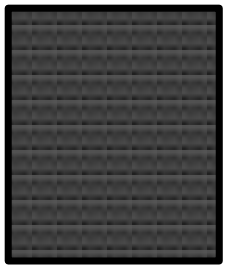
CptS 442/542 (Computer Graphics)
Unit 7: Filling Polygons

Bob Lewis

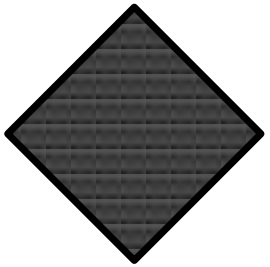
School of Engineering and Applied Sciences
Washington State University

Fall, 2017

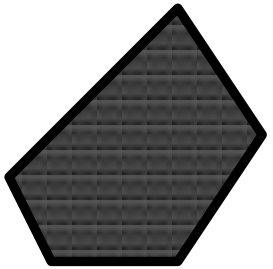
A Polygon Taxonomy



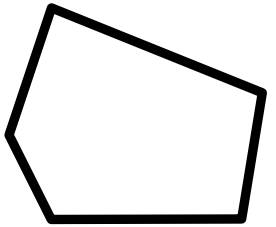
box or rectangle



regular

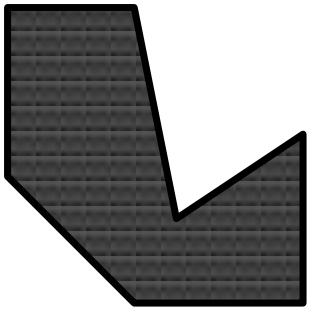


irregular

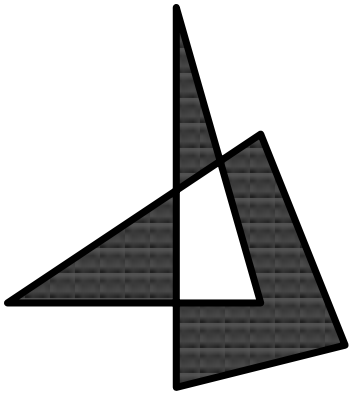


unfilled

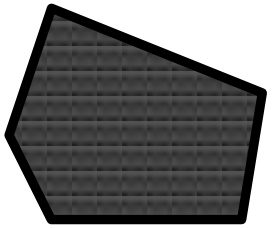
degenerate



concave



self-intersecting



filled

Axis-Aligned Rectangles

This one is easy:

```
for (y = yBot; y <= yTop; y++) {
    for (x = xLeft; x <= xRight; x++)
        setPixel(x, y);
}
```

but not all polygons are axis-aligned rectangles!

Flood Fill

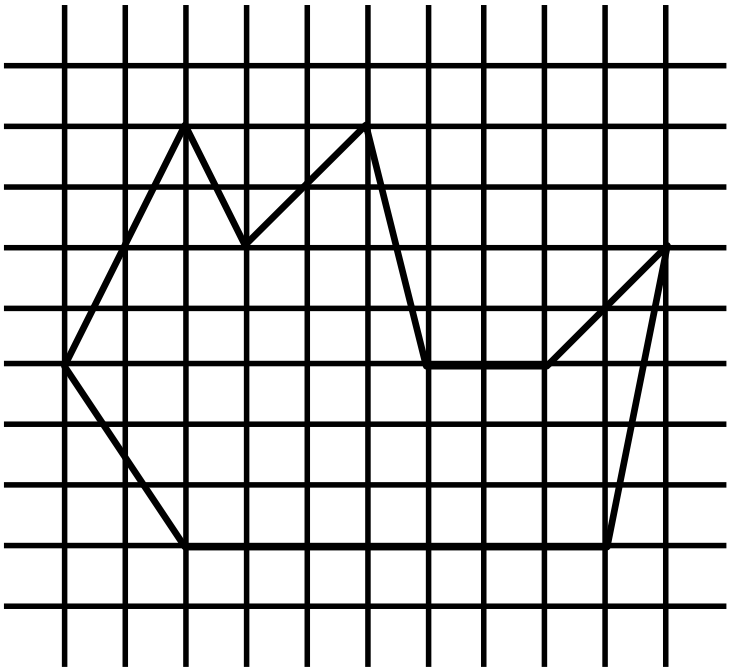
The classic pixel-based fill algorithm, it fills a region of value `old` bounded by regions of a different value:

```
void flood(x, y, old, new)
{
    if (pixel[x][y] == old) {
        pixel[x][y] = new;
        flood(x-1, y, old, new);
        flood(x+1,y, old, new);
        flood(x, y-1, old, new);
        flood(x, y+1, old, new);
    }
}
```

- easy to remember
- naïve: depends on coloring conventions
- works with any non-degenerate, non-self-intersecting polygon
- recursive (hard to do in hardware!)

An Arbitrary Polygon

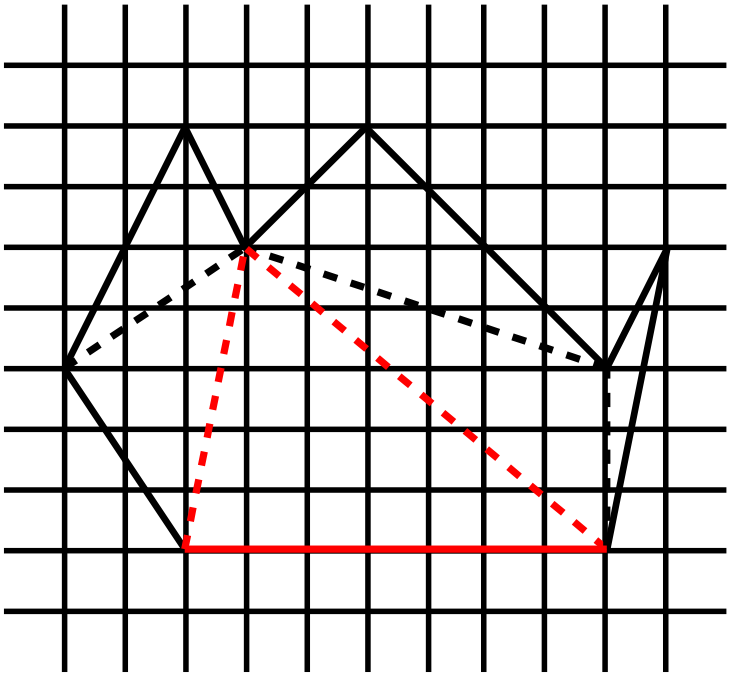
The general problem looks like this:



This is a (what kind of?) polygon. There are several algorithms for filling a concave polygon, but they're compute intensive, so...

