Lecture #04

Polygon Rasterization

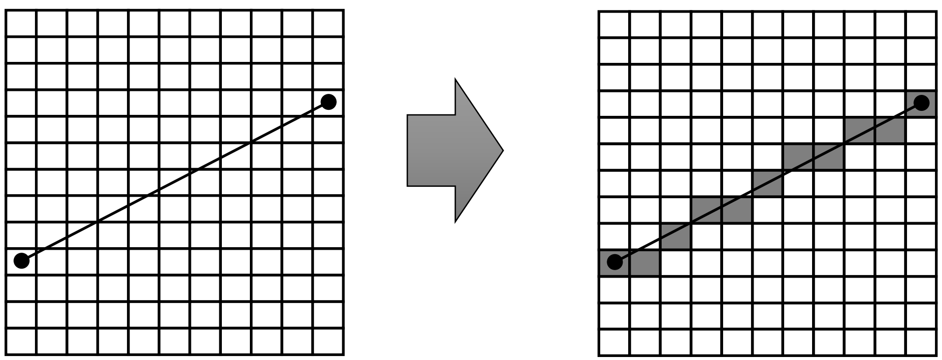
Computer Graphics
Winter Term 2020/21

Marc Stamminger / Roberto Grosso

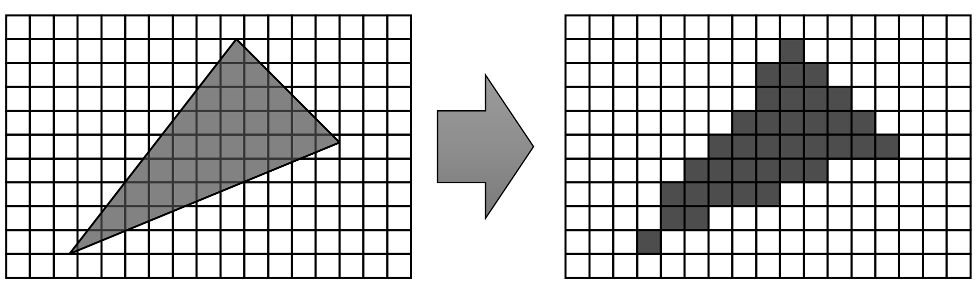
What is Rasterization ?

- Given a primitive, find the pixels that cover this primitive

- Line primitive:

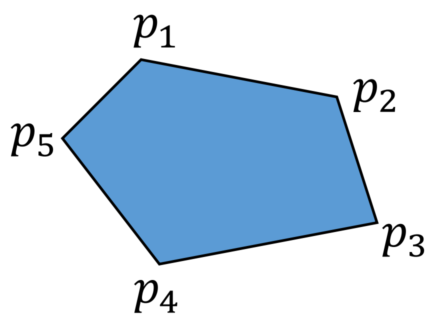


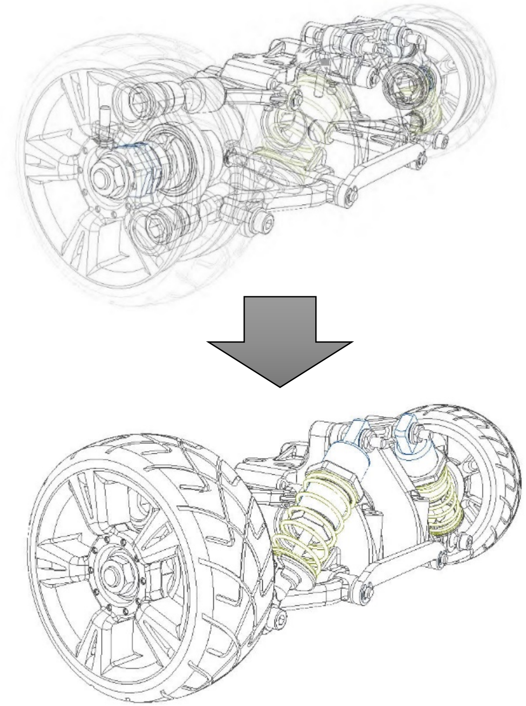
- Triangle primitive:



Rasterization - Primitives

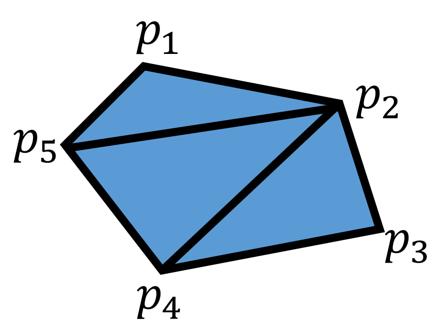
- mostly, we want to **fill** objects → **polygons**
- A **polygon** is defined by an ordered set of points (for now in 2D)

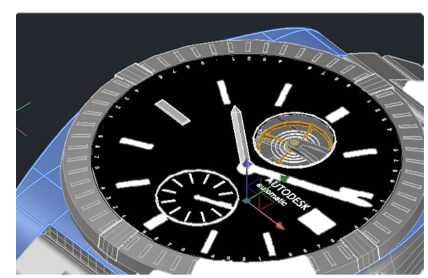




autodesk.com

- Every 2D shape can be approximated by a polygon
- Every 2D polygon can be split into **triangles** = **Triangulation**
- we use triangles as primitives, sometimes also polygons

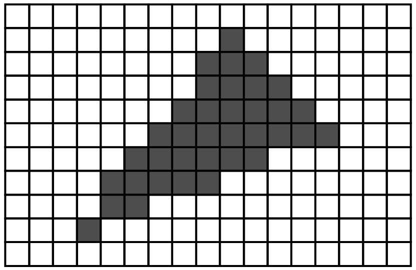




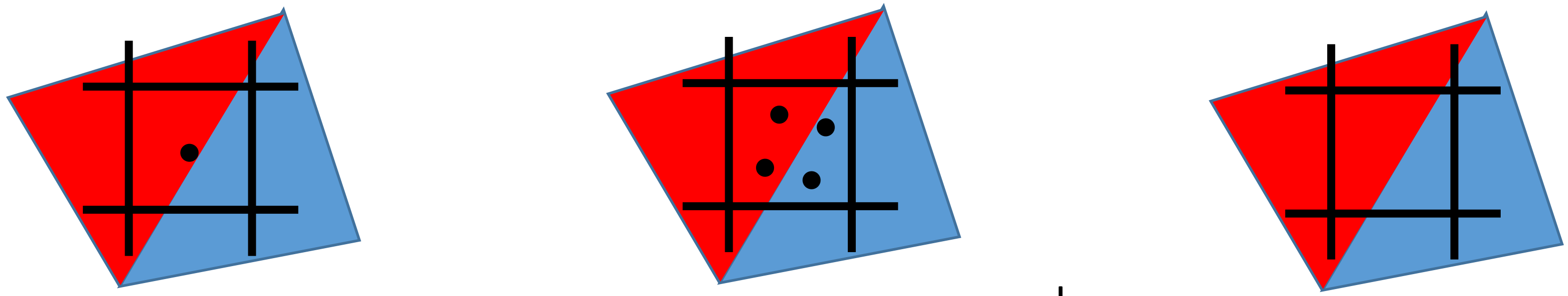
autodesk.com

Rasterization – Aliasing and Antialiasing

- For now: set pixel if its center is inside the shape
  → strong jaggies, well visible
  → this is one form of **Aliasing**
  → we will come back to aliasing later



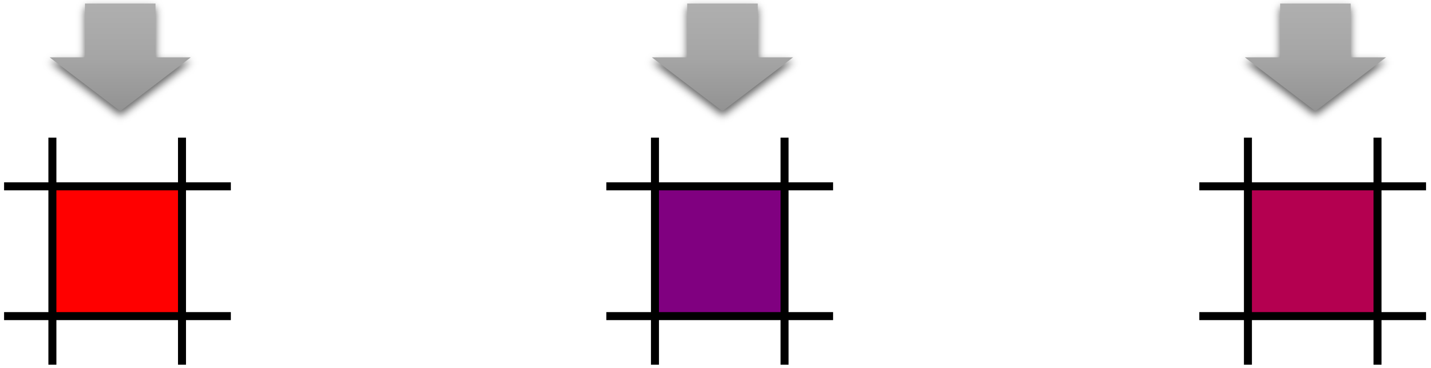
- Other rasterization rules:



look at pixel's center

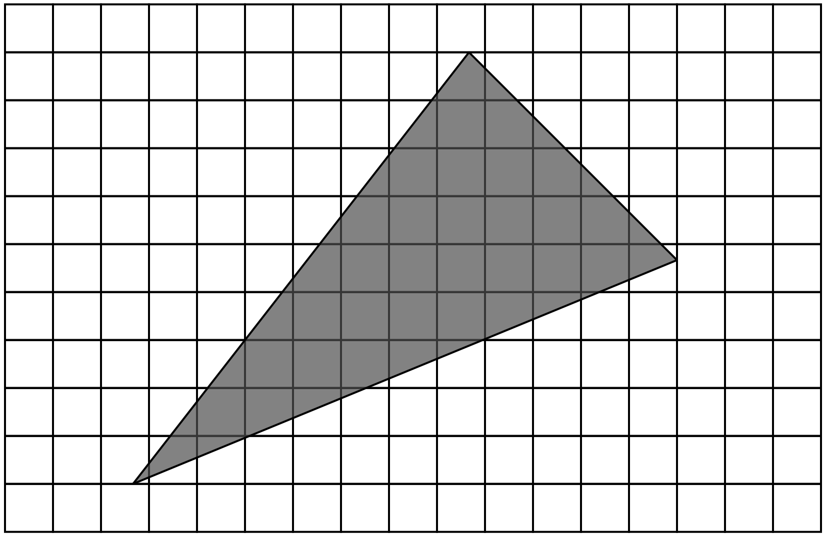
average over some sample positions within pixel

