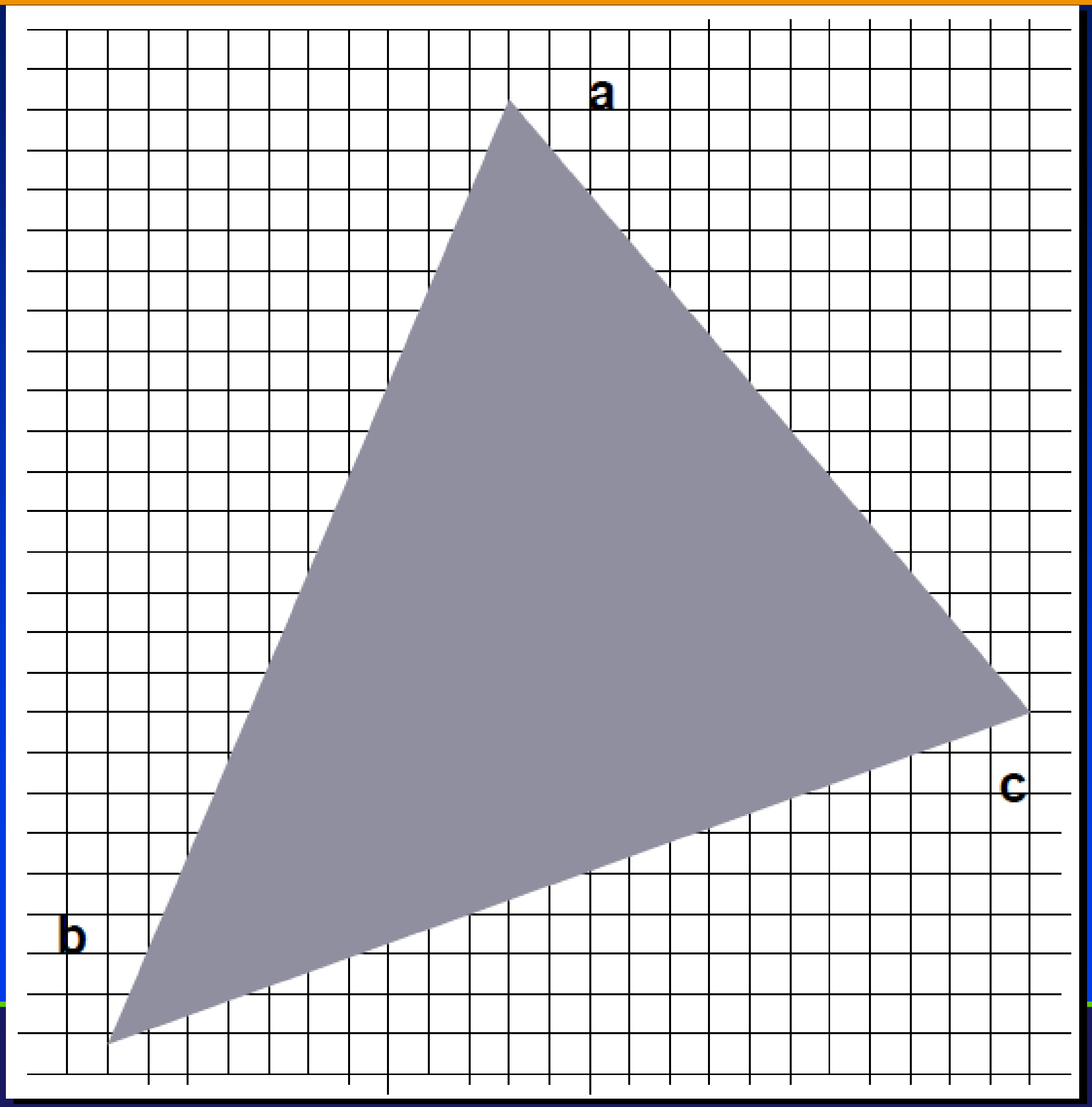
CSE328 Fundamentals of Computer Graphics: Theory, Algorithms, and Applications

Hong Qin
Department of Computer Science
Stony Brook University (SUNY at Stony Brook)
Stony Brook, New York 11794-2424
Tel: (631)632-8450; Fax: (631)632-8334
qin@cs.stonybrook.edu
http://www.cs.stonybrook.edu/~qin

Scan Conversion

- The earlier task allows us to draw line segments, polylines, curves, is it sufficient for 2D graphics?
- What are still missing for the rasterization task?
- Wireframe geometry and display is NOT enough
- We must have drawing routines to support the solid-shaded appearance (not only boundaries, but also all interior points of polygons)
- Scan conversion is achieving such goal

Scan Conversion



Simple Algorithms

- We start from a simple triangle T: a = (x1,y1), b = (x2,y2), and c = (x3,y3)
- The task is to find all pixels inside T
- Naïve algorithm (the worst algorithm)
  - For each pixel p do
  - If p is inside T, then draw-point(p) end if
  - End for
- For a single triangle, we MUST traverse all pixels, the worst performance