A Tessellated Polygon

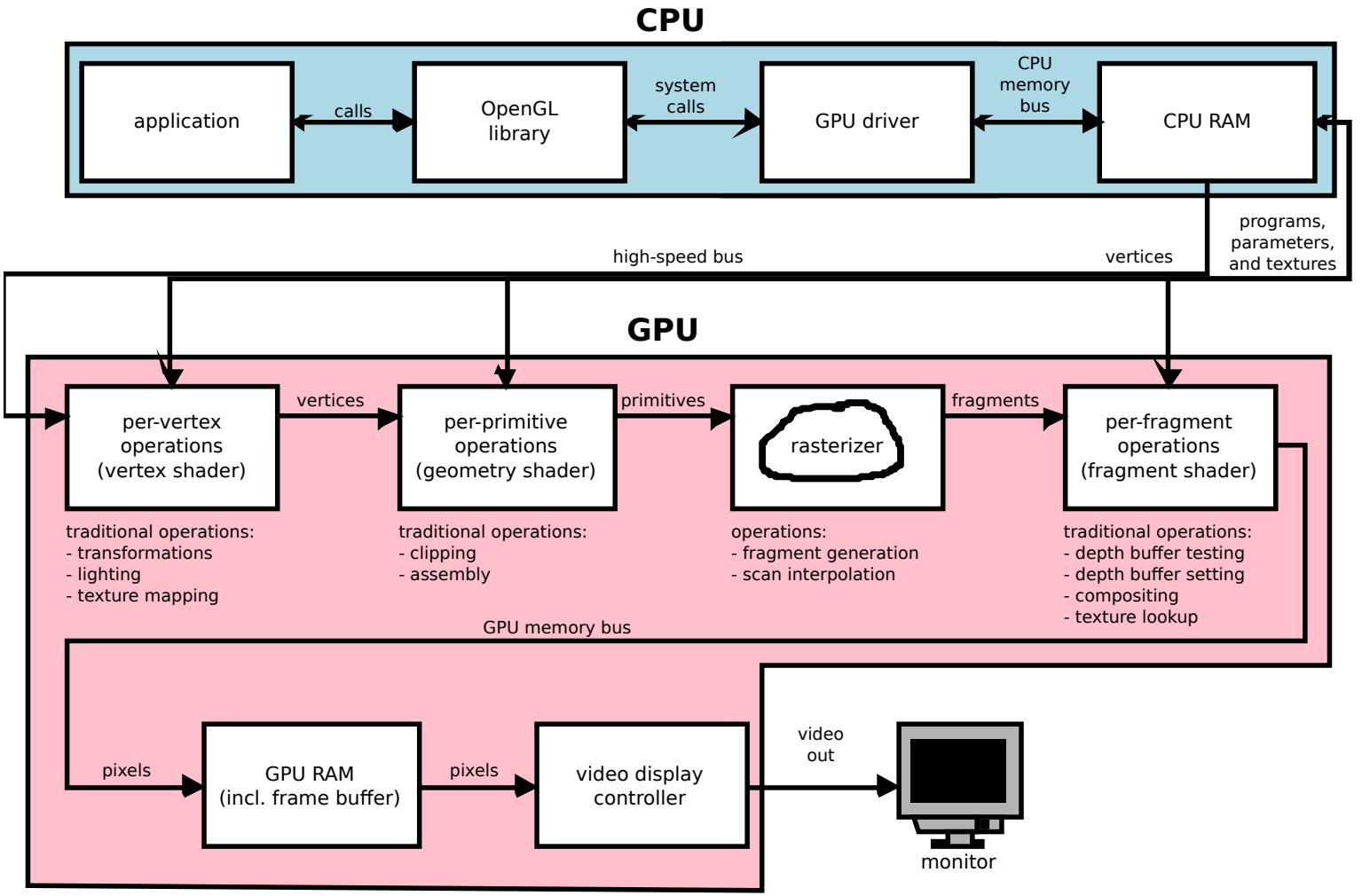
If necessary (and it often isn't), we tessellate the polygon up into triangles:



So let's "fill" (or "scan convert" or "rasterize") a single (red) triangle. (Note: "divide-and-conquer")

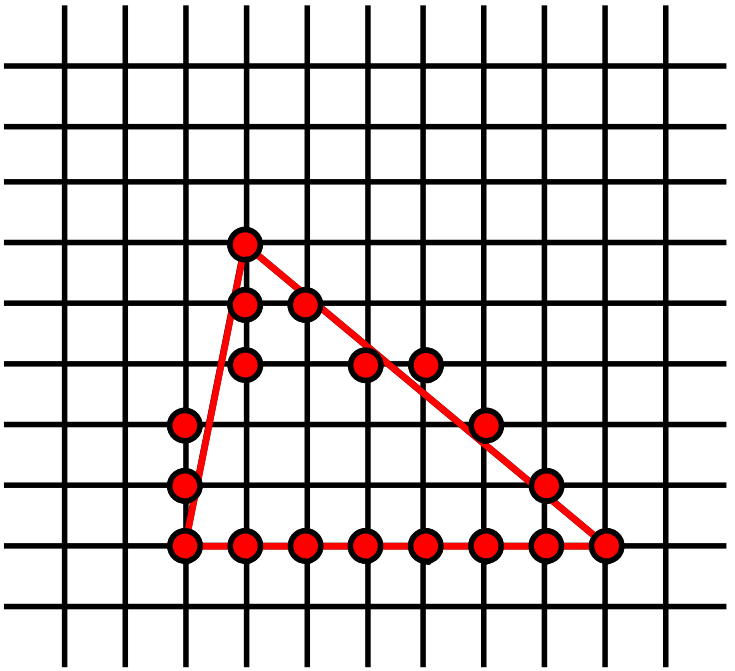
Filling Triangles

Recall: Where in the rendering pipeline do triangles get filled?



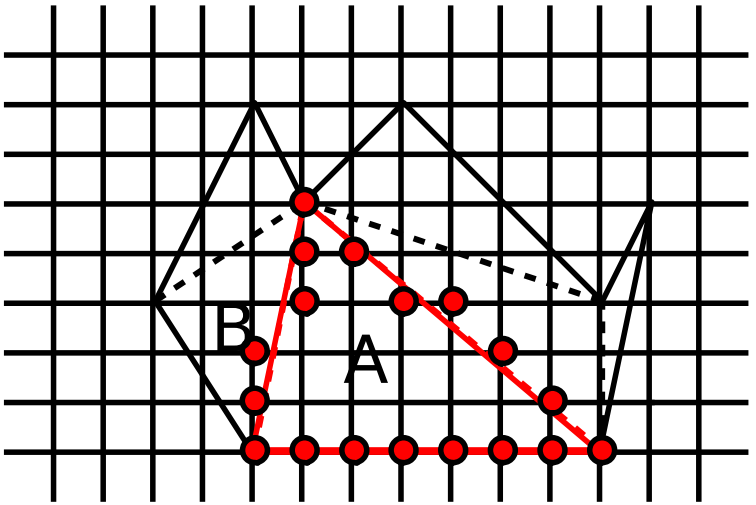
First Attempt I

Let's use something we already know how to do: use the midpoint algorithm to draw edges.



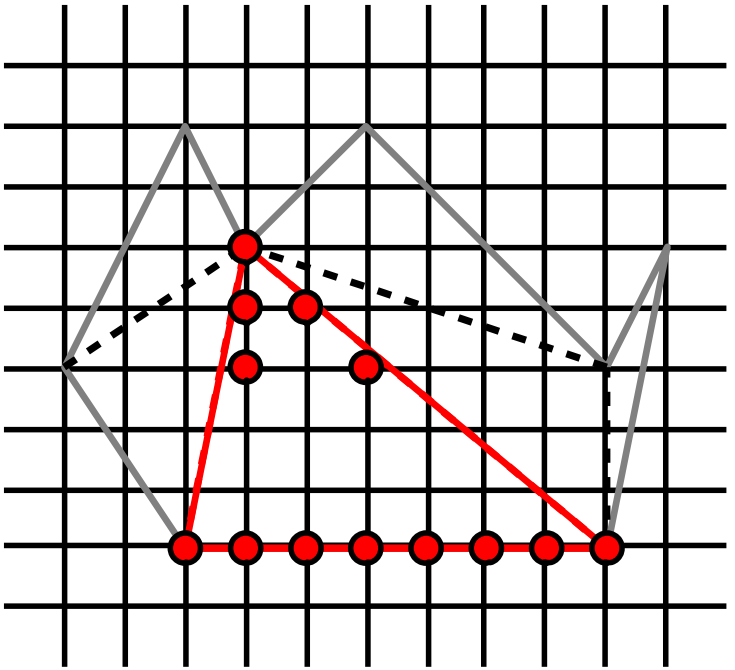
First Attempt II

But this causes problems with adjacent triangles:



First Attempt III

So, what if we only set line pixels that lie on the inside?



