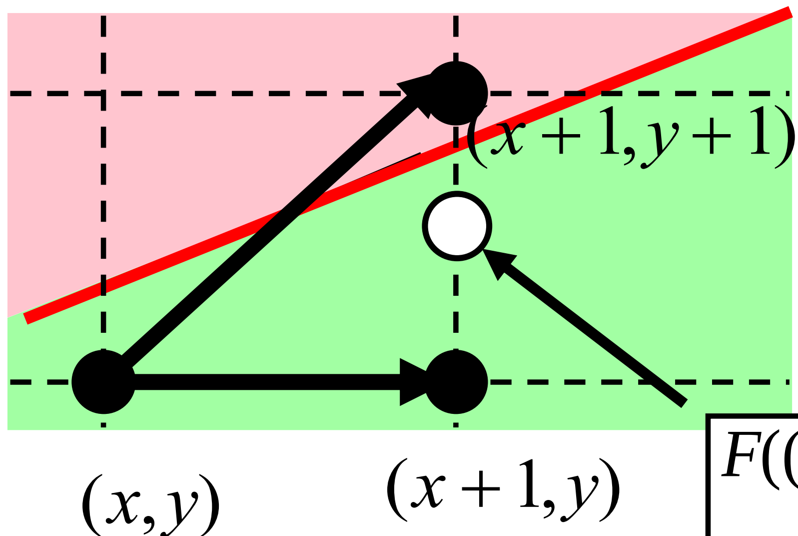
Line Drawing: Which Pixel is Next?



- Assume:
  - 0 < slope < 1
  - sorted endpoints, $x_0 < x_1$
- At each step:
  - Current point is *(x,y)*
  - Next point is pixel (x+1,?) that's closest to the actual line
  - Do we increment *x* and *y* or only *x*?
- Use the implicit function to decide!

Use the Implicit Function

- **Idea: Test the half-way point (x+1, y+1/2)**



$F((x+1, y+1/2)) > 0$?

yes : increment x and y

no : increment x

Trick: Incrementally Update *F*

$$\mathbf{P} = (x, y), \Delta = (1, 1/2)$$

$$F(\mathbf{P}) = \mathbf{N} \cdot (\mathbf{P} - \mathbf{P}_0)$$

$$F(\mathbf{P} + \Delta) = \mathbf{N} \cdot (\mathbf{P} + \Delta - \mathbf{P}_0)$$

$$= F(\mathbf{P}) + \mathbf{N} \cdot \Delta$$

- What we care about here is only the sign of **F**, so multiply the function by 2 to avoid floating point calculation

Trick: Incrementally Update *F*

$$F(\mathbf{P}) = 2\mathbf{N} \cdot (\mathbf{P} - \mathbf{P}_0)$$

$$F(\mathbf{P}+\Delta) = 2\mathbf{N} \cdot (\mathbf{P}+\Delta-\mathbf{P}_0)$$

$$= F(\mathbf{P}) + 2\mathbf{N} \cdot \Delta$$

- **Computing *F*(P) requires a dot product:**
  - **2 multiplications and 1 add**
- **But computing *F*(P+Δ) requires only 1 add**
  - **The 2N• Δ term is constant - it only needs to be calculated once**
- **Δ is [1,0] or [1,1]**

Decision Variable F

$$F_0 = F(\mathbf{P}_0 + [1, 1/2])$$
$$= F(\mathbf{P}_0) + \mathbf{N} \times [2,1]$$

$$\mathbf{N} = [dy, -dx]$$

$$F = F + 2\mathbf{N} \times \Delta$$

where

$$\Delta = [1,0] \text{ or } [1,1]$$

i.e.,

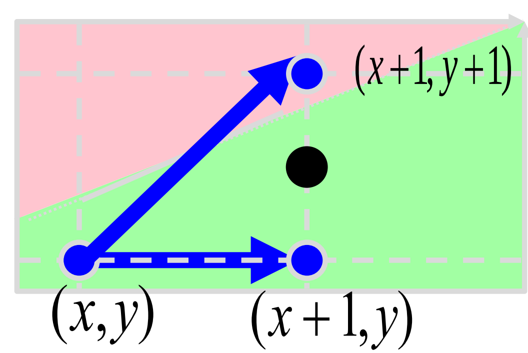
$$F = F(P_0) + 2dy - dx$$

$$If\ F<0 \quad F = F + 2dy$$

or

$$If\ F>=0\ F = F + 2\,dy - 2dx$$

- **Initialize *x, y*, F**
- **Loop until end of line:**
  - **draw pixel *(x,y)***
  - **increment *x***
  - **if F>0, increment *y***
  - **increment F according to whether Δ is [1,0] or [1,1]**