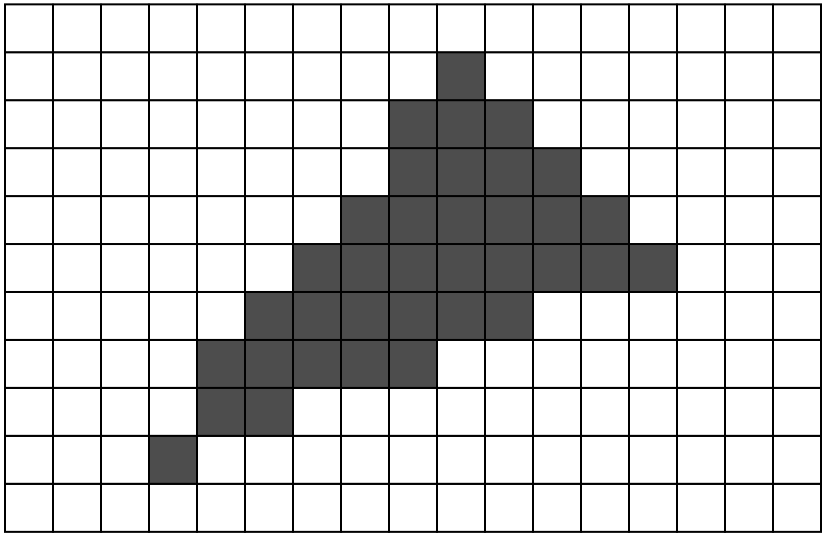
compute coverage



Polygon Rasterization

- Problem statement
  - Given a 2D-polygon with n vertices $P_1, \dots, P_n$
  - Color all pixels with center inside the polygon





Seed-Fill Algorithm

- Idea 1: rasterize boundary, fill interior → **seed fill algorithm**

- Rasterize boundary as seen before
- To fill, start at one point (seed), e.g. the center of a triangle
  - Set it to fill color
  - look at neighbor pixels:
    if not set, call seed fill for these pixels recursively

- Recursive algorithm → BAD ☺

Seed-Fill Algorithm

- Recursive algorithm

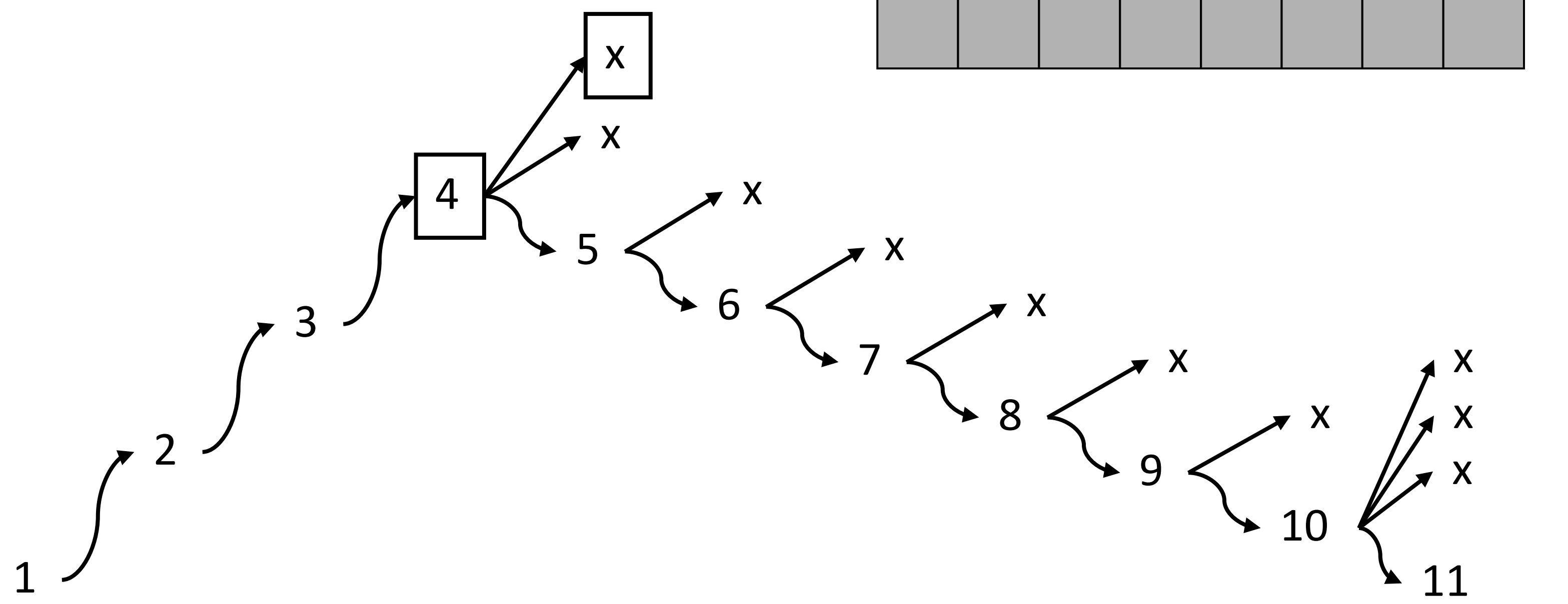
```
seedfill (x,y,fillcolor)
    if (color(x,y) == fillcolor)
        return; //boundary reached or fillcolor already set
    color(x,y) = fillcolor;
    seedfill(x+1,y);     //right
    seedfill(x-1,y);     //left
    seedfill(x,y+1);     //up
    seedfill(x,y-1);     //down
```

- Cons: Very deep recursion possible (requires large stack), rather inefficient

Seed-Fill Algorithm

- Example
  - 1: seed point
  - Recursion tree

	10	9	8	7	6	5	
	11	12	1	2	3	4	
	18	13	14	15	16	17	



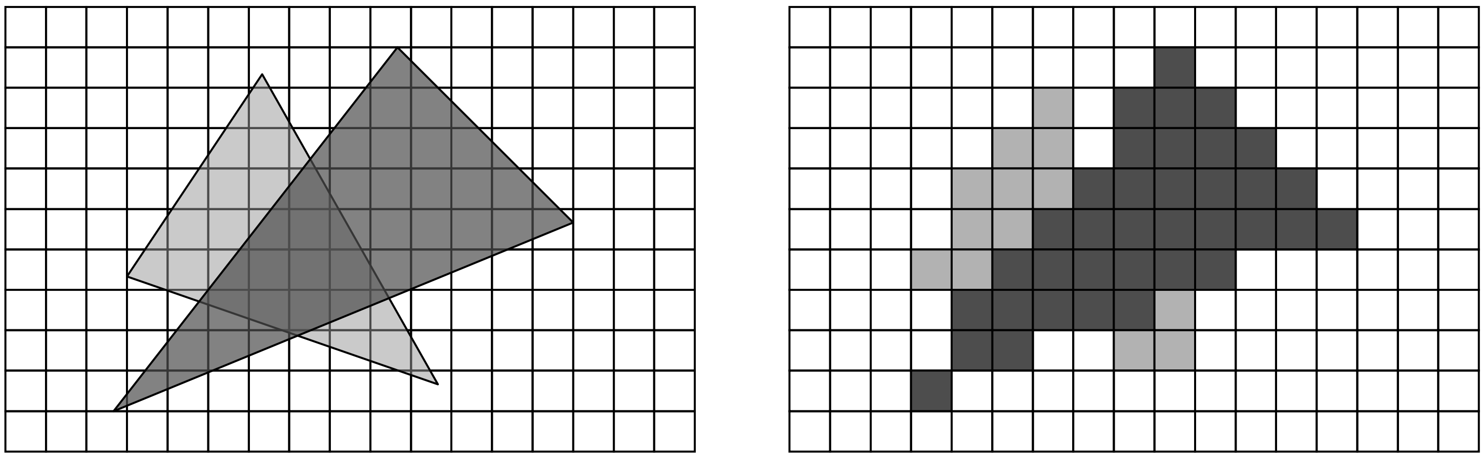
Seed-Fill Algorithm

- Apply for Polygon Rasterization:
  - Draw boundary of polygon using Bresenham **in unique color**
  - Pick a point inside
  - Do seed fill from this point with this unique color
  - Replace unique color by desired one

- Evaluation for rasterization of polygons
  - Single color only (no shading, see later)
  - How to find seed position?
  - Not very efficient !

