CSD2341

Computer Graphics Programming




Area Fills

Third Edition

- Hearn and Baker 3.15
  - polygon classifications
  - Inside-outside tests
- Hearn and Baker 4.10-4.13

Fourth Edition

- Hearn, Baker & Carithers 4.7
  - polygon classifications
  - Inside-outside tests
- Hearn, Baker & Carithers 6.10-6.13

Math review: Polygons

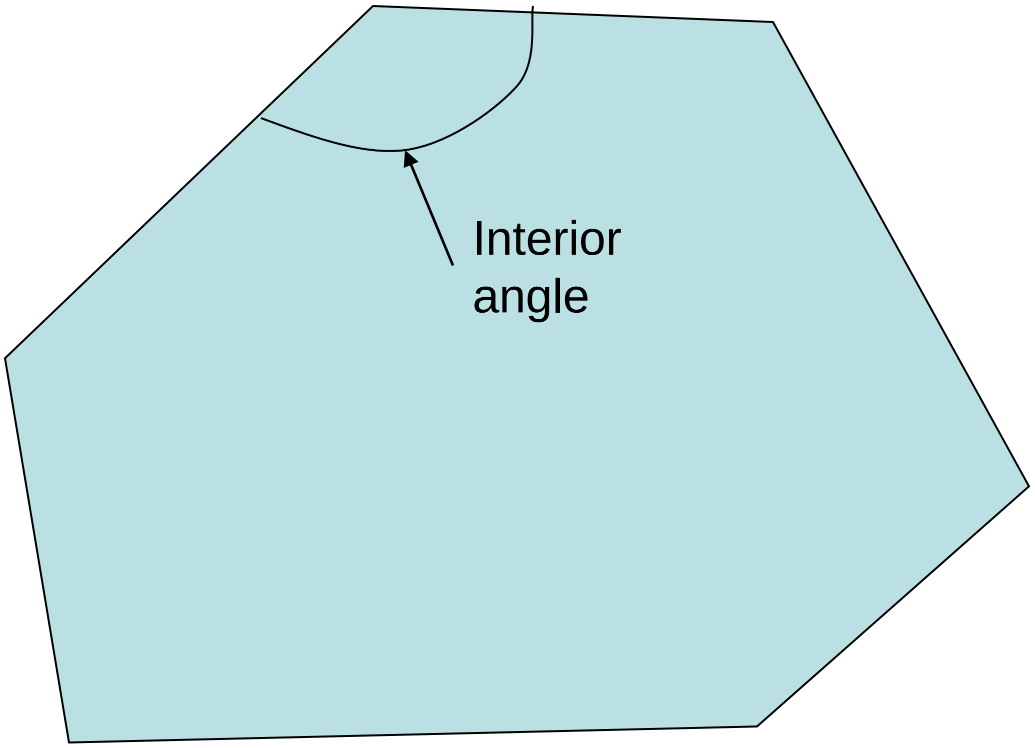
- A **polygon** is a shape defined by joining successive pairs of points (**vertices**) in a plane (including joining the last vertex to the first)
- Each successive pair of vertices is joined by a line segment – an **edge** or **side**.
- Your textbook calls this a **closed polyline**, and requires that a polygon does not intersect itself (but this is not always part of the definition).

Math review: Polygons

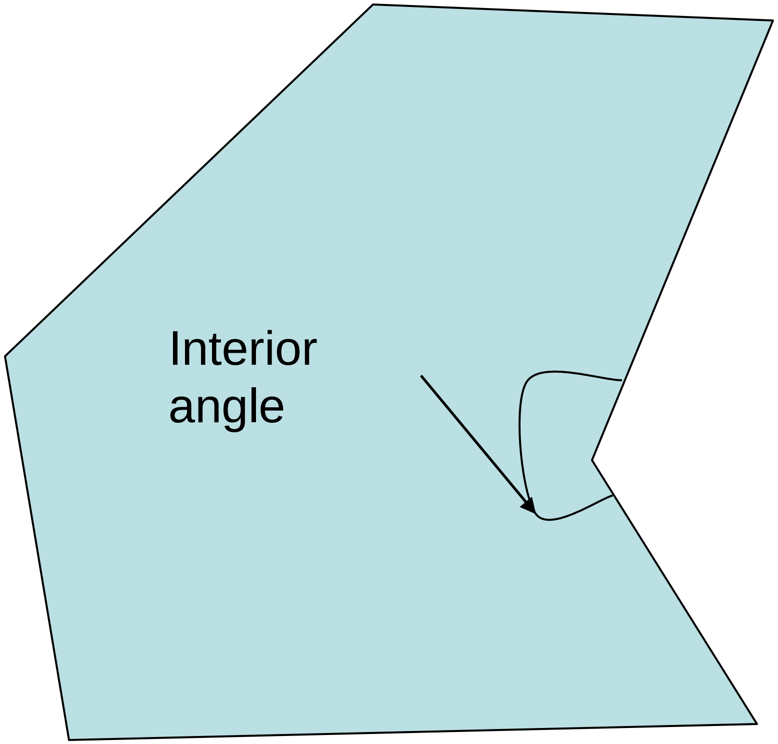
- Polygons divide the plane into three parts
  - The outside or **exterior**
  - The polygon itself
  - The inside or **interior**
- In graphics, we use polygons to define areas to be filled with a colour or pattern – we fill the interior
- In 3D graphics, we often use connected networks of polygons to describe the surface of an object.

Math review: Convex Polygons

- All interior angles < 180°



Math review: Concave Polygons



Splitting concave polygons

- Lots of graphics algorithms only work on convex polygons.
- Therefore, concave polygons need to be broken down into a number of convex ones.
- Sometimes, polygons are even split into triangles – triangles are very easy and fast to calculate with.
- Text book discusses methods for doing this – read it, but this part is not examined.