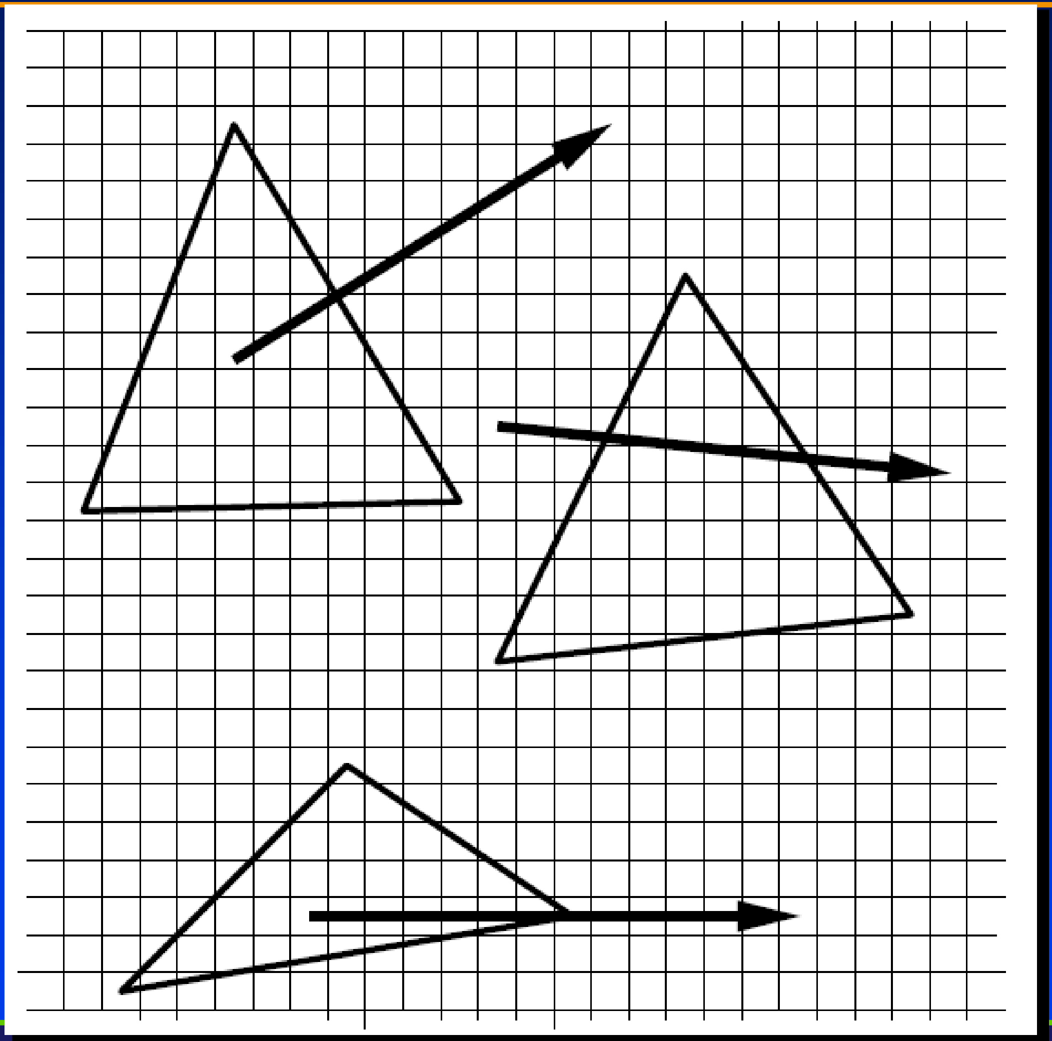
Slight Improvement

- We start from a simple triangle T: v1=(x1,y1), v2=(x2,y2), and v3=(x3,y3)
- We compute its bounding box B (later we will investigate the hierarchical modeling for the bounding volume hierarchy) first
  - For each pixel p that is inside B do
  - If p is inside T, then draw-point(p) end if
  - End for
- Essentially, the scan conversion MUST solve this problem, given a T and a pixel (or point in general), can we determine: p is a part of T

Ray Casting (Ray Firing)

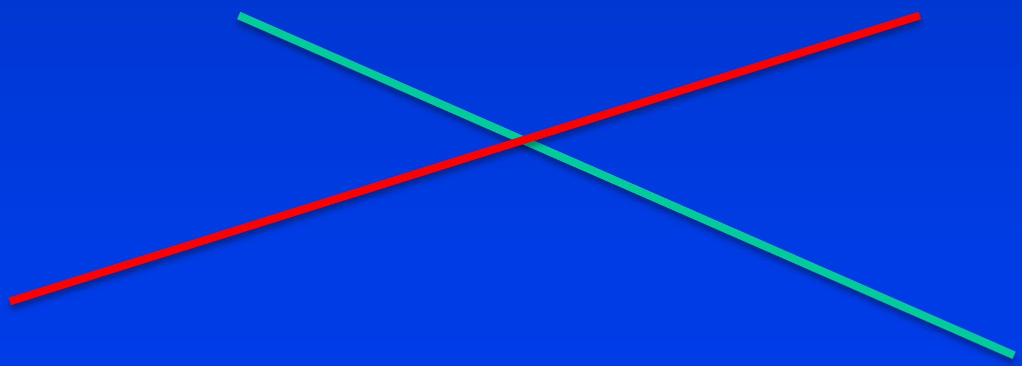
- We start from a simple triangle T: v1=(x1,y1), v2=(x2,y2), and v3=(x3,y3) and a point
  - (1) draw a ray from p outward along any direction
  - (2) count the number of intersections of this ray with triangular boundaries for T
  - (3) If ODD, then p is inside T, otherwise, p is not a part of T
- Is this method correct?

Polygon Scan Conversion



Scan Conversion

- What happens if the ray pass through a vertex of a simple triangle T: (x1,y1), (x2,y2), and (x3,y3)
- How do you actually count the number of intersections with a triangular boundary?
- How do you actually compute the intersection?



Computing Intersections

- Mathematically speaking: $f(x,y)=0$; $g(x,y)=0$, simple solve them for possible solutions
- In reality (computer graphics), we don't really do the above way!
- Why?
- How do we handle this in computer graphics?

Computing Intersections

- First, consider a boundary of a polygon, we do NOT use its explicit representation at all. Instead, we use $f(x,y)=ax+by+c=0$;
- Second, consider a ray geometry, once again, we do NOT use its explicit representation at all. Instead we are using its parametric representation: $ray(p, v) = p + v*t$, where t is a spatial parameter, $ray(p, v)$ works for (x,y) simultaneously!

Computing Intersections

- Parametric equation

$$x(t) = x_0 + t(x_1 - x_0)$$

$$y(t) = y_0 + t(y_1 - y_0)$$

- Vector expression

$$\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0)$$

$$\mathbf{p}(t) = (1 - t)\mathbf{p}_0 + t\mathbf{p}_1$$

- The parameter is between 0 and 1 to describe a line segment, the ray can be expressed in the same way

Computing Intersections

- Combine the two equations together (one is the implicit equation, another one is the parametric equation), f(ray(p,v))=0, where t is the ONLY parameter (to be solved)
- What is the geometric meaning of t?
- We are going to have more mathematically rigorous process on the parametric representation and its power and potential later in this course!

