Parallel image reductions over polygon masks

Dániel Berényi

Freelancer

GPU Day 2024

Motivation

Quality control / selection in manufacturing

A. Types of the defects in metal

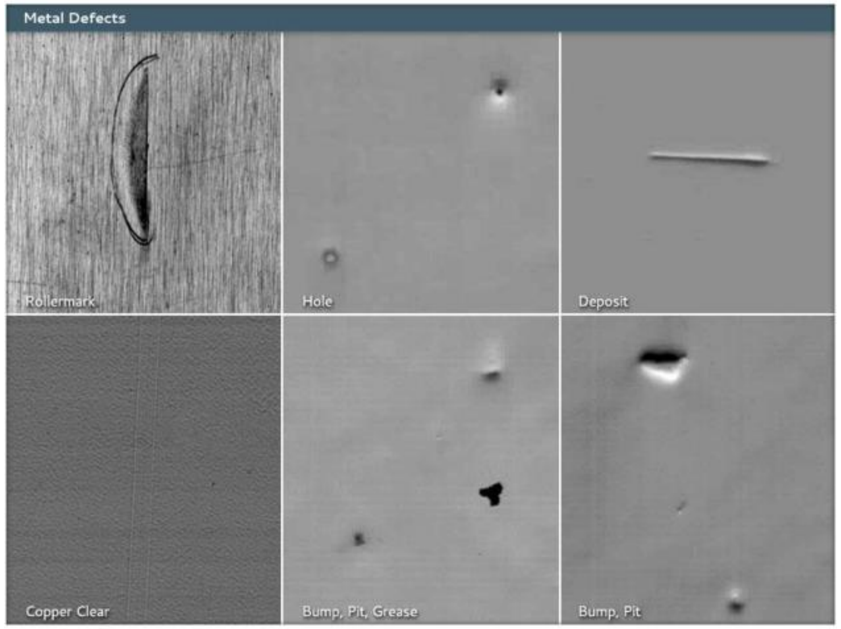
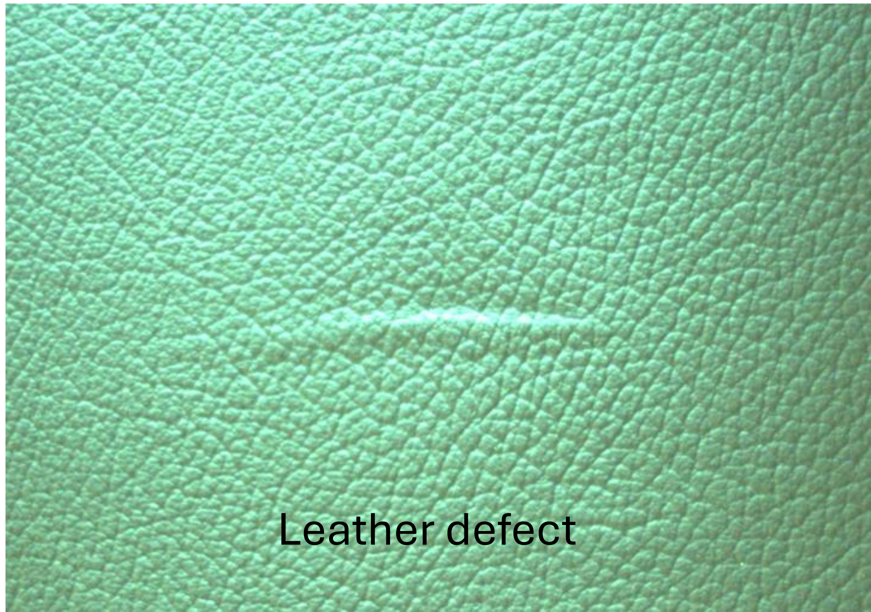


Fig. 1 Different types of surface the defects in metals

Metal Inspection for Surface defect Detection by Image Thresholding D. M. Lohade, P. B. Chopade