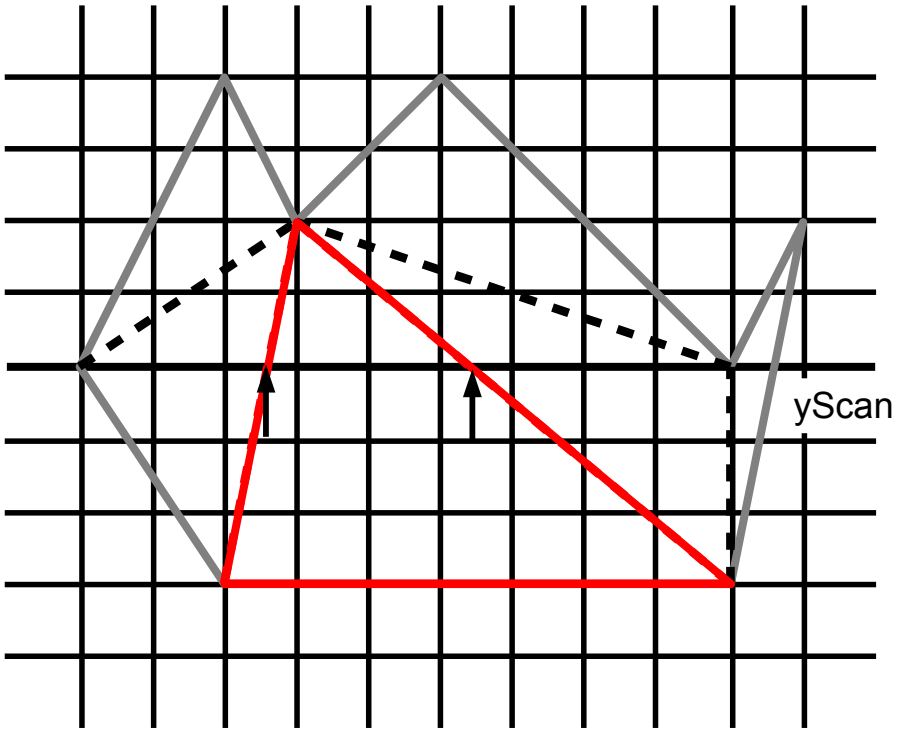
This is getting cumbersome and ugly and we don't have a solution for interior pixels.

Let's look at the problem again...

Processing a Scan Line I

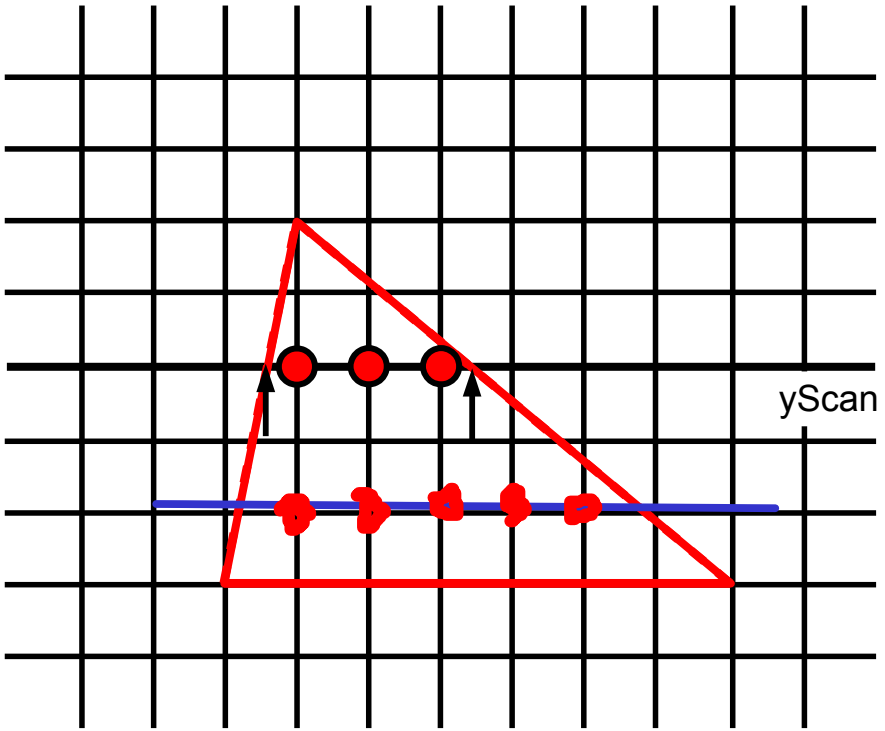
Instead of looking at vertices, let's consider a single scan line:

- Identify intersections.
- Sort (i.e. for a triangle, swap) them in increasing x order.



Processing a Scan Line II

- Find entry and exit points and fill in interior pixels.



- Then do this for all the other scan lines.

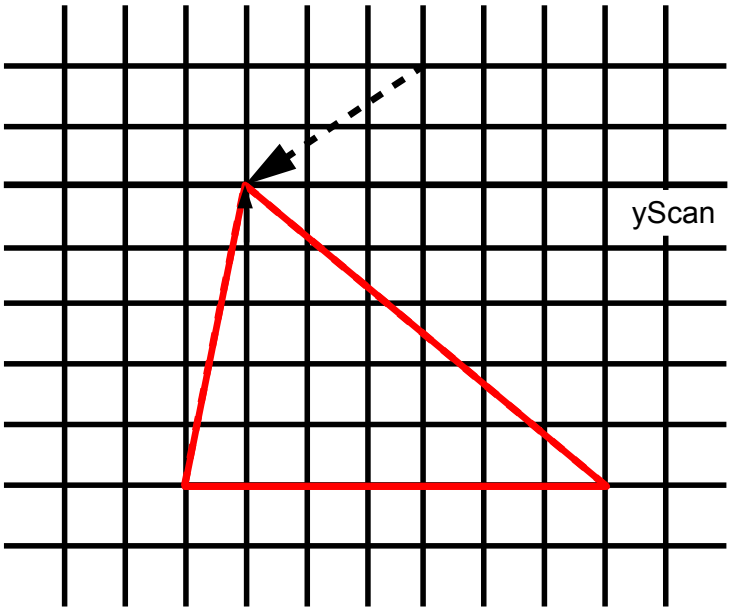
A First Attempt at a Scan Line Algorithm

```
for each scan line yScan:
    A = [] # set of intersections
    for each edge crossing yScan:
        x = x coordinate of yScan intersection
        insert x in A in increasing order
    if A (= [xL, xR]) is not empty:
        fill intervening pixels
        # ...which are on integer coordinates
        #    [ ceil(xL)...floor(xR) ]
```

This is a good start, but it's not very efficient. (Why?)