Scan Conversion

- We start from a simple triangle T: v1=(x1,y1), v2=(x2,y2), and v3=(x3,y3) and a point
- Consider the edge (v1v2) and formulate the implicit expression for this line

$$l_{1,2}(x,y) = a_{1,2}x + b_{1,2}y + c_{1,2}$$

- Pick a sign so that the evaluation of v3 is negative!
- This defines a half-plane

$$h_{1,2} = \{(x,y) : l_{1,2}(x,y) <= 0\}$$

Scan Conversion

- We start from a simple triangle T: v1=(x1,y1), v2=(x2,y2), and v3=(x3,y3) and a point
- Repeat the similar process for two other edges (v1v2) and (v2v3)

$$T = h_{1,2} \cap h_{1,3} \cap h_{2,3}$$

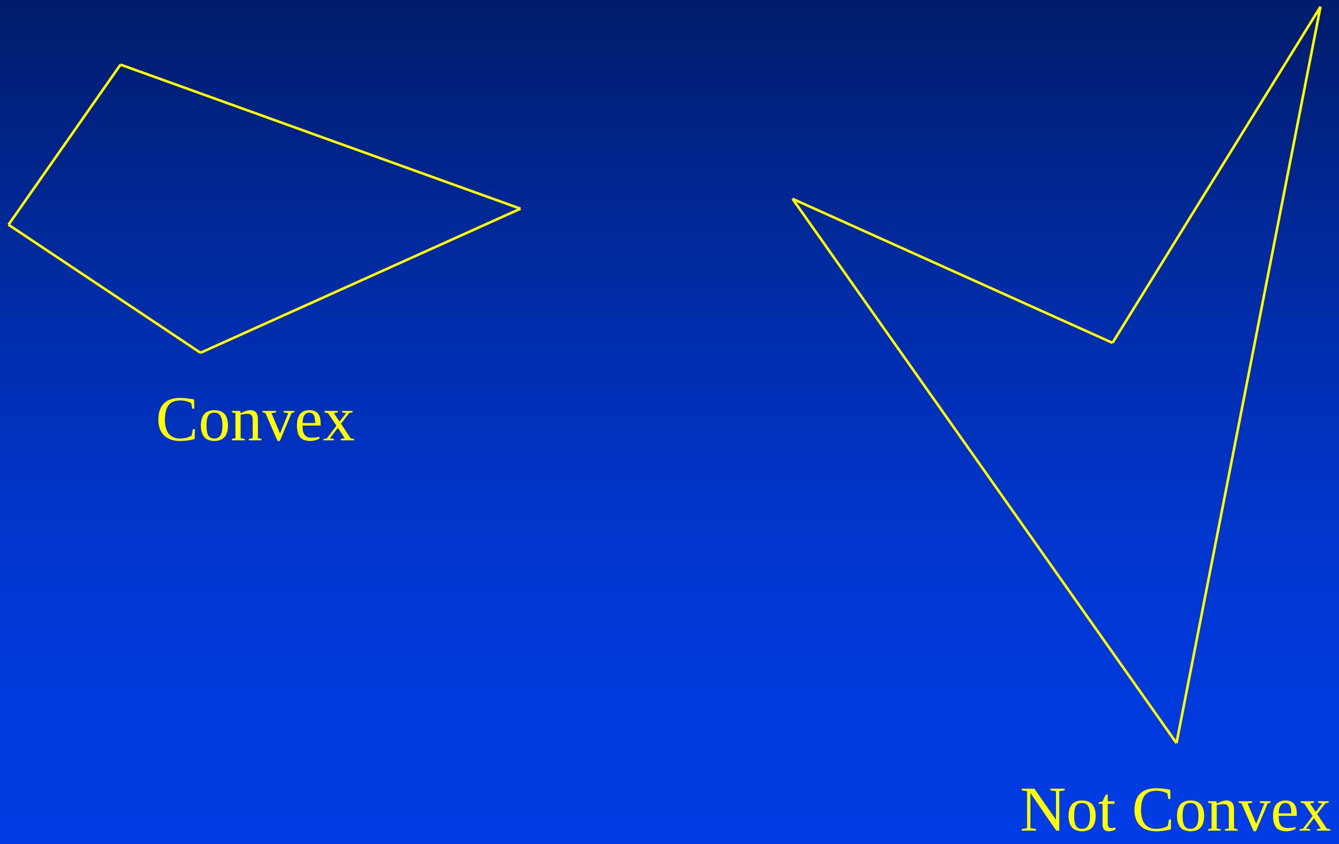
- It is equivalent to say, a pixel (point) is a part of a triangle if this point belongs to three half-planes simultaneously

$$l_{1,2}(p_x, p_y) <= 0$$

$$l_{1,3}(p_x, p_y) <= 0$$

$$l_{2,3}(p_x, p_y) <= 0$$

- What about Concave polygon?

