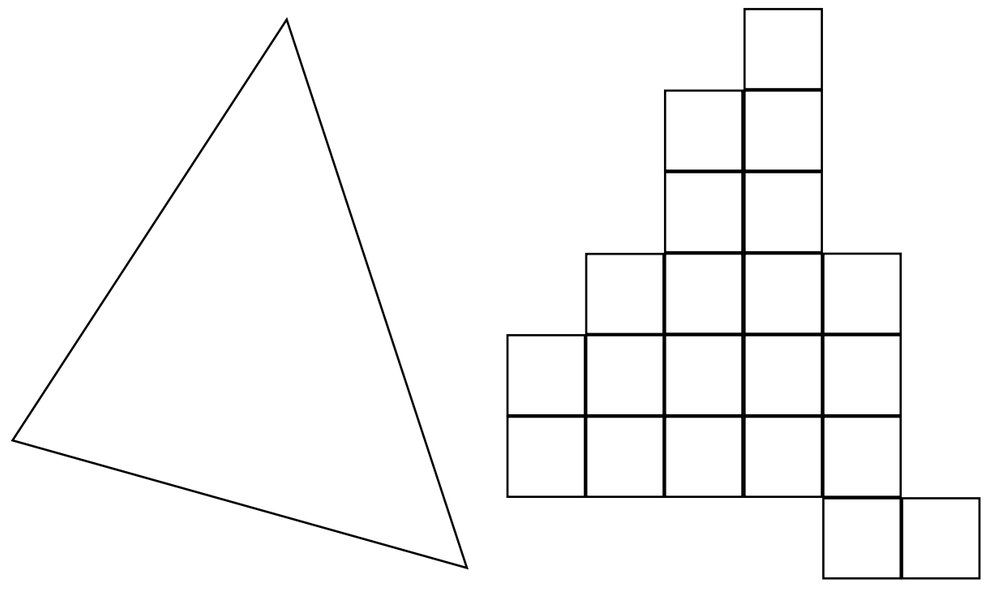
Other low-level algorithms:

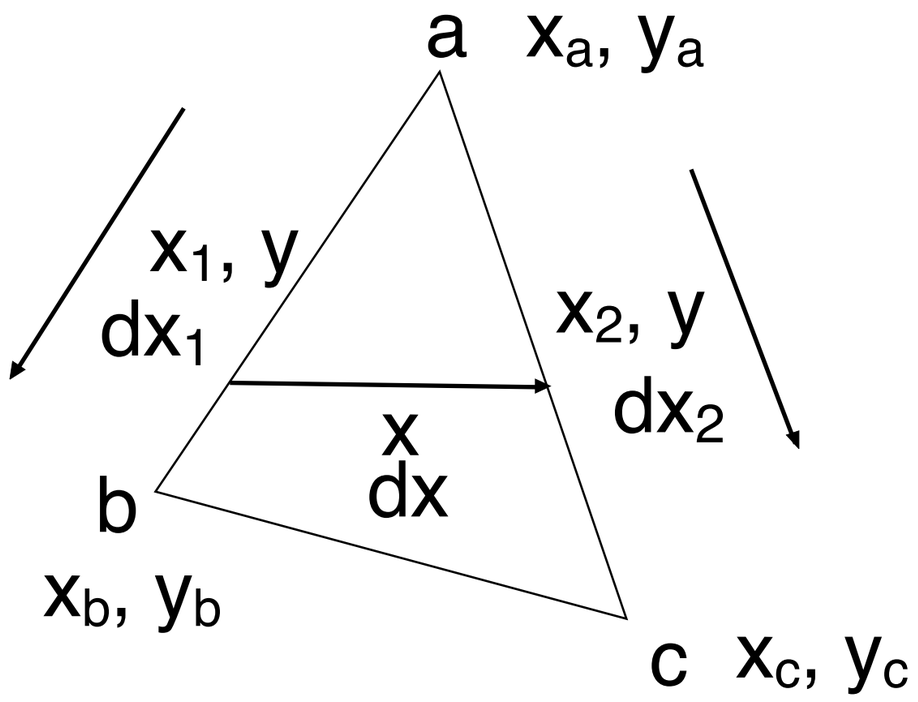
Polygon rendering
Scan-line polygon fill
Inside-outside tests
Flood fill

Polygon rendering: Scan conversion



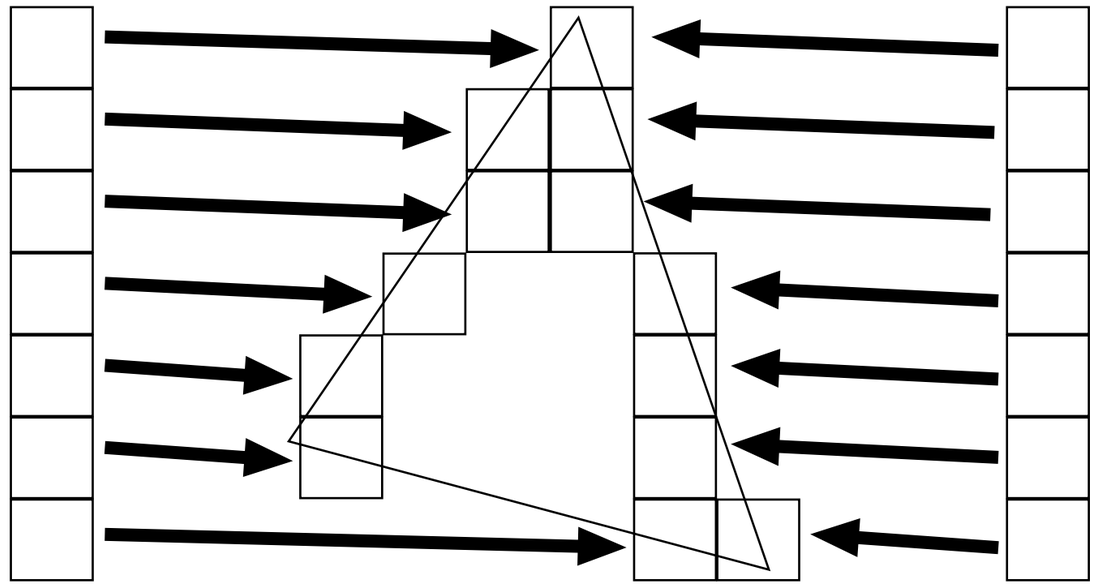
Given a triangle, find the pixels to fill

Method 1: Single-phase



Walk along edges

Method 2: Two-phase method



Write edges to span-buffer, then fill each interval

Method 2: Two-phase method

Write edges to span-buffer:

Use a modified DDA algorithm
Always step in Y. Horizontal gaps is no issue.