Inside/outside tests for polygons

- Some graphics algorithms need a way to tell whether a point is inside or outside the polygon
- This needs to work for polygons with self-intersections too.
- There are a couple of different methods to determine this:

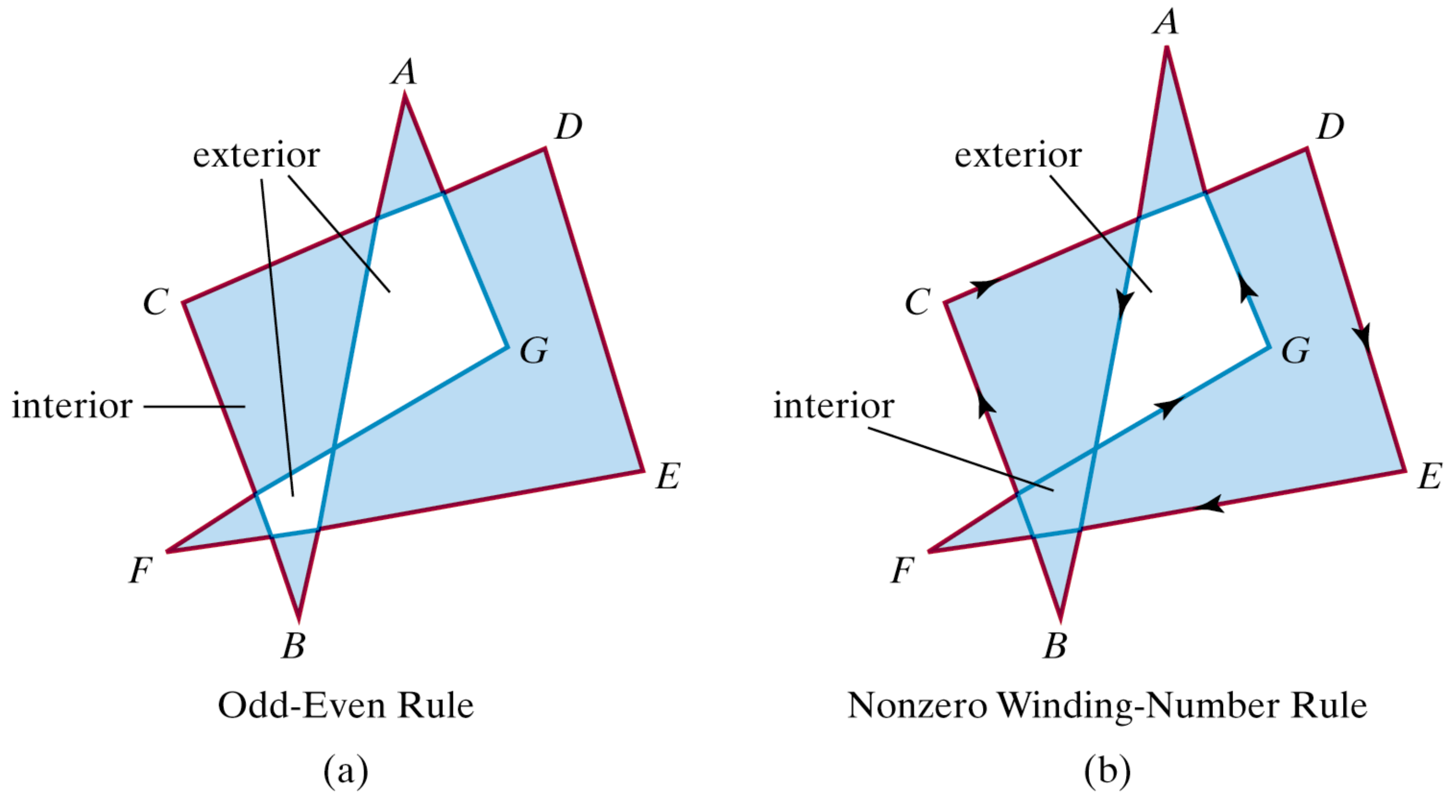


Figure 3-46

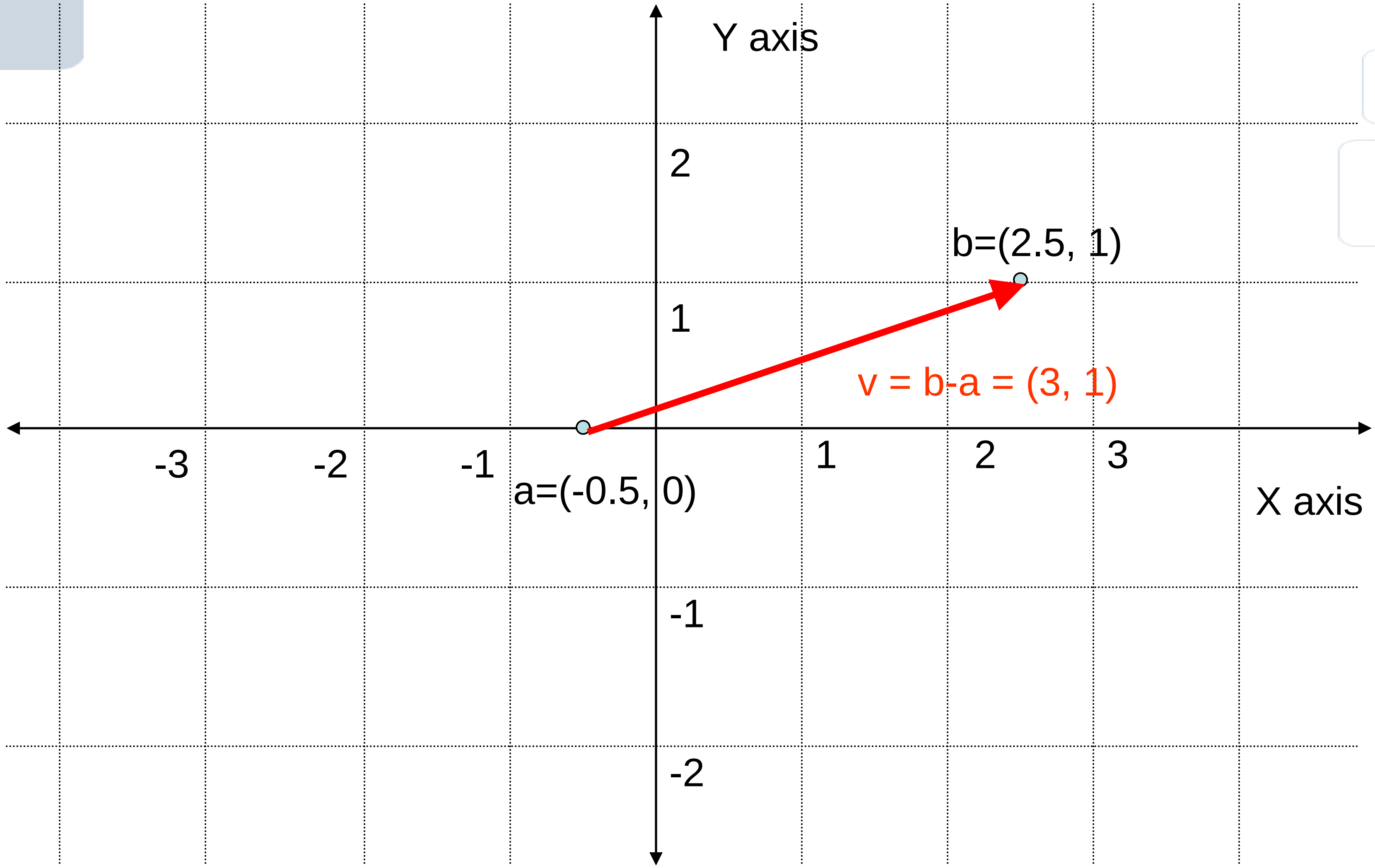
Identifying interior and exterior regions of a closed polyline that contains self-intersecting segments.

