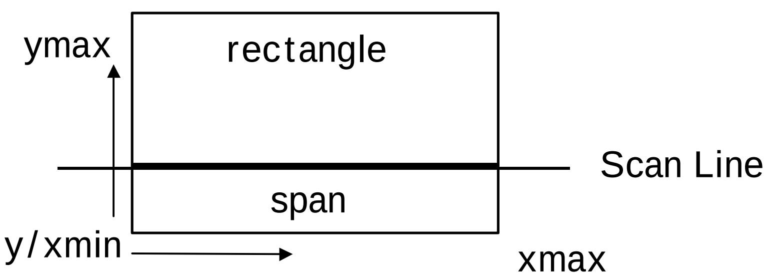
Filling Algorithms

- decide what pixels to fill
- decide what value to fill them (solid/pattern)

1. Primitives: rectangles/polygons
   - scan line algorithms (text sections 3.5-3.8)
2. Regions of pixels
   - fill algorithms (text section 19.5)

Filling Rectangles

- fill each span (segment of scan-line containing the rectangle) from x_{min} to x_{max} while traveling from y_{min} to y_{max} (reversing the order is trivial of course).



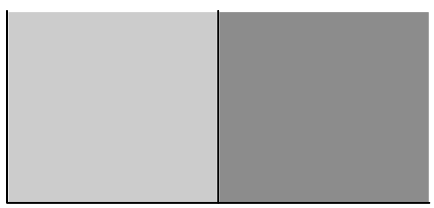
Span: a contiguous sequence of pixels on a scan line

Spans exhibit a primitive's COHERENCE, the degree to which parts of an environment or its projection exhibit local similarities.

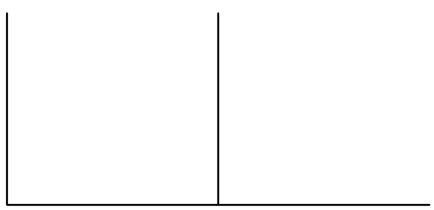
- Spatial coherence:
  primitives do not often change from pixel to pixel within a span or consecutive span lines (look only for pixels where change occurs, such as boundaries)
- Span coherence:
  primitives do not often change from span to span (ex, all pixels set to same value for solidly shaded polygon).
- Scan-line coherence:
  not much change between successive scan-lines (ex, consecutive scan-lines that intersect rectangle are identical).
- Edge coherence:
  edges of polygon intersect successive scan-lines (continuity of edges, will be useful later).
- Coherence greatly increases efficiency of scan-line algorithms (can output an entire span or scan-line rather than pixel by pixel).

- Problem: boundary pixels may be drawn several times for shared edges. (what colour should a shared edge be?)

- Partial solution, only draw “left” & “bottom” edges (skip right & top)

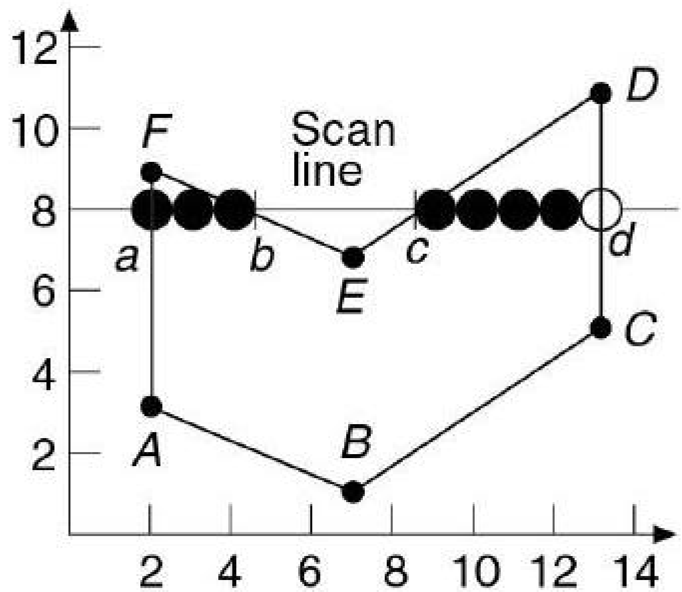


- Problem with this is that the left/bottom vertex still drawn twice, not so good for unfilled polygons (there is no perfect solution)