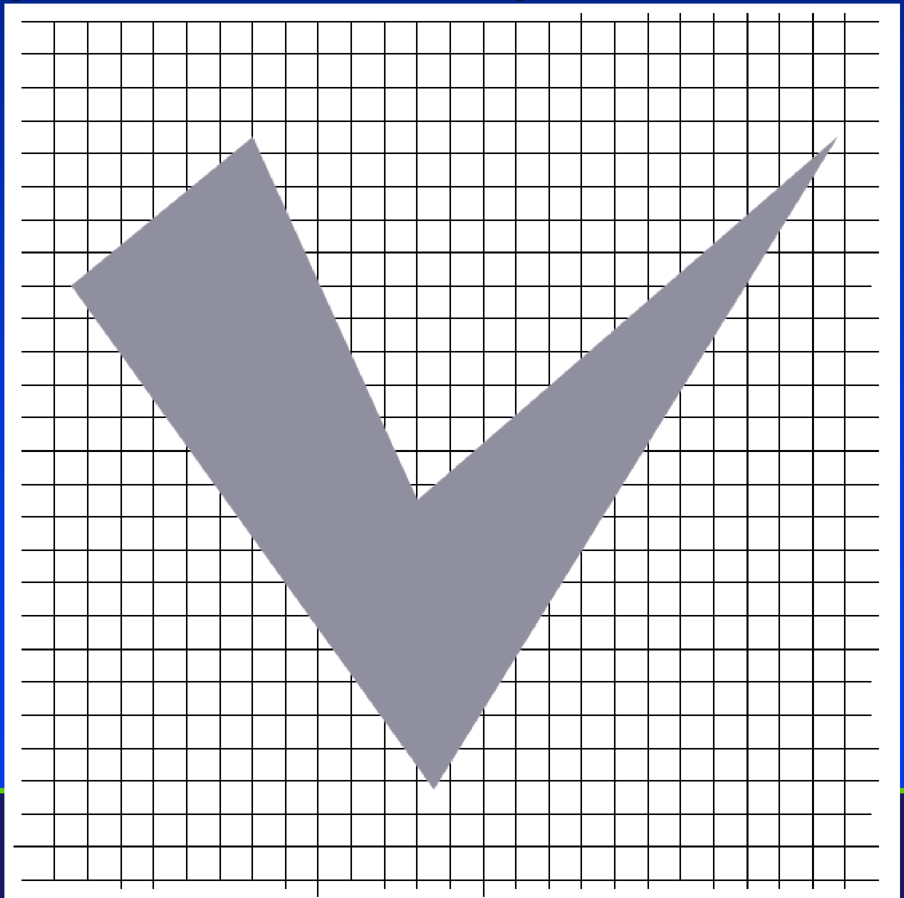

Convex
Not Convex

Convex

- A polygon is convex if…
  - A line segment connecting any two points on the polygon is contained in the polygon.
  - If you can wrap a rubber band around the polygon and touch all of the sides, the polygon is convex

Concave Polygon

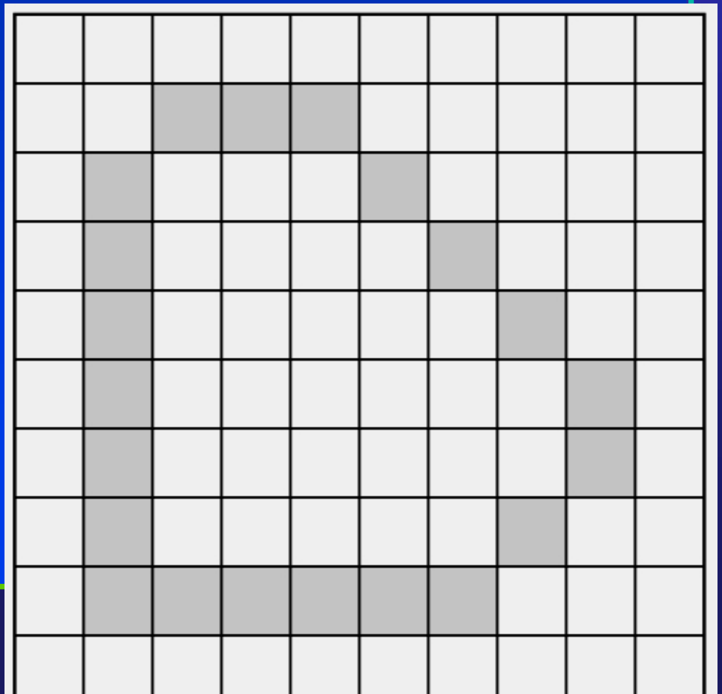
- We now consider a concave polygon T: (x1,y1), (x2,y2), (x3,y3), ……. (xn, yn)



Scan-Converting a Polygon

- General approach: any ideas?
- One idea: *flood fill*
  - Draw polygon edges
  - Pick a point (x,y) inside and **flood fill** with DFS

```
flood_fill(x,y) {
  if (read_pixel(x,y)==white) {
      write_pixel(x,y,black);
      flood_fill(x-1,y);
      flood_fill(x+1,y);
      flood_fill(x,y-1);
      flood_fill(x,y+1);
} }
```



Polygon Classification

- Simple convex
- Simple concave
- Non-simple (with self-intersection)
- Once again, a bounding box can help, and the idea of using ray-casting is also GOOD!
- However, these approaches would NOT take advantage of (spatial) coherence
- Adjacent pixels in the image space are likely sharing the similar graphics properties such as color and appearance

Sweeping Lines

- Our observation – spatial coherence

  If $p \in T$, then neighboring pixels are probably in the triangle, too (Coherence)

- Idea

(1) sweep from top to bottom
(2) maintain intersections of T and sweep-line "span"
(3) paint pixels in the span

Sweep-line Algorithm

- Algorithm

Initialize x_l and x_r

For each scan line covered by T do Paint pixels $(x_l, y), \ldots, \ldots, (x_r, y)$ on the current span