Working out directions for the winding number method

- We need to work out which direction each edge crosses the reference vector
- Two methods:
  - Using vector cross product
  - Using vector dot product

Math review: vectors

- A vector is the difference between two points
- Think of it as an arrow starting at one point and ending at the other.
- See Appendix A-2



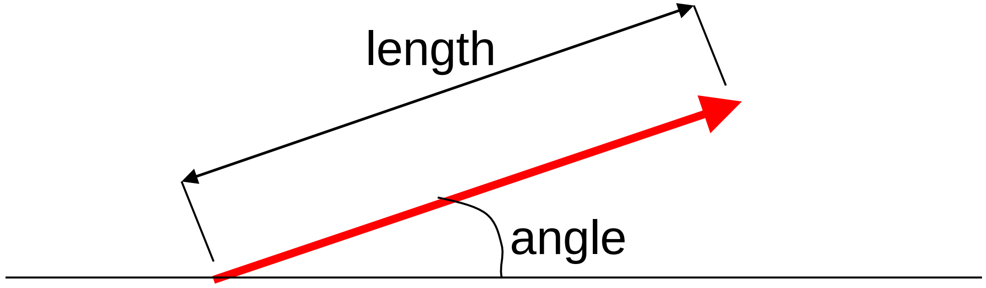
Y axis
2
b=(2.5, 1)
1
v = b-a = (3, 1)
-3
-2
-1
1
2
3
a=(-0.5, 0)
X axis
-1
-2

Vector calculations

- Vectors have coordinates e.g. (3, 1) means 3 units in the x-direction and 1 unit in the y-direction
- To calculate coordinates, just subtract the corresponding coordinates of the two end points.
- $(2.5, 1) - (-0.5, 0) = (2.5 - (-0.5), 1 - 0) = (3, 1)$

Vector calculations

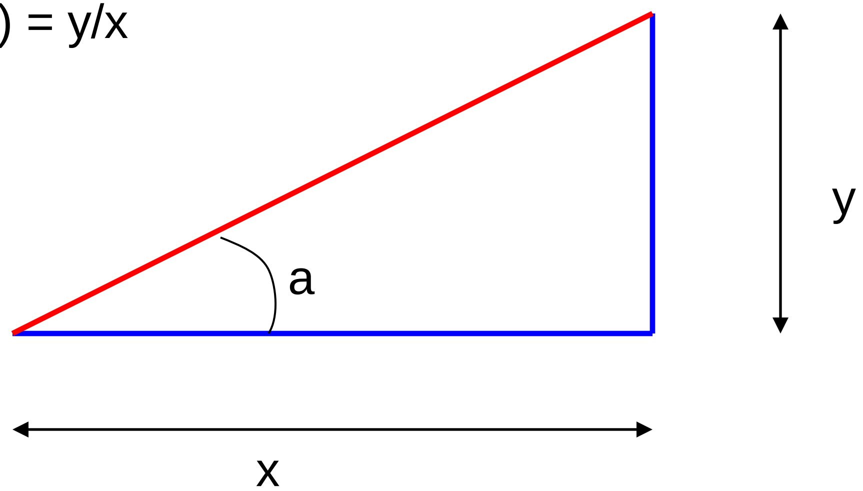
- Vectors have a direction and a length
- If v = (x, y)
  - Direction: angle = arctan(y/x) (in radians)
  - Length = sqrt(x*x + y*y)



arctan?

- What's arctan? First, remember what tan (short for tangent) is:

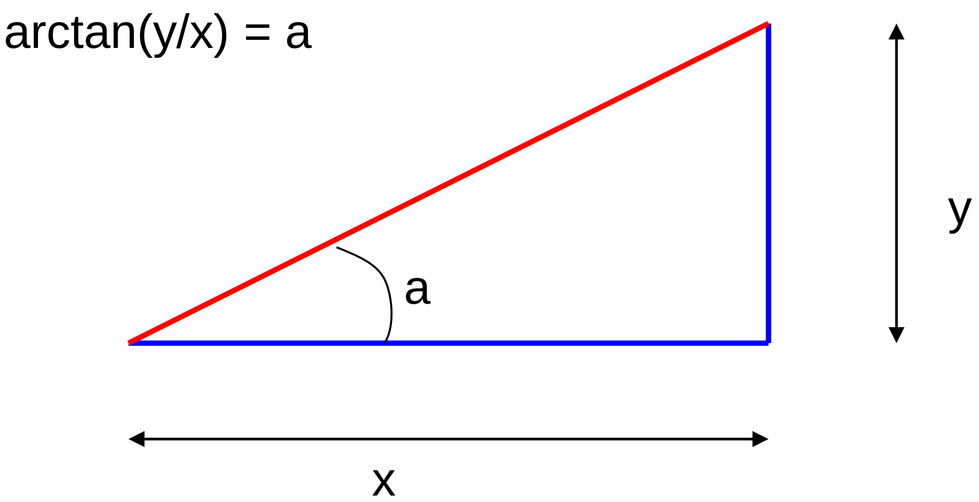
$\tan(a) = y/x$



- That is: the tan of the angle "a" is y/x, i.e. the slope of the red line.