Polygon Rasterization

- Better: directly find the pixels within a polygon

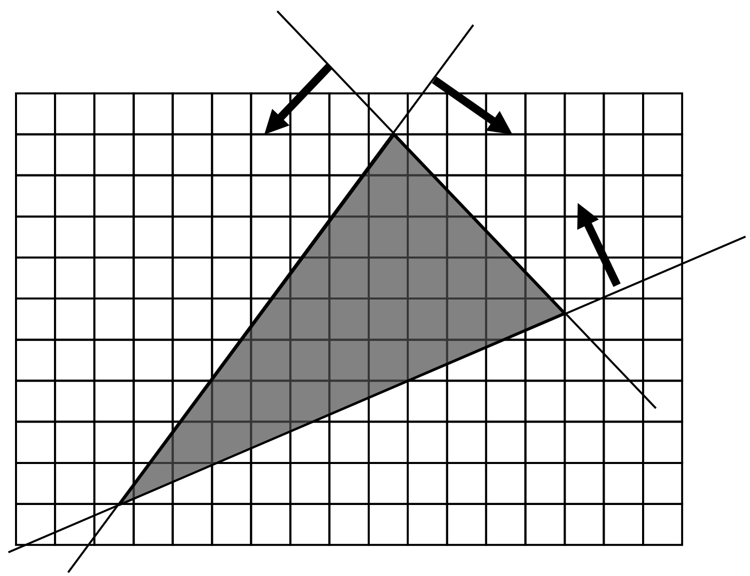


Triangle Test

- Brute force solution for triangles

```
for each pixel (x,y)
        for each edge E
                if (x,y) on wrong side of E
                        continue with next pixel
        set pixel (x,y)
```

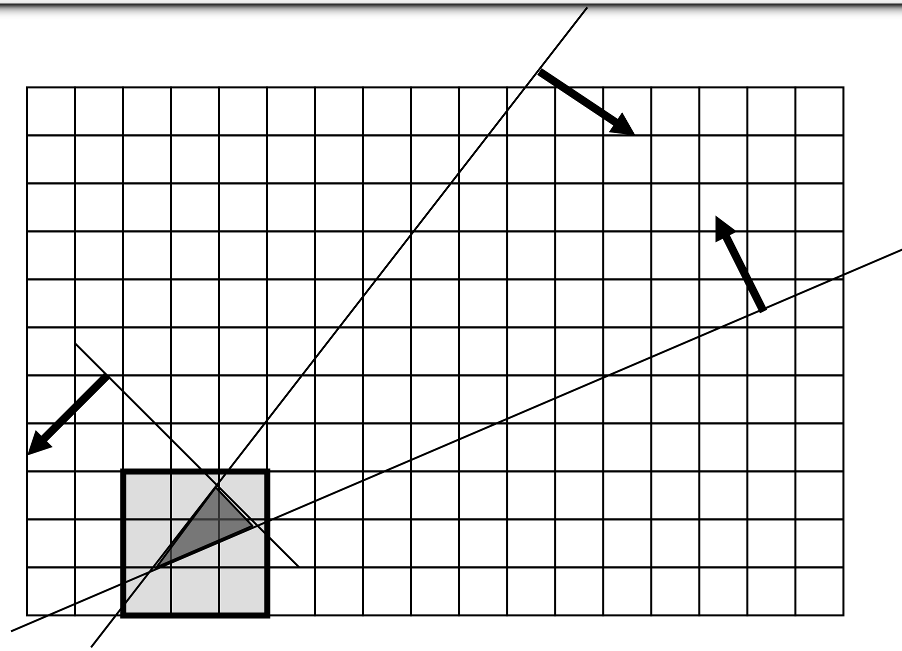
- very wasteful for small triangles



Triangle Test

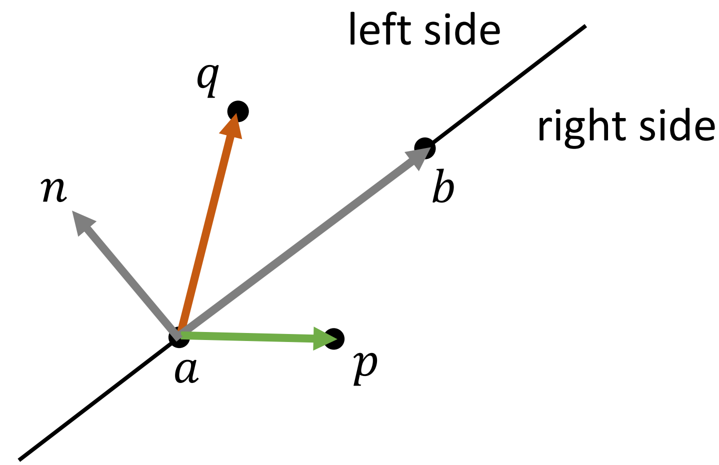
- Brute force solution for triangles
  - Improvement: Compute only for the screen bounding box of the triangle

```
for each pixel (x,y) in bounding box
        for each edge E
                if (x,y) on wrong side of E
                        continue with next pixel
        set pixel (x,y)
```



$X_{min}, X_{max}, Y_{min}, Y_{max}$ of the triangle vertices

Triangle Test



- Edge test:
  - ab defines direction and separates plane to “left” and “right” half
  - normal vector n defines these halves: $n = \begin{pmatrix} a_2 - b_2 \\ b_1 - a_1 \end{pmatrix}$ points to the left
  - edge test by using dot product:

$$p \text{ "left"} \Leftrightarrow (p - a) \circ n > 0 \Leftrightarrow p \circ a \; - a \circ n > 0$$

  - with homogeneous coordinates:

$$p \text{ "left"} \Leftrightarrow \underbrace{\begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}}_{\text{“edge” vector}} \circ \begin{pmatrix} a_2 - b_2 \\ b_1 - a_1 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

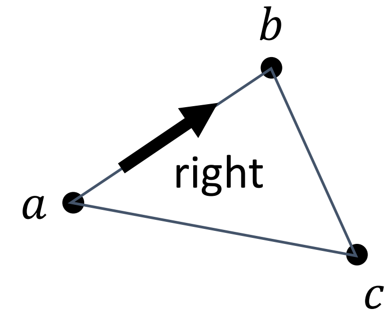
→ precompute and use within loop for fast test

Triangle Test

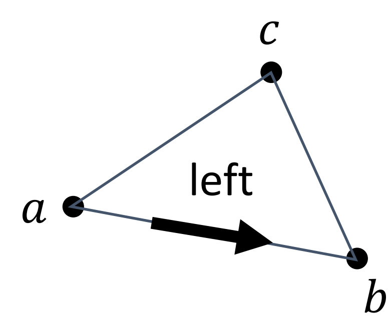
- Which is the "right" side ?
- Depends on orientation of triangle…

- Check orientation by computing determinant (see also transformations/reflections)
- $D = |b - a \quad c - a| > 0$
  → positive orientation
  → "left" is right

- We can also code this into the edge vector
  → simply negate edge vector in case of negative orientation

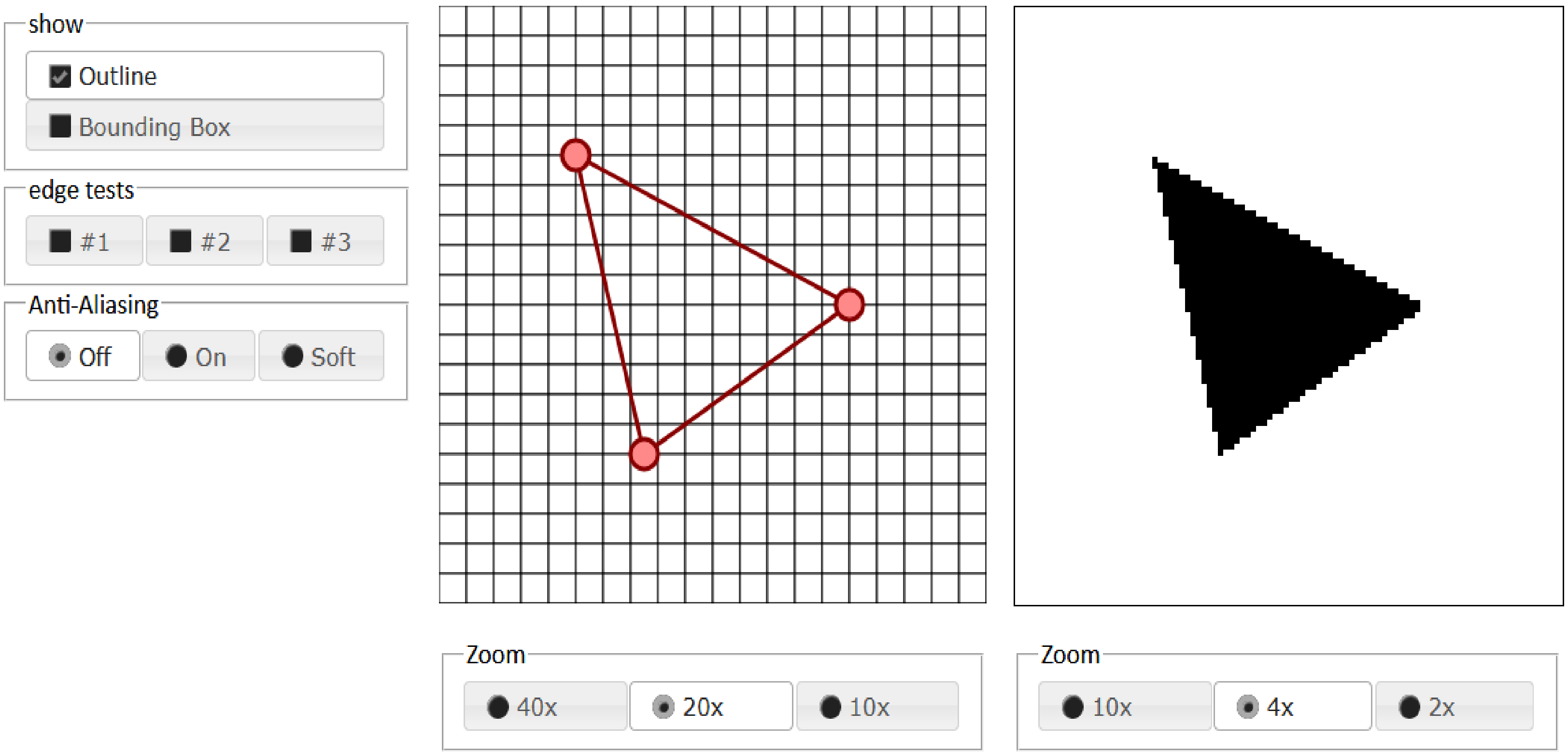


"negative" orientation
"clockwise"



"positive" orientation
"counterclockwise"

