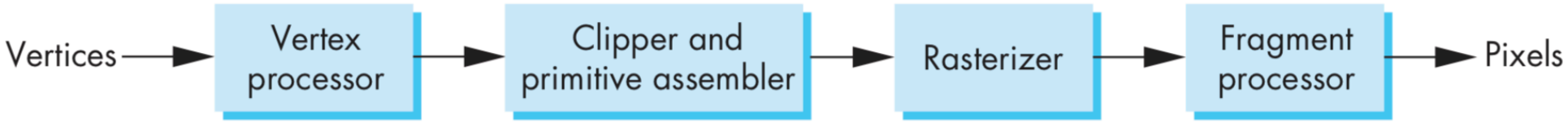
COMP3271 Computer Graphics

Clipping & Rasterization

2019-20

Graphics Pipeline Overview



Clipping — To eliminate objects (or part of objects) that lie outside the viewing volume

Rasterization — To produce fragments from the remaining objects that are visible in the final image

Two Rendering Approach

For every pixel, determine which object that projects on the pixel is closest to the viewer and compute the shade of this pixel

- Ray tracing paradigm

For every object, determine which pixels it covers and shade these pixels

- Pipeline approach
- Must keep track of depths

Clipping

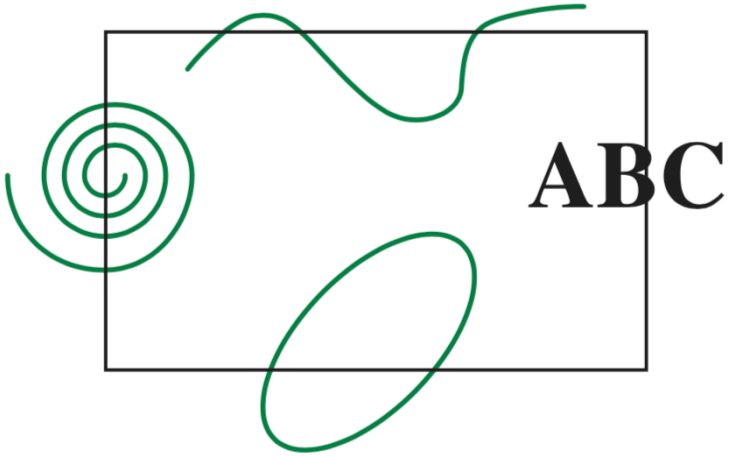
2D against clipping window

3D against clipping volume

Easy for line segments & polygons

Hard for curves and text

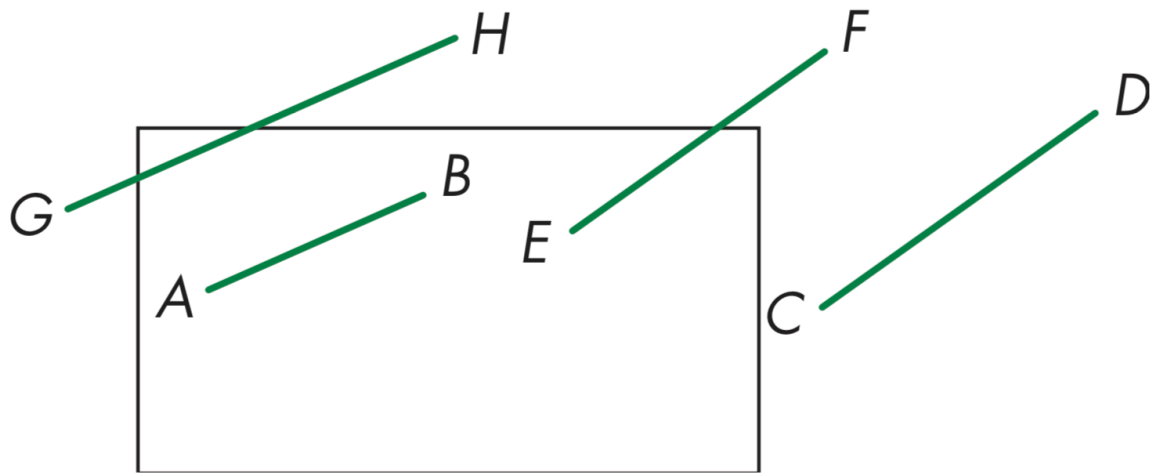
- Convert to lines and polygons first



Clipping 2D Line Segments

Brute force approach: compute intersections with all sides of clipping window

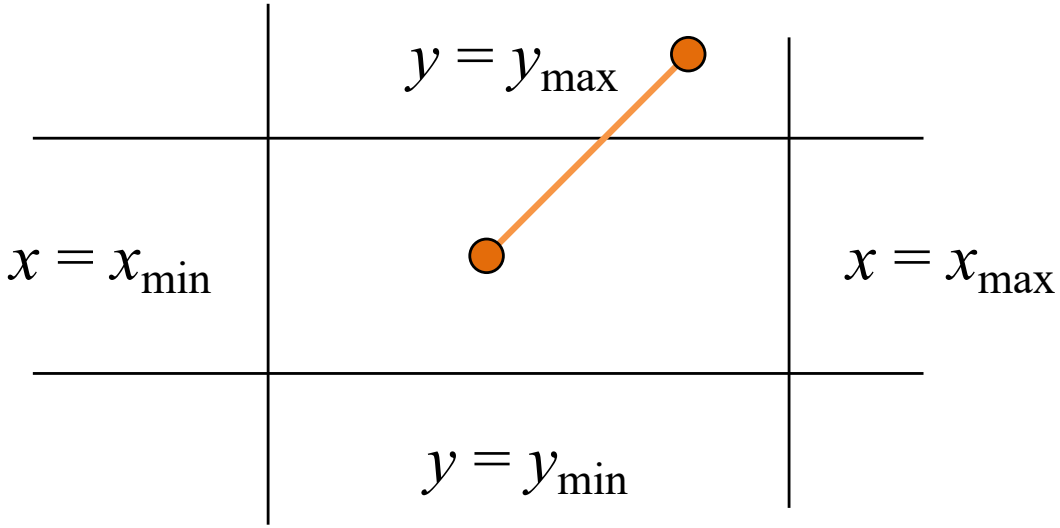
- Inefficient: one division per intersection