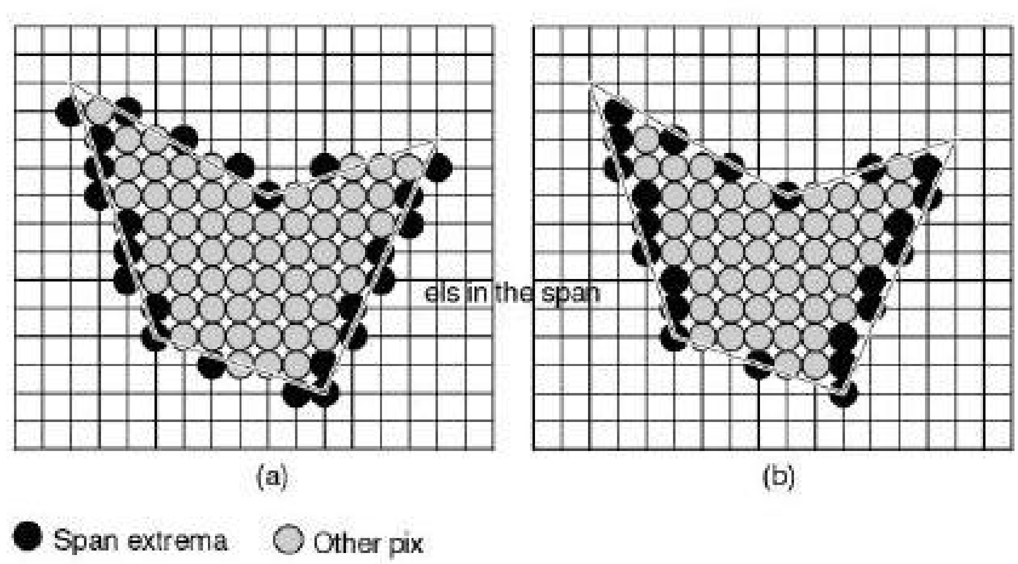


Filling Polygons

- Basic Idea: intersect the polygon with consecutive scan-lines and check for points of intersection (ie. Compute and fill the spans)



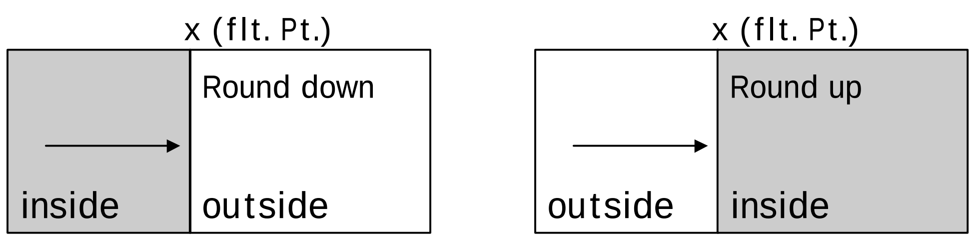
- Could determine span extrema (outermost pixels of a span), using midpoint algorithm, but watch out for extrema outside of polygon (want to fill the interior)



Incremental Algorithm

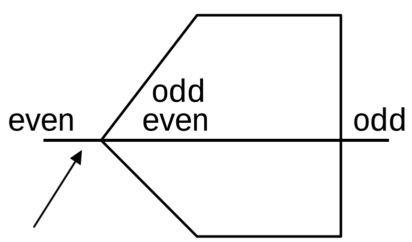
1. Find the intersection of the scan-line with all edges of the polygon.
2. Sort the intersections by increasing x.
3. Fill in all pixels between pairs of intersections that lie interior to the polygon. (Odd-parity rule: parity initially even, each intersection inverts the parity - draw when odd only).

   3.1 What is the interior pixel for a fractional x intersection?

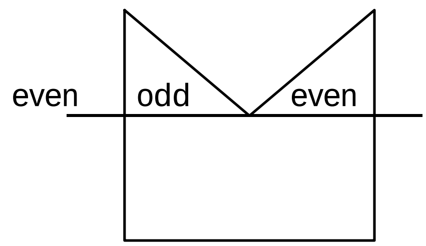


   3.2 Intersection at integer pixel coordinates? Leftmost extrema visible (interior), rightmost extrema exterior (not visible).