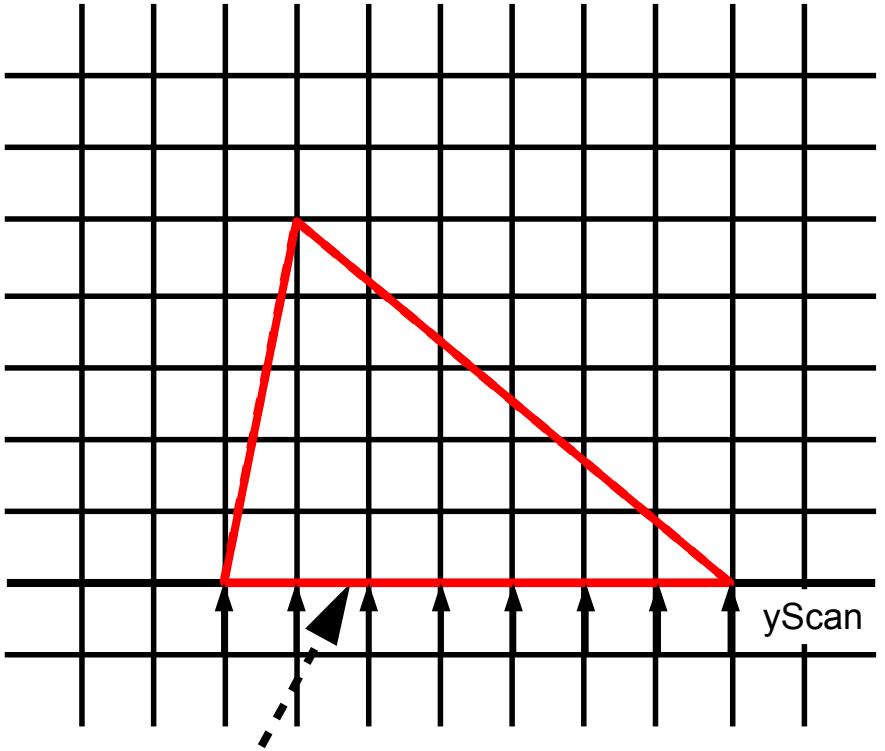
And What About This Case?

We need to determine when we're inside the triangle. To do this, we count crossings. If even: we're outside. If odd: we're inside.



Is there a scan line crossing here or not?

And This One?



Does this edge cross the scan line or not?

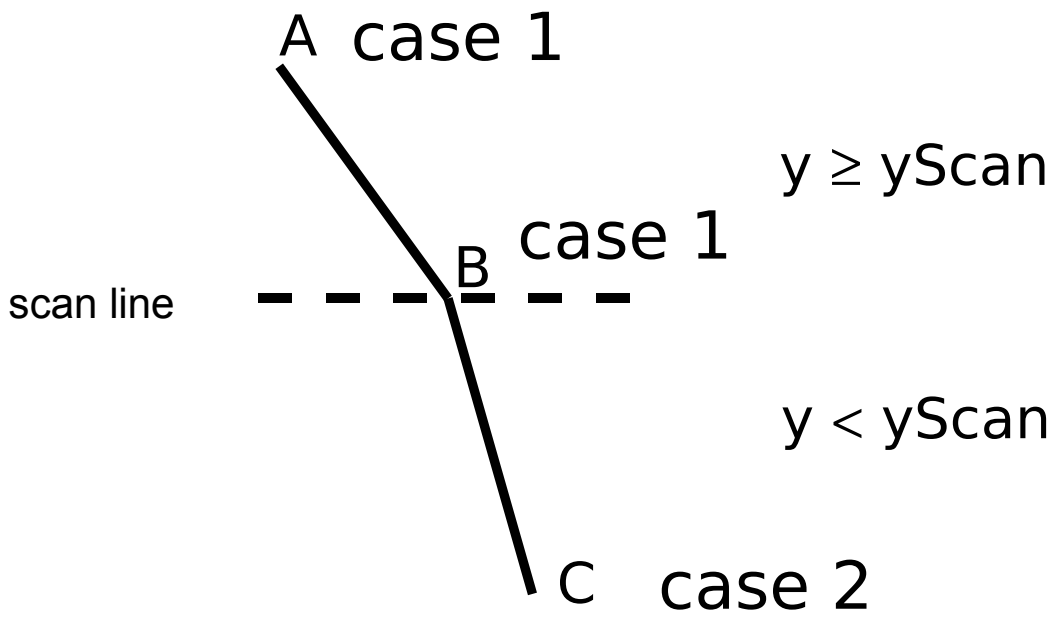
Solution: Watch Scan Line Crossings Closely

Although it looks like there are three cases to deal with (`y > yScan`, `y = yScan`, and `y < yScan`), it's much easier to deal with two:

- Case 1: `y >= yScan`
- Case 2: `y < yScan`

The line is crossed when we go from one case to the other. (Note: `y > yScan` and `y <= yScan` also work!)

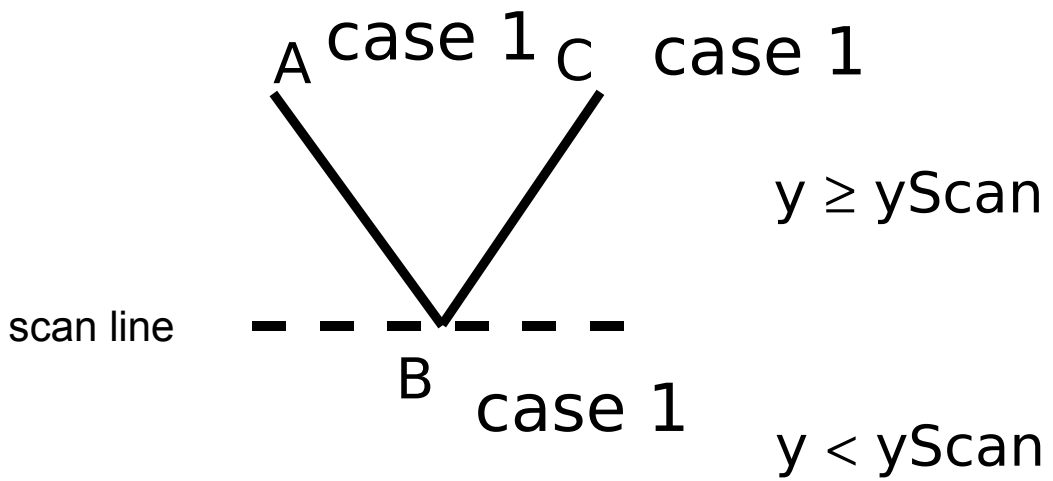
Scan Line Crossing Example 1



AB does not cross the scan line, but BC does, resulting in one intersection.

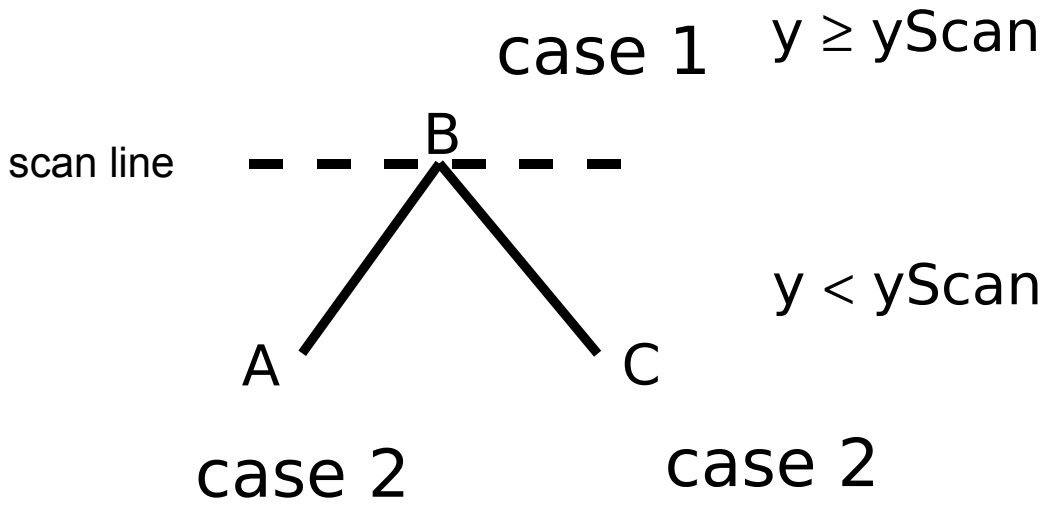
Is this what we want to have happen?