Cohen-Sutherland Algorithm

Idea: eliminate as many cases as possible without computing intersections

Start with four lines that determine the sides of the clipping window

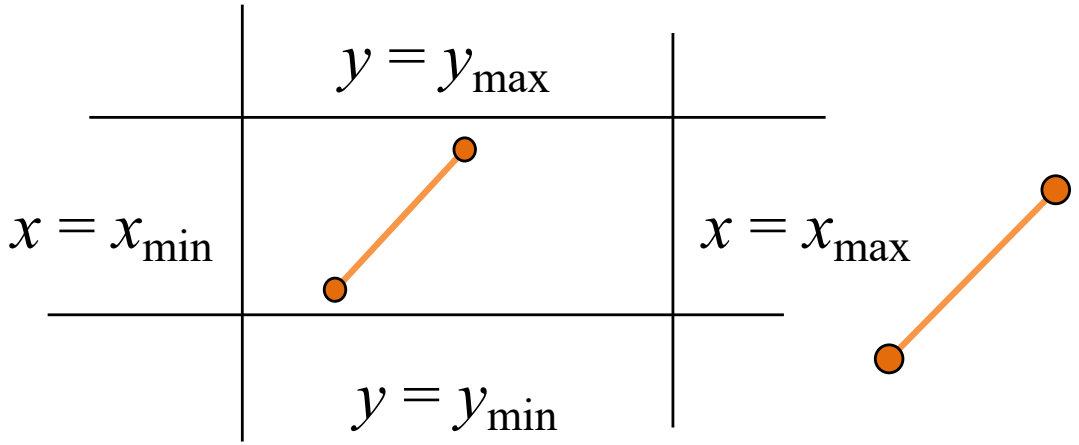


Viewing volume must be axis-aligned

The Cases

Case 1: both endpoints of line segment inside all four lines

- Draw (accept) line segment as is



Case 2: both endpoints outside all lines and on same side of a line

- Discard (reject) the line segment

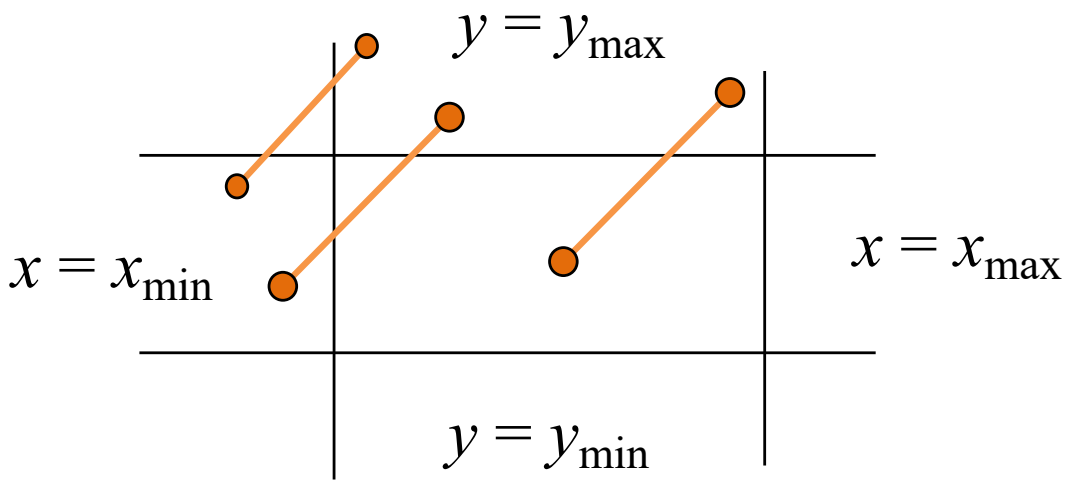
The Cases

Case 3: One endpoint inside, one outside

- Must do at least one intersection

Case 4: Both outside

- May have part inside
- Must do at least one intersection

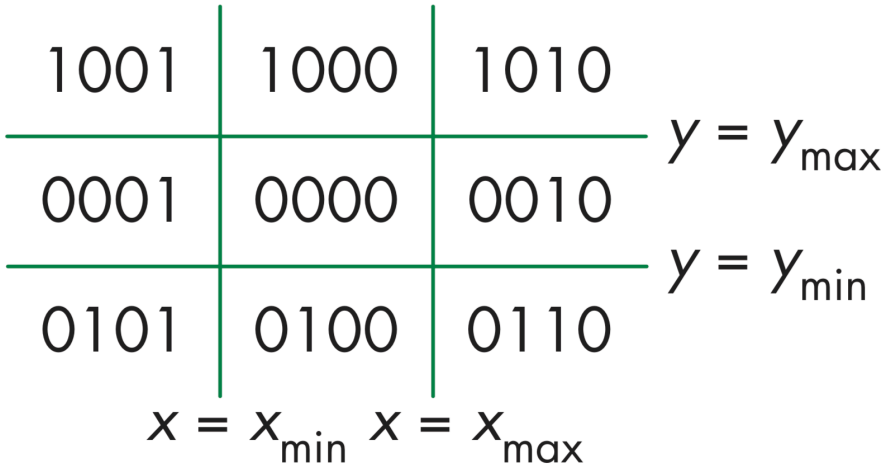


Defining Outcodes

For each endpoint, define an **outcode**

$$b_0 b_1 b_2 b_3$$

$b_0 = 1$ if $y > y_{max}$, 0 otherwise
$b_1 = 1$ if $y < y_{min}$, 0 otherwise
$b_2 = 1$ if $x > x_{max}$, 0 otherwise
$b_3 = 1$ if $x < x_{min}$, 0 otherwise



Outcodes divide space into 9 regions

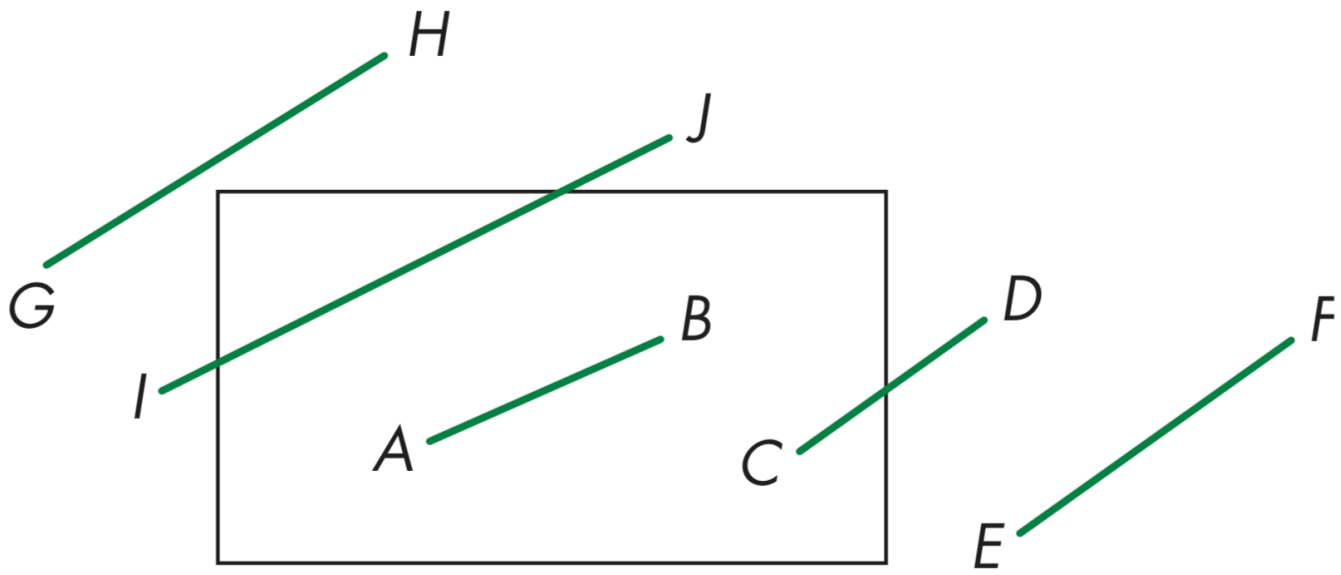
Computation of outcode requires at most 4 subtractions