3.3 Intersection at vertices? Problem:

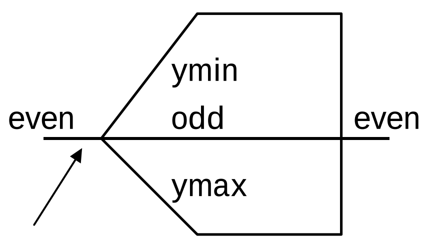


Intersects 2 edges
tf/ counts twice

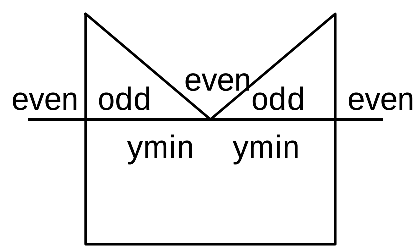


If count once, still
have problems...

solution, count y_{min} of edges bu not y_{max}.



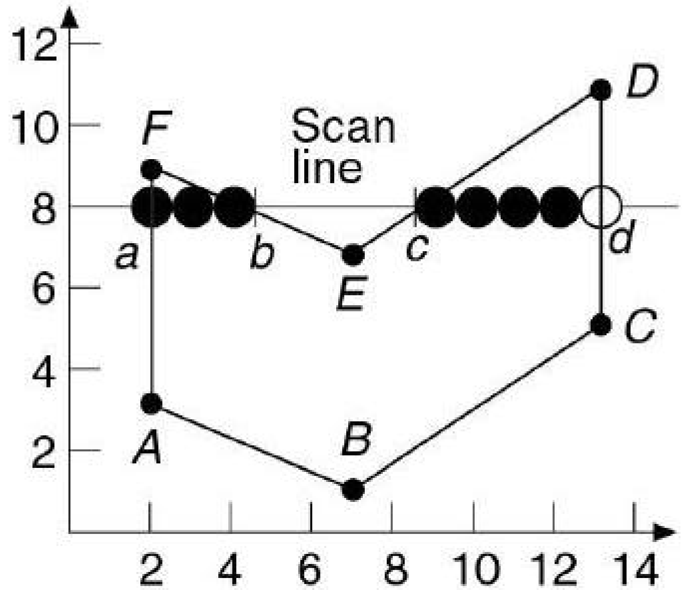
Count once for
ymin



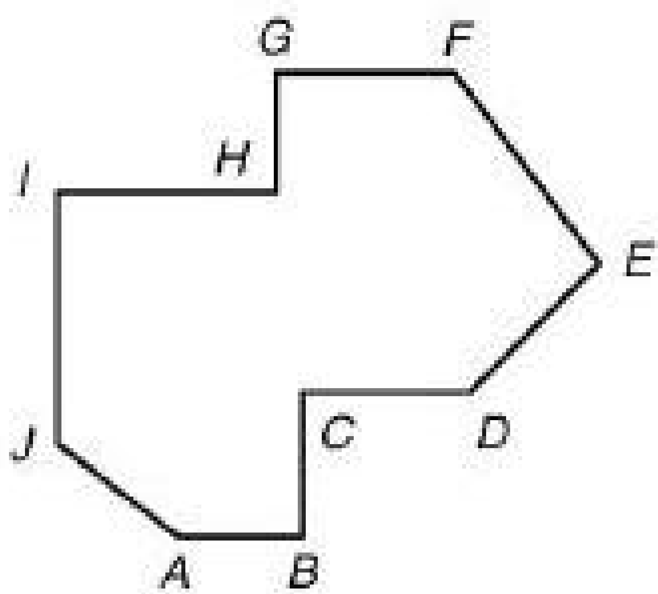
Counted 2x for
ymin of 2 edges

3.4 Horizontal Edges? Bottom edges drawn, top edges not. Since bottom edges will begin with a y_{min} they will be odd parity.