Fill each interval:

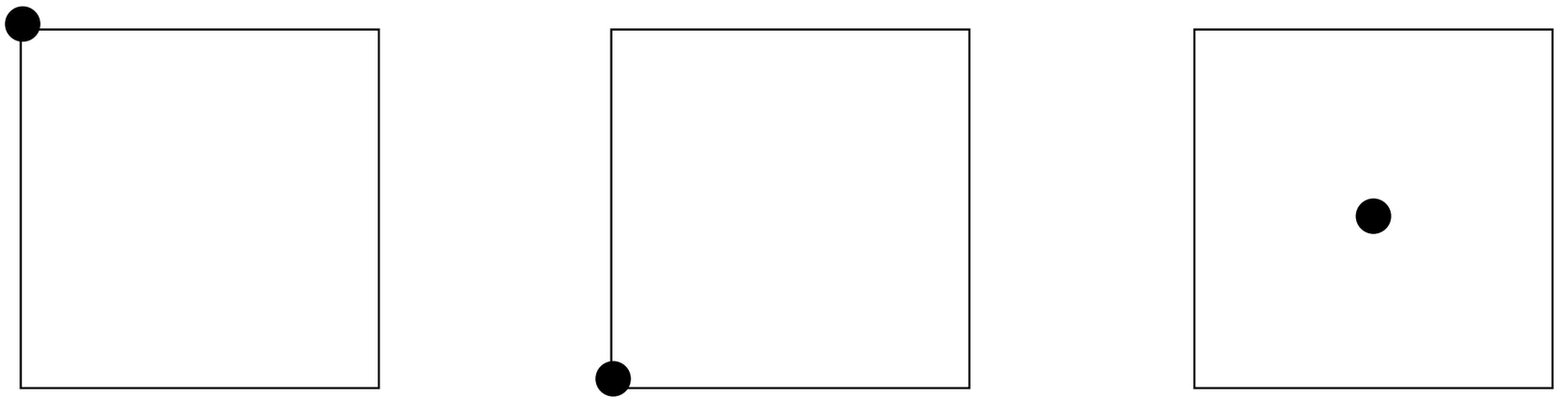
Essentially a simple “for” loop

Easier to implement, suitable for software rendering.

Pixel addressing and object geometry

So far, I have been careless with one question:

Where in the pixel (x,y) is the point (x,y)?



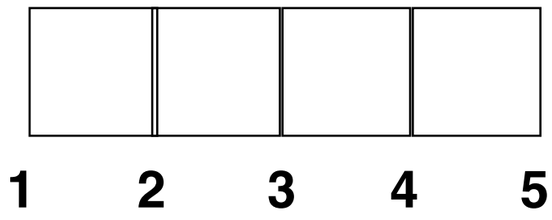
Is it here, top-left…?

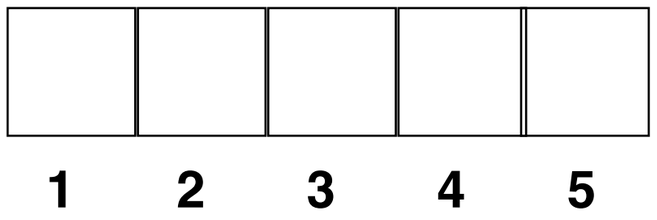
…or here, bottom-left…?

…or perhaps here, centered?

This is the “hotspot” of the pixel

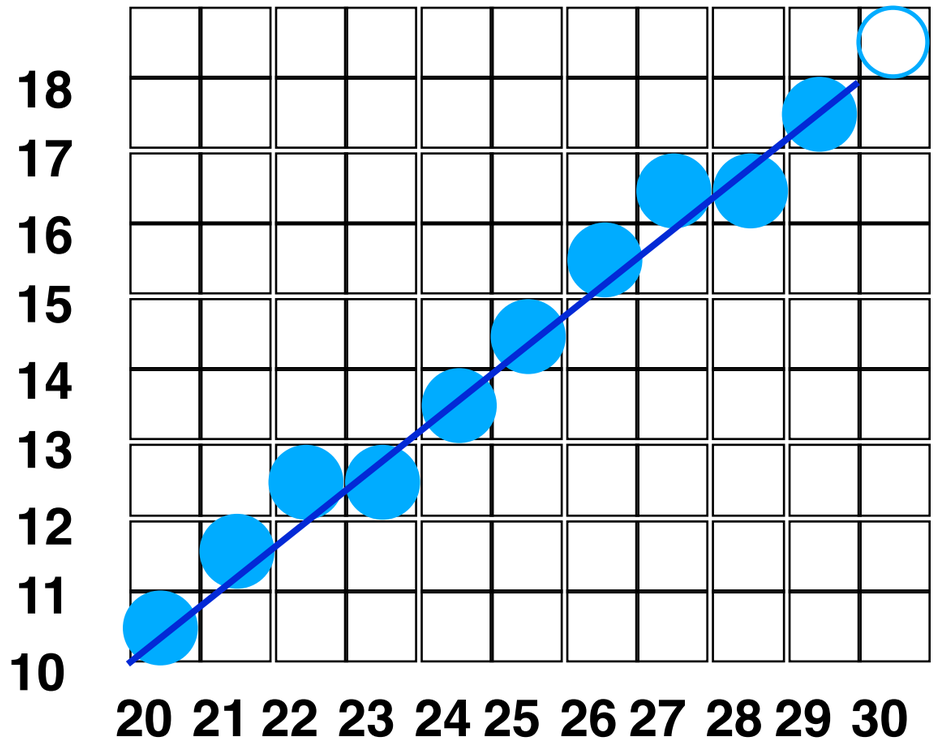
Is a line from 1 to 5
4 or 5 pixels long?





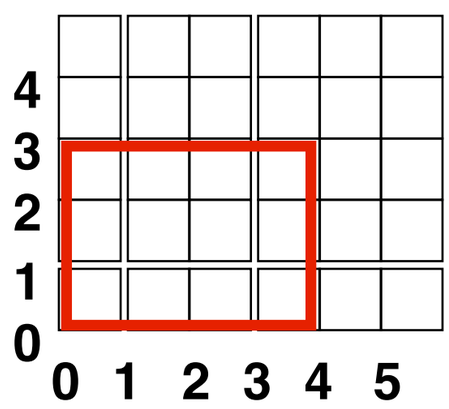
There is no universal answer
– but you may need to know!

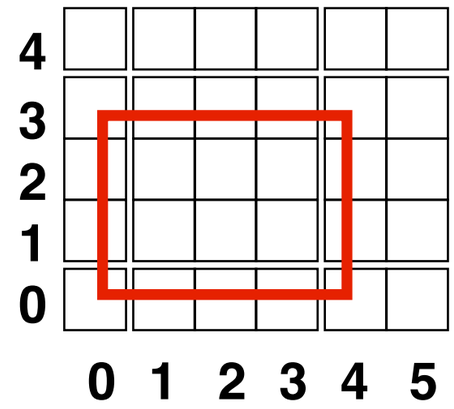
Example: The line (20,10) to (30,18):



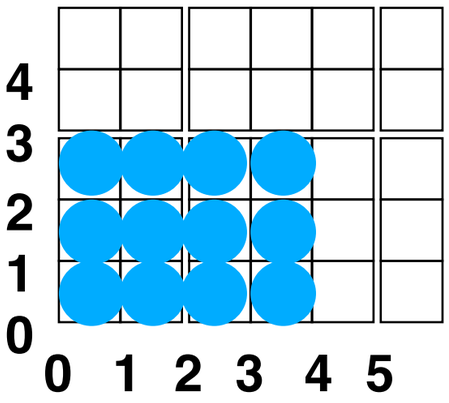
Should the pixel at (30,18) be painted or not?

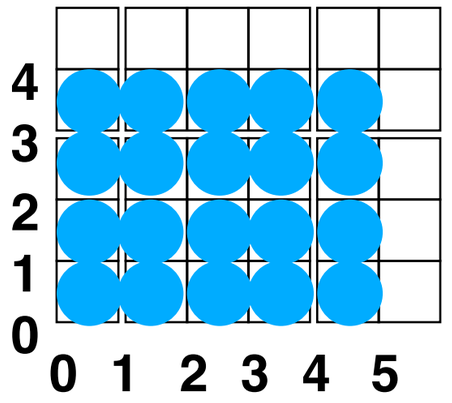
Example: The rectangle (0,0)(4,0)(4,3)(0,3):





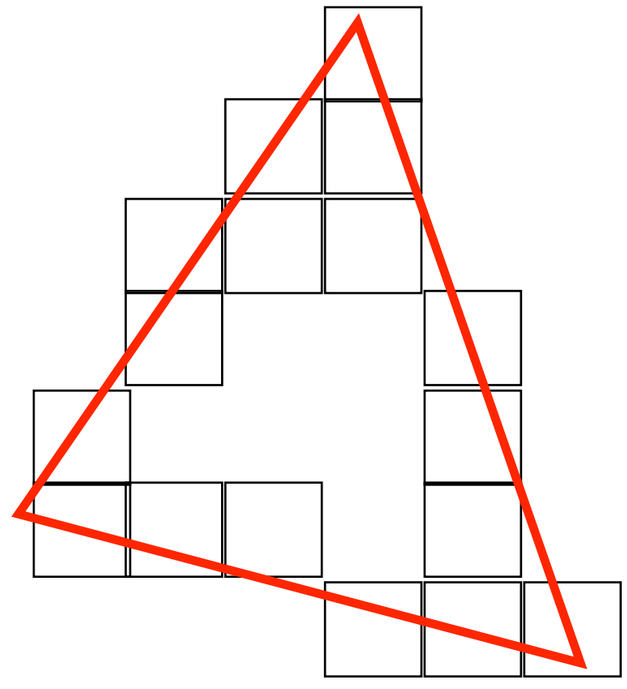
How is it defined?