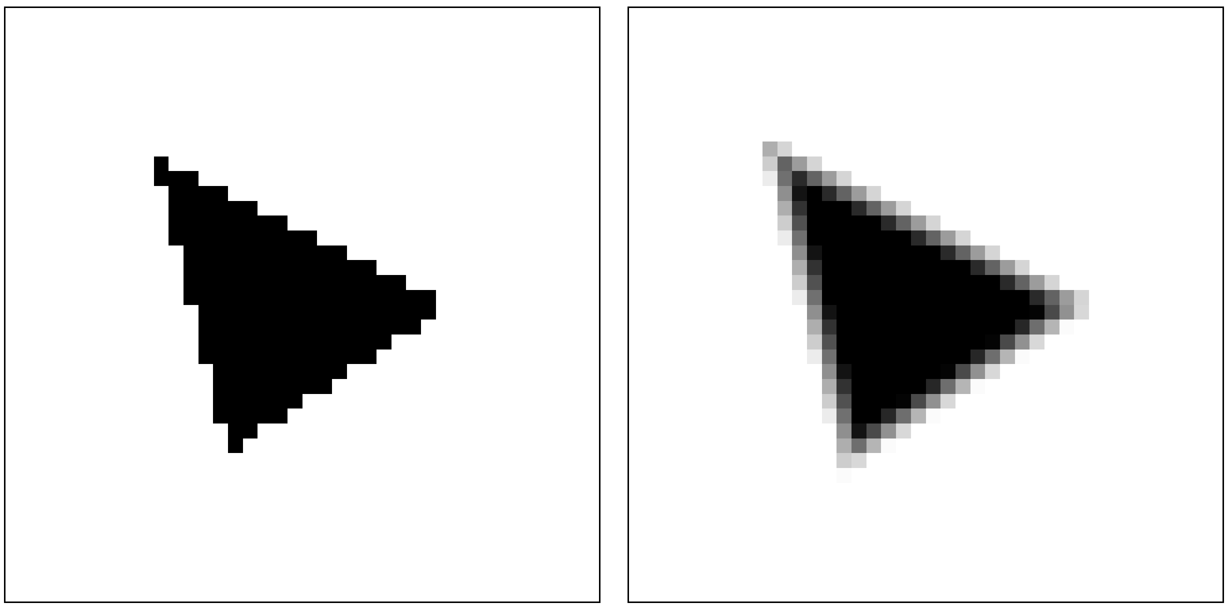
Triangle Test



- Edge test only tests “left” → does not work if orientation is changed (check!)

Triangle Test

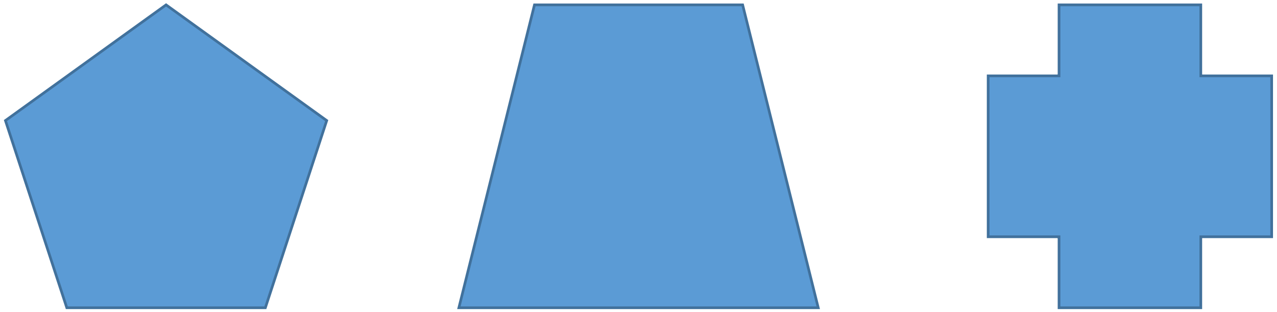
- normalize n in edge test → scalar product delivers the distance to the edge
- Can be used for anti-aliasing of triangle edges:



- How ?

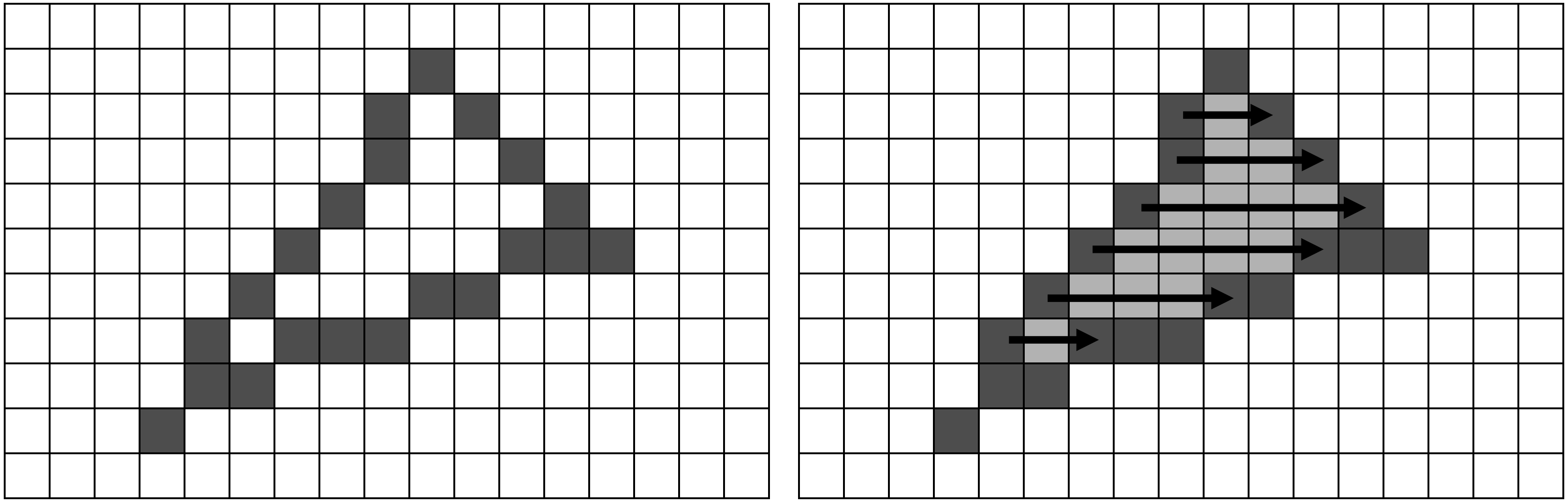
Arbitrary Polygons

- Does this test work for arbitrary polygons ?



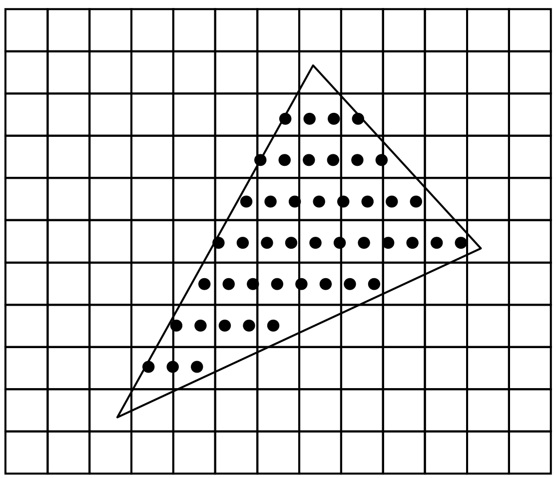
Polygon Rasterization

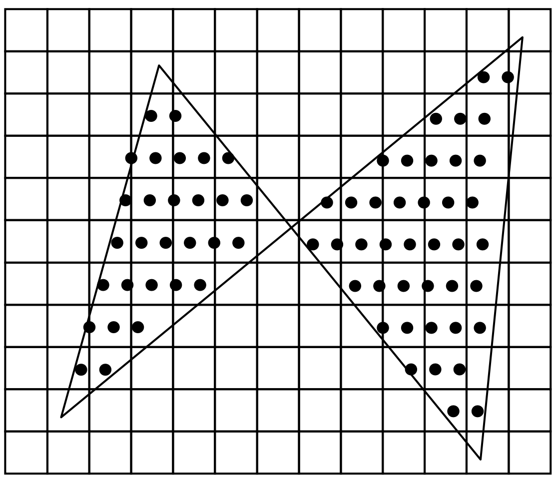
- Alternative idea: scanline rasterization



Scanline Algorithm

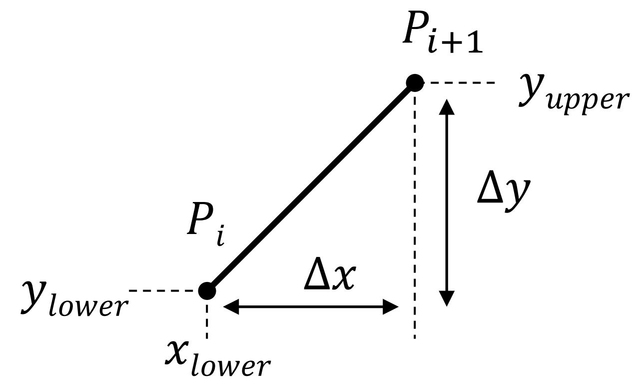
- Idea Scanline Algorithm
  - Proceed scanline by scanline from bottom to top
  - Find intersections of scanline with polygon
  - Fill these intersections





Scanline Algorithm

- Data Structures
  - Edge table (ET)
    - List of all polygon edges (upwards only!)
    - Content per edge
    - Linked list
    - Sorted by ylower
  - Note that 1/m is the x-increment when stepping to above scanline

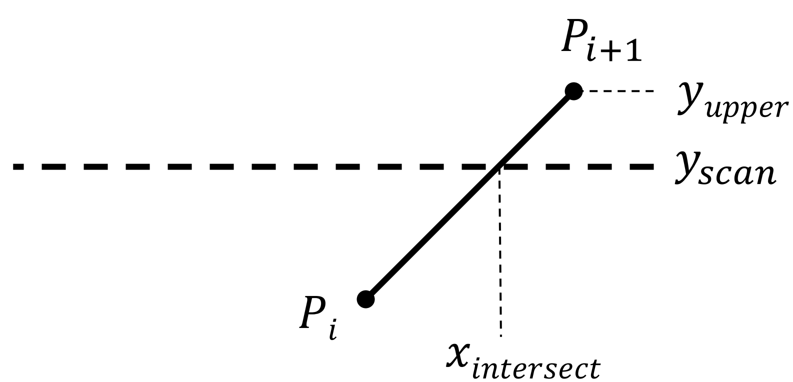


y_{lower}	x_{lower}	y_{upper}	$1/m = \Delta x/\Delta y$	$\bullet\!\longrightarrow$ next