Examples – Figure 3.22

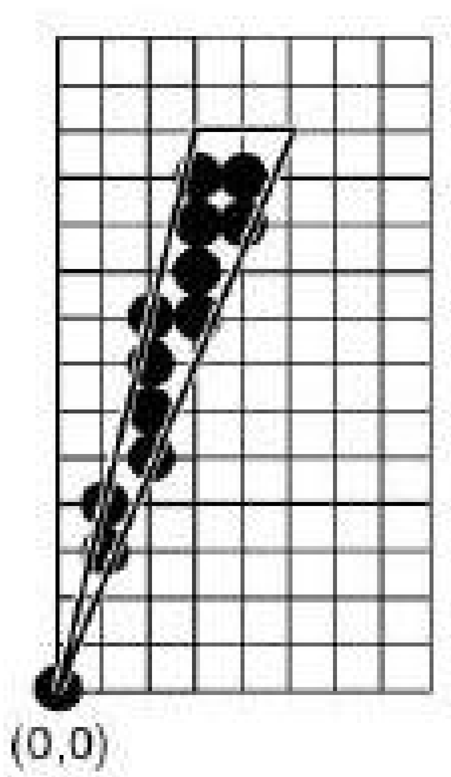


Horizontal Edges – Figure 3.24



Little problem with little (thin) polygons

- The edges lie so close together that the area does not contain a single pixel



Some pixels not drawn since not interior, left or bottom... GAPS!

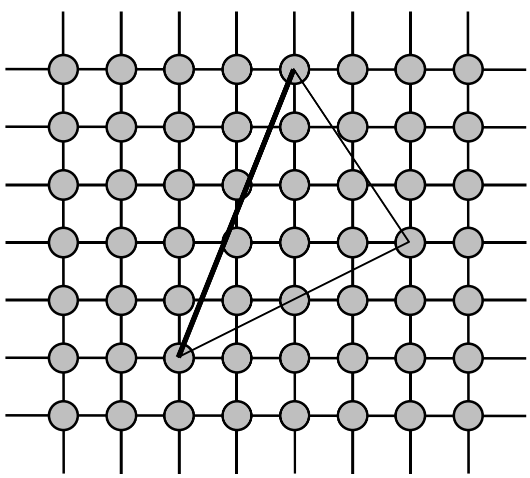
Scan-Line Algorithms

1. Find the intersections of the scan-line with all edges of the polygon.
   - Must be computed in a clever way, or can be SLOW.
   - Brute Force: test each polygon edge with each scan-line (brutally slow!)
   - Use edge coherence (many edges intersected by scan-line i are also intersected by scan-line i+1).
   - Can compute new x intersection with scan-line i+1 using old intersection with scan-line i. $x_{i+1} = x_i + \frac{1}{m}$ (remember midpt. line algorithm, but here stepping by 1 in y).

Edge Coherence Algorithm:

(slope > +1 that are left edges)

- Draw a pixel at endpoint (x_{min}, y_{min})
- As y is incremented, x will increment by 1/m where $m=(y_{max}-y_{min})/(x_{max}-x_{min})$
- x will have an integer and a fractional part
- As we iterate, the fractional part will overflow and the integer part will have to be incremented
- When fractional part is zero, draw the pixel at (x,y) that lies on the line. When fractional part is nonzero, round up (interior point)
- When fractional part becomes greater than 1, we increment x and subtract 1 from the fractional part



(2,1) – (4,6)
x_{min} = 2   m=5/2 1/m=2/5

y=1 x=2
y=2 x=2+2/5 → 3
y=3 x=2+2/5+2/5 → 3
y=4 x=2+4/5+2/5
=2+6/5
=3+1/5 → 4
y=5 x=3+1/5+2/5 → 4
y=6 x=3+3/5+2/5 → 4