Using Outcodes

Consider the 5 cases below

Line AB: outcode(A) = outcode(B) = 0

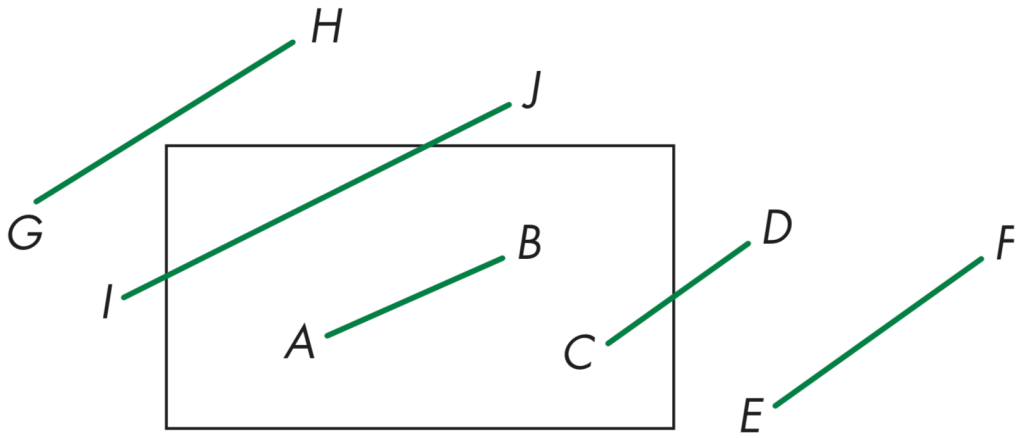
- Accept line segment



Using Outcodes

Line CD: outcode (C) = 0, outcode(D) ≠ 0

- Compute intersection
- Location of 1 in outcode(D) determines which edge to intersect with
- Note if there were a segment from C to a point in a region with 2 ones in outcode, we might have to do two intersections



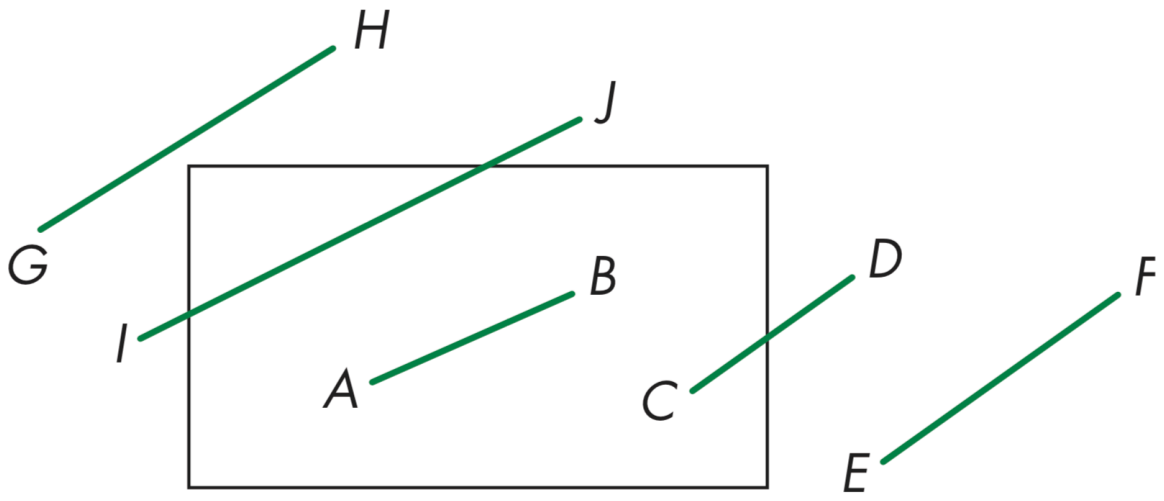
Using Outcodes

Line EF:

outcode(E) logically ANDed with outcode(F)

(bitwise) ≠ 0

- Both outcodes have a 1 bit in the same place
- Line segment is outside of corresponding side of clipping window
- reject



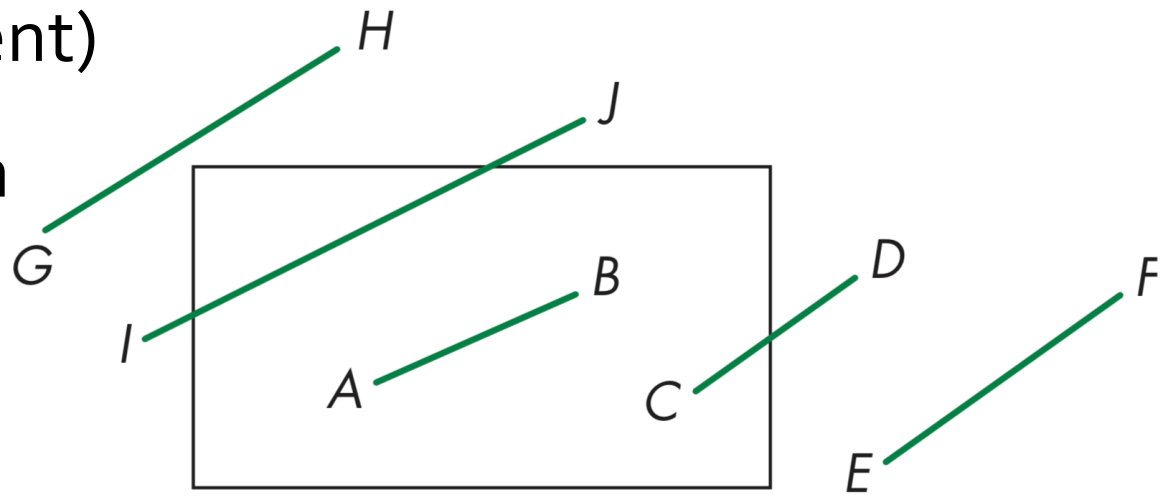
Using Outcodes

Lines GH and IJ: same outcodes, neither zero but logical AND yields zero

Shorten line segment by intersecting with one of sides of window

Compute outcode of intersection (new endpoint of shortened line segment)