Scan Line Crossing Example 2



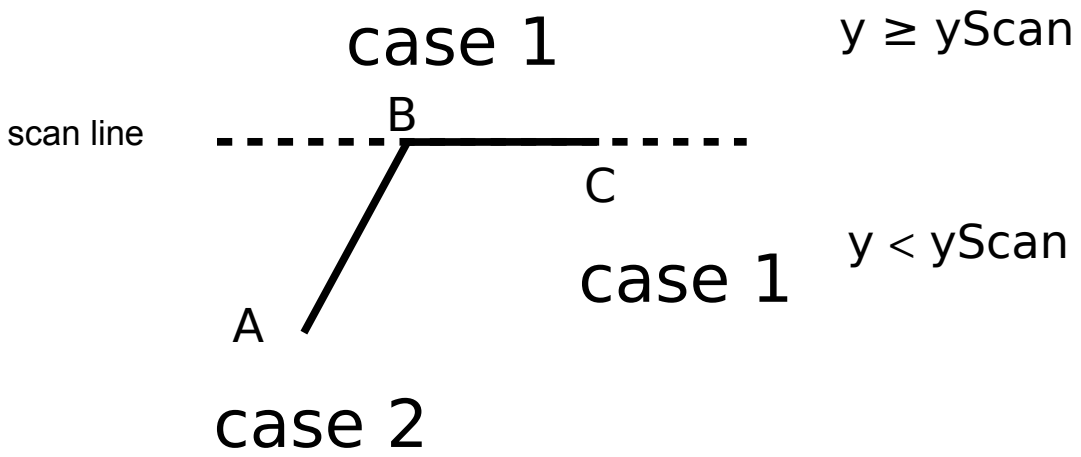
Neither AB nor BC cross the scan line, resulting in no intersections. Is this okay?

Scan Line Crossing Example 3



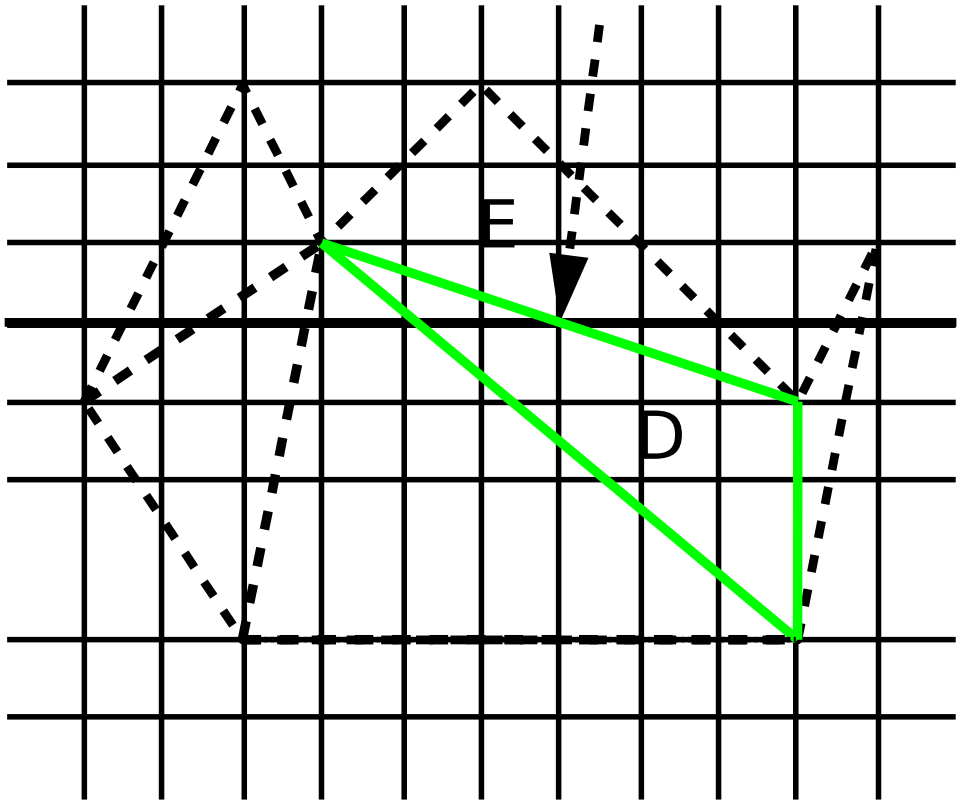
Both AB and BC cross the scan line, resulting in two intersections. How's this?

Scan Line Crossing Example 4



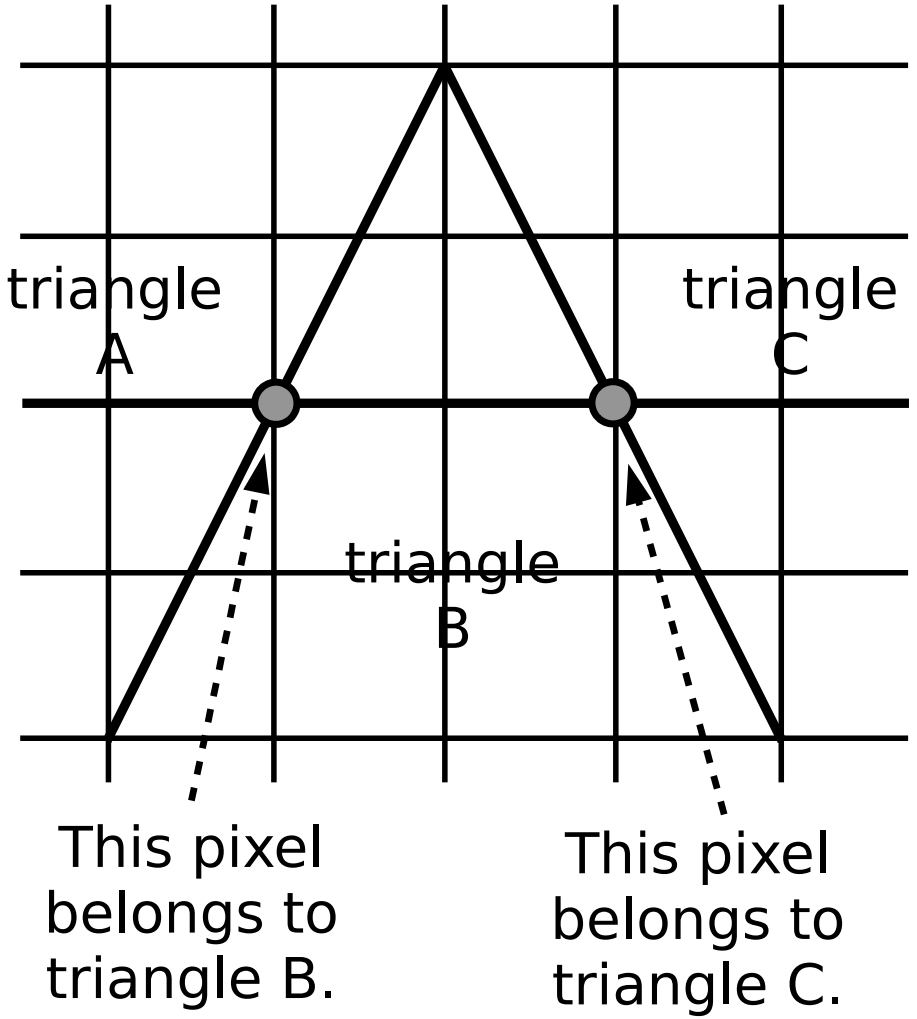
Well?