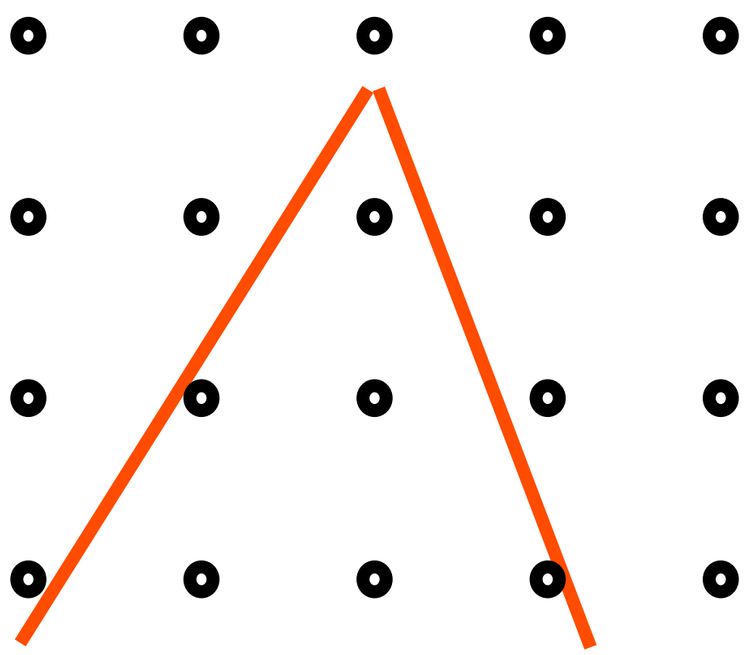
How should it be drawn?

Which pixels are inside?



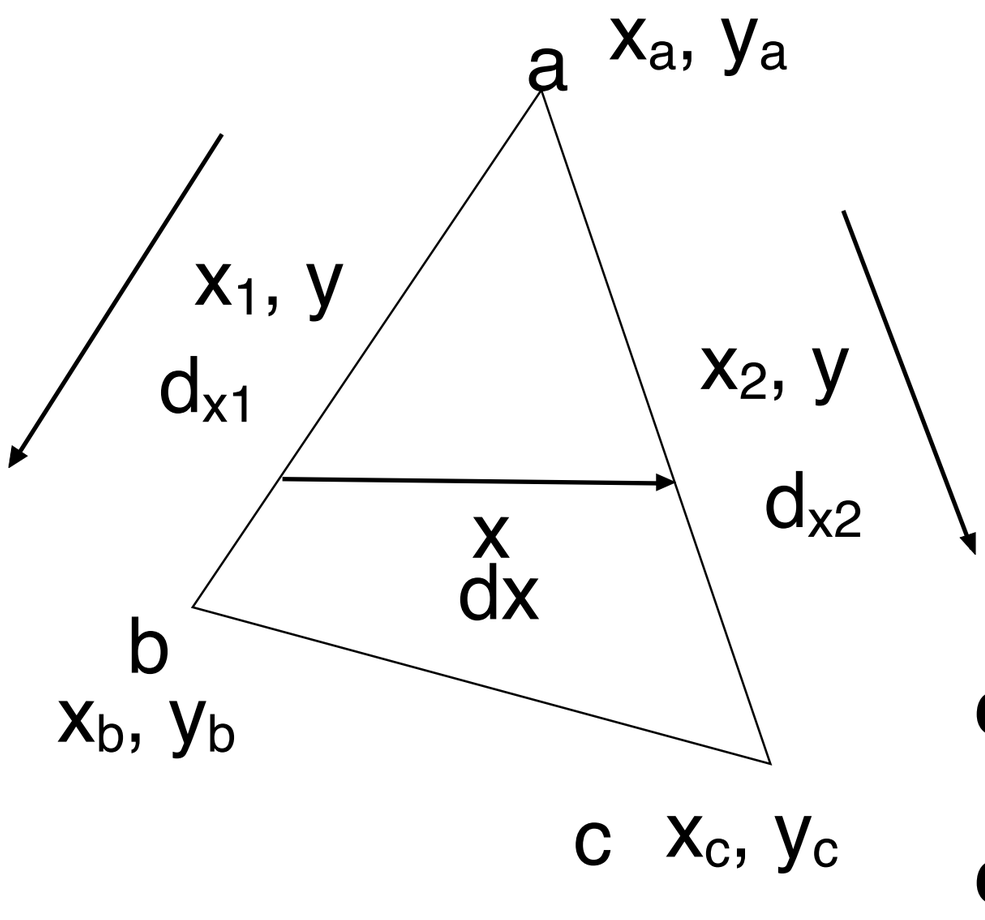
Without a proper definition, we will get errors, visible “gaps” between polygons!

Sub-pixel precision rendering



If the pixel point, according to the pixel definition, is inside, it should be included