Bresenham Line Algorithm

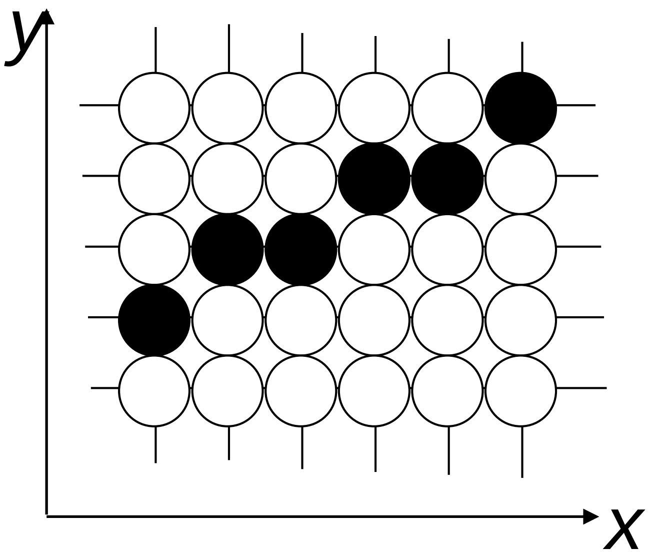
This does the right thing (same as DDA) at a **cost of only 2 or 3 integer adds per point**. (assumes sorted endpoints, 0<slope<1)