Keeping Track of Edges of Interest to a Scan-Line

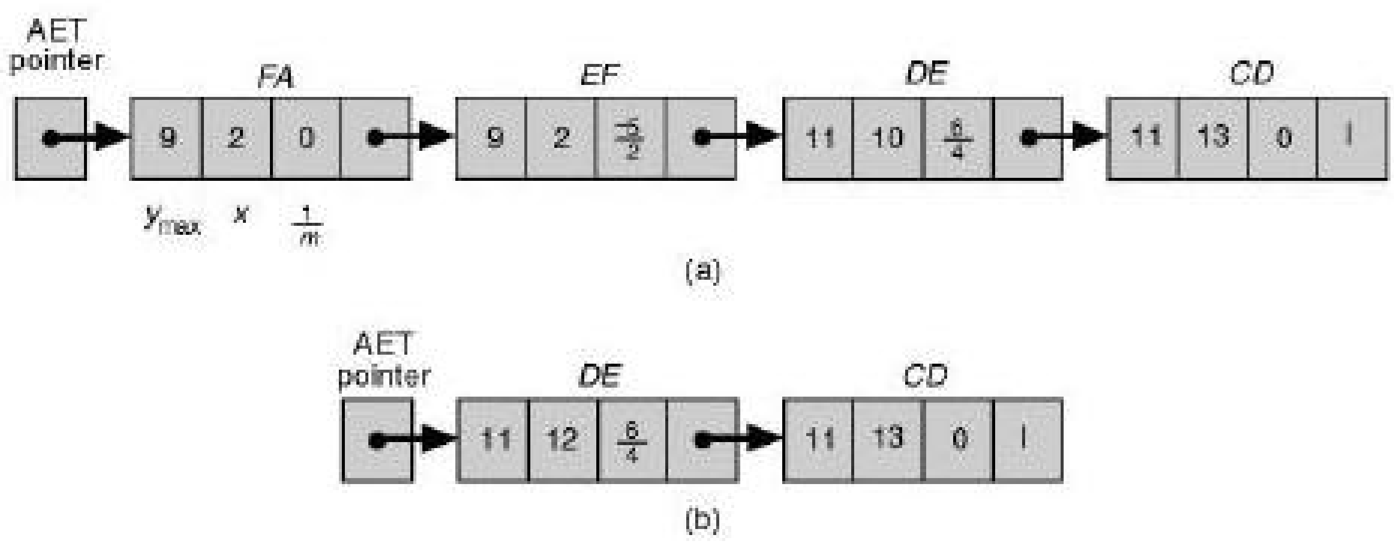
Active Edge Table (AET)

- set of edges (with intersection pts.) intersected by the current scan-line.
- sorted by x intersection values
- fill span of each pair of x intersection values
- updated for each scan-line (assume y+1)