Wood defects


Leather defect

Motivation

Quality control / selection in manufacturing

Many goods are produced in wide rolls, sheets, boards, or on conveyor belts

~1-2 meters wide

https://commons.wikimedia.org/wiki/File:Tissue_Paper_Production_Machine.jpg

Desired fault detection size can be as low as

1.0mm - 0.5mm

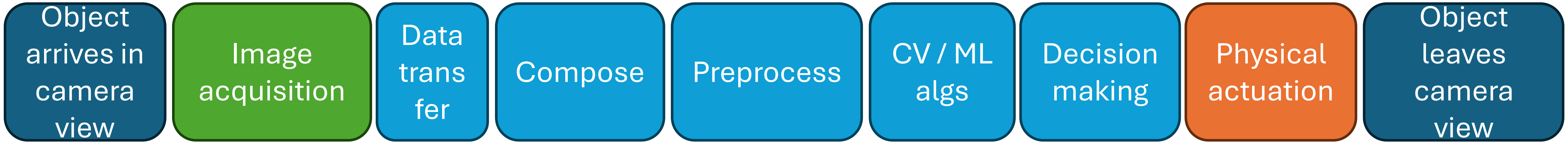
Spatial resolution of up to 10k pixels may be needed

Motivation

Quality control / selection in manufacturing

Movement speed is high, the entire decision making has ~1-10 seconds to decide an accept/reject

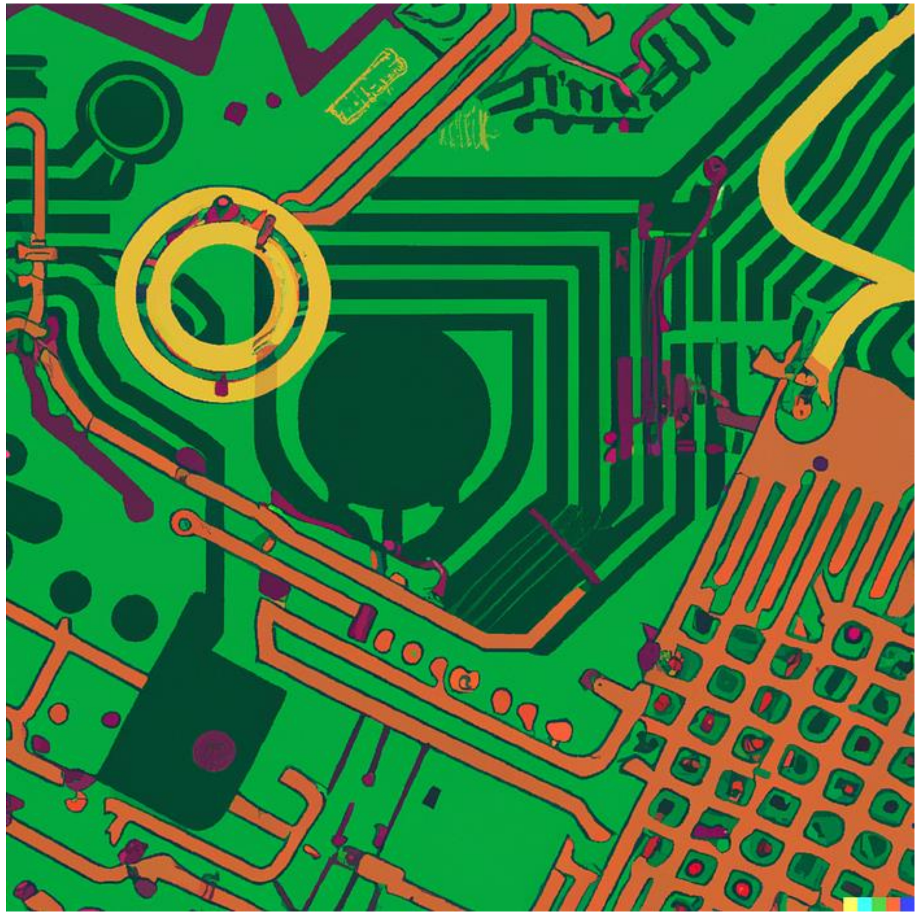
A complex image processing pipeline should finish in this short time



Motivation - polygons

Many applications across industries are working with polygon boundaries, e.g.:

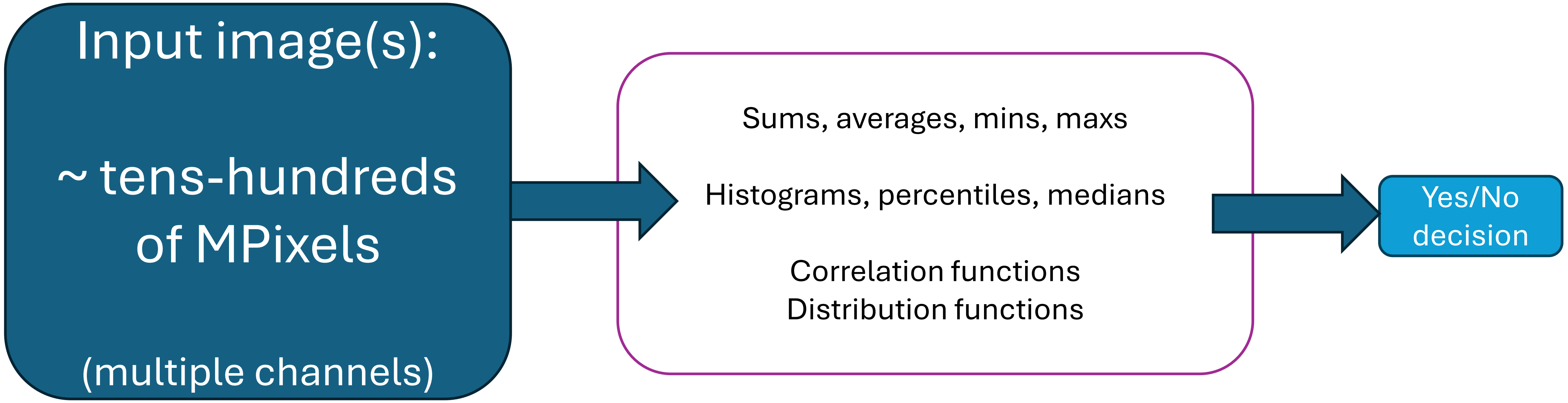
- PCB QC
- Wood QC
- Mining / ore processing
- Civil engineering
- Cartography
- …



https://medium.com/@nazlicanto/pcb-defect-detection-with-segformer-b56947732914

Reductions

As with many other image processing tasks,
it is all about reductions:



Masked reductions

As with many other image processing tasks,
it is all about reductions,

Sums, averages, mins, maxs

Histograms, percentiles, medians

Correlation functions
Distribution functions

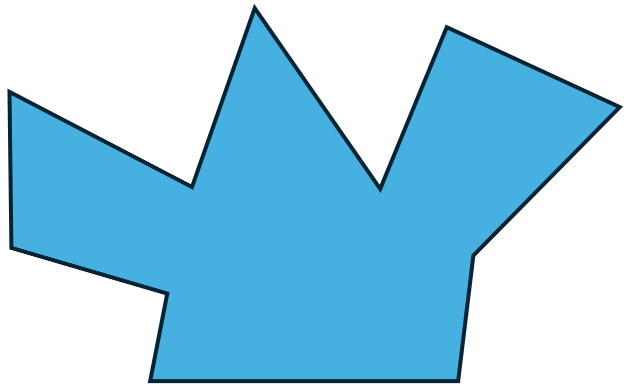
But in many cases, we would like to restrict the area of reductions to some specific region.

Typical masks include:
Pixelwise masks, rectangles, or polygons…

Polygons

Polygons might come from many different sources, we may, or may not make particular assumptions about them.

For the current task, lets assume that we are dealing with simple polygons (no self-intersection, no holes, but can be concave):



Polygons

Polygons might come from many different sources, we may, or may not make particular assumptions about them.

Let’s assume that they are represented as an array of coordinate pairs:

```
11.73125,  -15.8507
12.015,    -15.56944
12.51271,  -13.42438
12.97,     -12.78444
```

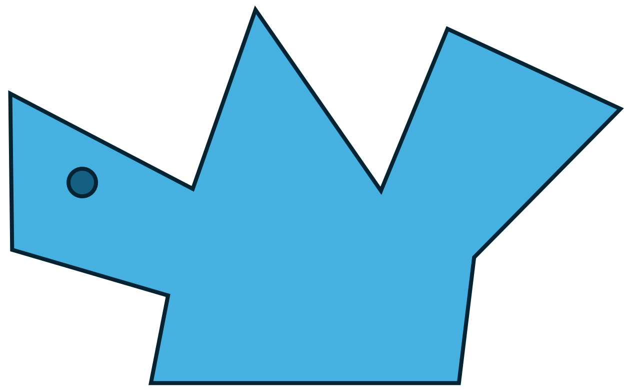
Point in polygon problem

The solution to the masked reduction problem needs to load and aggregate those, and preferably only those pixels that are inside the polygon area.