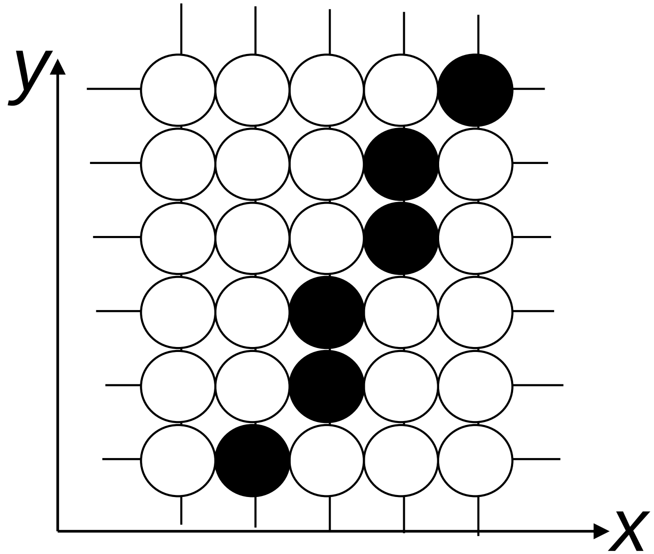
```
void draw_line(int x0, int y0, int x1, int y1) {
    int x, y = y0;
    int dx = 2*(x1-x0), dy = 2*(y1-y0);
    int dydx = dy-dx, F = dy-dx/2;

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

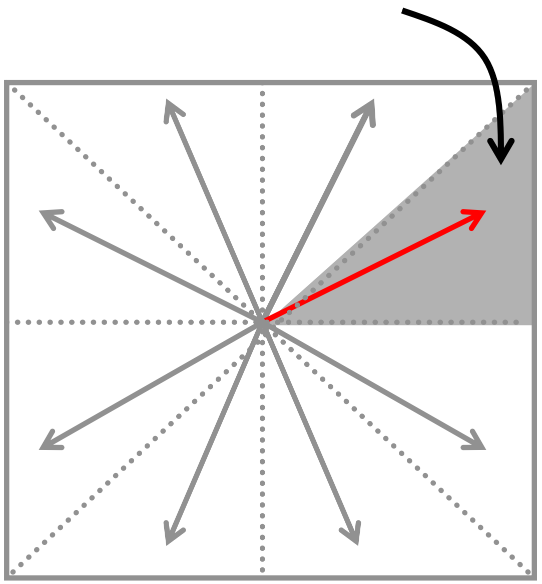
Line Drawing, Cases by Octant

- The algorithms for drawing lines need to step along one pixel at a time in the “fast” direction, which depends on the slope of the line
- We also have to worry about reversed end point order (drawing from large to small X, for example).
- This gives us 8 cases.



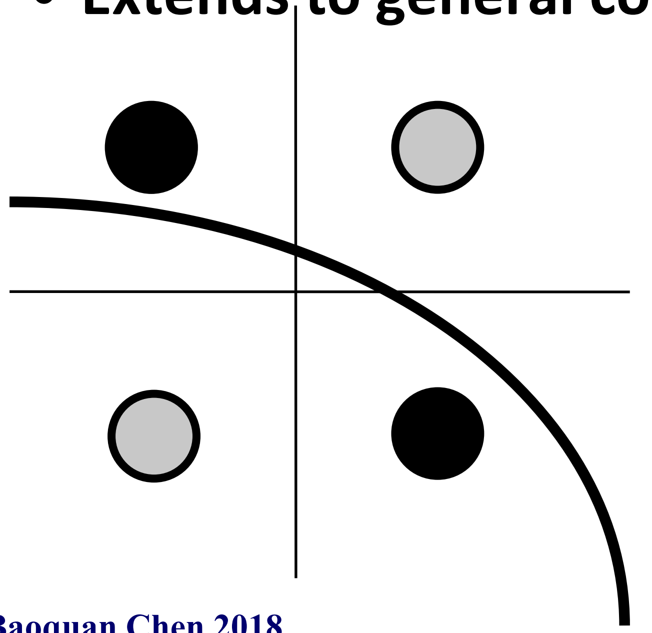


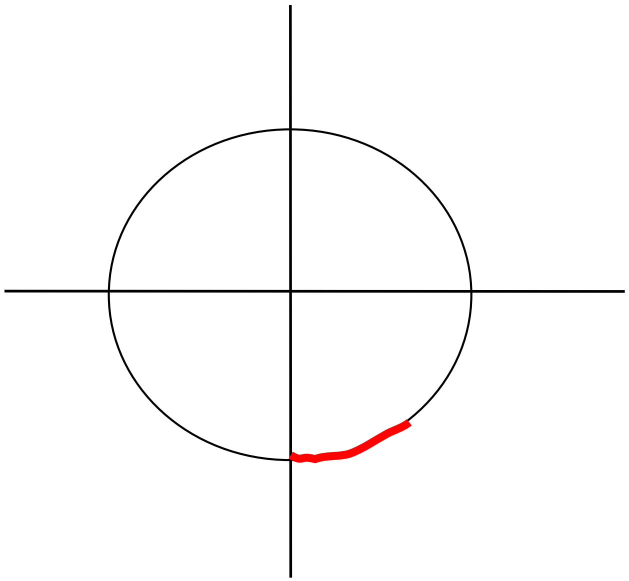
We’ll assume slope is between 0 and 1



Bresenham Algorithm for Circles

- Same approach as line algorithm
  - use a decision variable formula derived for a circle ($F = x^2 + y^2 - r^2$)
- Eightfold symmetry
  - only compute the points for one octant - use sign flips to give the rest
- Extends to general conics (ellipses...)





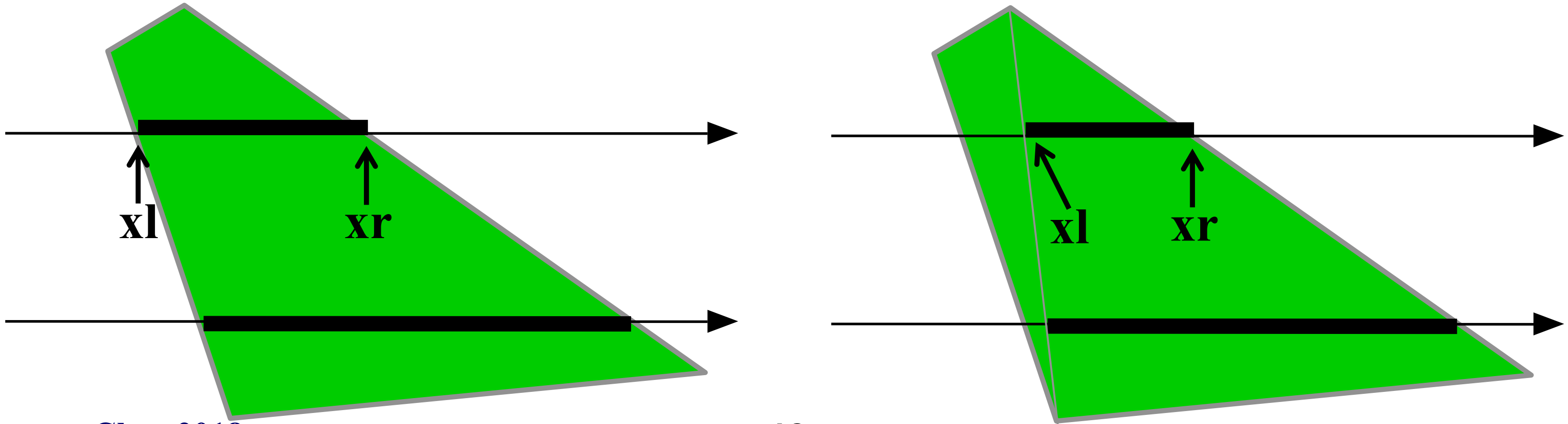
Bresenham Circle Algorithm

This draws a circle by calculating in one octant and re-using the resulting point 8 times