  delete $\quad y_{max} < y+1 \; (y_{max} = y)$

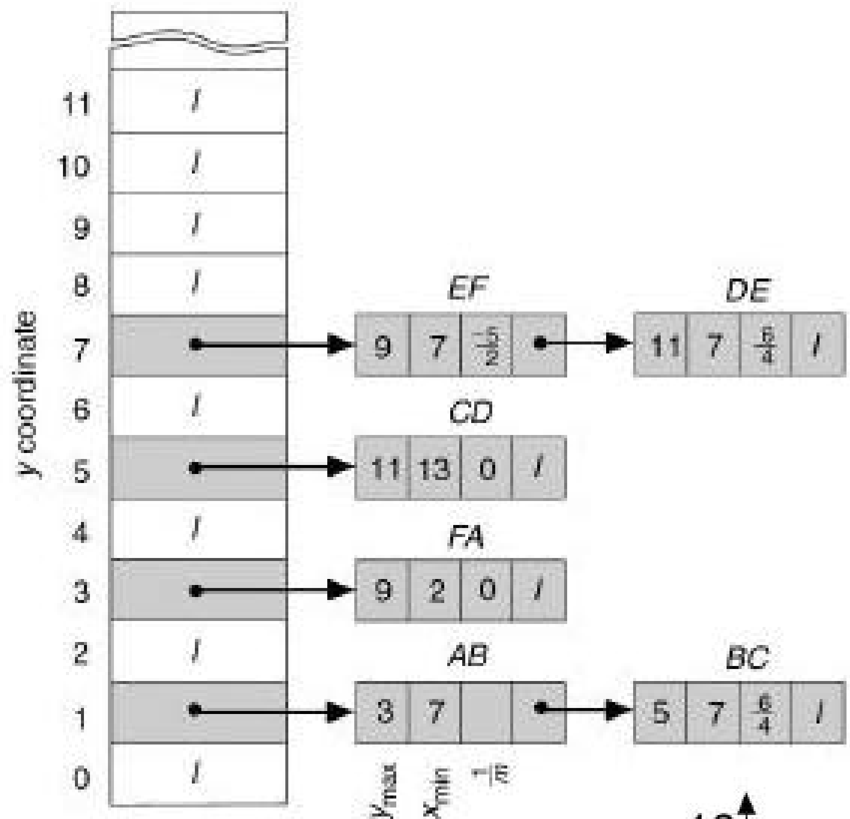
  add $\quad y_{min} = y+1$

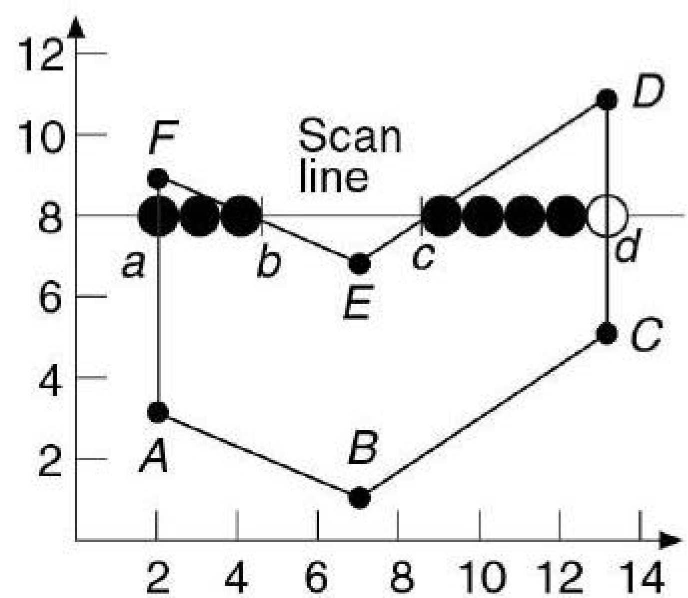
  compute $\quad$ new x intersection for edges in AET



Edge Table (ET)

- global table containing all edges sorted by decreasing y. (usually bucket-sorted: one bucket per scan-line)
- edges in a bucket sorted by increasing x.





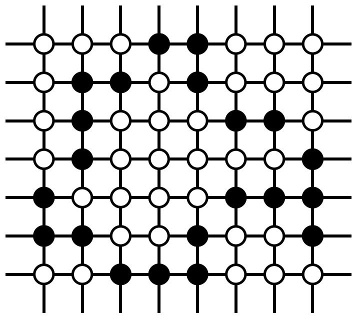
Scan-Line Algorithm

1. Set y to the smallest y coordinate that has an entry in the ET (ie. Y for first non-empty bucket).
2. Initialize AET to be empty
3. Repeat the following until both AET & ET are empty.
   - 3.1 Move edges from ET to AET if y_{min} = y, then sort AET on x (easier since ET presorted).
   - 3.2 Remove edges from the AET if y_{max} = y, then sort the AET on x
   - 3.3 Fill in pixels between x pairs in the AET
   - 3.4 Increment y by 1 (next scan line)
   - 3.5 Update x for new y for edges in AET

(also include flag for left or right edge)

Filling Regions of Pixels (text section 19.5)

- good for filling regions or non-self intersecting polygons (like flood fill, or paint can in Mac)
- region: collection of pixels



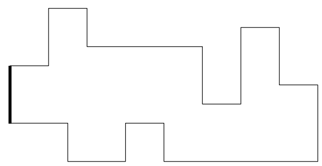
- interior defined regions: largest connected region of pts whose value is the same
- boundary defined regions: largest connected region of pts whose value are NOT some boundary value

Each algorithm can be divided into four components:

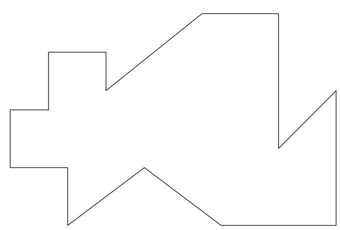
- propagation method (determine next point to be considered)
- start procedure (initialize algorithm)
- inside procedure (determines if a pixel should be filled)
- set procedure (changes the colour of a pixel)

Two Types of Regions

- 4-connected
  - pixels connected L, R, U, D



- 8-connected
  - pixels connected by L, R, U, D,
    UR, UL, DR, RL



Two definitions of pixel regions

- interior defined
  - all pixels inside the region have a given colour and no boundary pixels have this colour (can also have "holes" in it of a different colour)
- boundary defined:
  - the region is defined by a set of pixels of a boundary colour and no interior pixels have this colour (can also have interior "holes" which have the boundary colour)