And How About These?



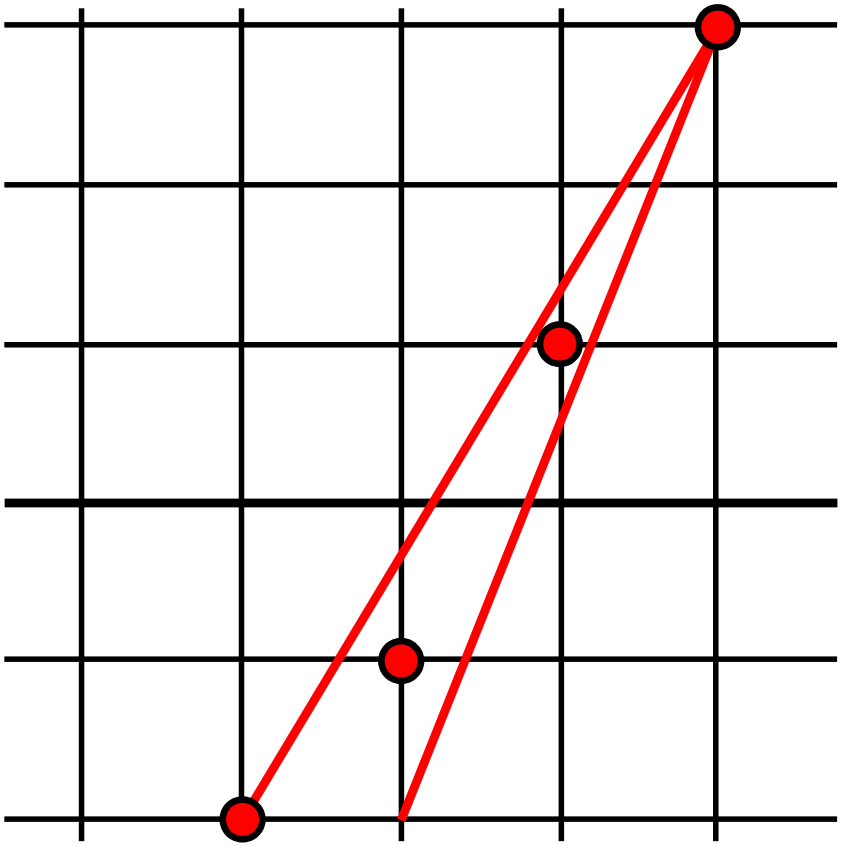
Who gets intersections that fall exactly on pixels?

Solution: Devise a *disambiguating rule*



Going from left to right, if you're entering a triangle, the pixel is interior. If you're leaving a triangle, the pixel is exterior.

Aside: Watch out for "Slivers"



Aliasing ("the jaggies") strikes again!

Improving Fill Efficiency

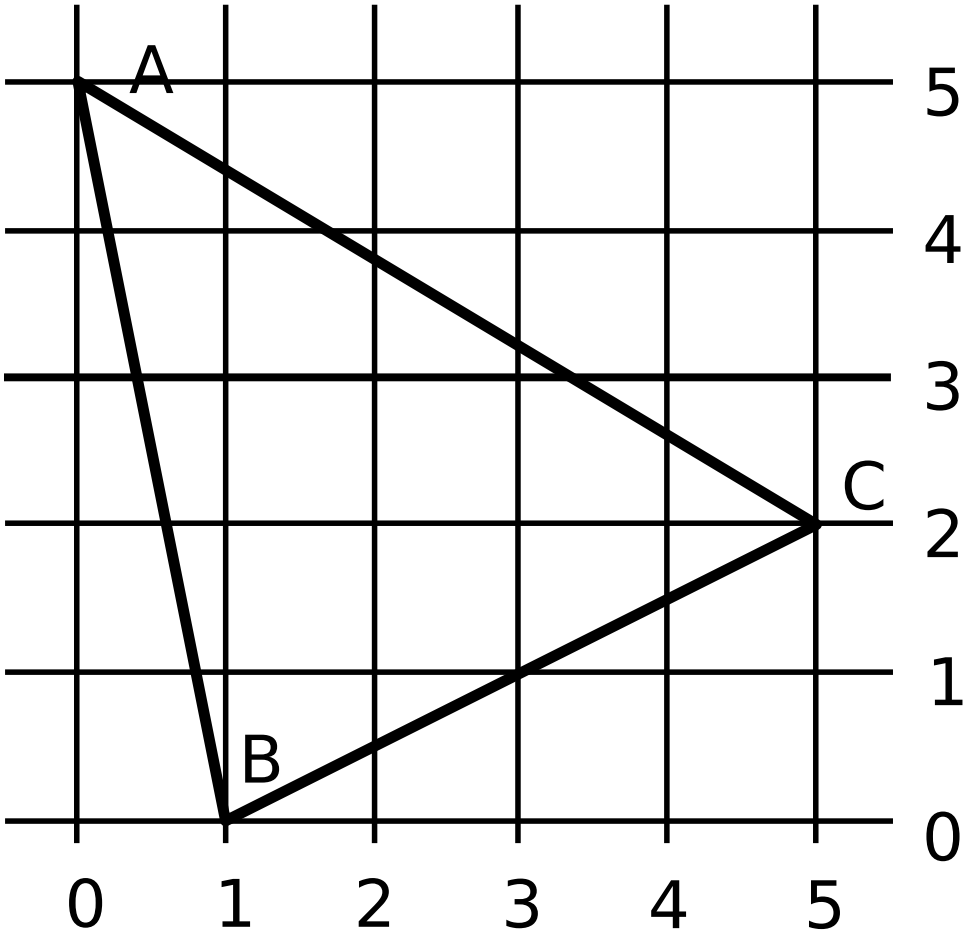
For each scan line, we need to recompute x_L and x_r and figure out if we leave or enter any new edges. Can we do this more efficiently? Insight: Use *edge coherence*

1. Many edges intersected by scan line i are also intersected by scan line $i+1$.
2. If we are scanning a line $y = mx + b$, then if x_i is the x-coordinate of the intersection on line i,