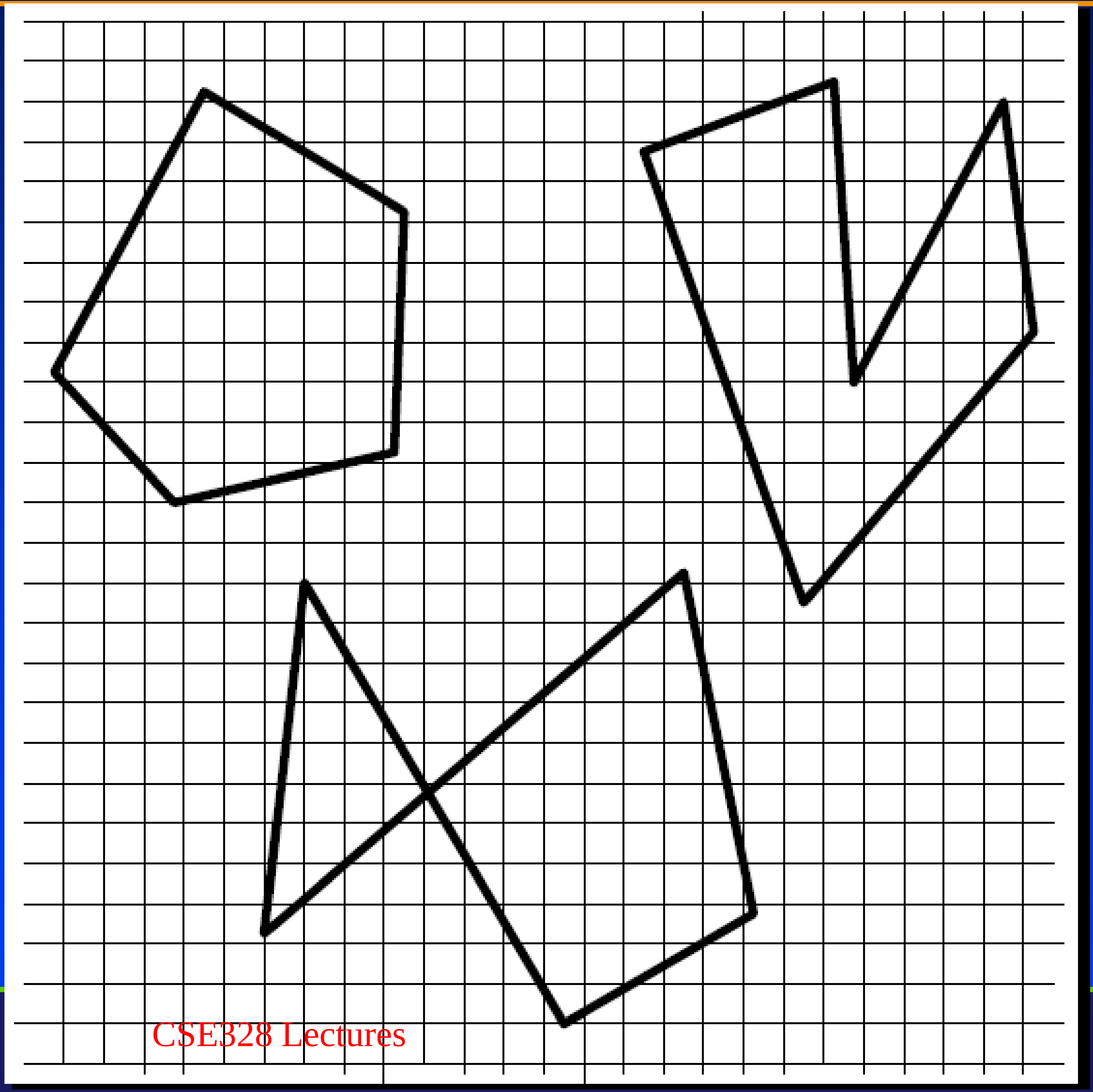
Incrementally update x_l and x_r

End for

- Question: how do we update x_l and x_r ?

- Answer: please recall our line-drawing algorithm!

Polygon Classification



Scan Conversion

More efficient algorithm

For each scanline

Identify all intersections $x_0, x_1, \ldots, x_{k-1}$

Sort intersections from left to right

Fill pixels between consecutive pairs of intersection

$$(x_{2i}, y), (x_{2i+1}, y)$$

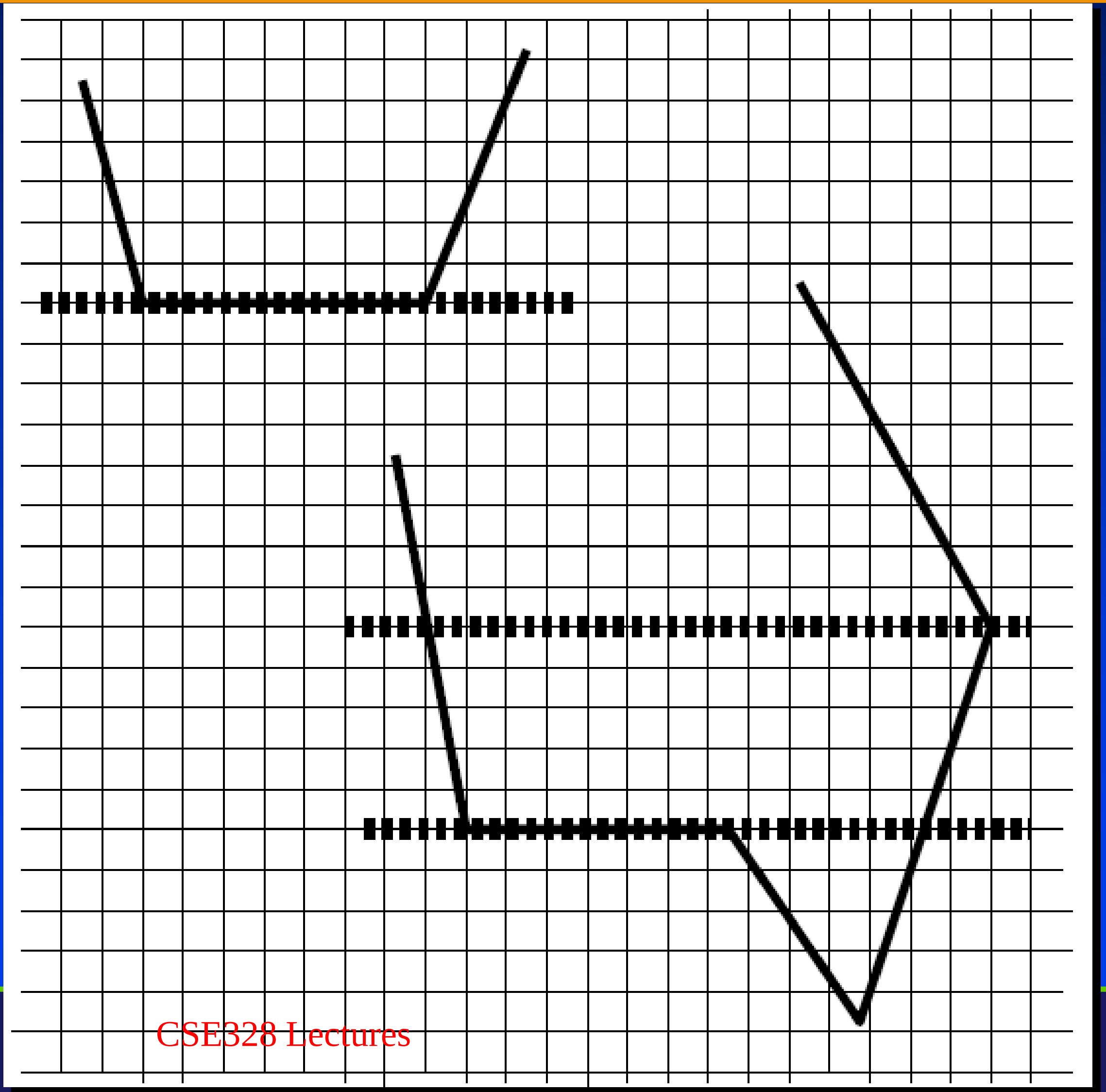
We must deal with "special cases" !