Re-execute algorithm



Efficiency

In many applications, the clipping window is small relative to the size of the entire data base

- Most line segments are outside one or more side of the window and can be eliminated based on their outcodes

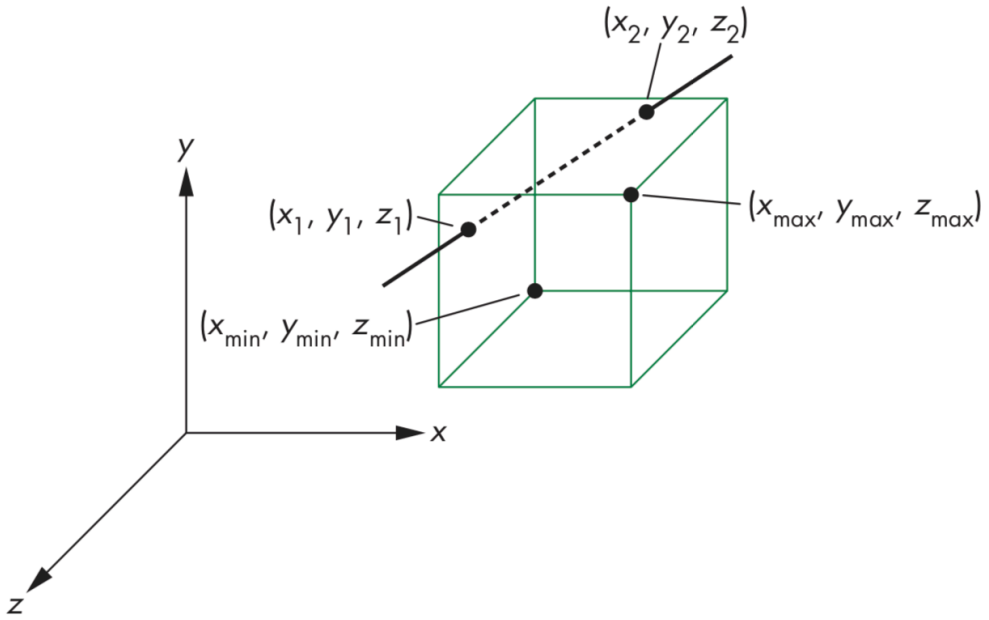
Inefficient as the clipping algorithm is recursive, and code has to be re-computed for line segments that must be shortened in more than one step.

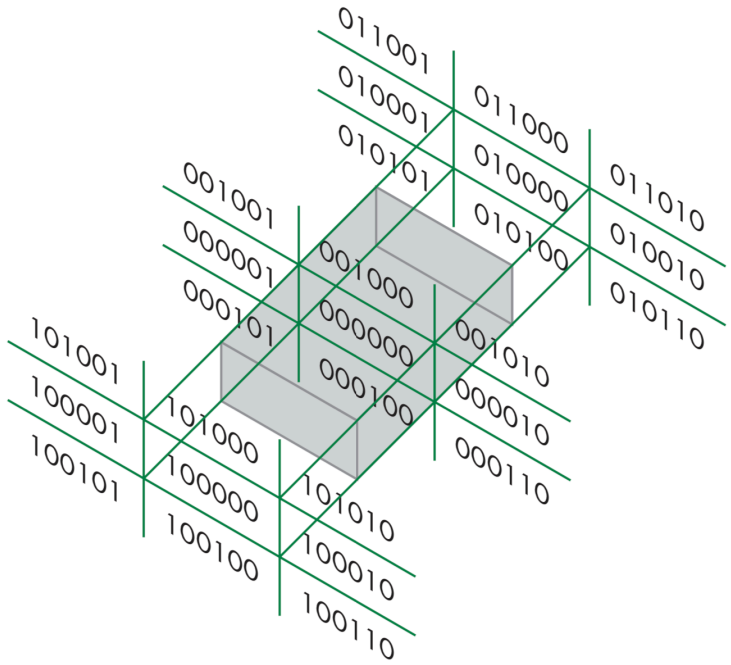
Cohen Sutherland in 3D

Easily extended to 3D

Use 6-bit outcodes

When needed, clip line segment against planes

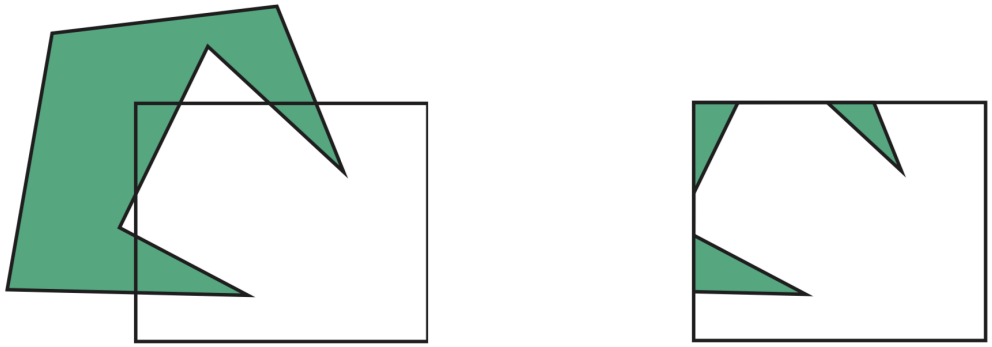




Polygon Clipping

Not as simple as line segment clipping

- Clipping a line segment yields at most one line segment
- Clipping a polygon can yield multiple polygons