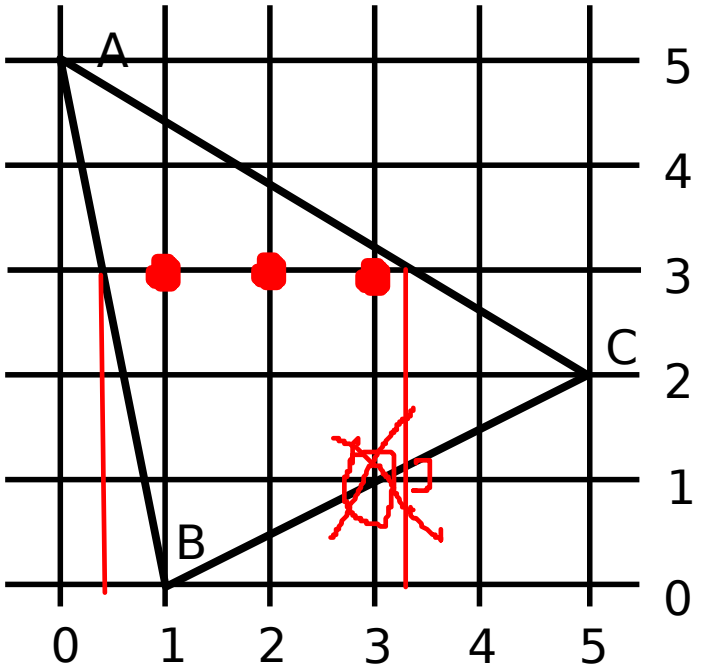
$$\begin{aligned} \Delta y &= m\Delta x \\ 1 &= m(x_{i+1} - x_i) \\ x_{i+1} &= x_i + \frac{1}{m} \end{aligned}$$

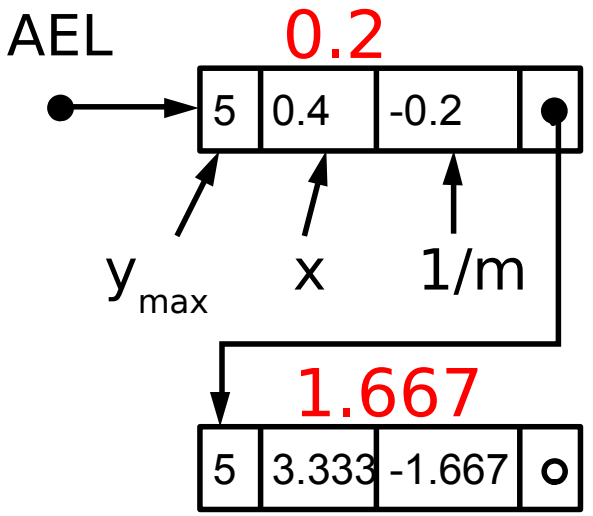
3. What if the line is horizontal?

Typical Triangle



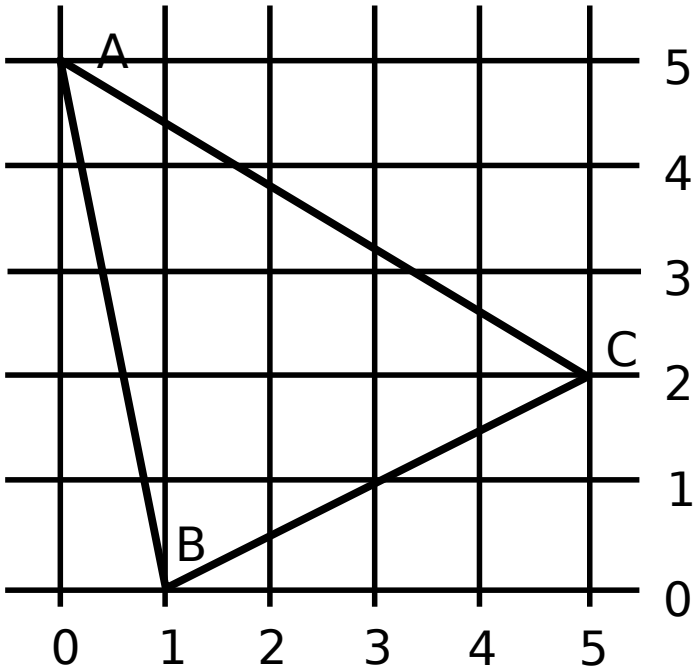
The Active Edge List

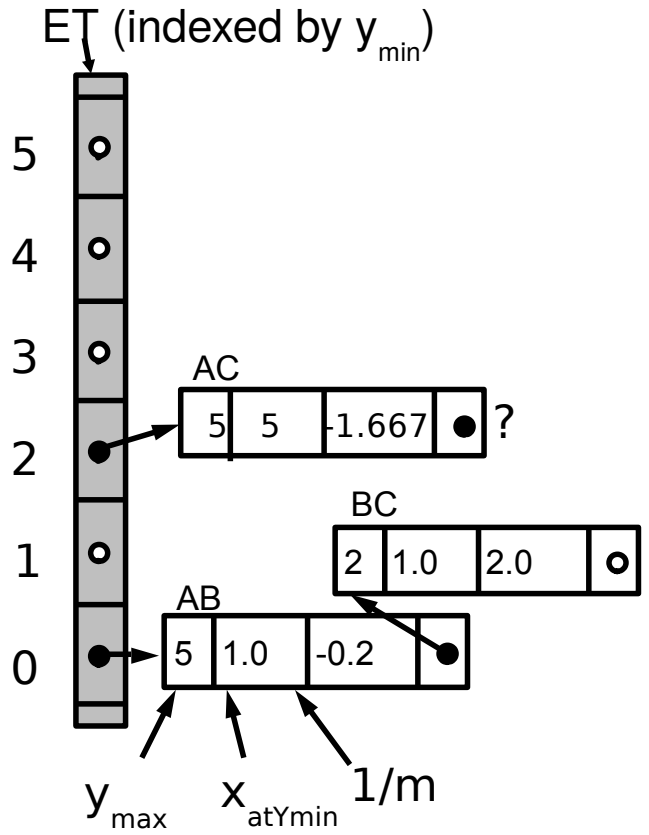




We could use a 2-element array for this, but this algorithm can actually work for multiple triangles, although we only show one here.

The Edge Table





x_atYmin is the value of x at y_min

The Active Edge Algorithm

```
{Build edgeTable}
y = {the smallest ymin in edgeTable}
activeEdgeList = []
while activeEdgeList != [] or edgeTable != []:
    mergeSortByX(activeEdgeList, all edges in edgeTable[y])
    scanConvert(activeEdgeList)  # note: be careful here!
    delete(activeEdgeList, all edges whose ymax = y)
    y = y + 1
    for each element i in activeEdgeList:
        x[i] = x[i] + 1/m[i]
```

Edge Table Issues

What if we have multiple triangles?

- Easy: Keep colors in each AEL and ET element. As you scan convert, change the color accordingly.

What if the polygons overlap?

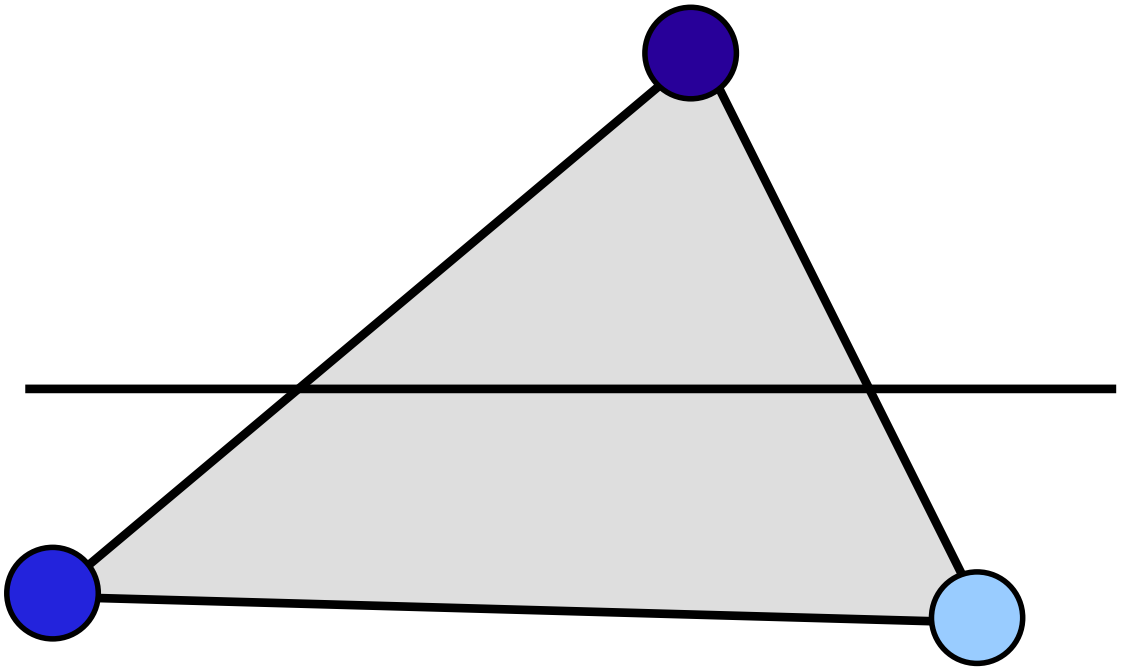
- Assign a priority to each polygon and use this to determine whether the color of the scanning pixel changes.