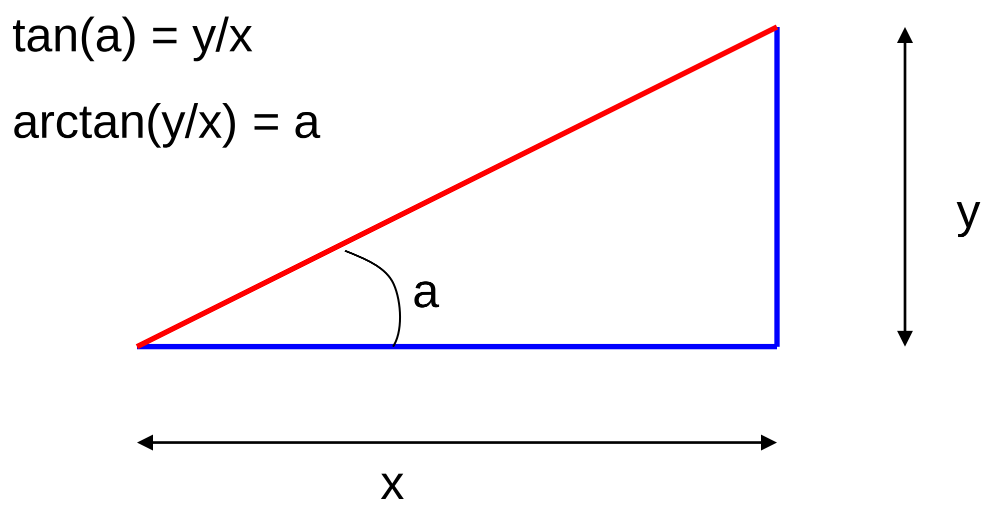
arctan?

- Another way of saying this is



- That is: the angle that matches the slope of the red line (has a tan of y/x) is “a”.

arctan?



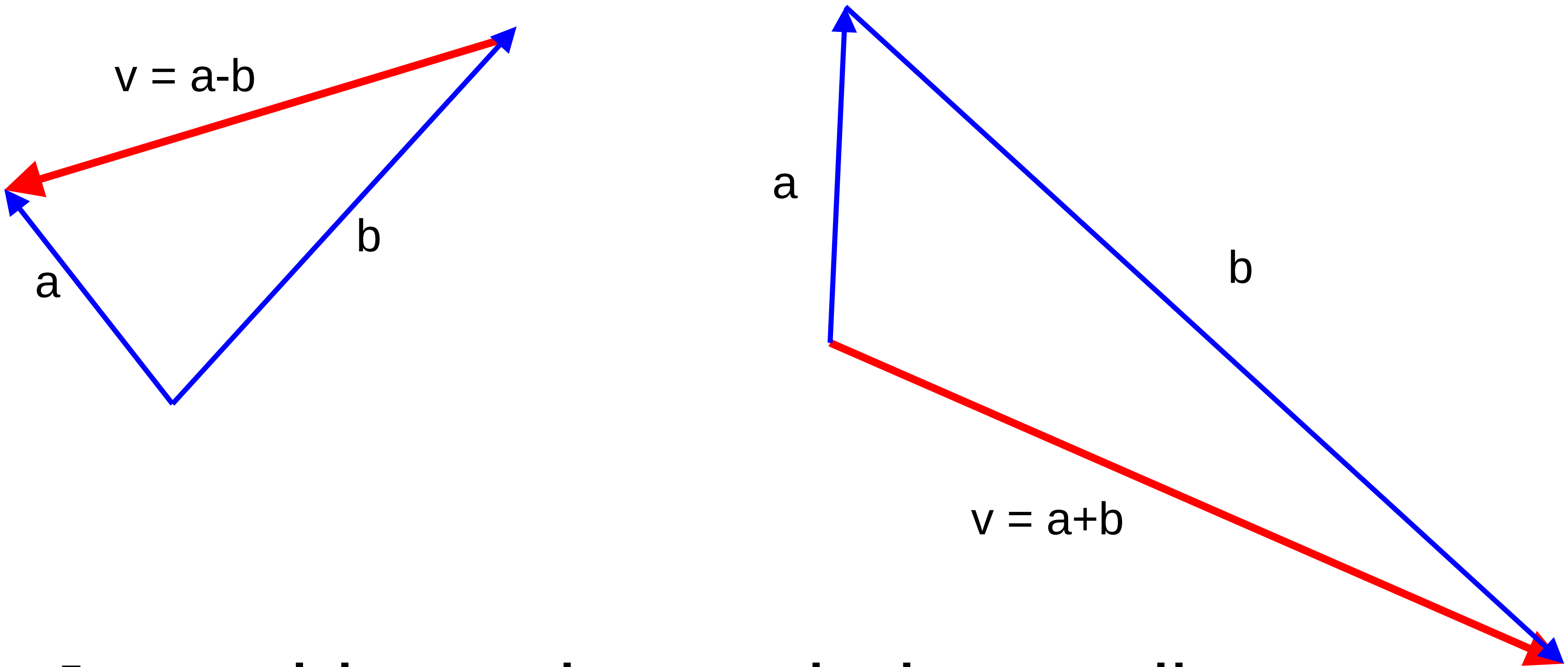
- Note that in these equations, a is measured in **radians**
- Radians can be converted to degrees:
  - – Degrees = 180*radians/π
  - – Where π is about 3.1416…