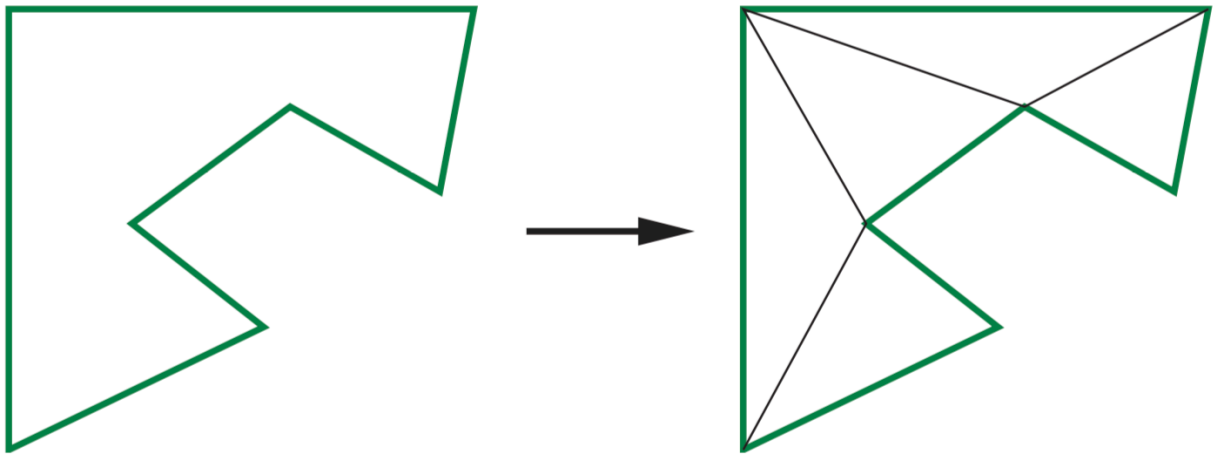
However, clipping a convex polygon can yield at most one other polygon

Tessellation and Convexity

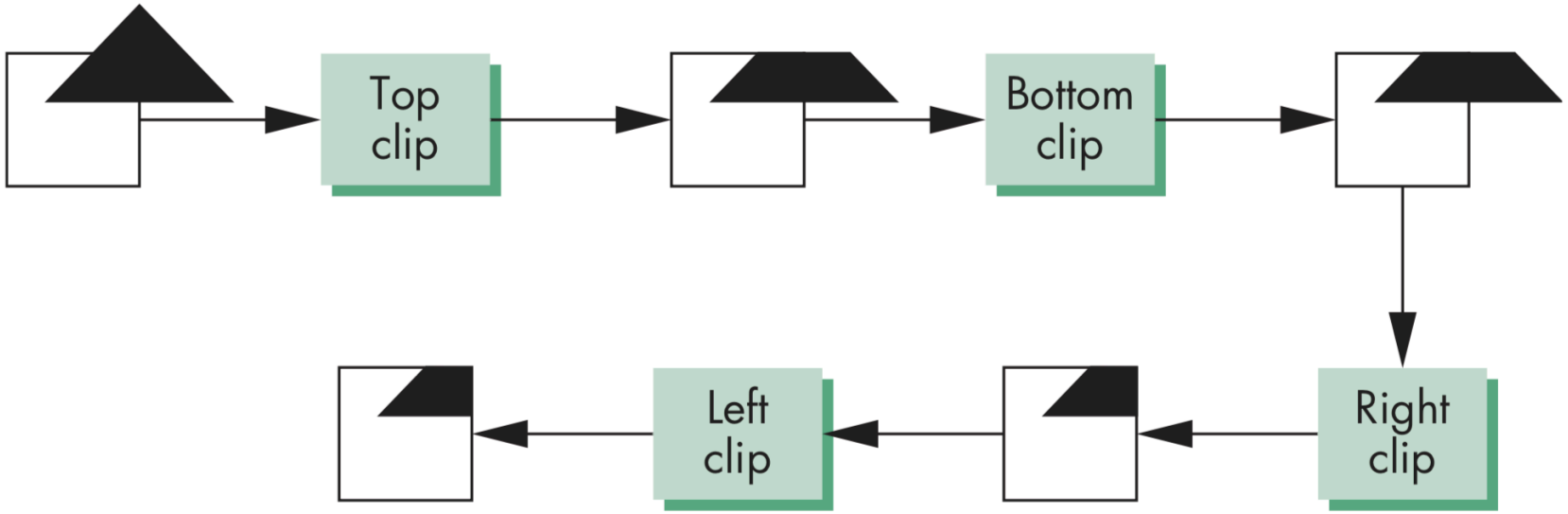
One strategy is to replace nonconvex (*concave*) polygons with a set of triangular polygons (a **tessellation**)