```
void draw_circle(int radius) {
   int x = 0, y = radius;
   int d = 1-radius;
   while (y>x) {
      if (d<0) /* select East point next */
         d += 2*x + 3;
      else {    /* select South-East point next */
         d += 2*(x-y) + 5;
         y--;
      }
      x++;
      draw_8_pts(x,y); /* draws point in each octant */
   }
}
```

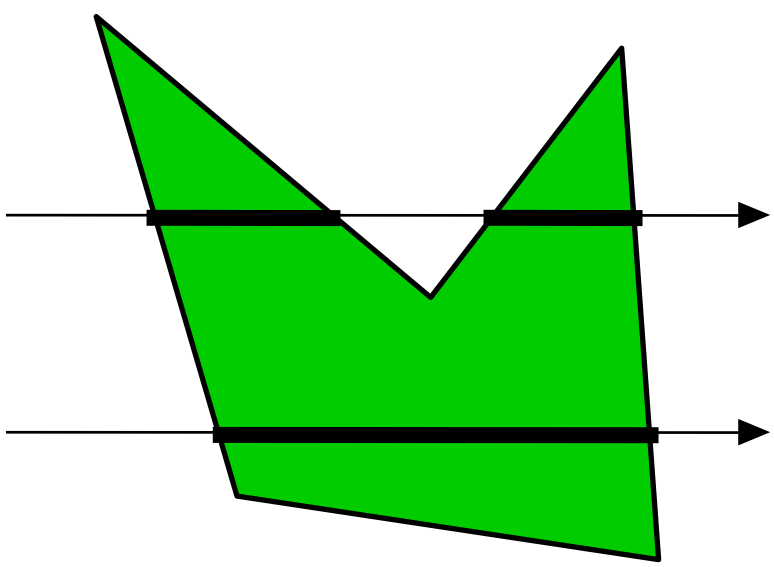
Scan Converting Filled, Convex Polygons

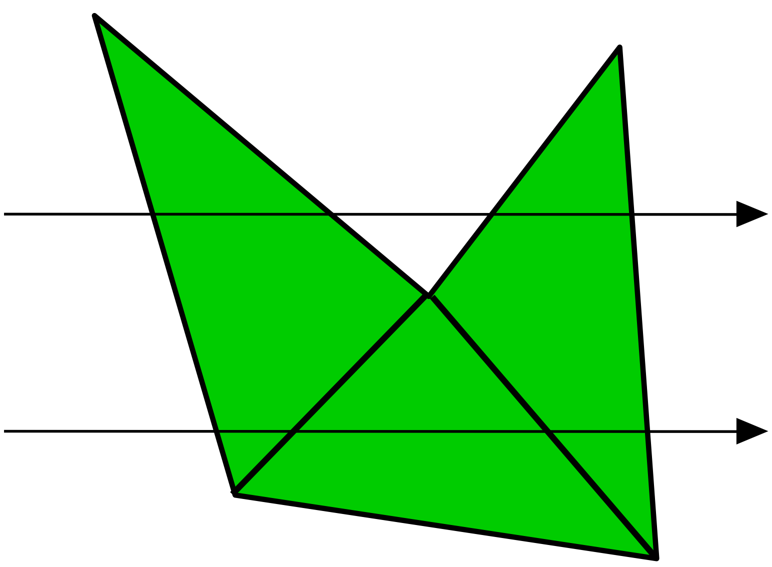
- Find top and bottom vertices
- Make list of edges along left and right sides
- For each scanline from top to bottom
  - There's a single span to fill
  - Find left & right endpoints of span, xl & xr, (can use Bresenham's algorithm