NOTE

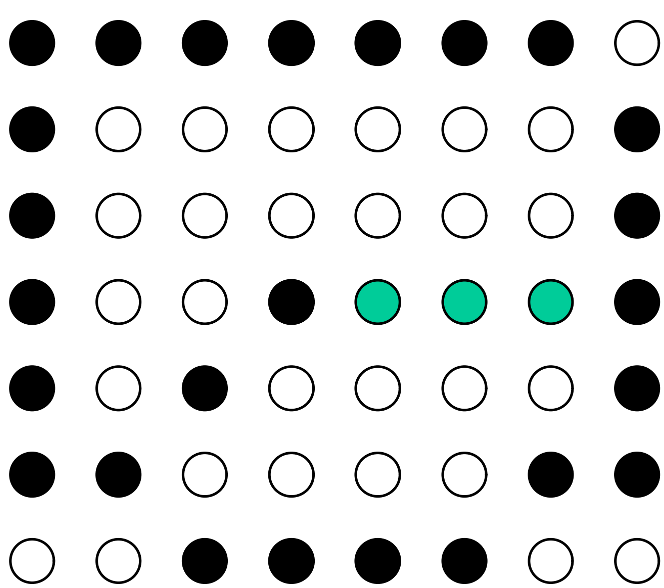
4 connected region

- interior define – 4 connected flood fill
- boundary defined – 4/8 connected boundary fill

8 connected region

- interior defined – 8 connected flood fill
- boundary define – 4 connected boundary fill

- Filling:
  - start with region (interior)
  - proceed in 4 directions (8) recrusively until
    - a) no more with same color (flood fill ≡ interior defined)
    - b) not hit boundary (boundary fill ≡ boundary-defined)



Algorithms

1/
```
Floodfill4(int x, int y, int old,
              int new){
if(pixel(x,y)==old){
    pixel(x,y) = new;
    FloodFill4(x, y-1, old, new);
    FloodFill4(x, y+1, old, new);
    FloodFill4(x-1, y, old, new);
    FloodFill4(x+1, y, old, new);
    }
}
```

2/
```
Boundaryfill4(int x, int y, int
                  bound, int new){
if((pixel(x,y)!=bound) &&
   (pixel(x,y)!=new)){
    pixel(x,y) = new;
    Boundaryfill4(x,y-1, bound, new);
    Boundaryfill4(x,y+1, bound, new);
    Boundaryfill4(x-1,y, bound, new);
    Boundaryfill4(x+1,y, bound, new);
    }
}
```

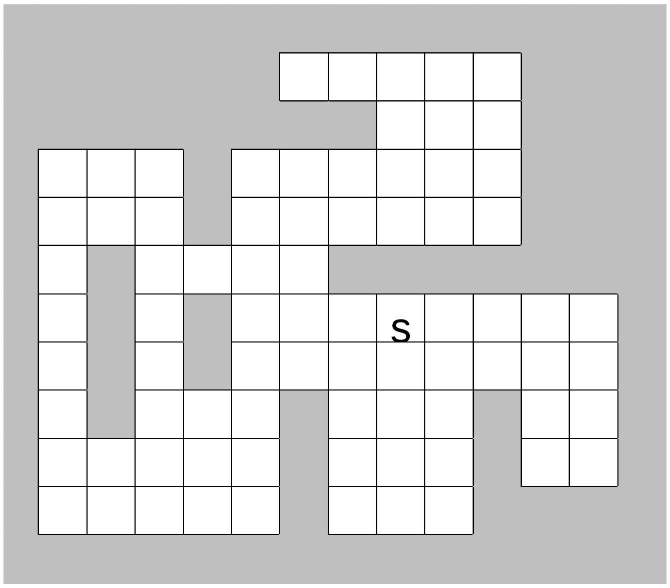
** highly recursive – stack can become very deep **