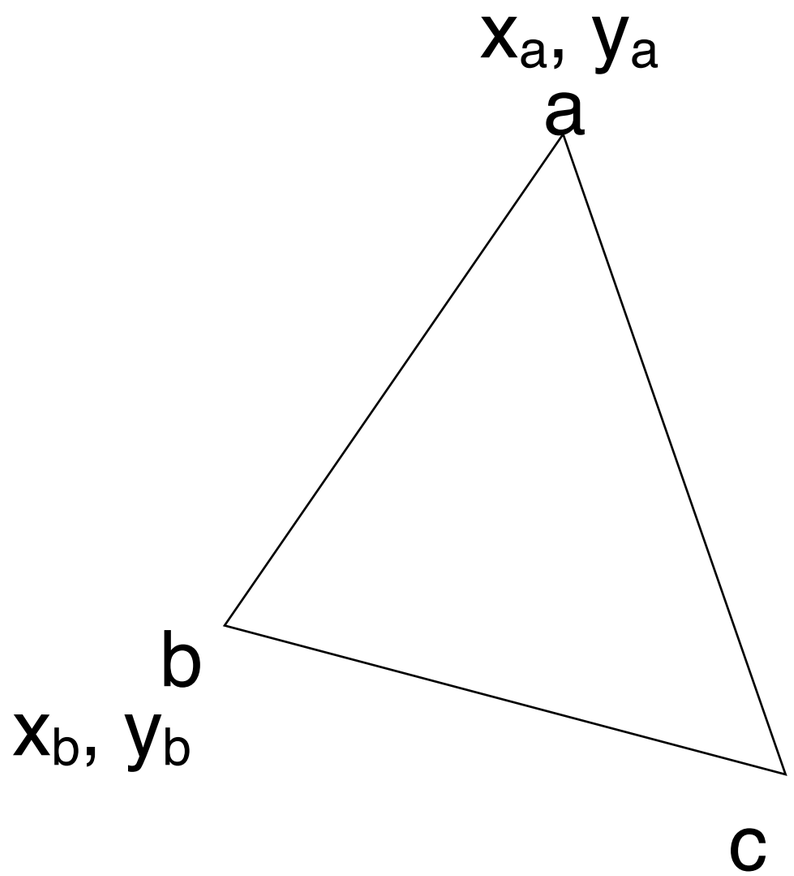
Single-phase revisited



$$d_{x1} = (x_b - x_a) / (y_b - y_a)$$

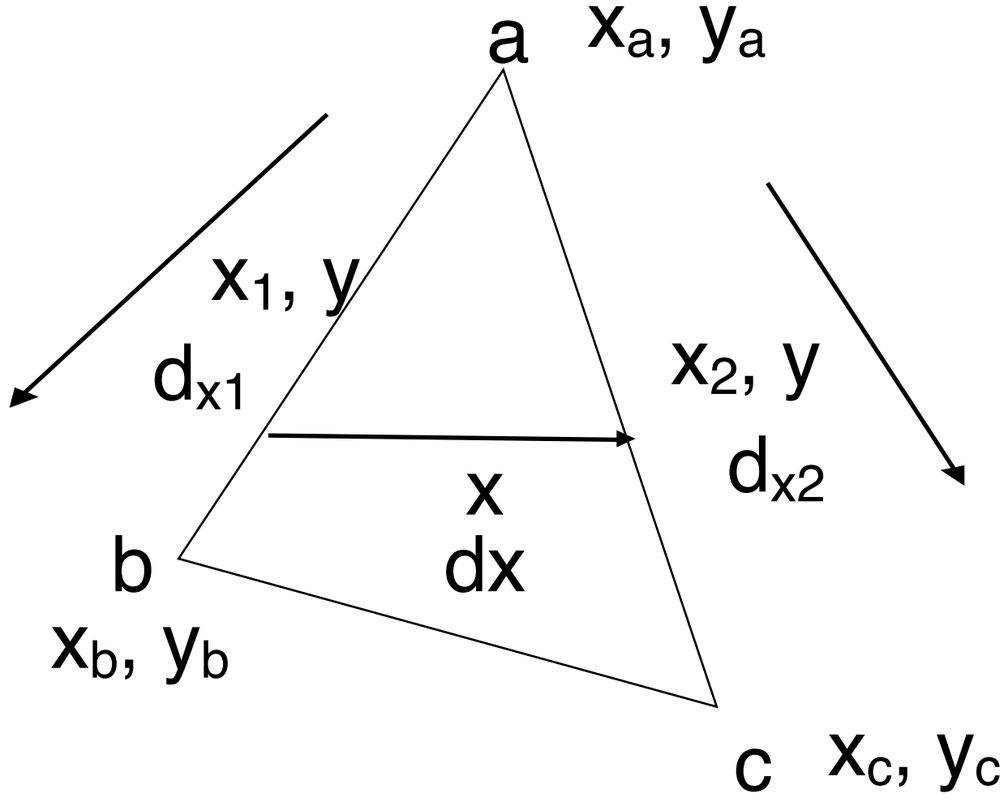
$$d_{x2} = (x_c - x_a) / (y_c - y_a)$$

Single-phase revisited



First: Sort a, b, c to top, middle, bottom

Single-phase revisited

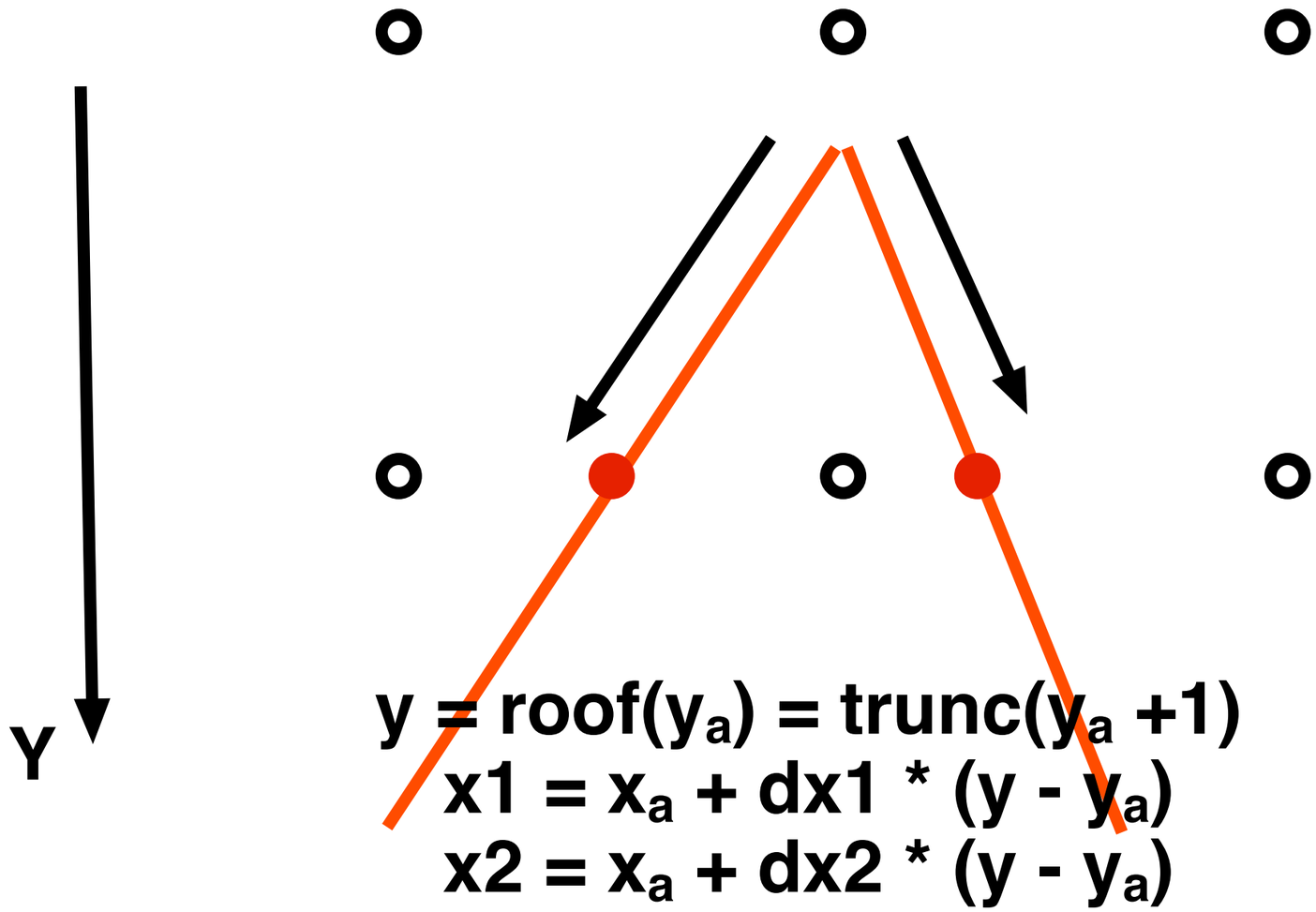


Separate to two parts:

Top-to-middle and middle-to-bottom

Recalculate d_{x1} at b

Sub-pixel precision rendering

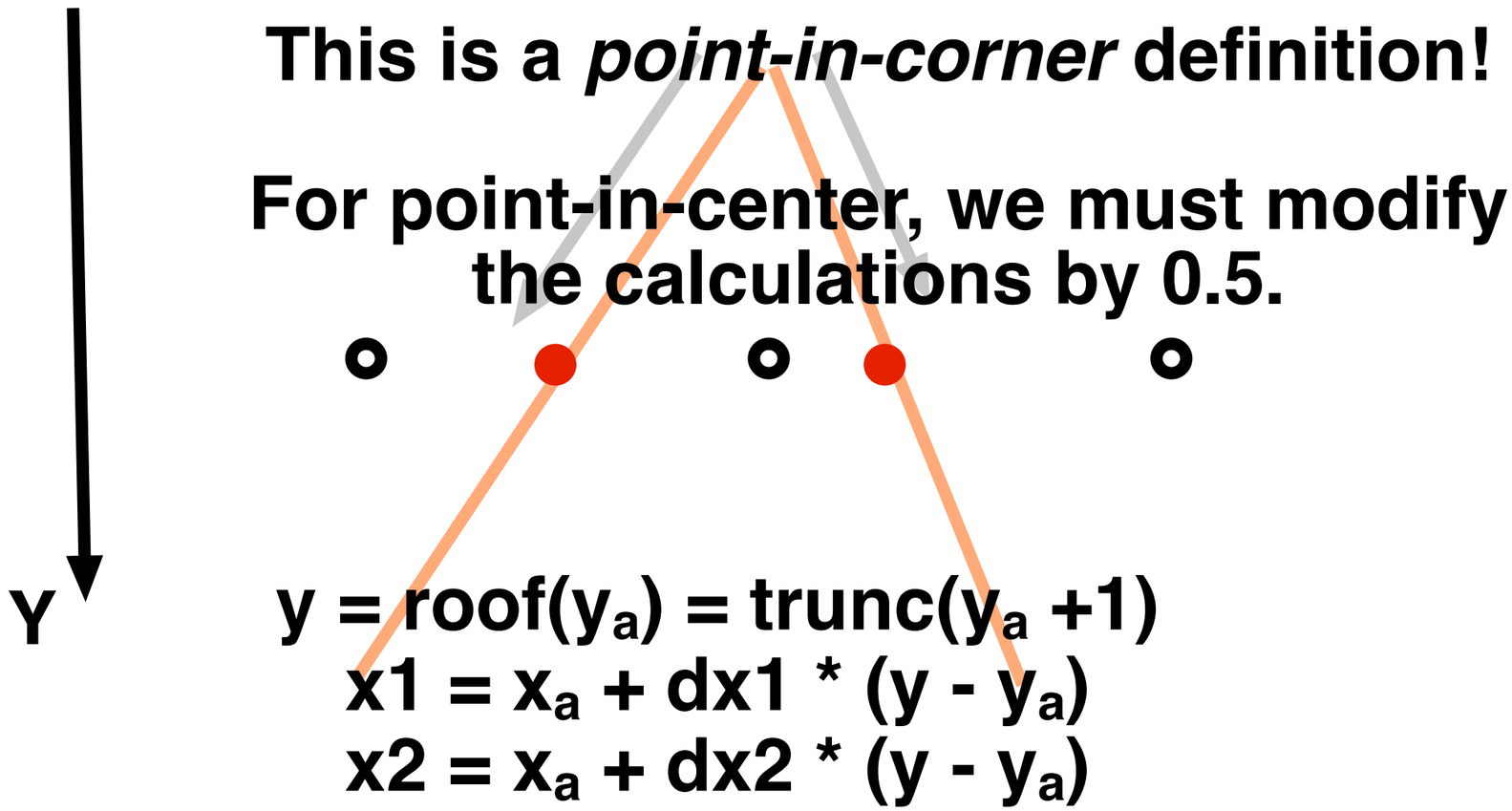


$y = roof(y_a) = trunc(y_a + 1)$

$x1 = x_a + dx1 * (y - y_a)$

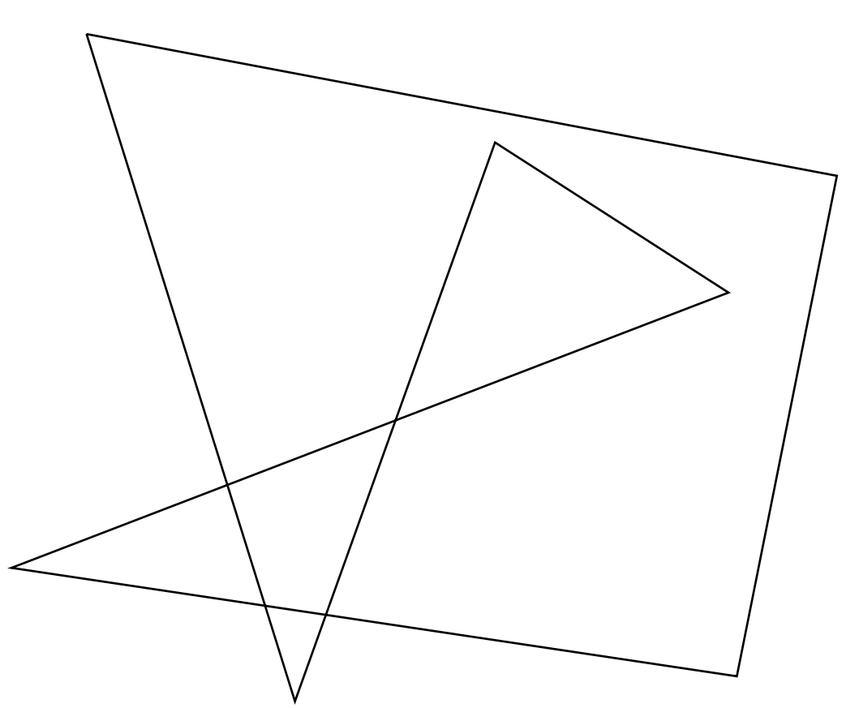
$x2 = x_a + dx2 * (y - y_a)$

Sub-pixel precision rendering

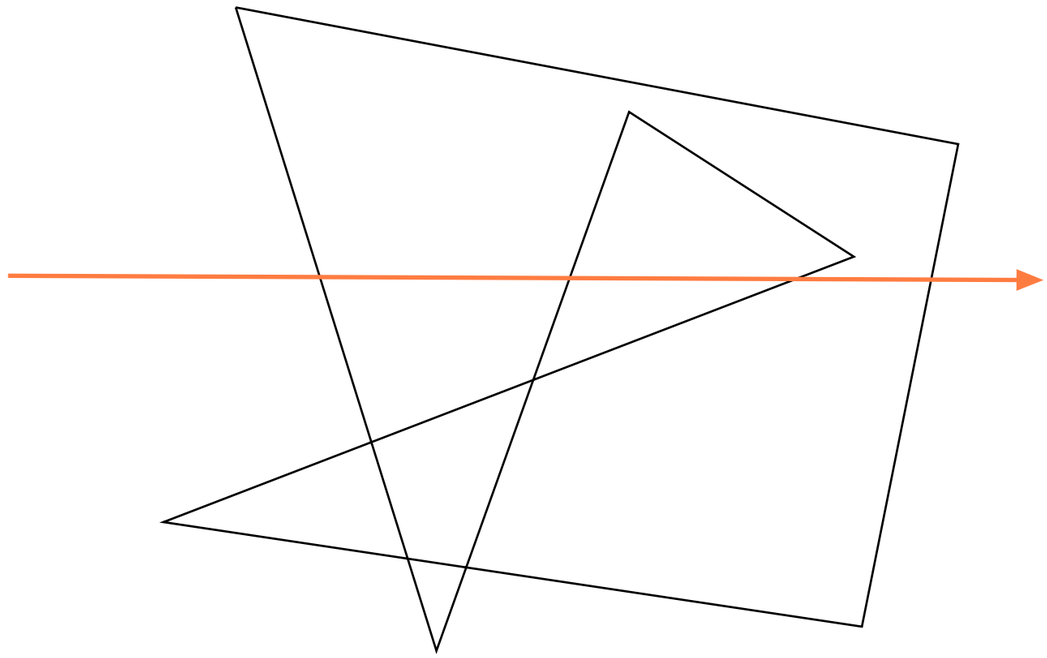


Filling an arbitrary polygon:

Scan-line polygon filling

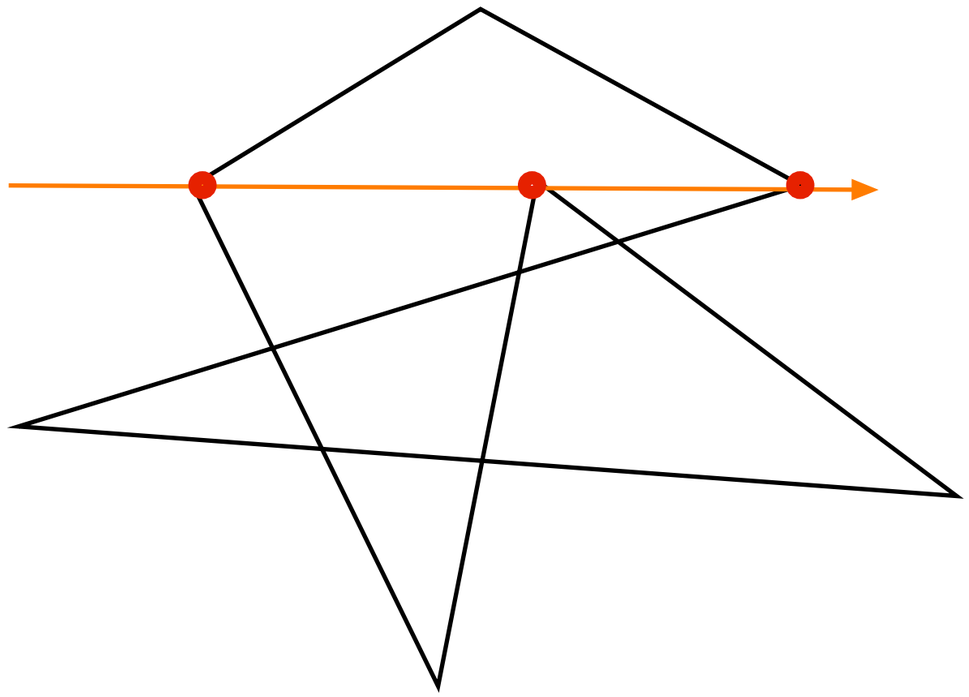


Scan-line polygon filling



**Go from left to right,
fill when there is an odd number
of edges to the left!**

Problem:
Scan-line through vertices



We get a count of two at each, which causes incorrrect filling!

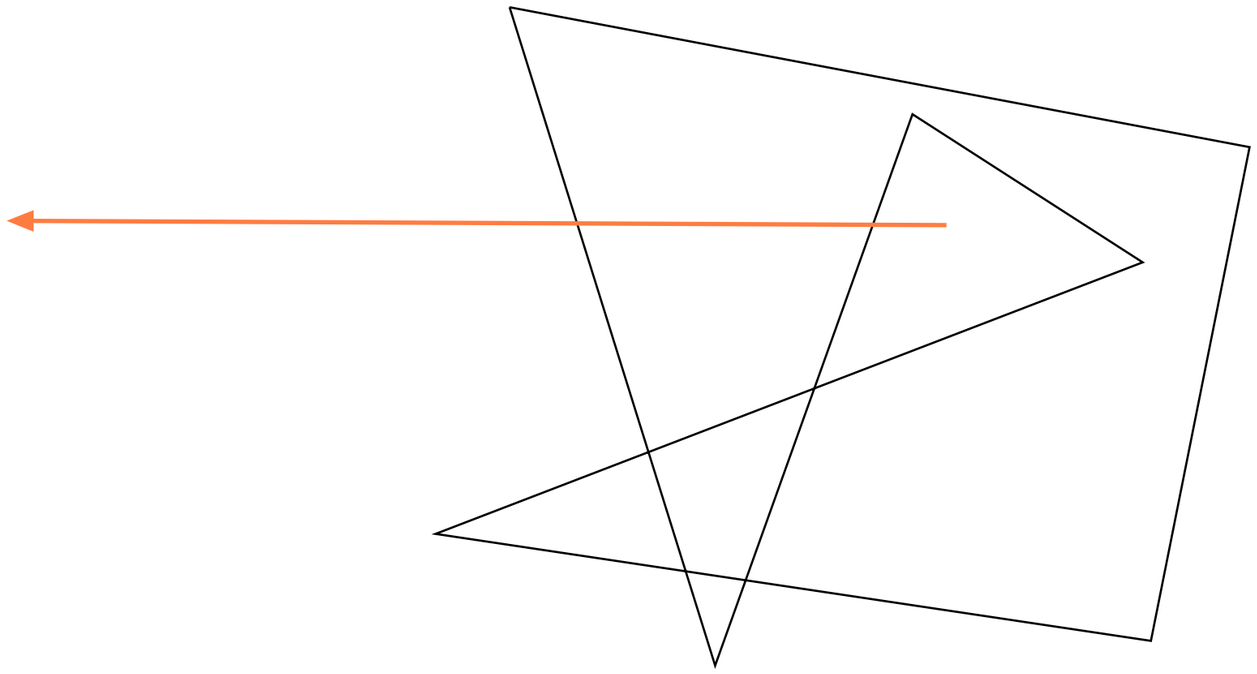
Solutions:

1) Do specific checks for vertices, and detect vertices where the two edges are on different sides. Then the two edges count as one!

2) Pre-processing: For vertices with the edges on different sides, shorten one with one scan-line.

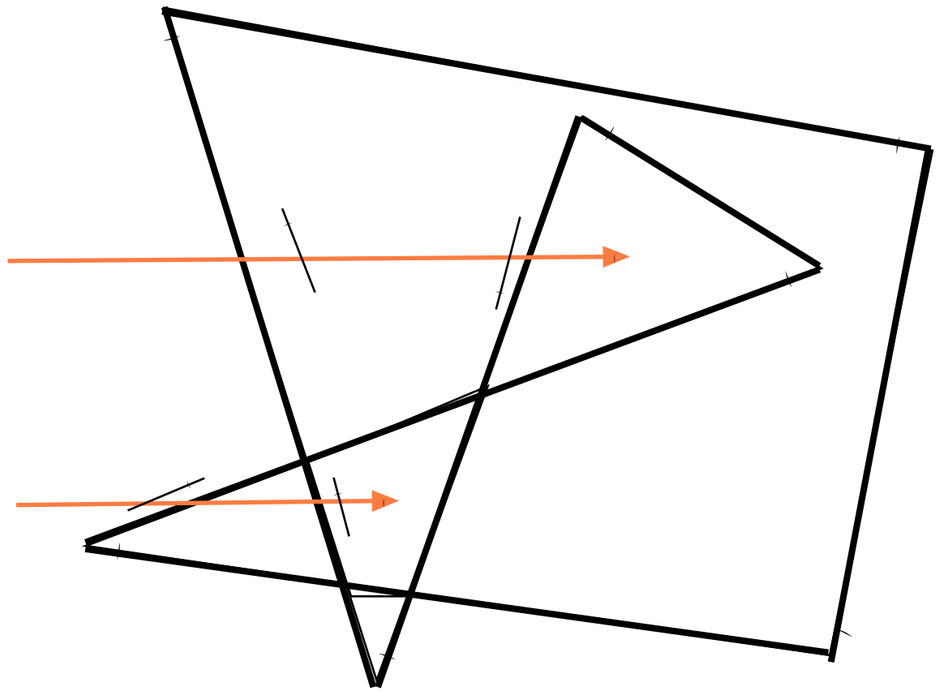
3) Use the pixel geometry definition! When properly used, the problem disappears!