  - Fill pixels inbetween xl & xr
- If you don't do all of the above carefully, cracks or overlaps between abutting polygons result!



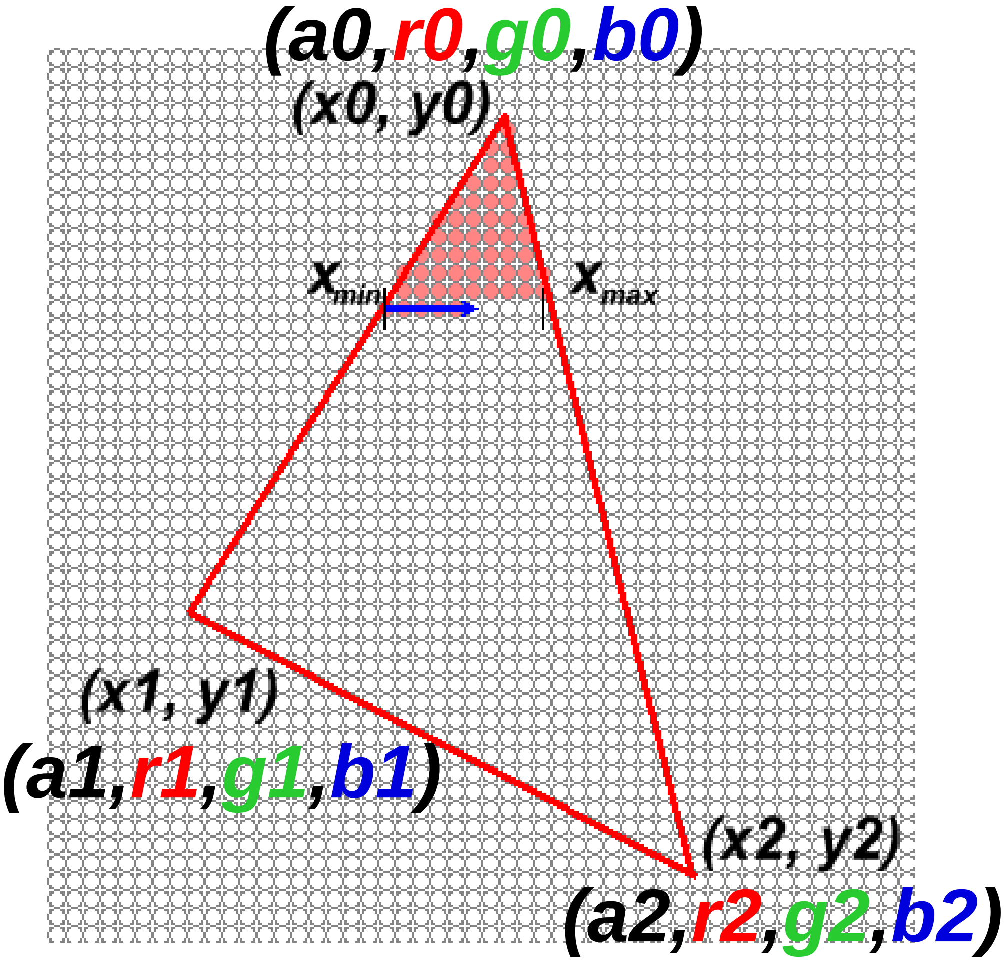
Scan Converting Filled, Concave Polygons

- For each scanline
  - Find all the scanline/polygon intersections
  - Sort them left to right
  - Fill the interior *spans* between intersections
  - *Parity Rule:* odd ones are interior, even are exterior
- Or, triangulation



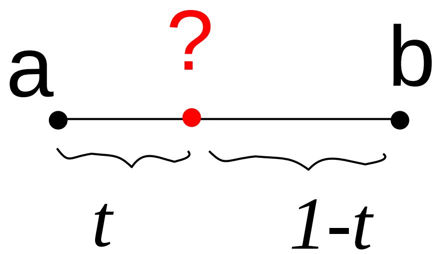


Color Interpolations



Review on Interpolation

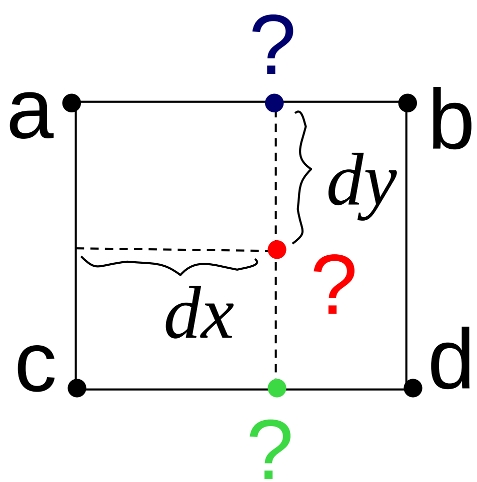
- **Linear Interpolation**



? $= a(1-t)+bt$

$= a + (b-a)t$

- **Bilinear Interpolation**