Also makes fill easier

Tessellation code in GLU library



Pipeline Clipping of Polygons

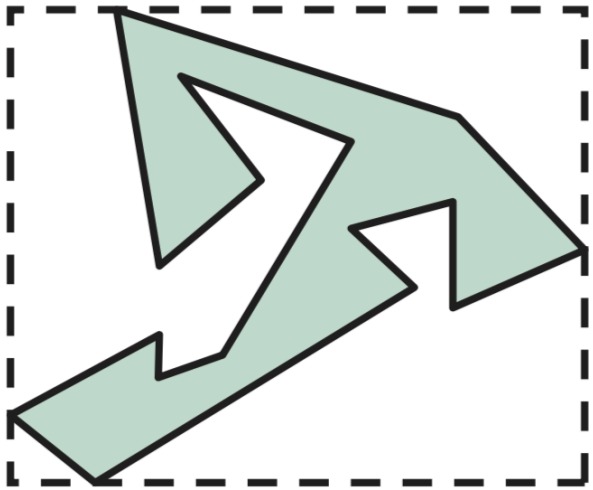


Extending to 3D: add front and back clippers

Bounding Boxes

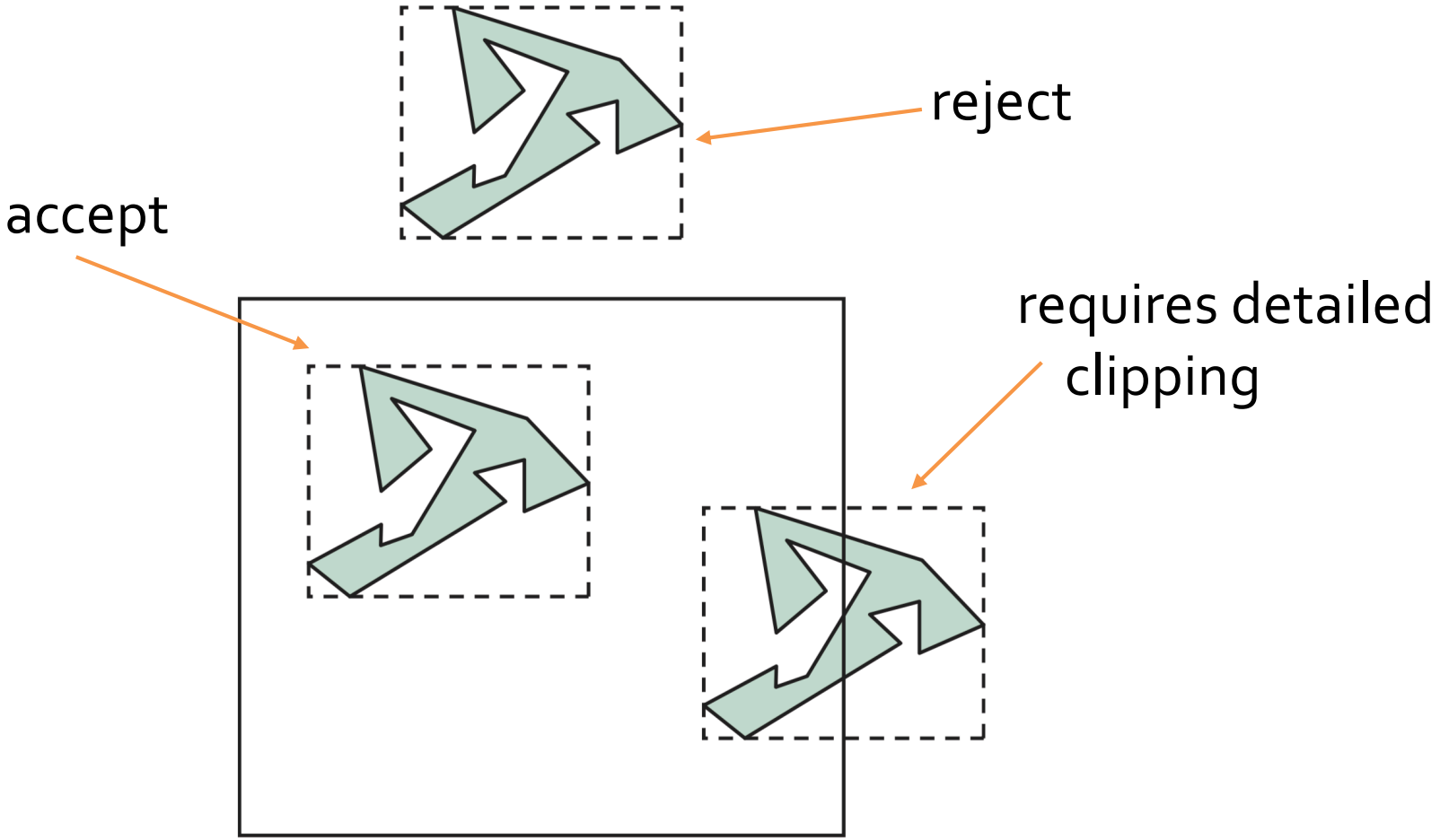
Rather than doing clipping on a complex polygon, we can use an **axis-aligned bounding box**

- Smallest rectangle aligned with axes that encloses the polygon
- Simple to compute: max and min of x and y



Bounding Boxes

Can usually determine accept/reject based only on bounding box



Rasterization

Rasterization (**scan conversion**)

- Determine which pixels that are inside a primitive specified by a set of vertices
- Produces a set of fragments
- Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices

Pixel colors determined later using color, texture, and other vertex properties

Issues on Rasterization

Moving from continuous geometry to discrete pixels is inexact

- we're attempting to approximate the primitive with pixels
- thus a certain amount of error is being introduced

Goal #1: Accuracy

- construct good approximations (i.e., low error)
- this can be hard because there may be many tricky cases

Goal #2: Efficiency

- this process is going to happen a lot
  - imagine we need to draw 10 million polygons/second
- one near-universal strategy: implement this process in hardware

Line Rasterization

We have a 2-D line segment inside the viewport

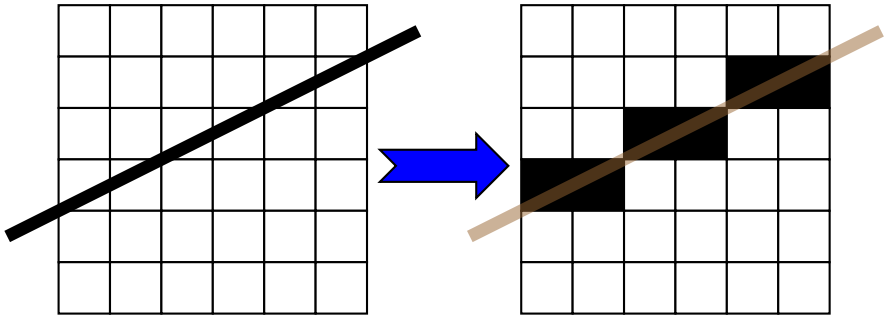
- it's been projected & clipped

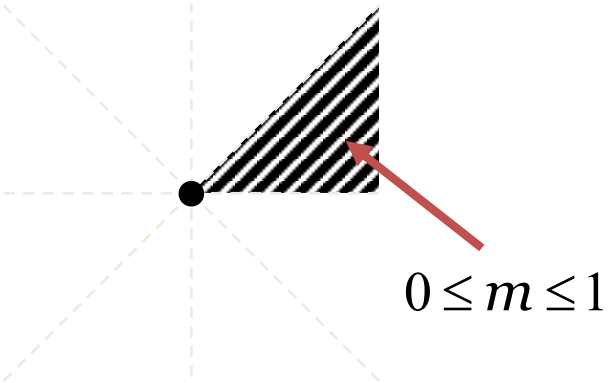
To simplify discussion, assume slope is between 0 and 1

- other cases are symmetric

Our goal: fill in pixels "on" line

- actually, most nearly on as measured at pixel centers
- Connected, but only one pixel wide





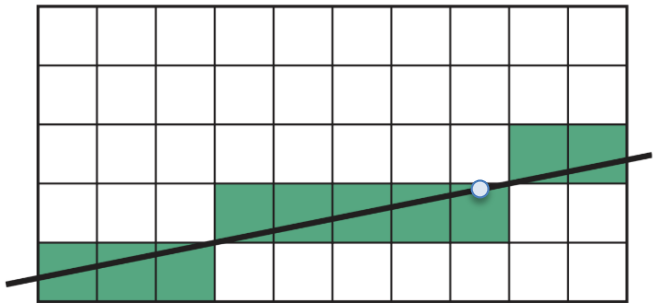
First Cut: Very Simple Line Algorithm

Compute equation of line

$$y = mx + b \quad \text{where } m = \frac{\Delta y}{\Delta x}$$

Now, start at the leftmost point and walk to the right

- increment x by 1 at each step
- for each x, compute y with equation
  - need to round y to integral coordinate
  - for instance, can use `rint(y)` or `floor(y + 0.5)`
- fill in pixel (x, y)



This is a correct algorithm, but it is inefficient

- requires floating point multiply/add/round for each pixel column

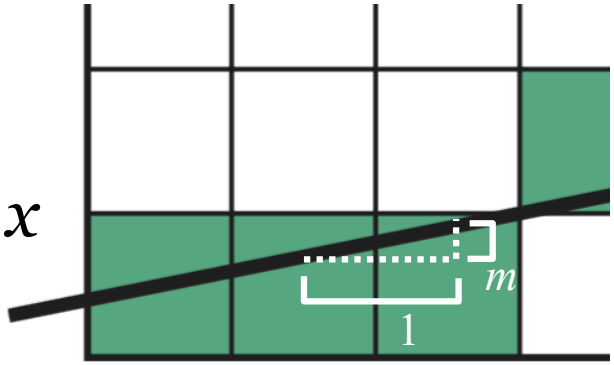
Fortunately, we can easily do better …

A More Efficient Incremental Algorithm

What does the slope of a line mean?