arctan in Java

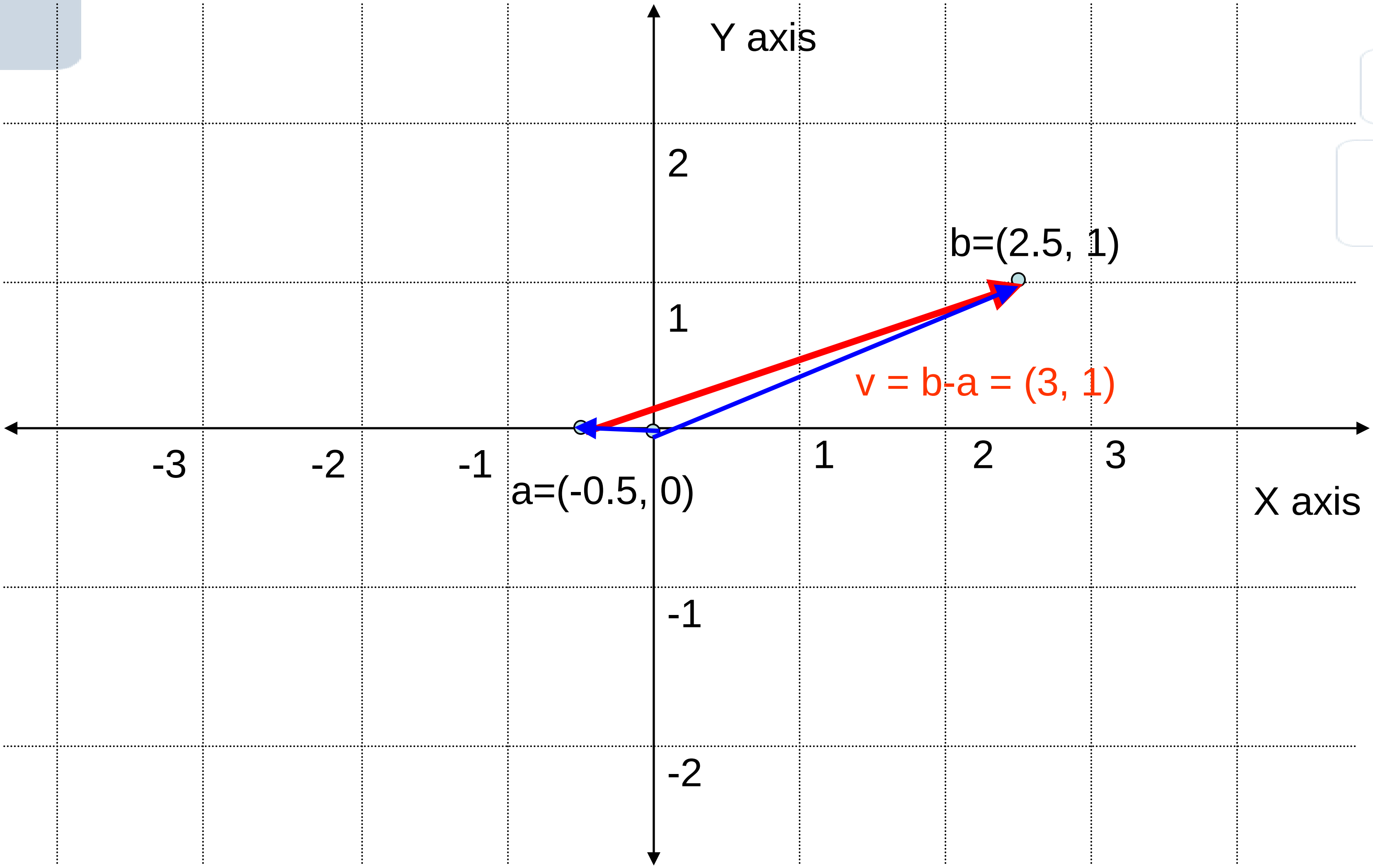
- To convert radians to degrees:
  - Math.radiansToDegrees(x)
- To convert degrees to radians:
  - Math.degreesToRadians(x)
- To get π:
  - Math.PI
- To calculate tan:
  - Math.tan(x) – if x is in radians
  - Math.tan(Math.degreesToRadians(x)) – if x is in degrees
- To calculate arctan:
  - Math.atan2(y, x) – the result will be in radians

Vector calculations

- Vectors can be added and subtracted



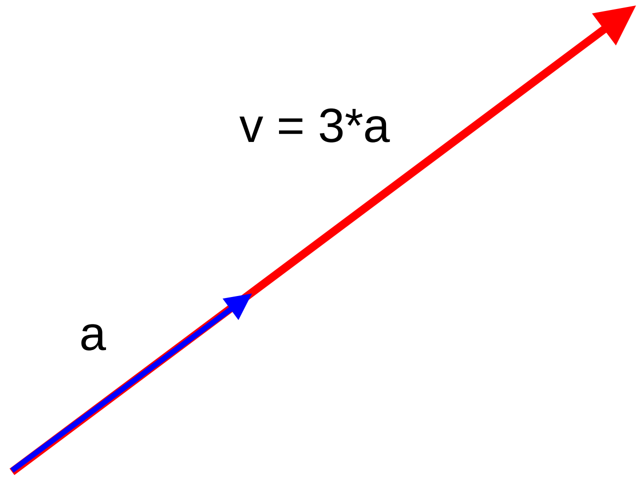
- Just add or subtract their coordinates



Y axis
2
b=(2.5, 1)
1
v = b-a = (3, 1)
-3
-2
-1
1
2
3
a=(-0.5, 0)
X axis
-1
-2

Vector calculations

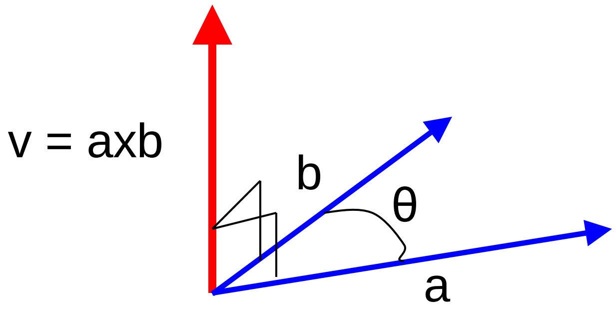
- A vector can be scaled



- Just multiply its coordinates by the scalar

Vector cross product

- Can be applied to two vectors in a plane
- The result is a vector pointing at right angles into or out of the plane (so it is a 3D vector)



- Formula
  - $v = a \times b = u * |a| * |b| * \sin(\theta)$, where u is a unit vector whose direction is determined by the right-hand rule
  - $V = (a_2 * b_3 - a_3 * b_2,\ a_3 * b_1 - a_1 * b_3, a_1 * b_2 - a_2 * b_1)$