? $= a(1-dx)+b\,dx$

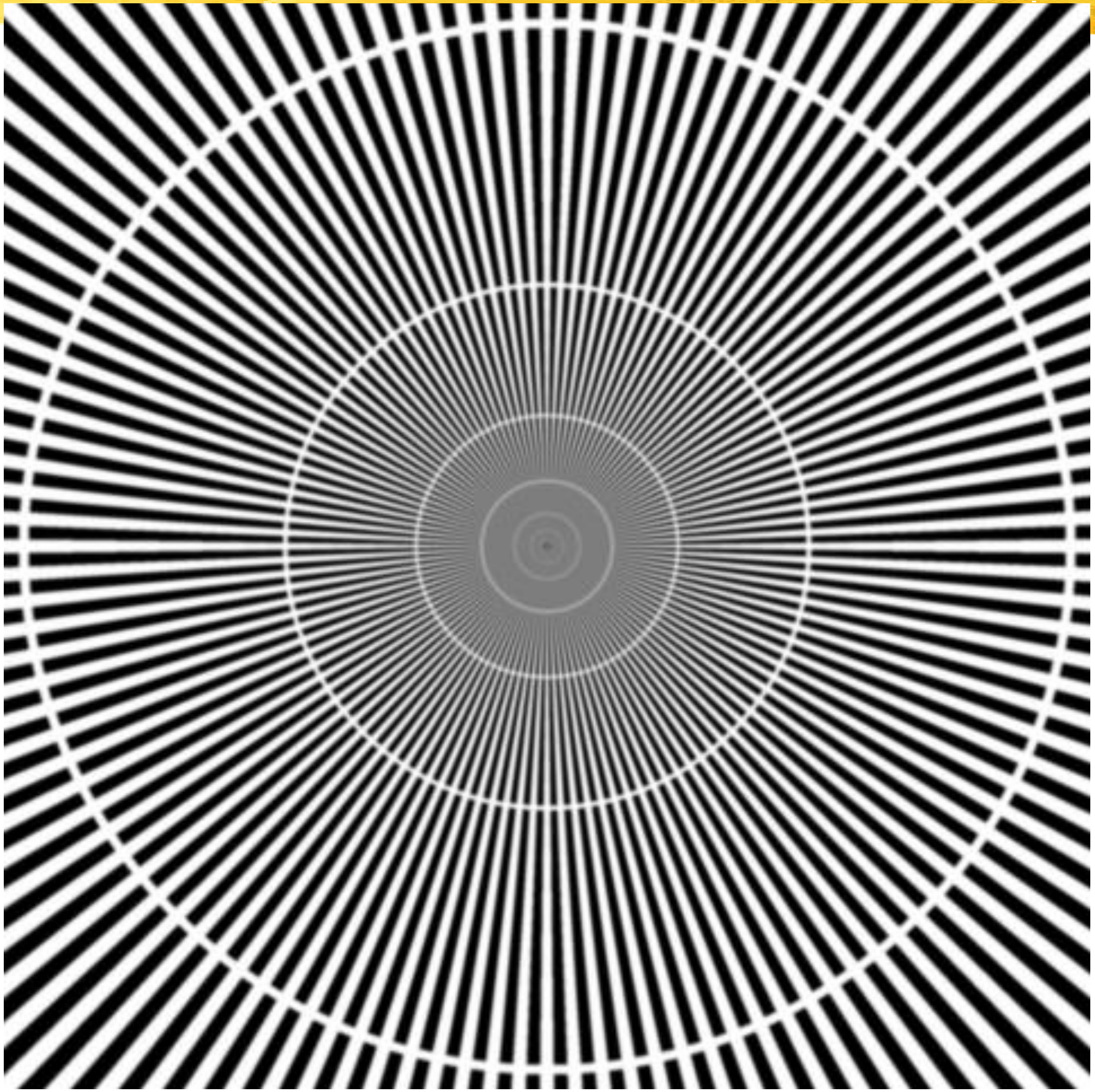
? $= c(1-dx)+d\,dx$

? = ?$(1-dy)$+?dy = ? + (?-?)dy

$= a(1-dx)(1-dy)+b\,dx(1-dy)$

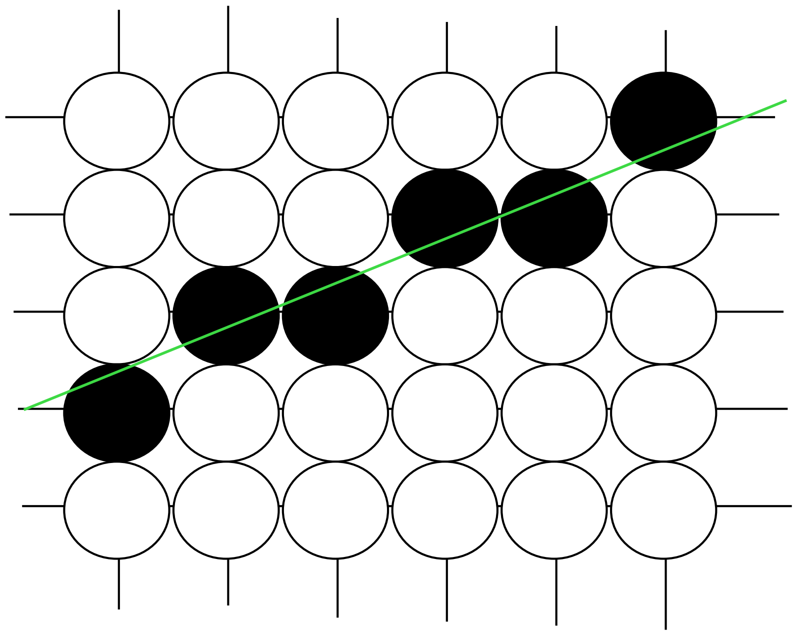
$+c(1-dx)dy+d\,dx\,dy$

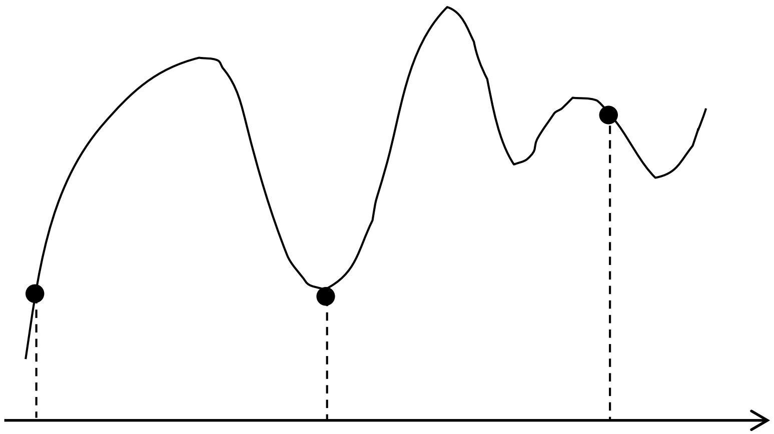
Again, How to Draw This?



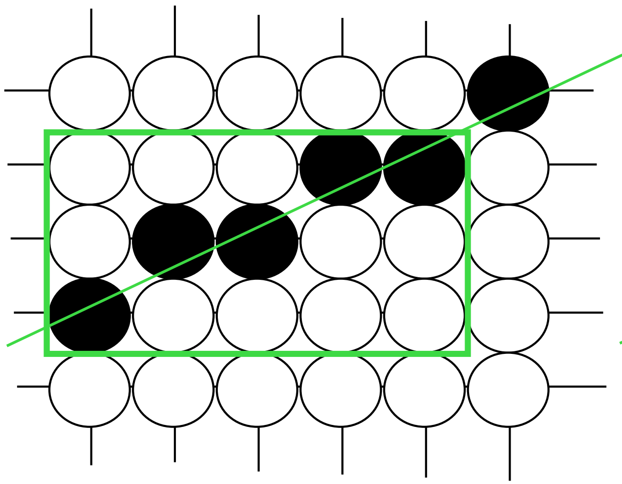
Aliasing

Samples don't capture geometry changes - Not dense enough!

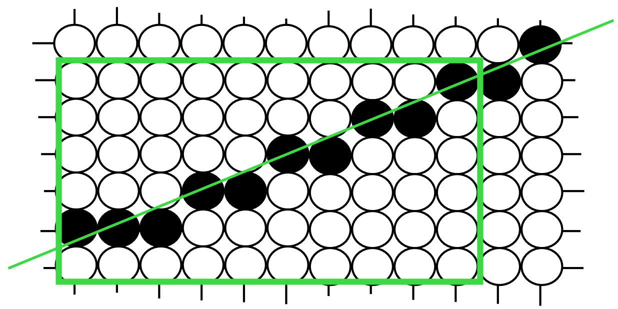




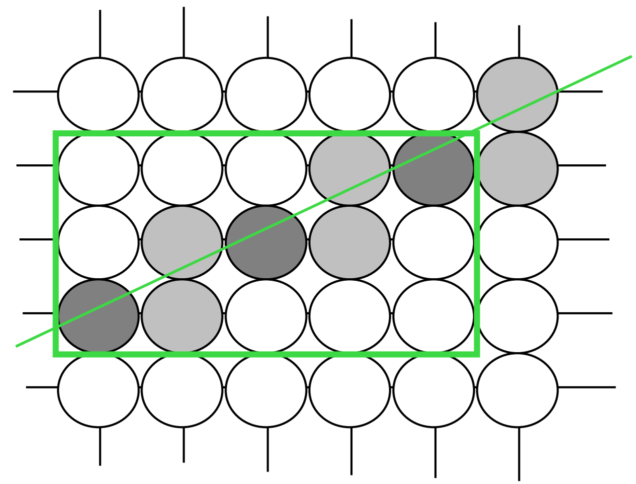
Antialiasing: Super-sampling



screen
resolution

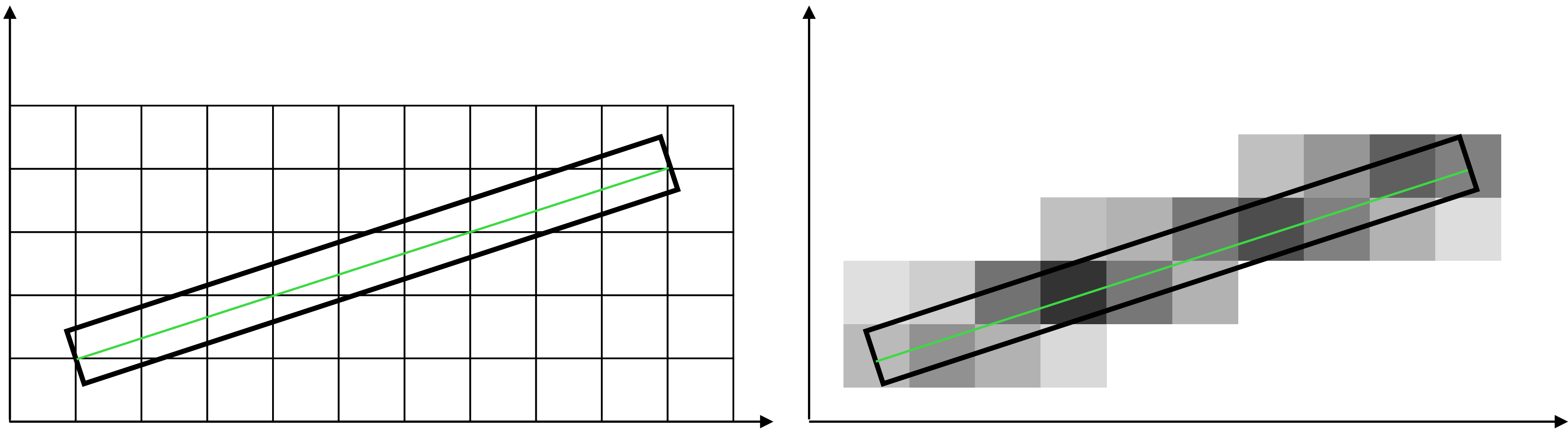


Increasing
resolution



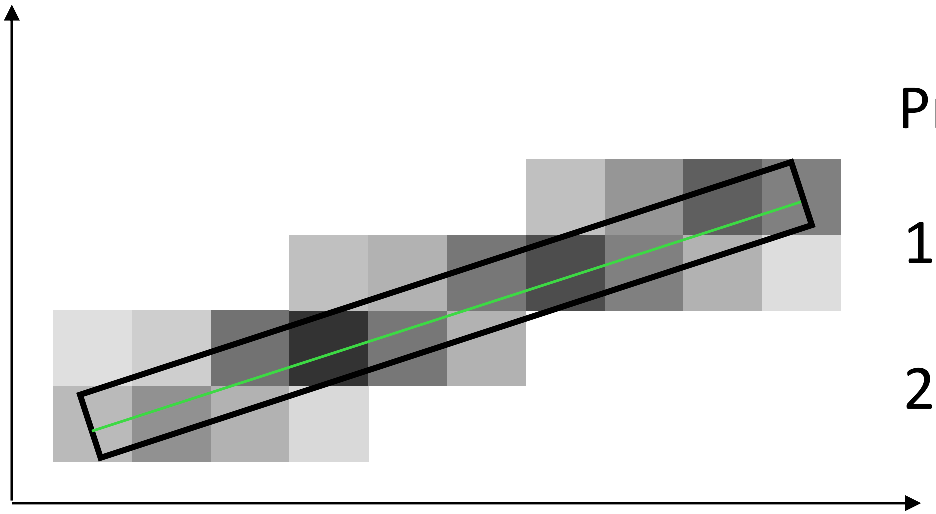
Back to screen
resolution