Inside-outside tests
Method 1: Odd-even rule



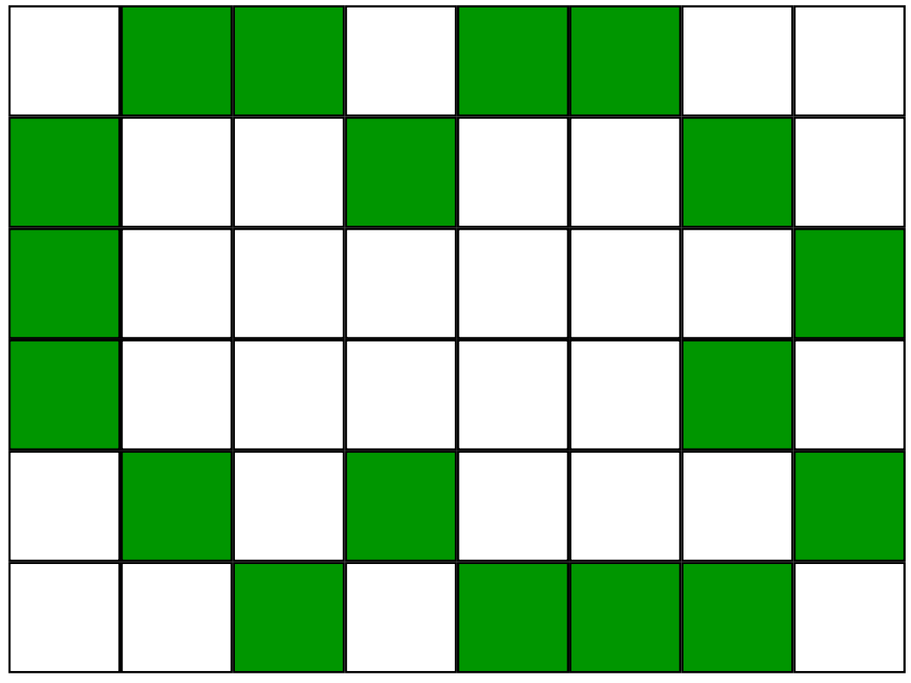
To learn if a pixel is inside the polygon, you can apply the same kind of test!

Inside-outside tests
Method 2: Non-zero winding rule



Check the directions of intersections!

Flood fill



A color defines the area to be filled

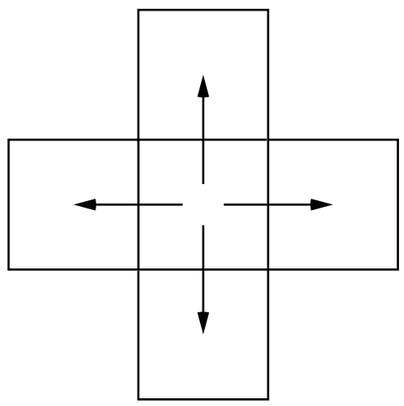
Simple recursive algorithm:

```
procedure FloodFill(x,y,fill,target);

current := GetPixel(x,y);
if (current = target) then
 SetPixel(x,y,fill);
 FloodFill(x+1, y, fill, target);
 FloodFill(x-1, y, fill, target);
 FloodFill(x, y+1, fill, target);
 FloodFill(x, y-1, fill, target);

procedure StartFloodFill(x, y, fill)
 target := GetPixel(x, y);
 if (fill <> target) then
  FloodFill(x, y, fill, target);
```

Not practical! Too deep recursions!

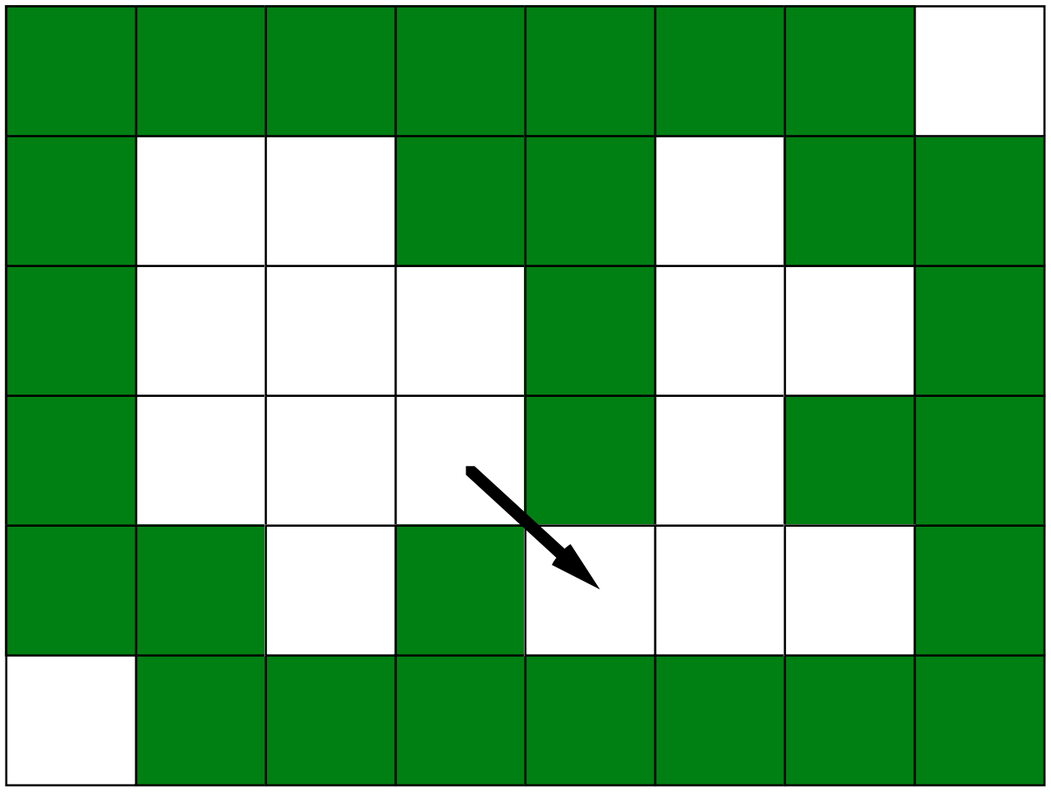


Flood fill using pixel spans:

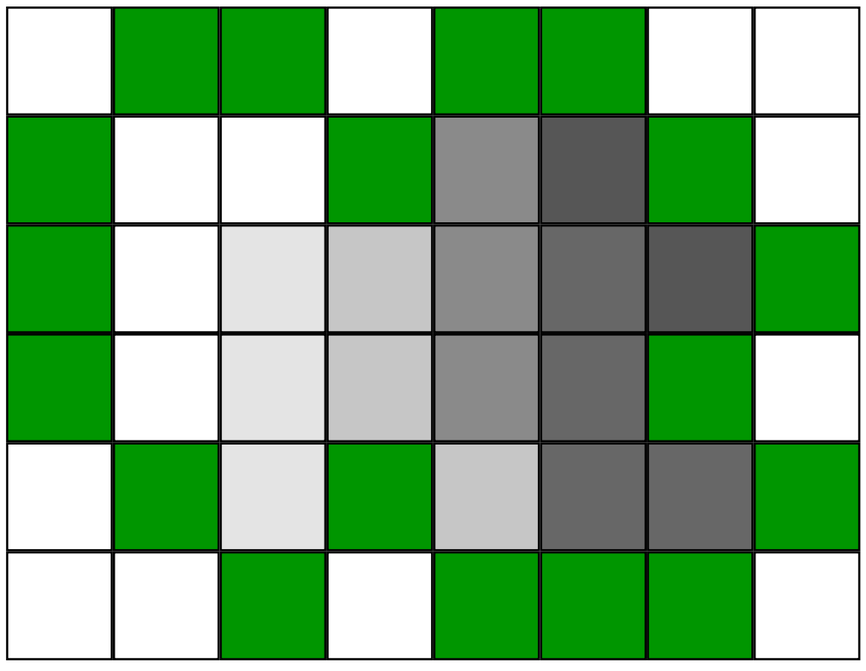
```
push starting pixel on stack
while stack not empty
 pull top pixel off stack
 for all fillable pixels A in the span
  fill the pixel
  for neighbor B above and below
   if pixel fillable and A or B at start of span
    push on stack
```

Efficient, low stack demand

Should we go through this crack or not?