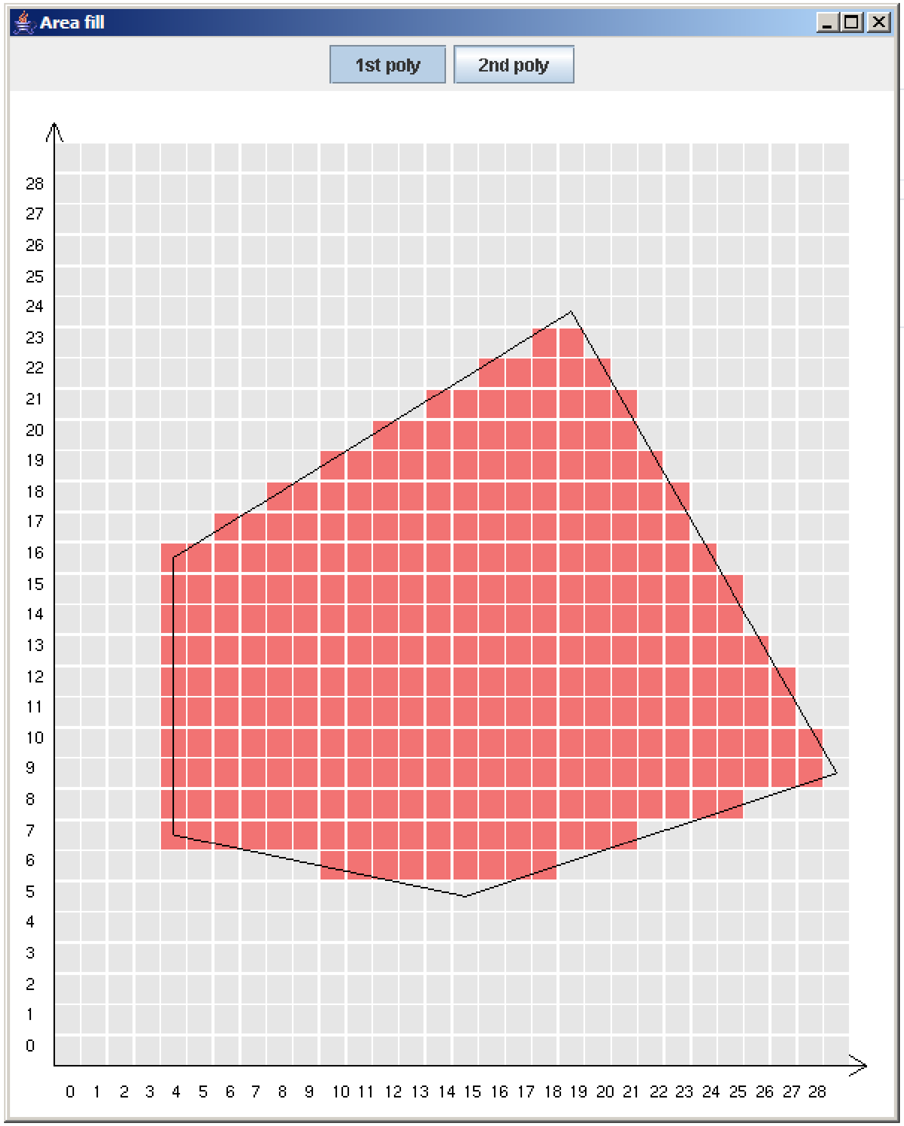
Vector scalar product

- Also called dot product, inner product
- Formula
  - $a \cdot b = |a| * |b| * \cos(\theta)$
  - $a \cdot b = a_1 * b_1 + a_2 * b_2 + a_3 * b_3$

- Now back to polygons….

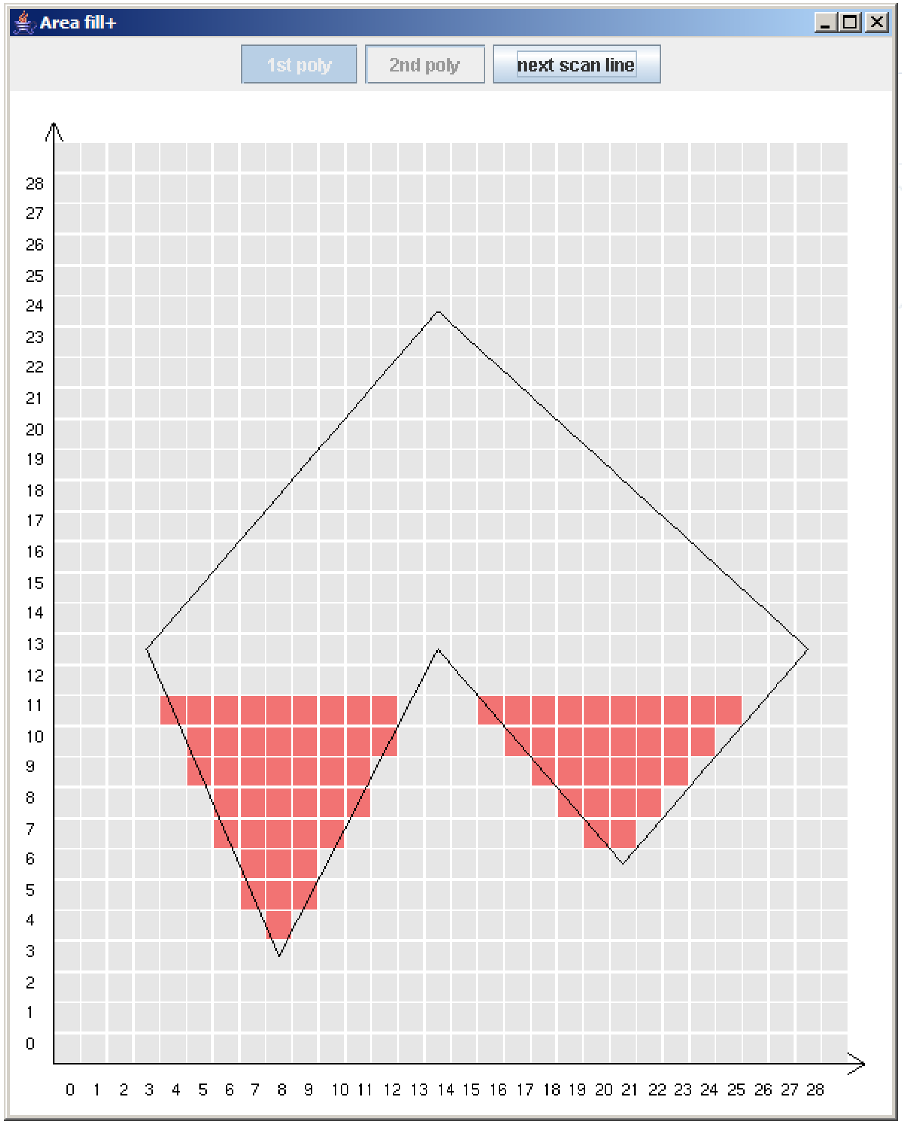
Polygon scan-conversion

- Say we want to “colour in” a polygon on a raster display
- Which pixels should we set?
- E.g. set pixel if the pixel centre is in the interior of the polygon (be careful about edges)
- We could do an inside-outside test for each pixel in the raster – this would be very inefficient.
- Instead….