- horizontal lines
- intersecting a vertex (double intersection)
- unwanted intersection

Scan Conversion

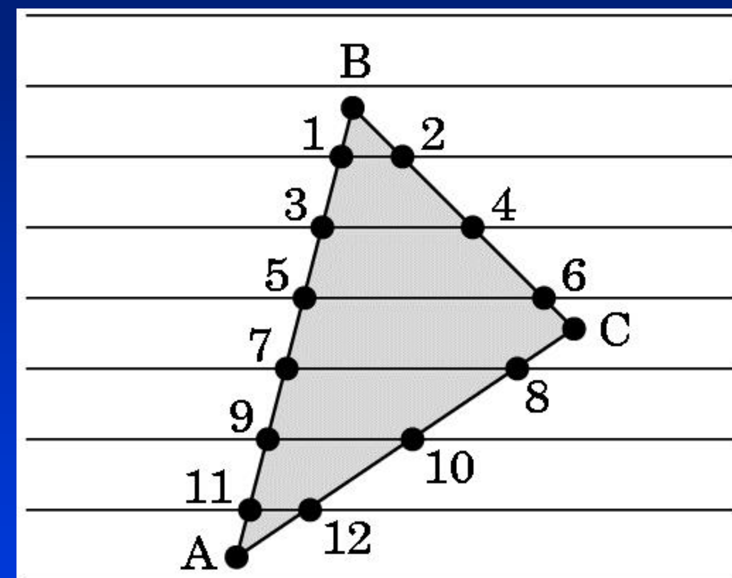
- We must speed up the edge intersection detection
- Data structure for efficient implementation
  - A sorted edge table
  - The active edge list
  - From bottom to the top
- Practical polygon scan conversion – based on polygon triangulation
- Extremely easy to handle for convex polygons
- Triangles are often particularly nice to work with because they are always planar and simple

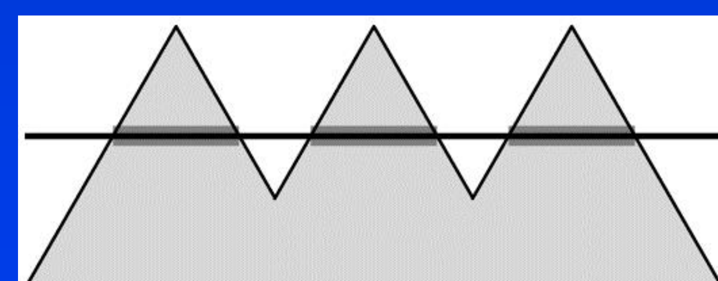
Special Cases



Scan-Line Approach

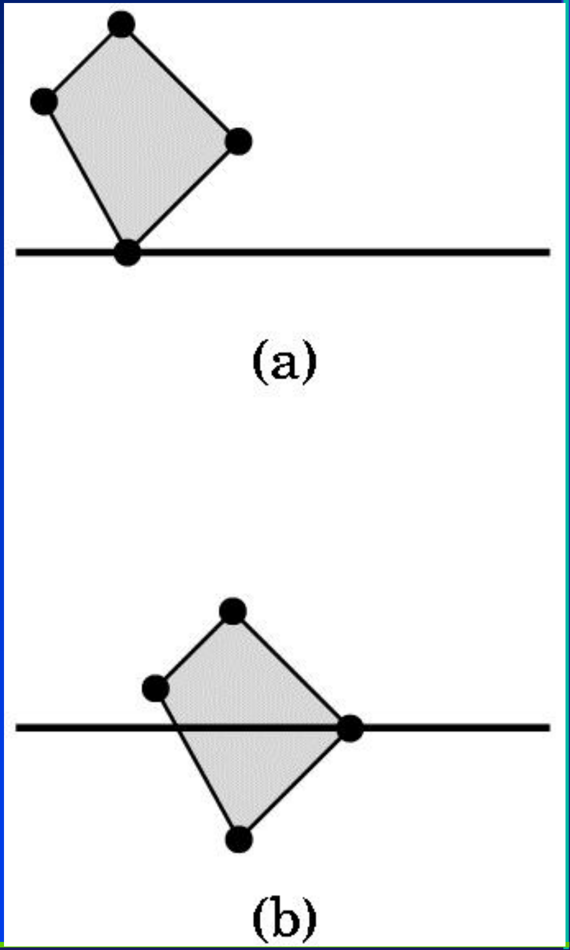
- More efficient way: use a scan-line rasterization algorithm
- For each y value, compute x intersections. Fill according to winding rule
- How to compute intersection points?
- How to handle shading?
- Some hardware can handle multiple scanlines in parallel