Antialiasing: Unweighted Area Sampling



- Line with ‘thickness’
- Pixel’s color, here ‘blackness’, depends on the intersection area between the thick line and the pixel square

Antialiasing: Unweighted Area Sampling



Properties:

1. Intenstity soley based on intersection area
2. Equal areas equal intensity ?

However, the same area closer to the pixel center should have greater influence than does one at a greater distance! This consideration leads

to

Weighted Area Sampling:

‘blackness’ = area*f(distance),

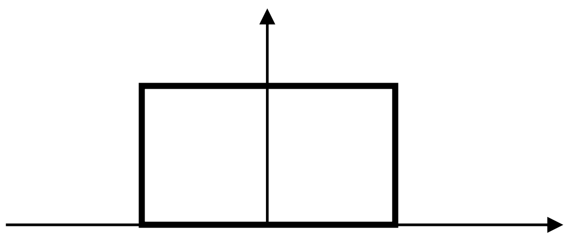
Where f: weighting function, dist: pixel center distance to the line

Antialiasing: Weighted Area Sampling

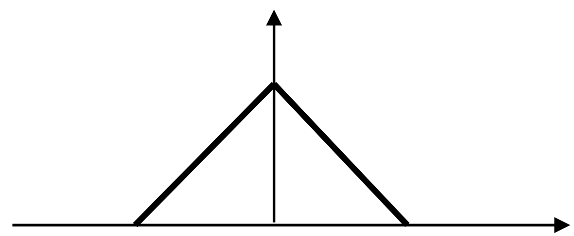
We can define many weighting functions!

Can be anything, BUT,

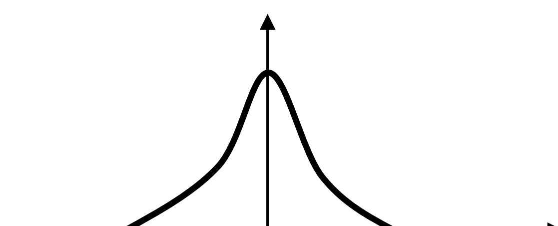
1. Finite non-zero region
2. Meaningful (e.g., decreasing from high to zero value when distance increases
3. Full ‘area’ equal to 1



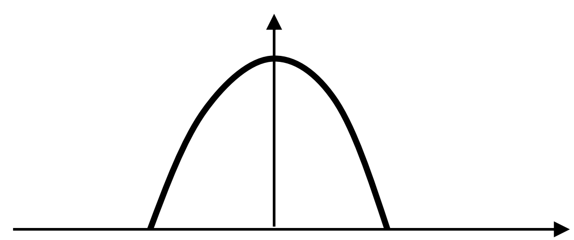
Box



Tent/Bartlet



Gaussian



Cone

Extend Everything to 3D

Voxelization

More Issues

The devils in the details:

1. Non-integer endpoints
   (occurs frequently when rendering 3D lines)
2. What if a line endpoint lies outside the viewing area?
3. How do you handle thick lines?
4. Optimizations for connected line segments
5. Lines show up in the strangest places
6. ……