What if the polygons have depth? (i.e., they are 3D)

- From plane equation of polygon, compute depth at each pixel and compare either with:
  - all other triangles that pixel is inside, or
  - a current "closest" pixel (actually, fragment) distance (this is OpenGL's depth test)

Smooth Shading I

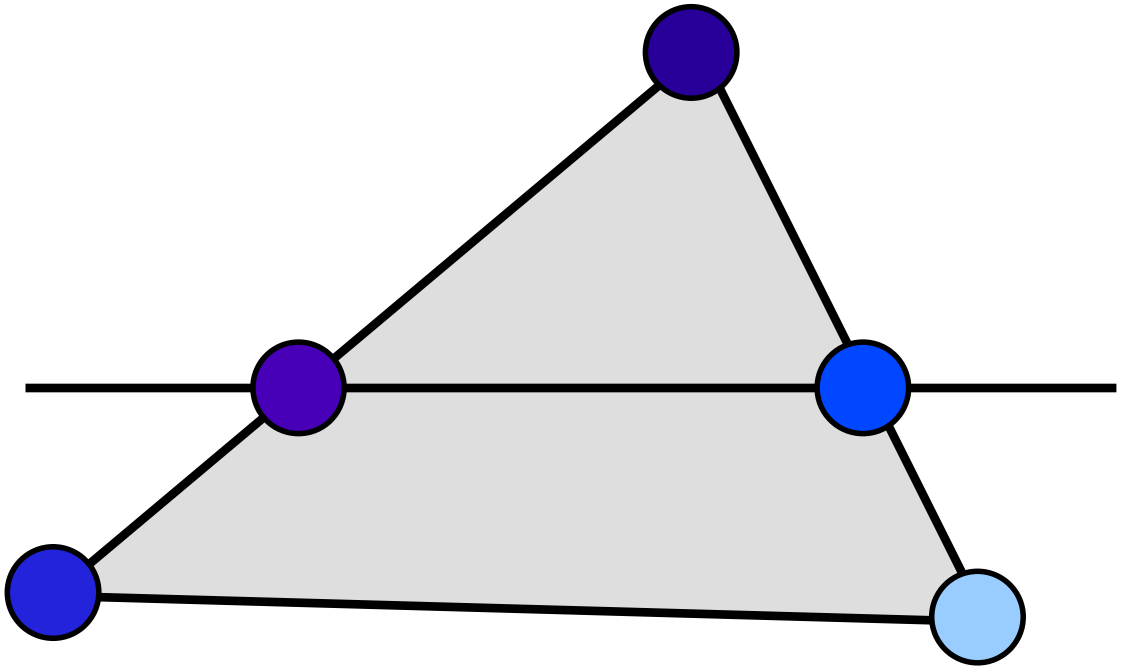
Traditionally, colors are bound to vertices:



but we can assign *different* colors to each vertex of a triangle.

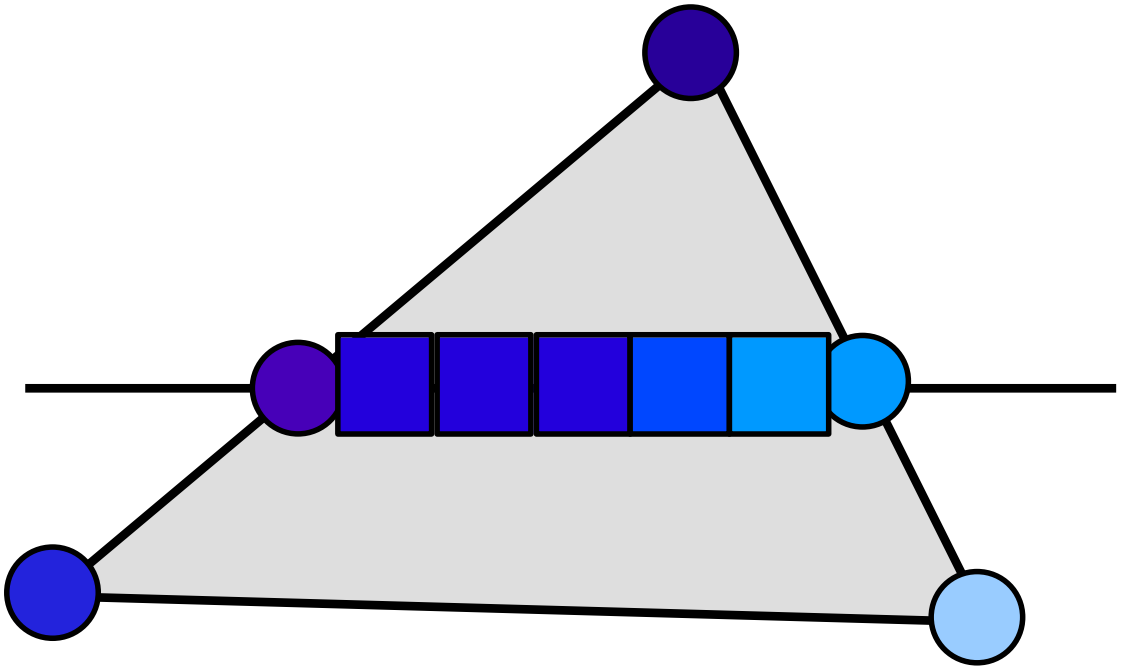
Smooth Shading II

Enhance the polygon fill (i.e., the active edge) algorithm to linearly interpolate (in RGB space) edge colors at scan endpoints:



Smooth Shading III

We then linearly interpolate endpoint colors along the scan line pixel-by-pixel.



(Squares indicate pixels drawn during scan conversion.)

Smoothly Shaded Polygons

We can compute colors at the vertices of polygons and interpolate them over the whole polygon.

Aside: Text Primitives

Attributes:

- Font (e.g. Helvetica, Freeform, Times, Courier, lots more)
- Face (normal, bold, italic, bold italic, underlined)
- Point Size (e.g., 6 point, 36 point) (What's a point?)
- Aliased/Antialiased (may use transparency)

Representations:

- bitmap
  - done with pixel operations (with transparency)
  - fast
- stroke
  - done with vertex operations (i. e., polygon filling)
  - easier to scale, rotate
  - characters modelled with curves, then tessellated into (concave, often) polygons and then into triangles.