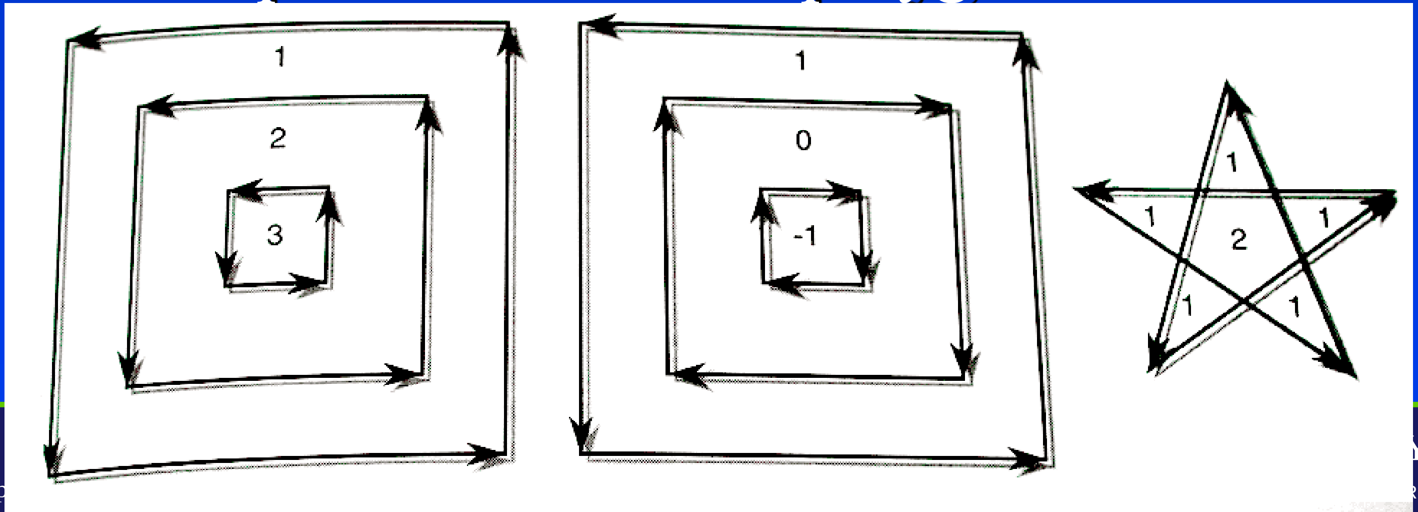
Singularities (Special Cases)

- If a vertex lies on a scanline, does that count as 0, 1, or 2 crossings?
- How to handle singularities?
- One approach: don’t allow. *Perturb* vertex coordinates
- OpenGL’s approach: place pixel centers half way between integers (e.g. 3.5, 7.5), so scanlines never hit vertices



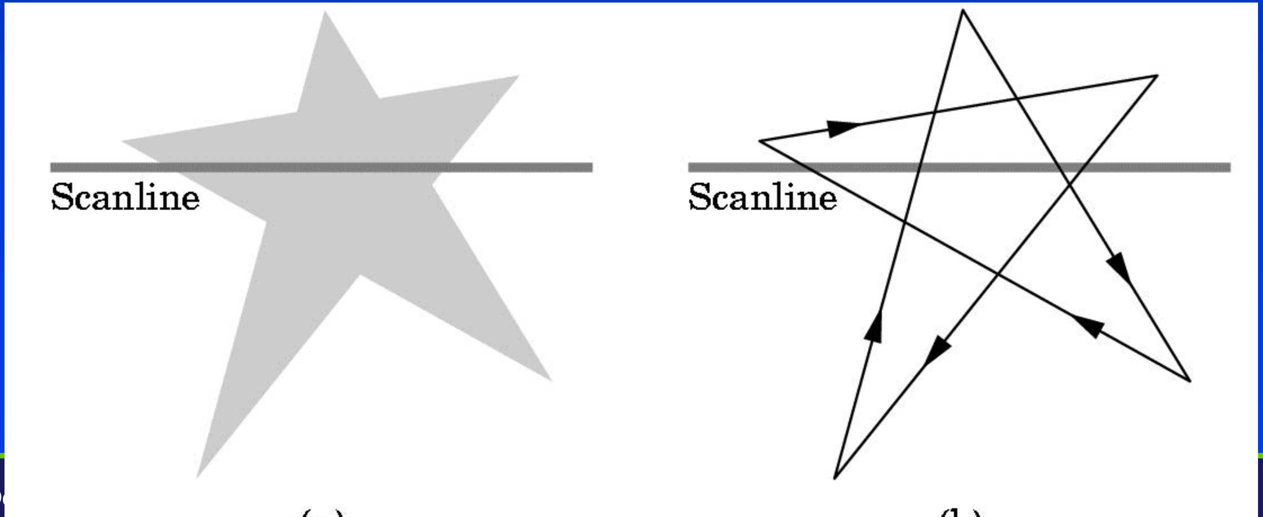
Winding Test

- Most common way to tell if a point is in a polygon: the winding test
  - Define “winding number” w for a point: signed number of revolutions around the point when traversing boundary of polygon once
  - When is a point “inside” the polygon?