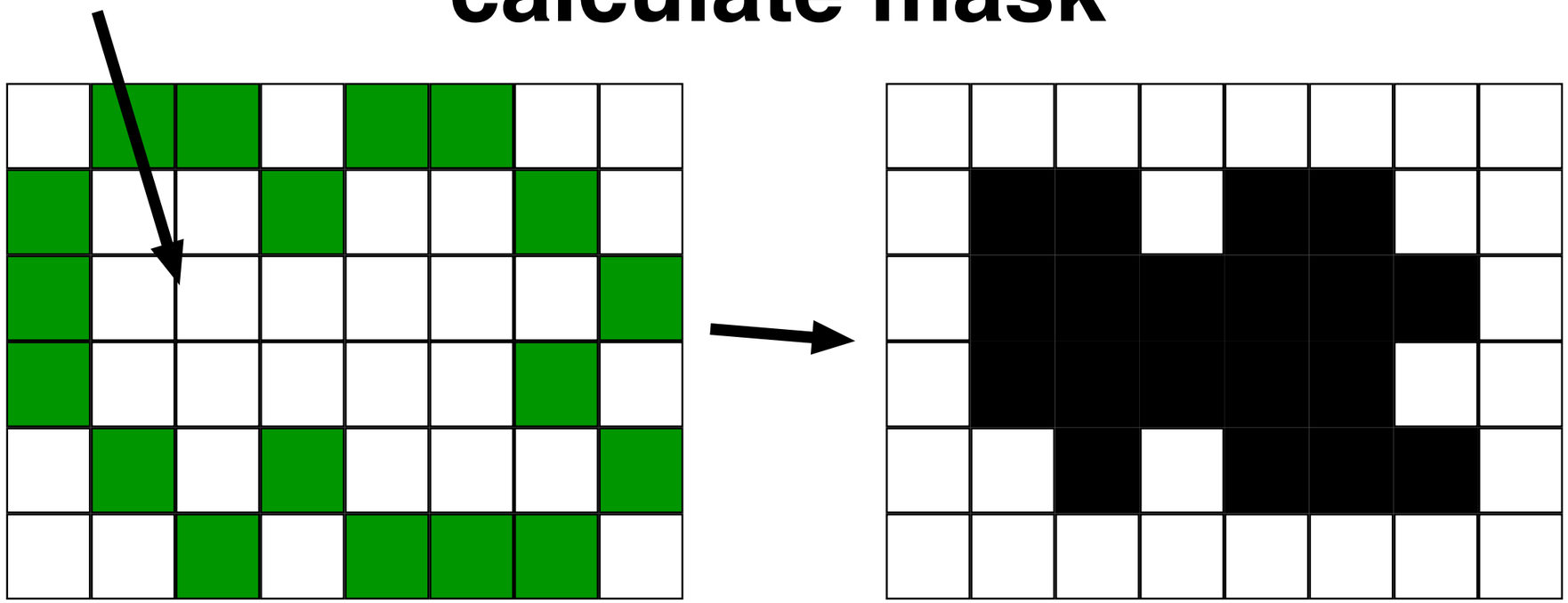


Better flood fill #1: flood fill color interval



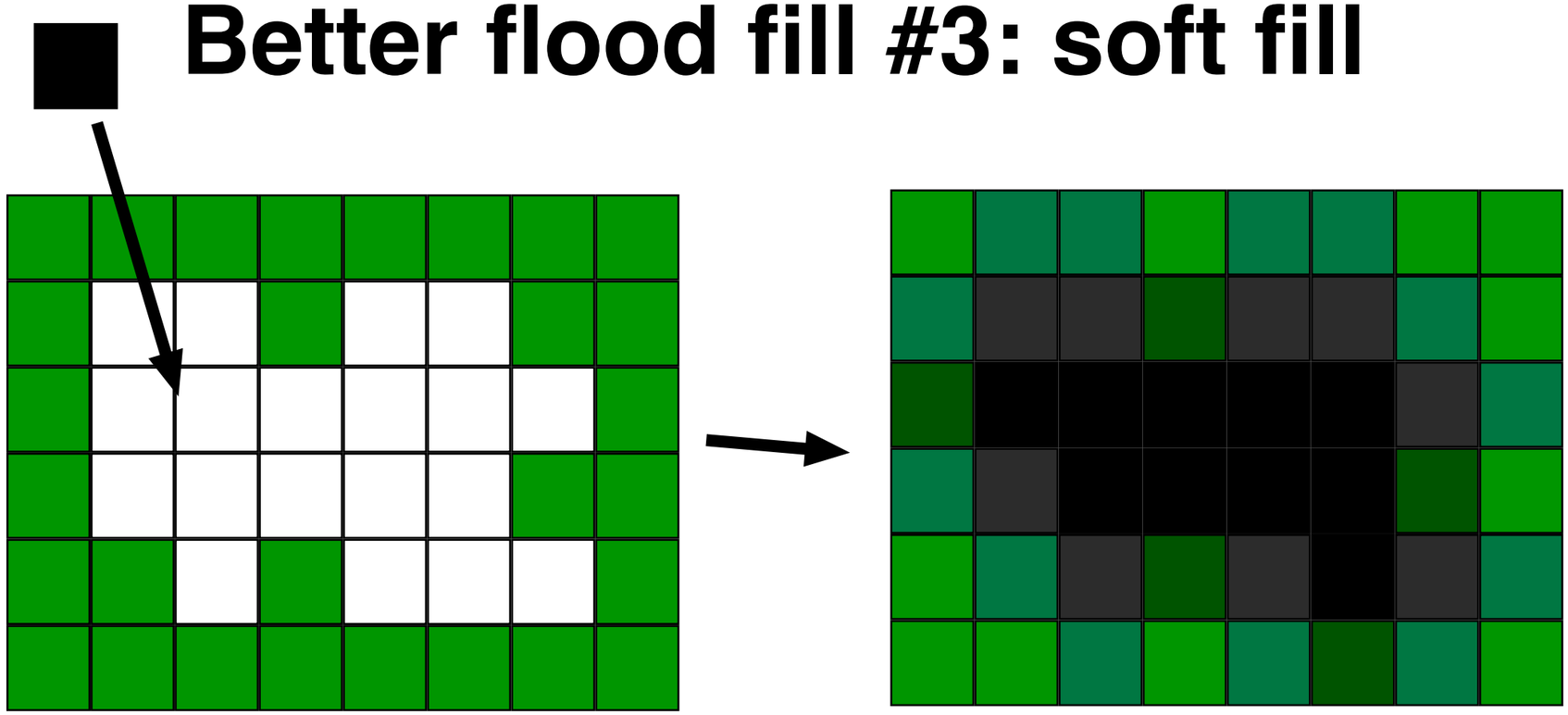
Fills same or similar colors

Better flood fill #2: calculate mask



Can fill into another image buffer!

Better flood fill #3: soft fill



Anti-aliasing effect at edges!

Conclusions about low-level algorithms

Not the most common ones to implement - but more common than you may think

Methods often applicable for other problems

Some 2D methods (like the inside-outside test) interesting in 3D generalizations