Scanline Algorithm

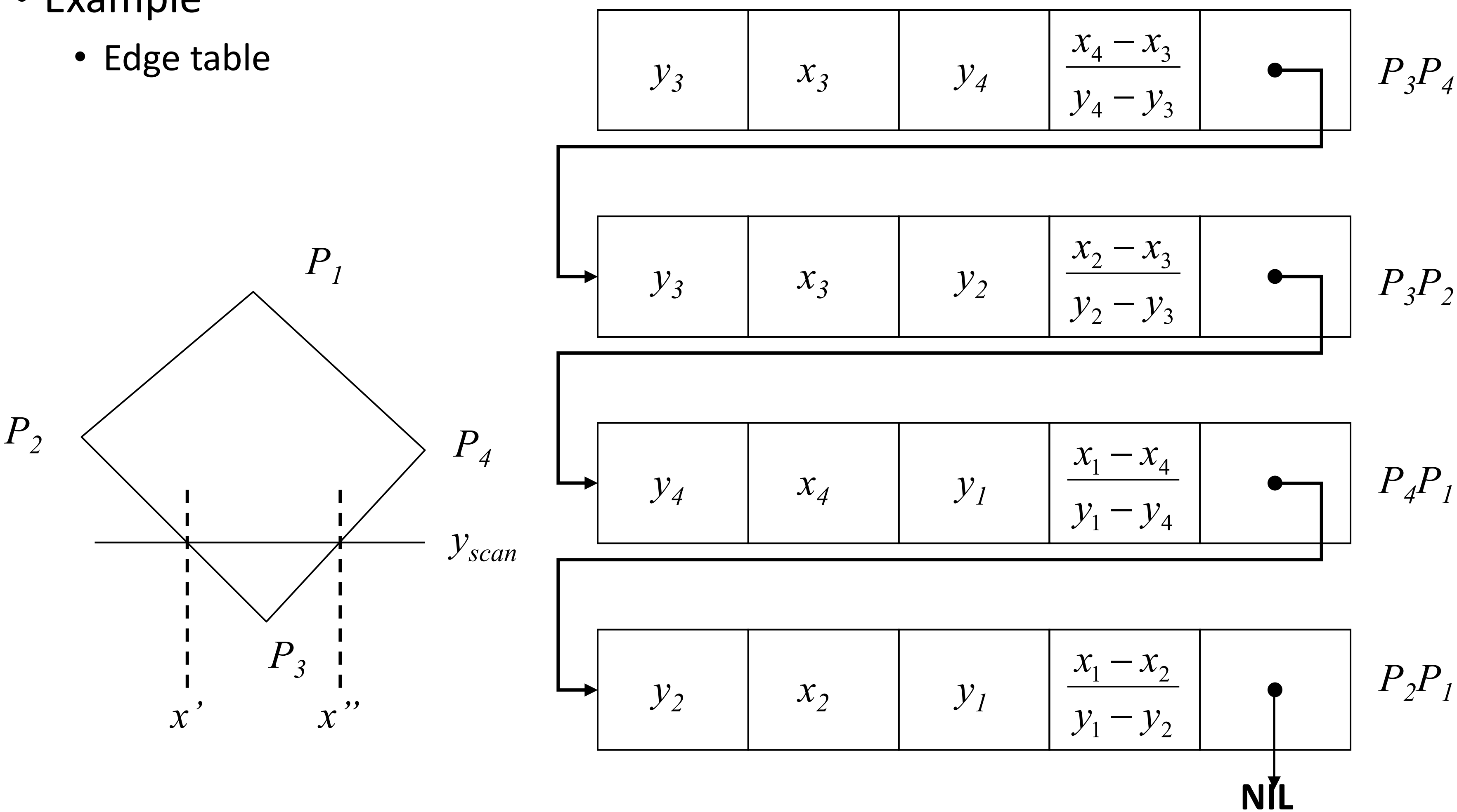
- Active Edge table (AET)
  - All edges from ET that intersect current scanline
  - Data per edge
  - Current scanline of y_{scan}
  - Current intersection of edge with scanline: $x_{intersect}, y_{scan}$
  - Sorted by $x_{intersect}$

$x_{intersect}$	y_{upper}	$1/m = \Delta x/\Delta y$	$\rightarrow$ next



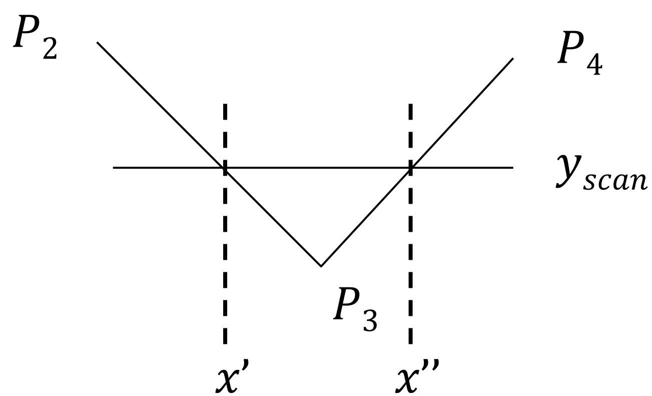
Scanline Algorithm

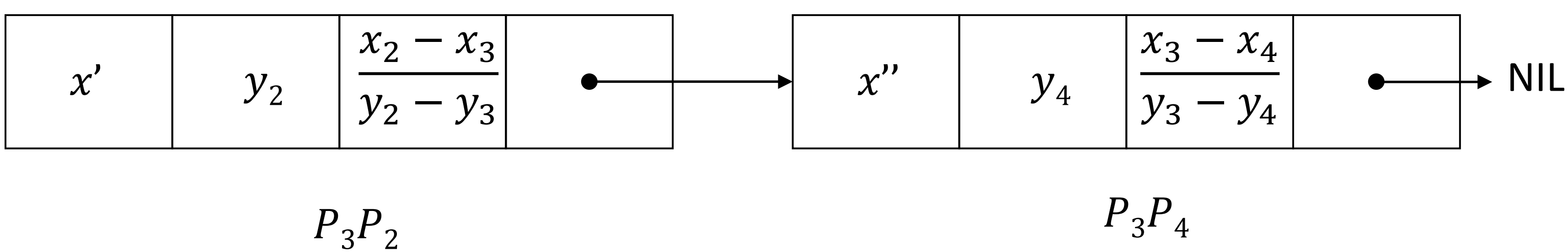
- Example
  - Edge table



Scanline Algorithm

- Current scanline $y_{scan} \Rightarrow$ AET





Scanline Algorithm

- Remark on incrementing x
  - $x_{old} = \frac{1}{m}(\mathrm{y_{scan}} - \mathrm{y_{lower}}) + \mathrm{x_{lower}}$
  - $x_{new} = \frac{1}{m}(y_{scan} + 1 - y_{lower}) + x_{lower} = x_{old} + \frac{1}{m}$
  - Where $m = \frac{y_{upper} - y_{lower}}{x_{upper} - x_{lower}}$

- So the update is $y \rightarrow y + 1,\ x \rightarrow x + \frac{1}{m}$

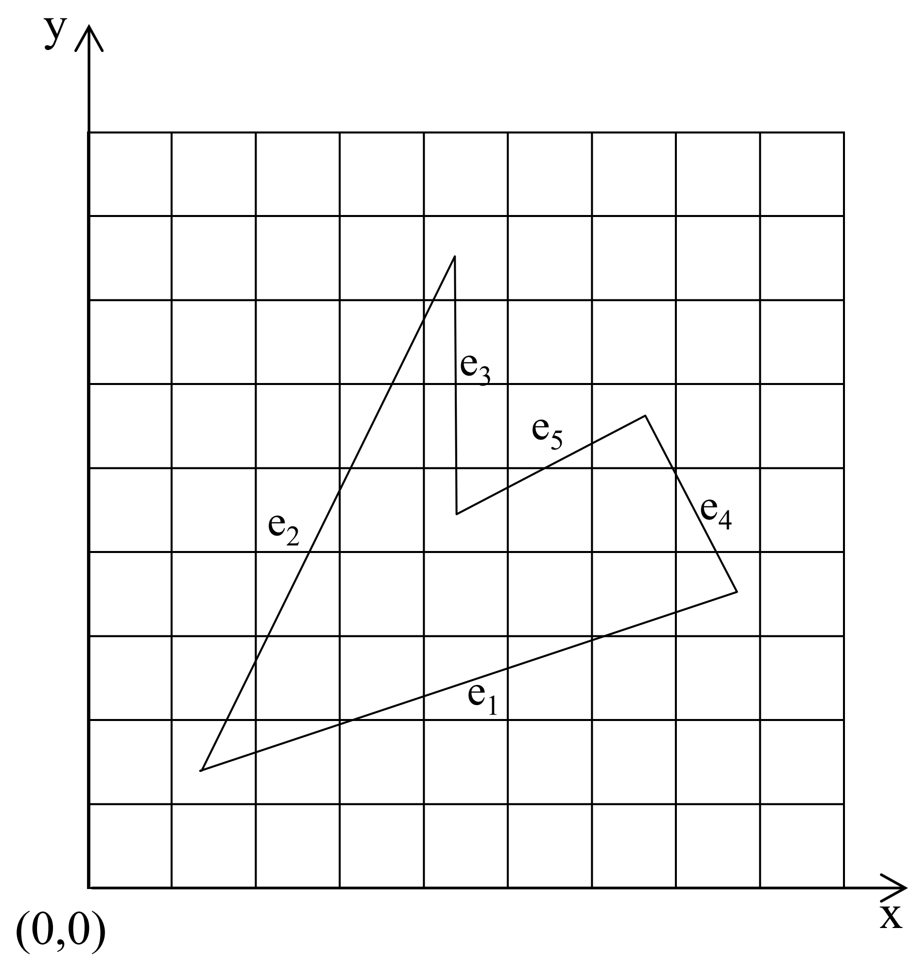
Scanline Algorithm

```
initialize ET
set AET to empty
set yscan to ylower of first entry in ET
    move all edges from ET with yscan == ylower to AET

while ET not empty or AET not empty
    sort AET for x
    draw lines from (AET[0].x,yscan) to (AET[1].x,yscan),
                   from (AET[2].x,yscan) to (AET[3].x,yscan), ……
    remove all edges from AET with yscan >= yupper
    for all edges in AET
        x:= x + 1/m
    yscan += 1
    move all edges from ET with yscan == ylower to AET
```