Scan-line polygon fill algorithm

- Works by counting intersections of scan-lines with polygon edges
- Pixels between pairs of intersections are filled
- Starts at bottom and works up one scan line at a time
- Uses incremental calculations for efficiency (coherence)



Scan-line polygon fill algorithm

- Tricky points
  - Scan line passes through vertex
    -Depends whether edges change direction
  - Horizontal edges
    -Depends if on top or bottom
  - Vertical edges
    -Depends if on left or on right
- Explore these using AreaFill.java
  - Try different cases
  - Examine scanConvert() method
- Work through simple example
  - Sorted edge table
  - Active edge list

Pseudo-code for creating edge table

Initialise sortedEdgeTable (entry for each scan-line should be empty)

For each edge of the polygon

if slope != 0

Set edge = upwards copy of edge

Set edge = shorten the edge by 1 scan-line

Set index = scan-line for start of edge

add edge to sortedEdgeTable[index], keeping sorted by starting x coordinate

Contents of sorted edge table

- Each entry contains
  - y coordinate of top of edge
  - x coordinate of start of edge
  - 1/m where m is the slope
- Update:
  - Set x = x + 1/m

Pseudo-code for scan-line algorithm

```
Create sortedEdgeTable (see previous slide)
Set row = scan line at bottom of polygon
Set activeEdgeList = sortedEdgeTable[row]
While activeEdgeList not empty
   Set inside = true
   for each edge in activeEdgeList
            if inside then
                        start = round up x coord of edge
            else
                        end = round down x coord of edge
                        fill between start and end
            Set inside = ! inside
   increment row
   remove edges that finished on the last row
   update x values for all edges in activeEdgeList
   add sortedEdgeTable[row] into activeEdgeList
   make sure activeEdgeList sorted by x coordinate
```

Filling with a pattern

- This can be done using a bit pattern array or texture pattern array
- If the array is n_x by n_y in size
  - Fill the pixel at (x, y) using the value at index (x mod n_x, y mod n_y)
  - Easy to incorporate into the scan-conversion algorithm

Gradient fill

- This is where the colour of pixels in the filled polygon varies in a smooth way.
- Can be done by associating a colour with each vertex of the polygon
  - Along an edge, interpolating the colours of the vertices at each end
  - Between edges, interpolate the colours two edges
- Can be incorporated into the scan-conversion algorithm and computed incrementally

Anti-aliasing

- Similar to the line drawing case – problems occur on edges of polygons
- Some possible anti-aliasing methods:
  - Pixel phasing (can be used for lines too)
  - Estimate how much of a pixel is inside the polygon
    - -Calculate the area of the intersection between polygon and pixel (very expensive)
    - -Supersampling – use sub-pixels
    - -Pitteway and Watkinson – similar to Bresenham's algorithm – use along polygon edges

Polygons tables etc

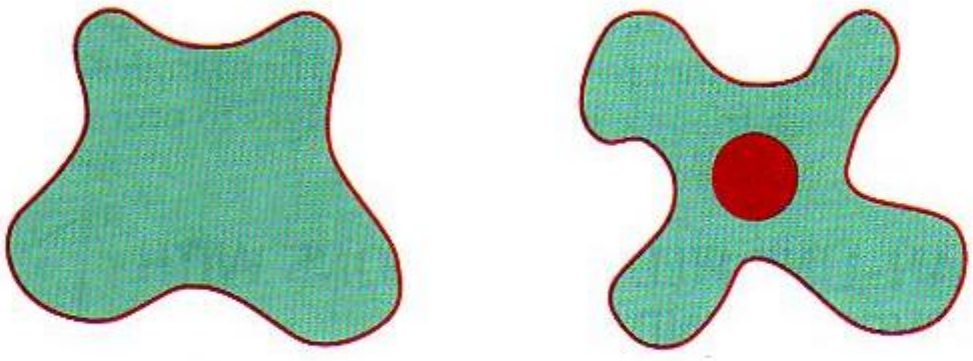
- The rest of 3.15 (4.7, 4th ed.) is to do with polygons in 3D
- Take a look, but it is not examined.

Region filling

- Scan-conversion good for vector-based applications
- For bitmapped applications (e.g. a paint package), use
  - Flood fill
  - Boundary fill

Boundary fill

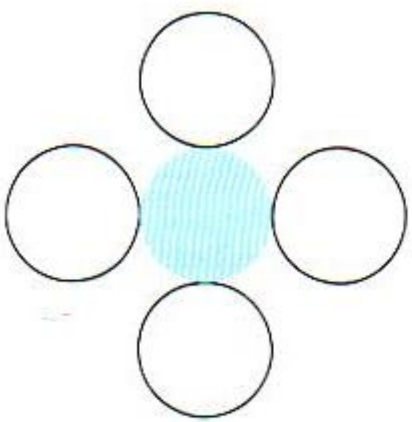
- Fill an area until a boundary of a particular colour is hit
- Start inside and search neighbouring pixels to fill until boundary is reached

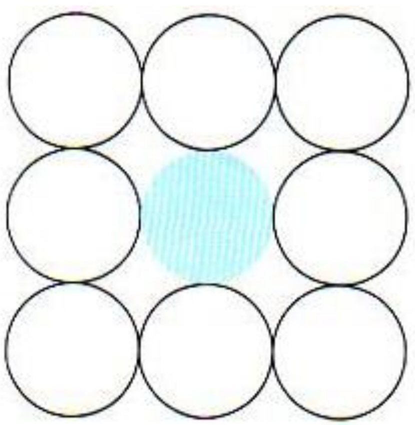


(a) (b)

Boundary fill neighbours

- Two possible definitions:
  - 4-connected
  - 8-connected
- Look at some examples with BoundaryFill program


(a)


(b)