- it's the change in y for a unit change in x
- this is exactly what we need to know!

$$y(x+1) = m(x+1) + b = (mx+b) + m = y(x) + m$$



Again, let's start at leftmost point and walk to the right

- increment x by 1 at each step
- increment y by m at each step
- fill in pixel (x, *round*(y))

Digital Differential Analyzer

This improved algorithm has a fancy name:
Digital Differential Analyzer (DDA)

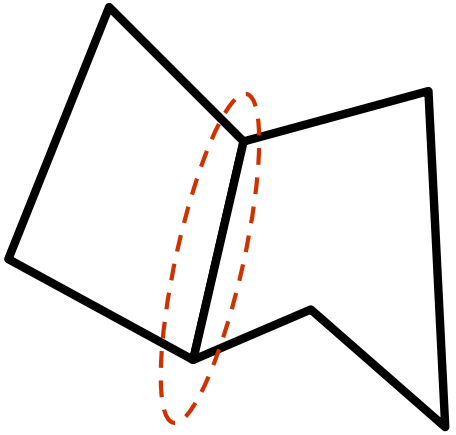
```
for (x=x1; x<=x2, x++) {
   y += m;
   write_pixel(x, round(y), line_color)
}
```

DDA requires one floating point addition per step

Can be further improved by an integer implementation (Brensenham algorithm)

Polygon Rasterization

We want to fill every pixel covered by a polygon



And we need to be really careful!

- suppose we have two adjacent polygons
- we don't want any overlap or any cracks
- visit every covered pixel exactly once

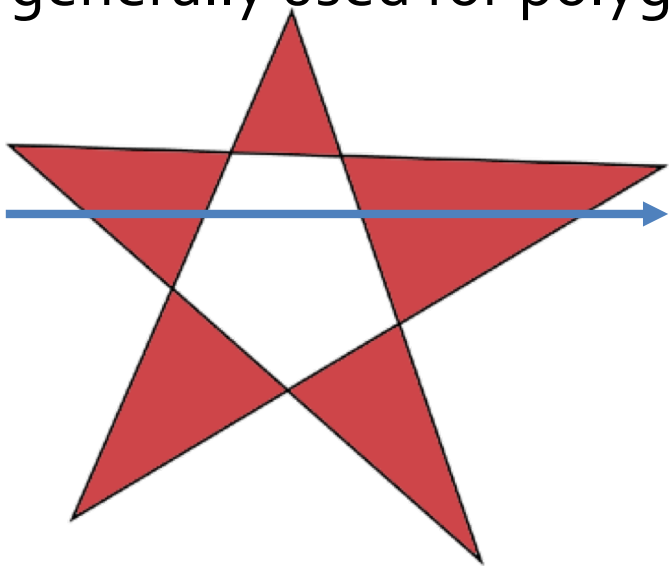
What's the Inside of a Polygon?

This is not obvious when the polygon intersects itself

- over time, people came up with some arbitrary definitions

Definition #1: **Odd–even rule**

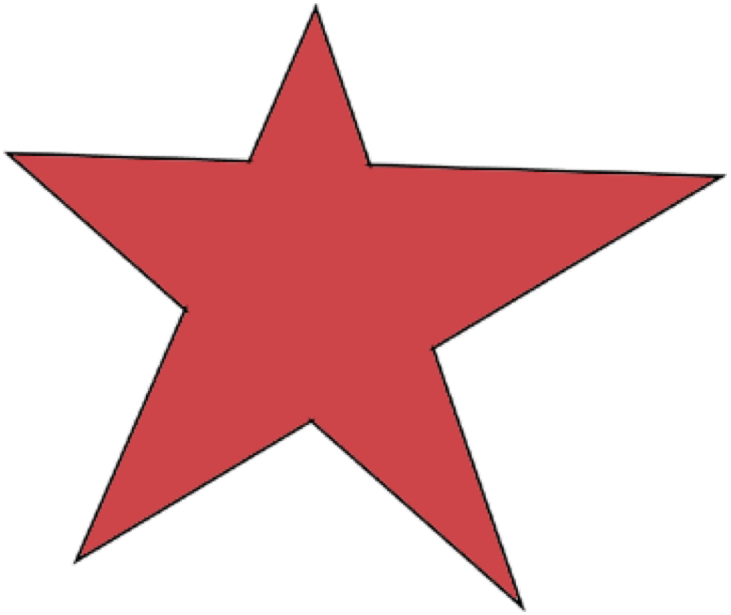
- pass horizontal line through shape; points with odd # crossings are **in**
- this is the one generally used for polygon rasterization