Scanline Algorithm

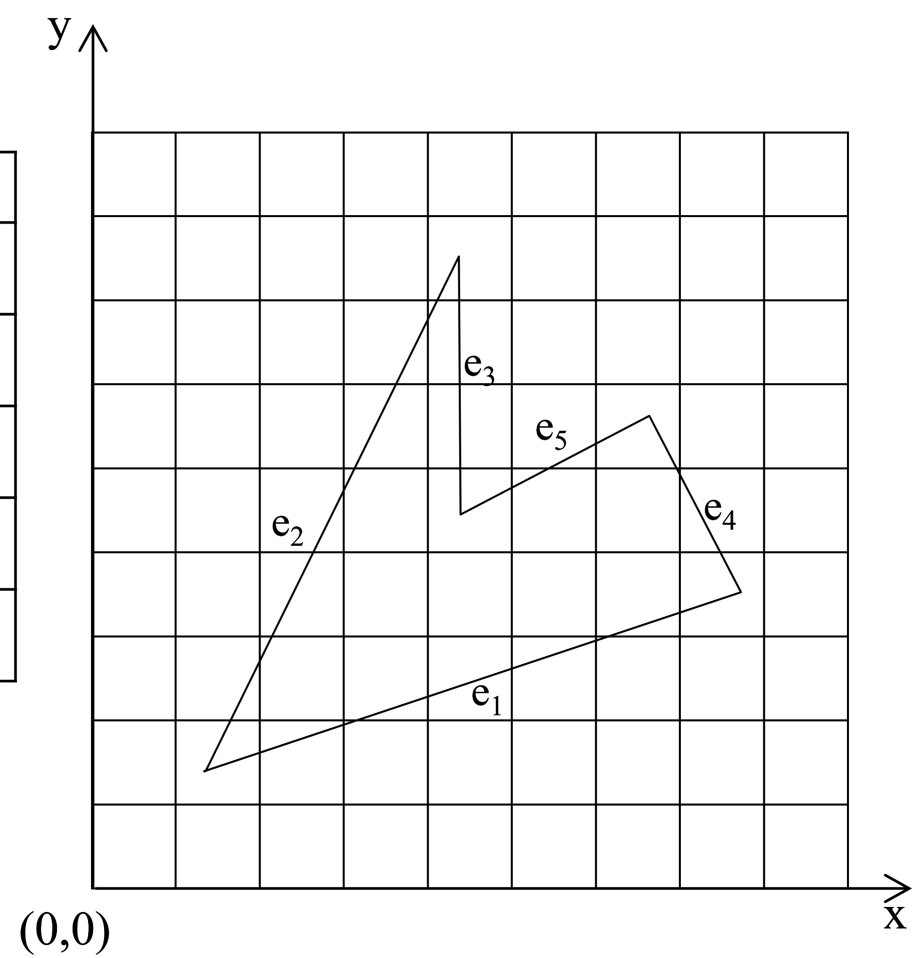
edge	y_{lower}	x_{lower}	y_{upper}	$1/m$
e_1	1	1	3	3
e_2	1	1	7	1 / 2
e_3	4	4	7	0
e_4	3	7	5	-3
e_5	4	4	5	2



Scanline Algorithm

ET: edge table, sorted on y_{lower}

edge	y_{lower}	x_{lower}	y_{upper}	$1/m$	Next
e_1	1	1	3	3	e_2
e_2	1	1	7	1 / 2	e_4
e_4	3	7	5	-3	e_3
e_3	4	4	7	0	e_5
e_5	4	4	5	2	NULL



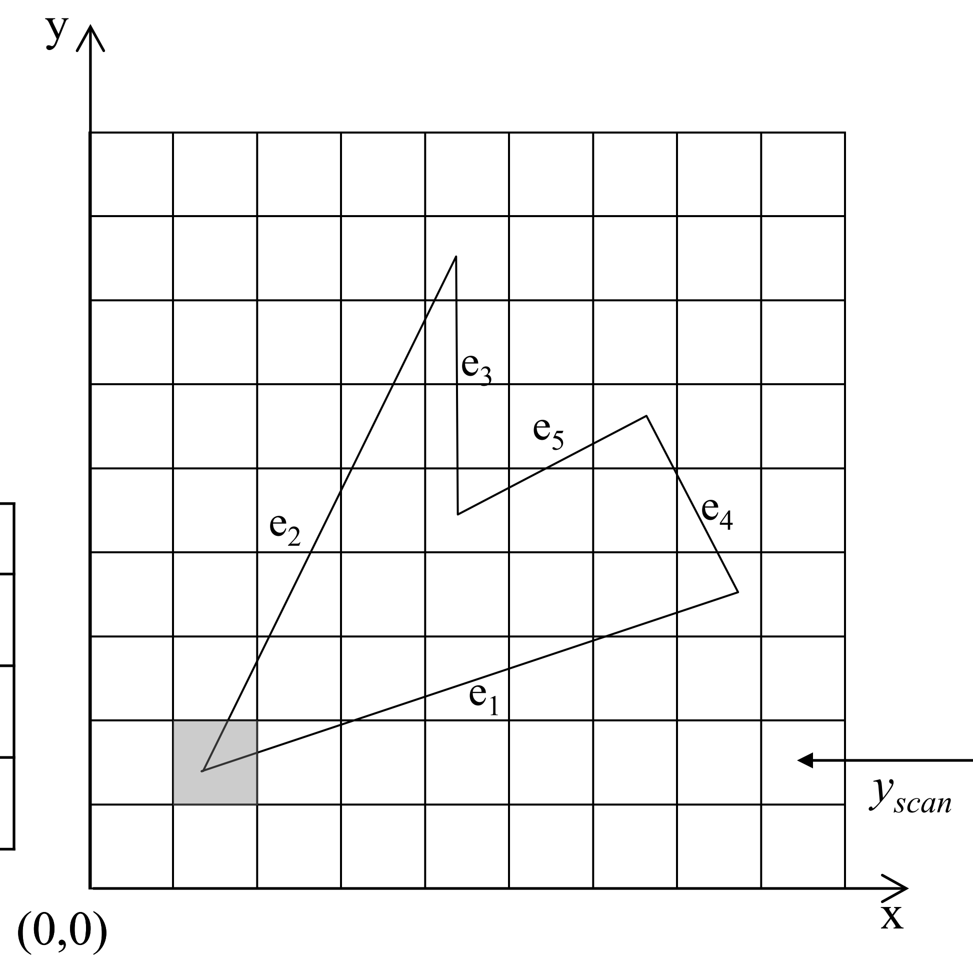
Scanline Algorithm

First scanline $y_{scan} = 1$
AET: edge table, sorted on $x_{intersect}$

edge	x_{inters}	y_{upper}	$1/m$	Next
e_1	1	3	3	e_2
e_2	1	7	1 / 2	NULL

ET: edge table, sorted on y_{lower}

edge	y_{lower}	x_{lower}	y_{upper}	$1/m$	Next
e_4	3	7	5	-3	e_3
e_3	4	4	7	0	e_5
e_5	4	4	5	2	NULL



Scanline Algorithm

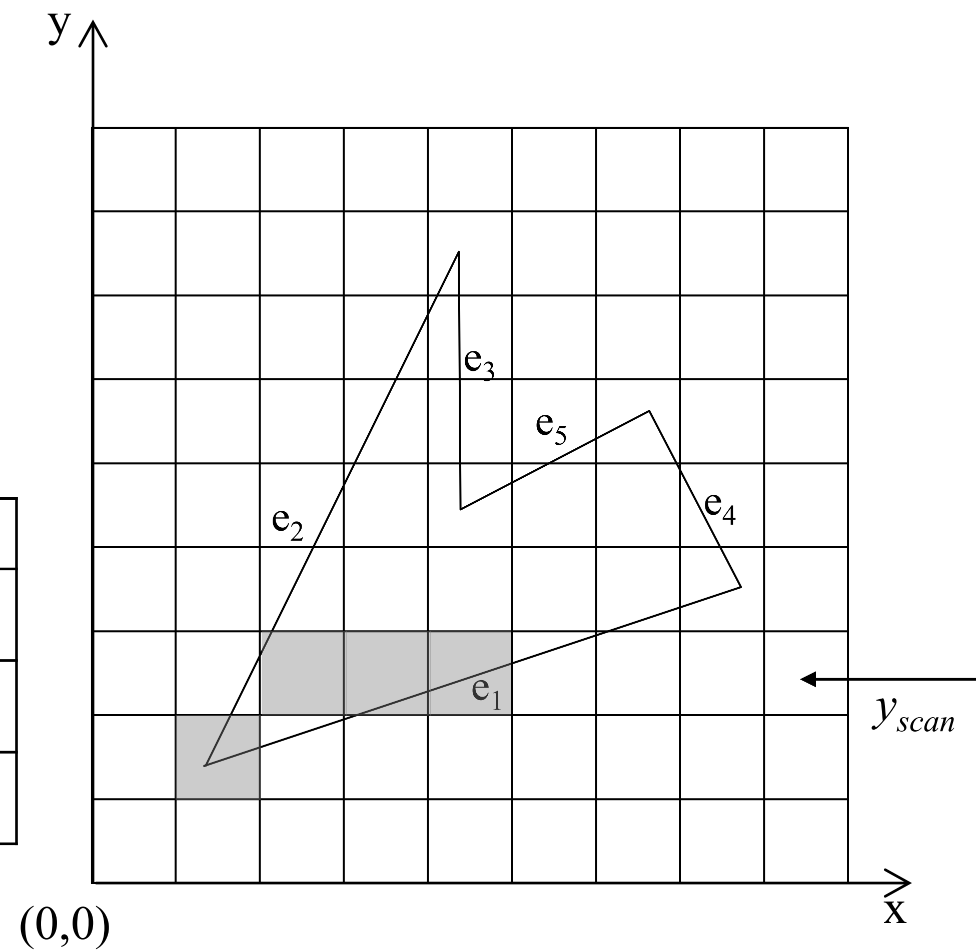
Scanline $y_{scan} = 2$

AET: edge table, sorted on $x_{intersect}$

edge	x_{inters}	y_{upper}	$1/m$	Next
e_2	3/2	7	1 / 2	e_1
e_1	4	3	3	NULL

ET: edge table, sorted on y_{lower}

edge	y_{lower}	x_{lower}	y_{upper}	$1/m$	Next
e_4	3	7	5	-3	e_3
e_3	4	4	7	0	e_5
e_5	4	4	5	2	NULL