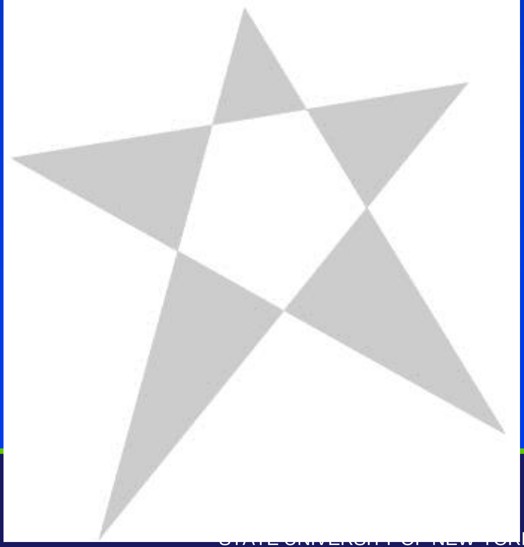


Rasterizing Polygons (Scan Conversion

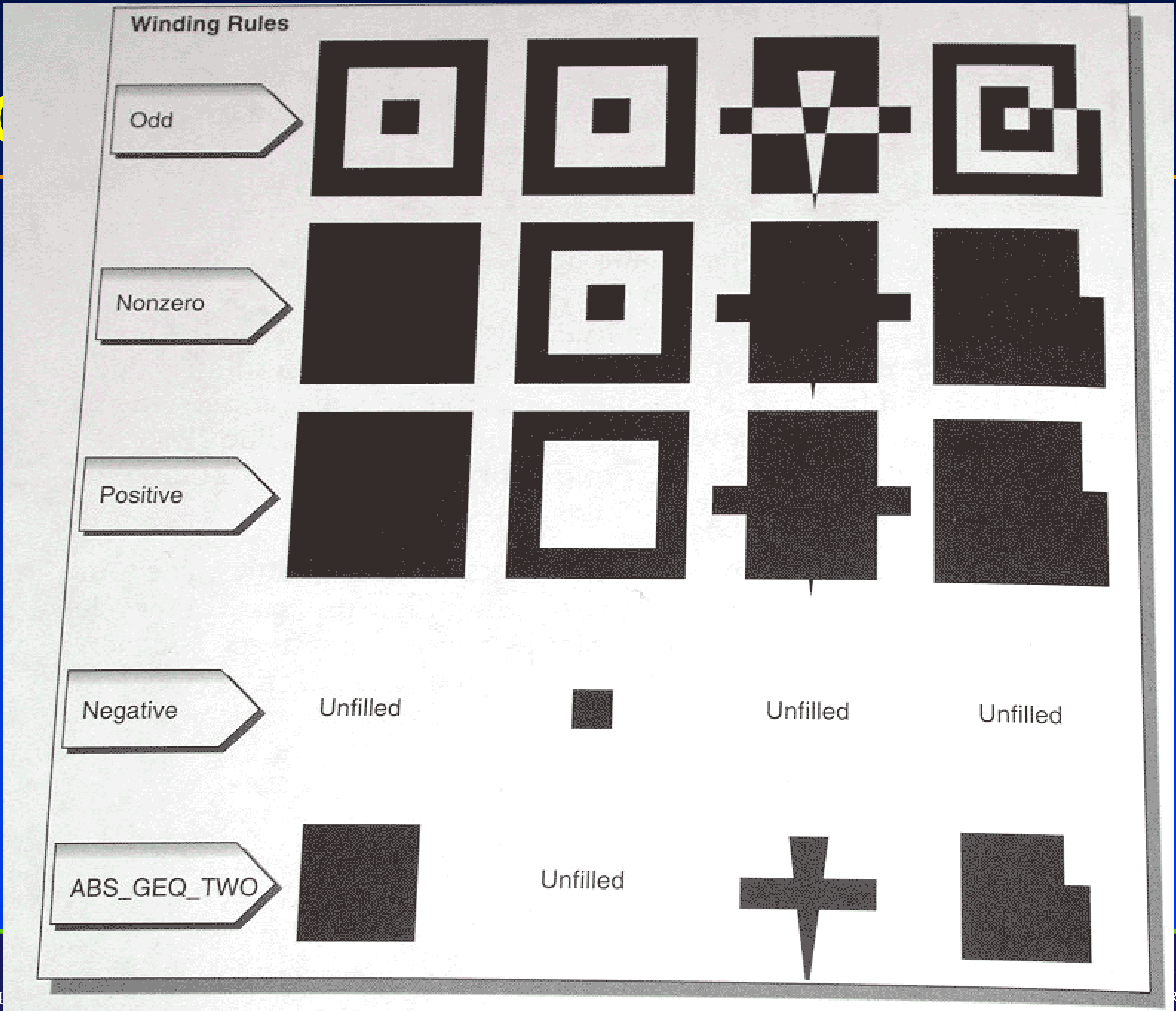
- Polygons may be or may not be simple, convex, or even flat. How to render them?
- The most critical thing is to perform inside-outside testing: how to tell if a point is in a polygon?



(a) (b)

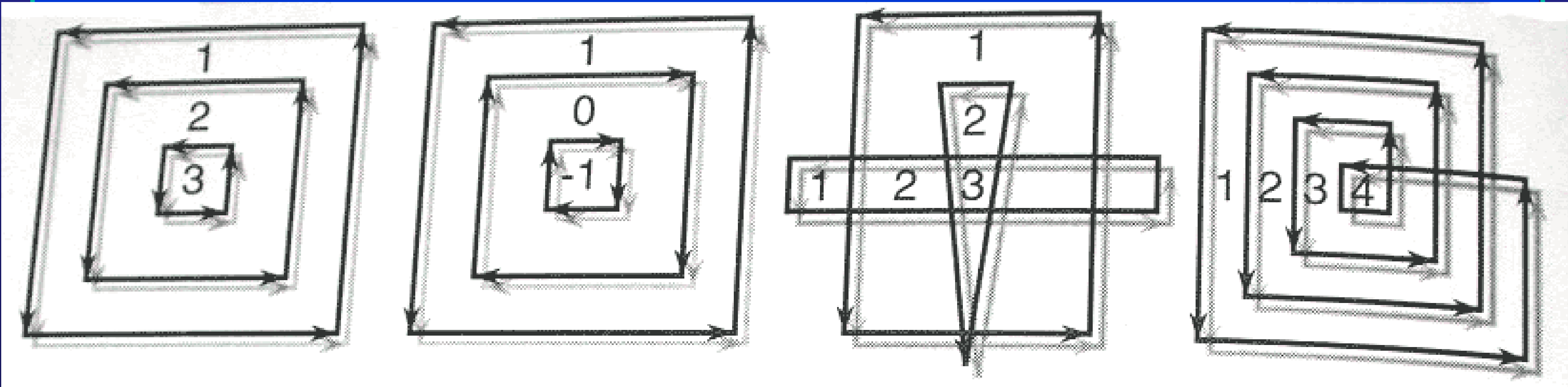


Win


Winding Rules
Odd
Nonzero
Positive
Negative
Unfilled
Unfilled
Unfilled
ABS_GEQ_TWO
Unfilled

OpenGL and Concave polygons

- OpenGL guarantees correct rendering only for simple, convex, planar polygons
- OpenGL tessellates concave polygons
- Tessellation depends on winding rule you tell OpenGL to use: Odd, Nonzero, Pos, Neg, ABS_GEQ_TWO



Scan Conversion

- At this point in the pipeline, we have only polygons and line segments. Render!
- To render, convert to pixels (“fragments”) with integer screen coordinates (ix, iy), depth, and color
- Send fragments into fragment-processing pipeline