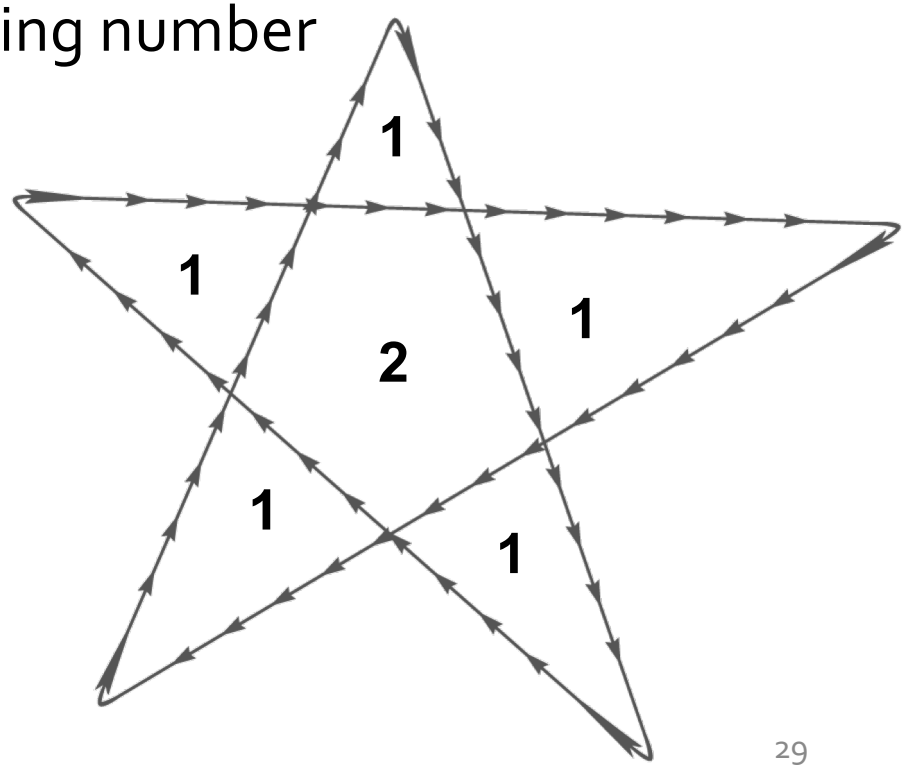


What's the Inside of a Polygon?

Definition #2: **Winding rule**

- walk around entire polygon; add up # of times you encircle a point
  - clockwise (+1) or counter-clockwise (-1)
- fill points with non-zero winding number

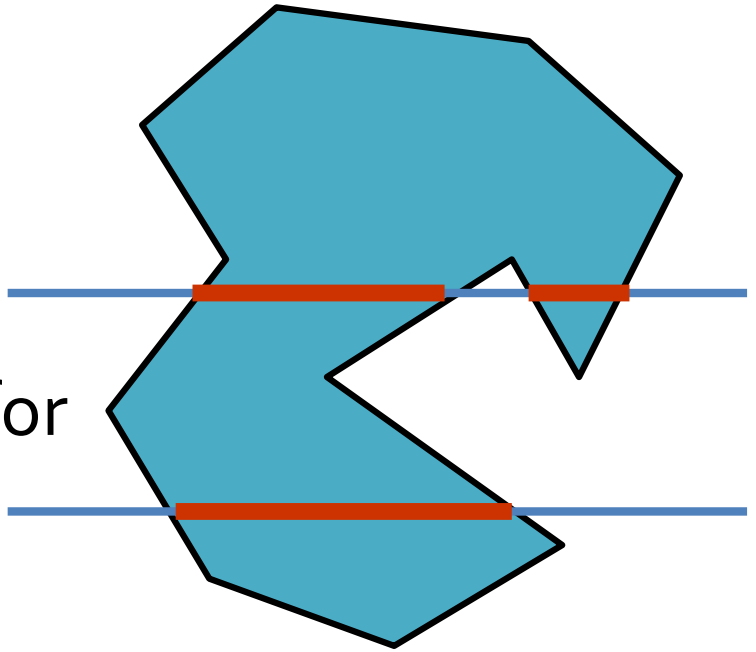




Scan Converting Polygons

Loop over all scanlines covered by polygon

- find points of intersection, from left to right
- fill all the interior spans
  - these are the odd spans
  - as per the odd–even rule



Some special cases to watch out for

- horizontal edges
- scanline passing through vertices

Efficiently Tracking Scanline Intersections

We could do something simple, but inefficient

- directly compute intersection of every scanline with every edge

But we can do better by exploiting coherence of scanlines

- Create an Edge Table with all edges sorted by y_{min}
- Maintain Active Edge Table to hold list of edges intersecting current scanline sorted left to right

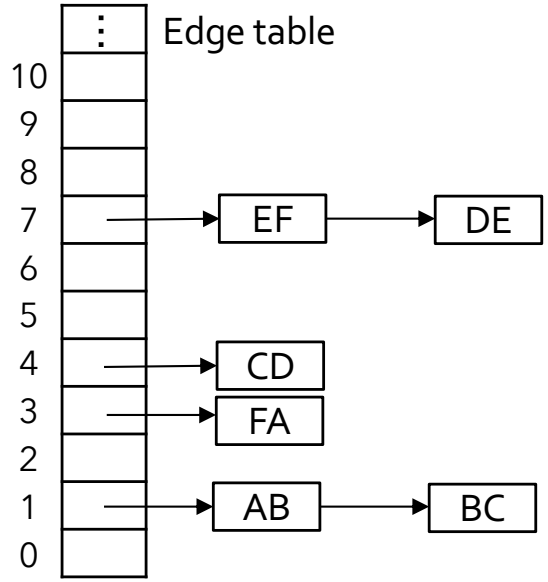
If we process the polygon from y_{min} to y_{max}

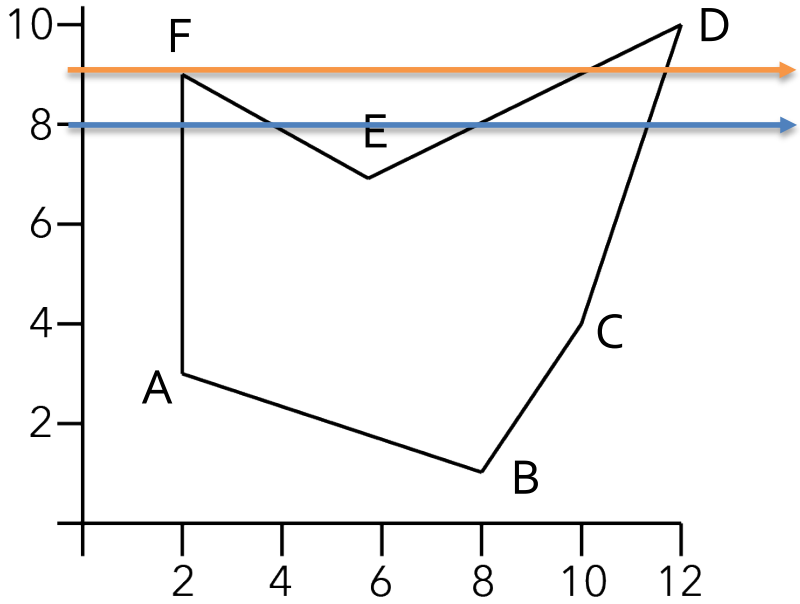
- add edge to AET at its y_{min} value
- remove edge at its y_{max} value
- when the AET is empty, we're done

can use something like DDA and Bresenham's line algorithm to efficiently track x-coordinate of intersections

Efficiently Tracking Scanline Intersections

Edge Table: all edges sorted by y_{min}
Active Edge Table: hold list of edges intersecting current scanline sorted left to right





Scanline y = 8

Active edge table: → FA → EF → DE → CD

Scanline y = 9

Active edge table: → DE → CD