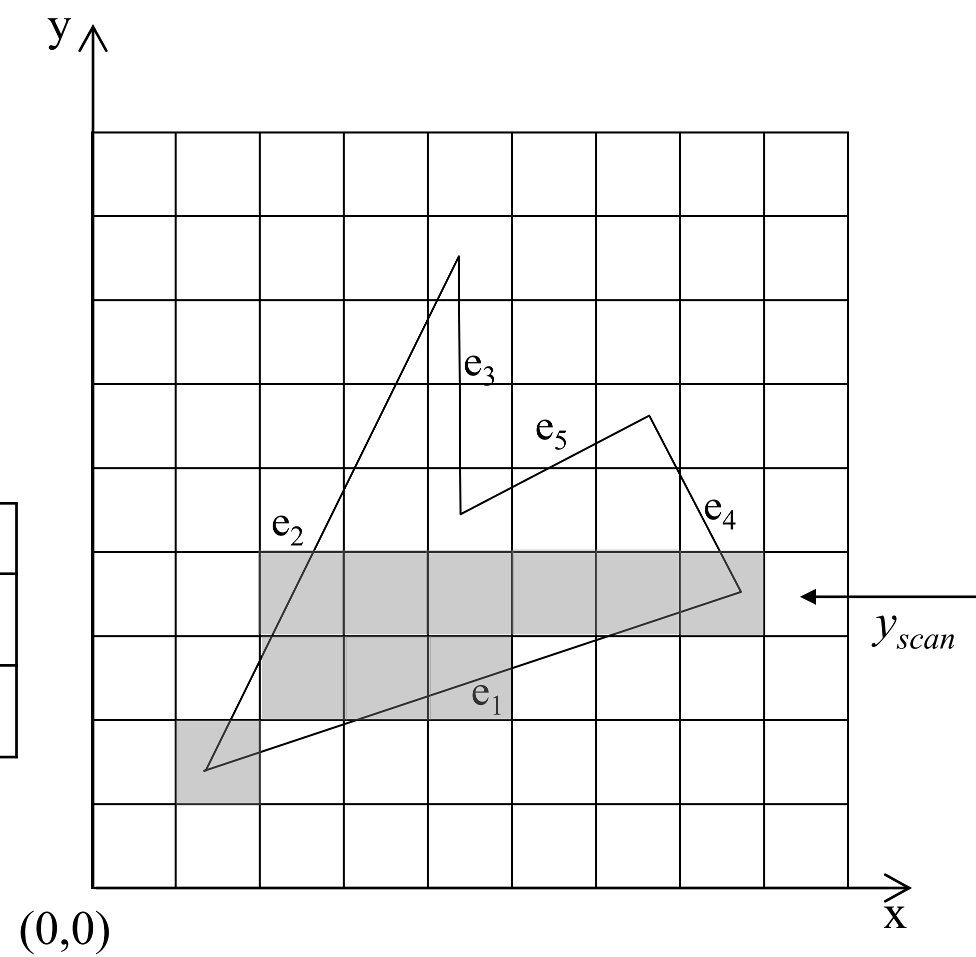
Scanline Algorithm

Scanline $y_{scan} = 3$
AET: edge table, sorted on $x_{intersect}$

edge	x_{inters}	y_{upper}	$1/m$	Next
e_2	2	7	1 / 2	e_1
e_4	7	5	-3	NULL

ET: edge table, sorted on y_{lower}

edge	y_{lower}	x_{lower}	y_{upper}	$1/m$	Next
e_3	4	4	7	0	e_5
e_5	4	4	5	2	NULL

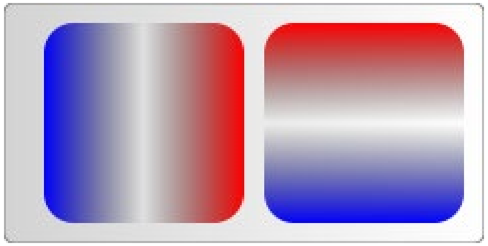


Scanline Algorithm

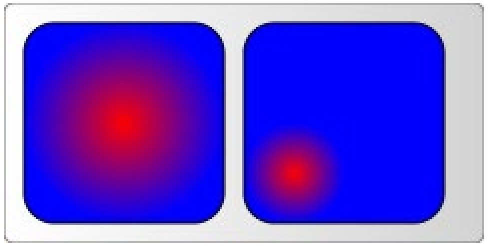
- Set pixels inside polygon to which color? → “**Shading**”

- We could define color gradients

  - e.g. SVG linear gradients



  - e.g. SVG radial gradients

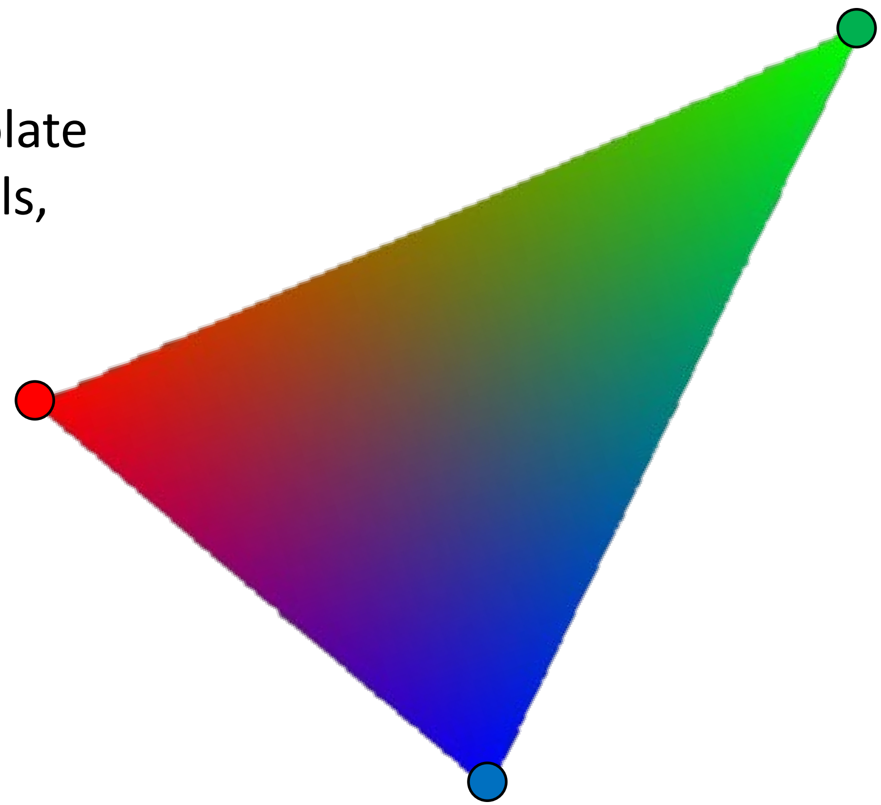


https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Gradients

Scanline Algorithm

- for our purpose, we want to define color values at the vertices of the polygon and interpolate these
  → **Gouraud Shading**

- Later on, we want to interpolate also other attributes (normals, texture coordinates, …)



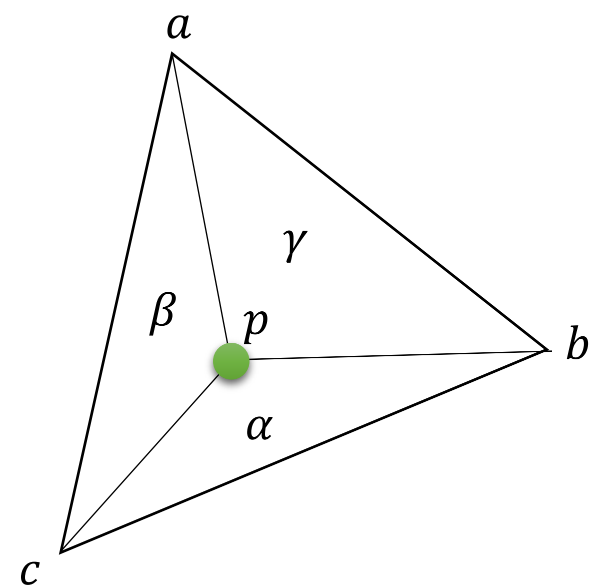
Gouraud Shading

- Interpolating intensities (or other attributes)

- Any point p inside the triangle abc can be described as an *affine combination* of the vertices

$$p = \alpha a + \beta b + \gamma c$$

with $\quad \alpha + \beta + \gamma = 1$
and $\quad 0 < \alpha, \beta, \gamma < 1$



- α, β, γ are the ***Barycentric Coordinates*** of p with respect to triangle abc