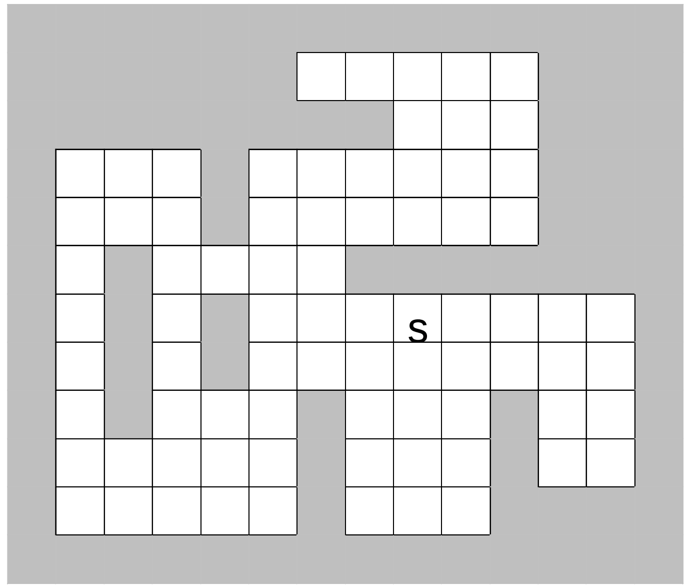
Span Filling: Region Coherence

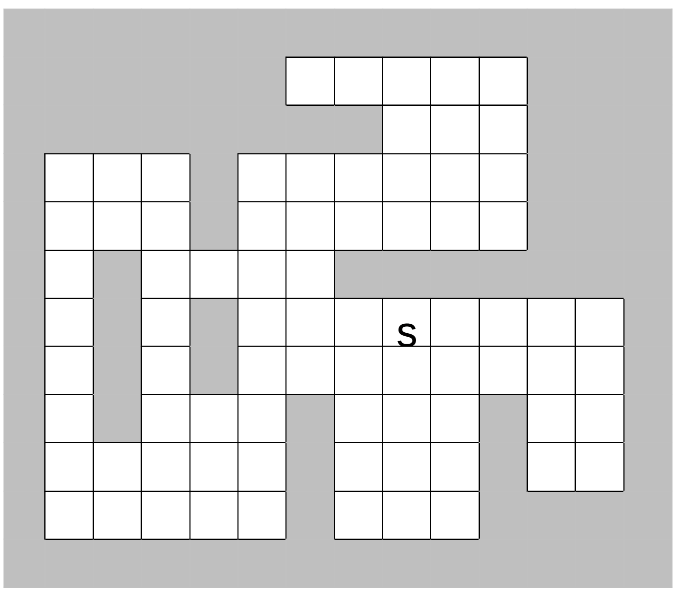
- efficiently fills in spans of pixels

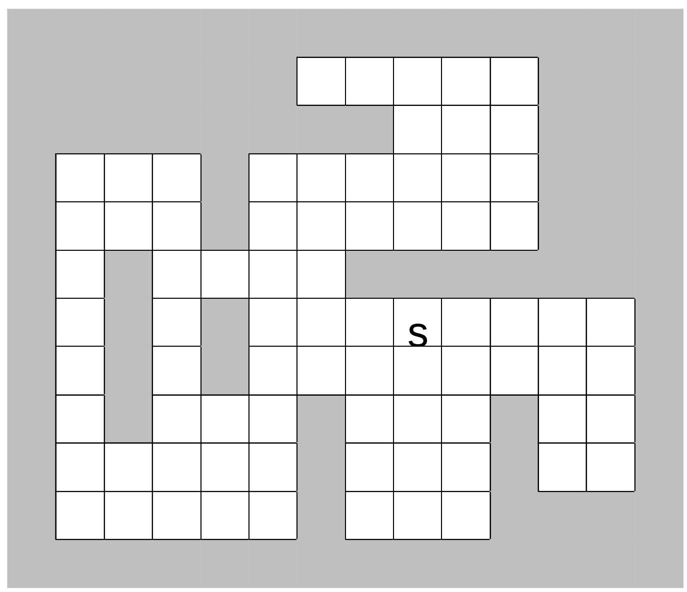
Algorithm:

- push seed pixel on stack
- while stack is not empty
  - pop stack to get next seed
  - fill in span defined by the seed
  - examine row above for spans reachable from this span and push the addresses of the rightmost pixels of each onto the stack
  - do the same for the row below the current span


S


S


S


S

Pattern Filling

- anchoring pattern to primitive

or

- pattern fills window and primitive “lets the pattern through”