Boundary Fill Algorithm

- From your text – recursive version in C++

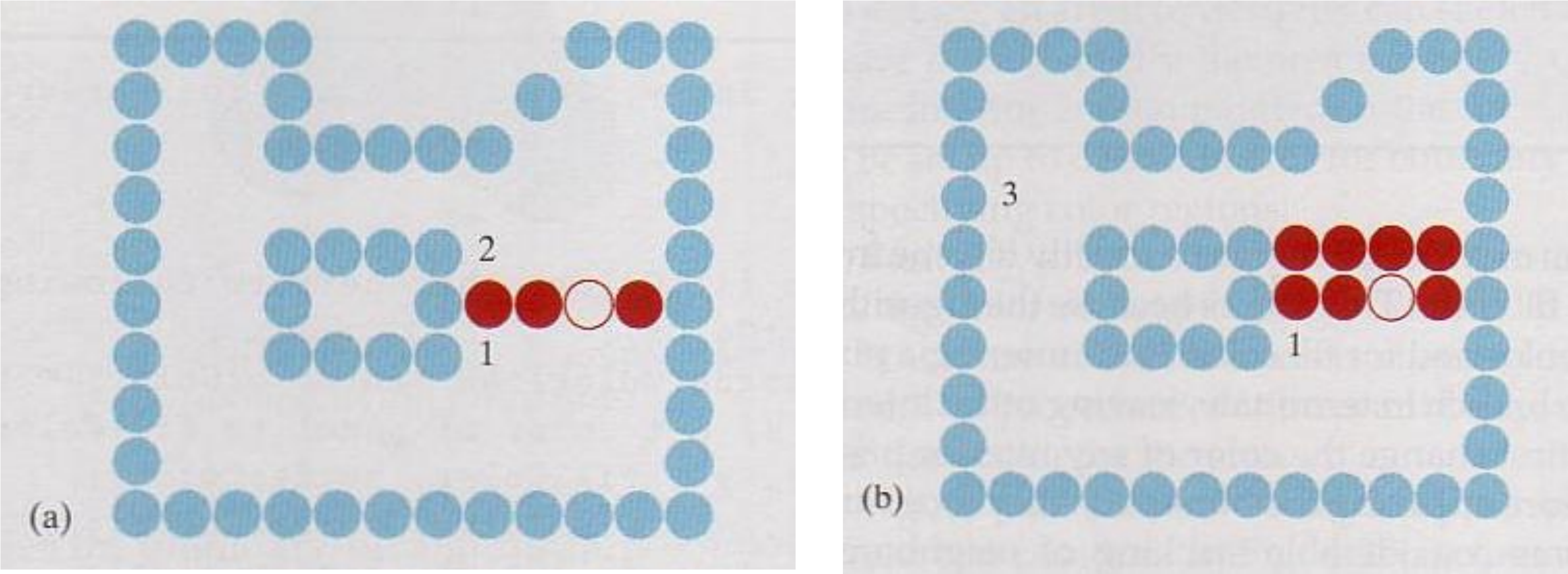
```
void boundaryFill4 (int x, int y, int fillColor, int borderColor)
{
   int interiorColor;

   /* Set current color to fillColor, then perform following oprations. */
   getPixel (x, y, interiorColor);
   if ((interiorColor != borderColor) && (interiorColor != fillColor)) {
      setPixel (x, y);    // Set color of pixel to fillColor.
      boundaryFill4 (x + 1, y , fillColor, borderColor);
      boundaryFill4 (x - 1, y , fillColor, borderColor);
      boundaryFill4 (x , y + 1, fillColor, borderColor);
      boundaryFill4 (x , y - 1, fillColor, borderColor)
   }
}
```

Boundary Fill problems

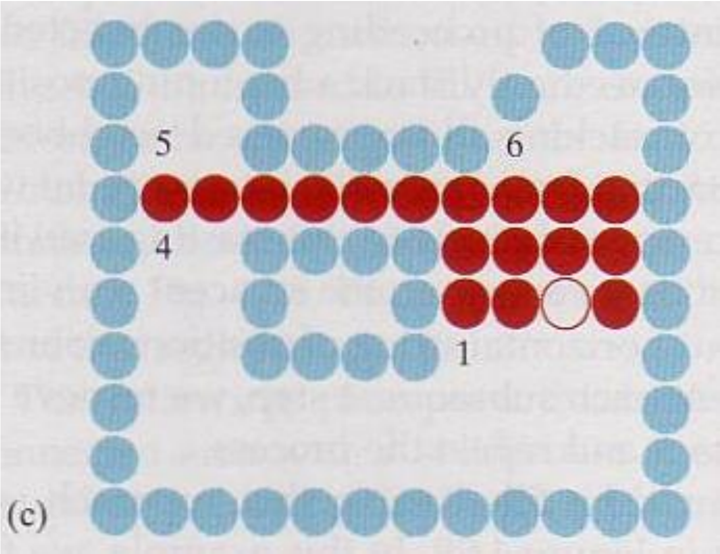
- Errors occur if any interior pixels already are same colour as fill colour
- This method uses lots of memory so you quickly run out of memory
- Although most texts give this as the standard algorithm, more efficient methods need to be employed
- Think about other ways that you could do it !

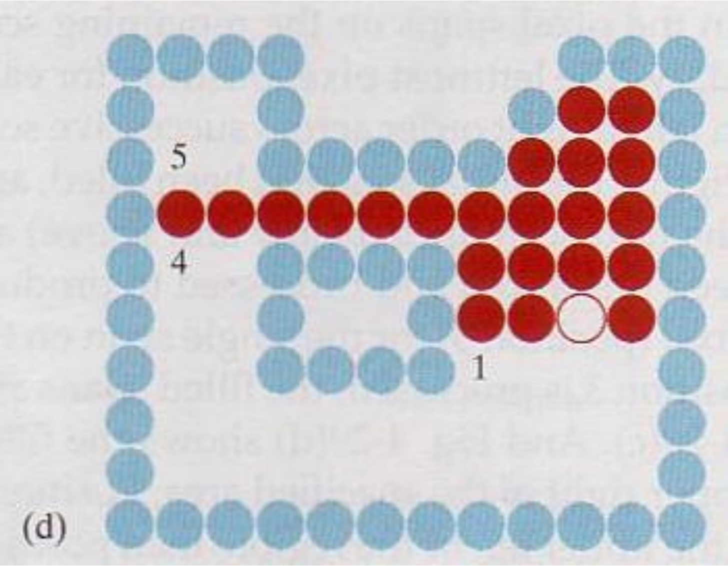
Better boundary fill

Work with "spans" of pixels to fill



Better boundary fill





Flood fill

- Start with a “seed” pixel and change its colour
- Recursively change neighbouring pixels that also have the seed colour
- Keep going until neighbours with different colours reached

Flood fill

- Similar to Boundary Fill algorithm
- 4-connected or 8-connected search methods
- Recursive – same problem with memory as Boundary Fill
- Again, most texts give this approach as the standard method
- Need to use more efficient methods to avoid memory problems
- Again, think of solution similar to boundary filling

Flood Fill algorithm

```
/* Set current color to fillColor, then perform following operations. */
getPixel (x, y, color);
if (color = interiorColor) {
   setPixel (x, y);     // Set color of pixel to fillColor.
   floodFill4 (x + 1, y, fillColor, interiorColor);
   floodFill4 (x - 1, y, fillColor, interiorColor);
   floodFill4 (x, y + 1, fillColor, interiorColor);
   floodFill4 (x, y - 1, fillColor, interiorColor)
```