Gouraud Shading

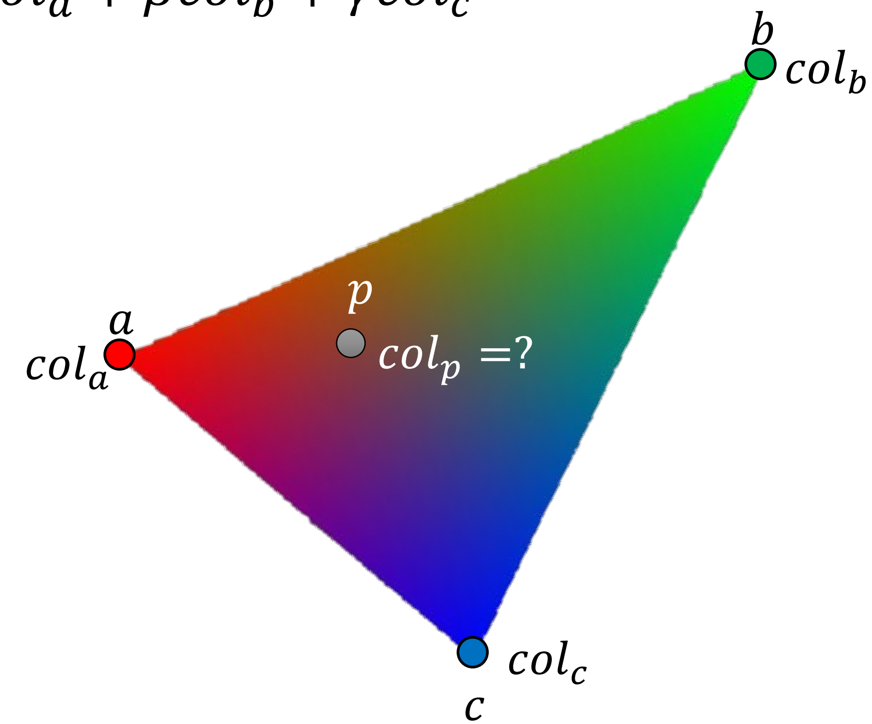
- If we know the barycentric coordinates of a point p inside a triangle

$$p = \alpha a + \beta b + \gamma c$$

- we can interpolate colors with the same weights:

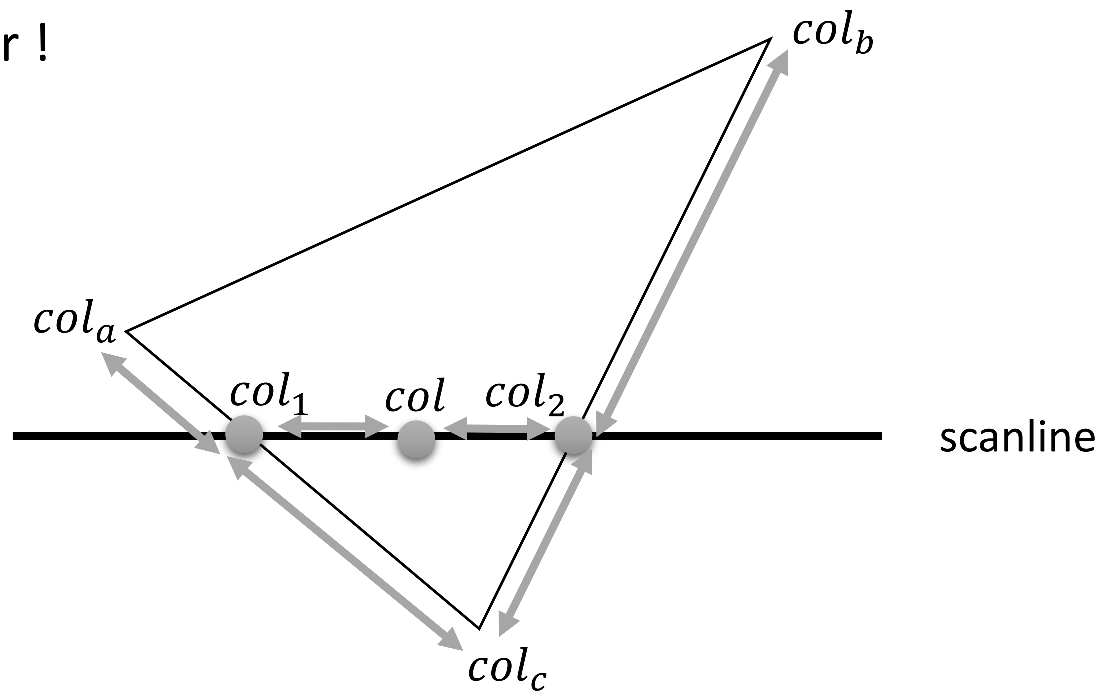
$$col_p = \alpha col_a + \beta col_b + \gamma col_c$$

→ *linear interpolation*



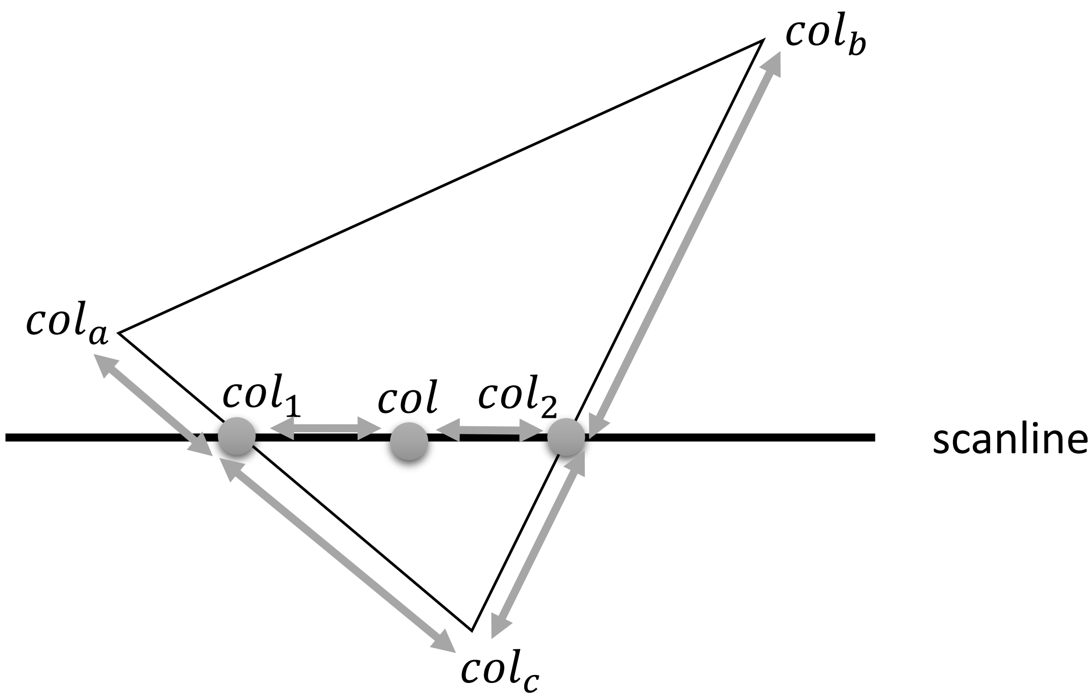
Gouraud Shading

- Algorithmically:
  - do linear interpolation of the attributes along the edges
  - within a span, interpolate linearily
- This is not bilinear, but linear !



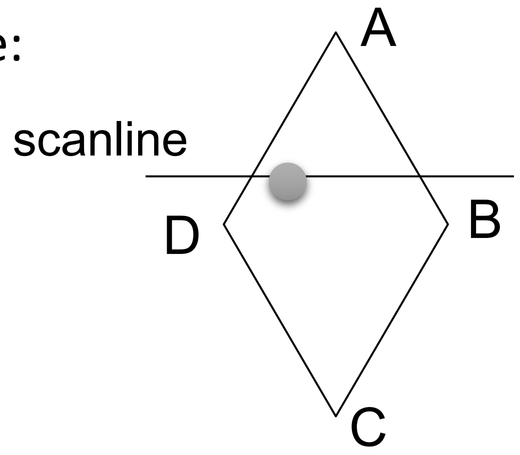
Gouraud Shading

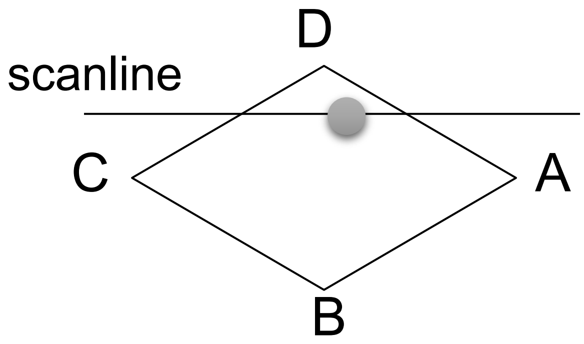
- Can be well combined with scanline rasterization
  - with each edge, store increment of attribute when going one scanline up
    → same idea as using $1/m$ to update x
  - do not only update x by $1/m$, but also attributes
  - when rasterizing a span, compute attribute updates for $x \rightarrow x+1$



Polygon Shading

- Problems
  - Shading only rotation invariant for triangles
  - for more than 3 vertices: color inside polygon changes with rotation → BAD !
- Example:





color depends on A,B,D

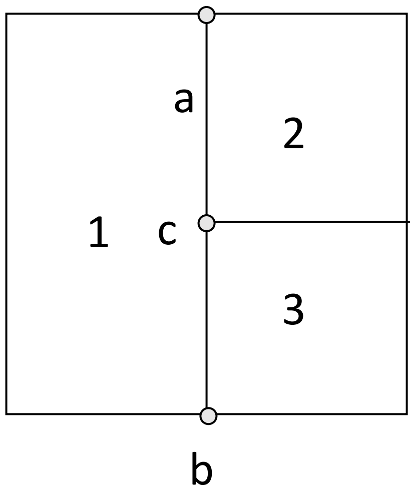
color depends on A,C,D

→ triangulate and rasterize triangles

→ but then the color depends on the triangulation…

Polygon Shading

- Problem: Vertex inconsistencies
  - Polygon 1
    - Color at c comes from interpolation between a and b
  - Polygons 2 and 3
    - c is separate vertex
  - Color seam along edge ab if color in c not chosen correctly



- Solution: avoid such hanging nodes, they also make other problems! (e.g., they can result in holes during rasterization)

Next Lecture

- An intro to GPU rendering