This is known as the point in polygon problem

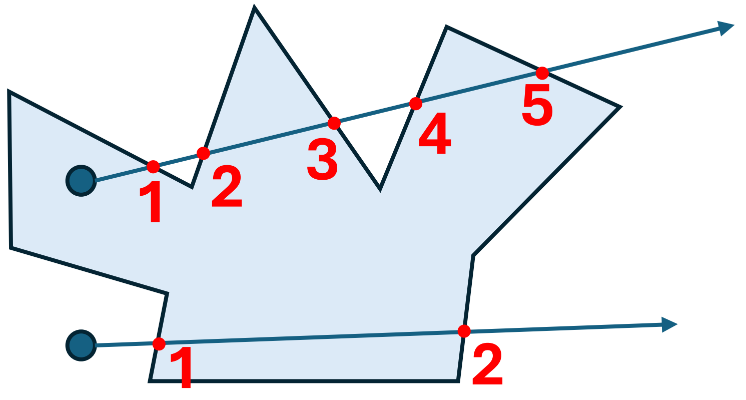
There are two widely used algorithms based on:

- Ray casting
- Winding number



Discussion based on:
http://profs.ic.uff.br/~anselmo/cursos/CGI/slidesNovos/Inclusion%20of%20a%20Point%20in%20a%20Polygon.pdf

Point in polygon problem

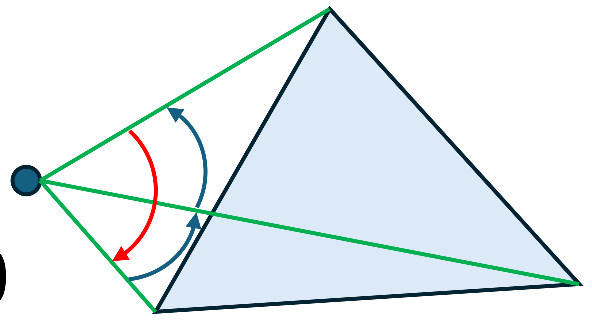


Ray casting:

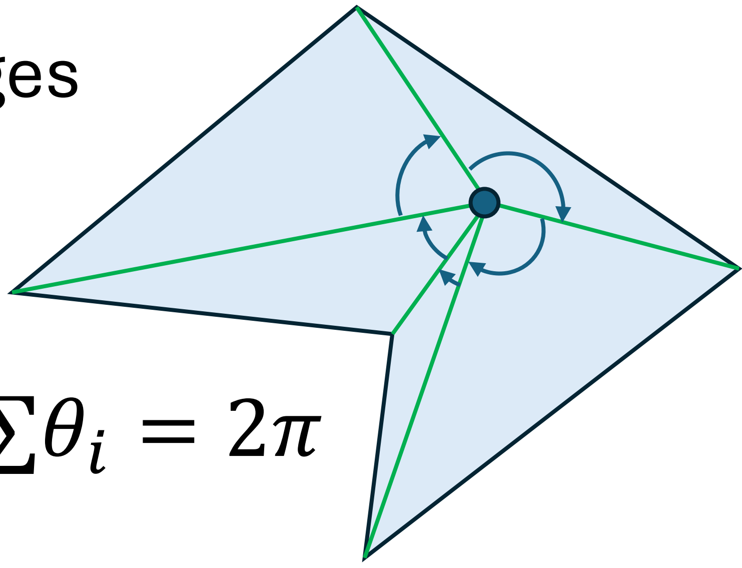
- Count the number of intersection points with the boundary, if it is odd, the source of the ray is inside

Winding number:

- Sum up the subtended (signed!) angles of the edges
  If it is non-zero, point is inside

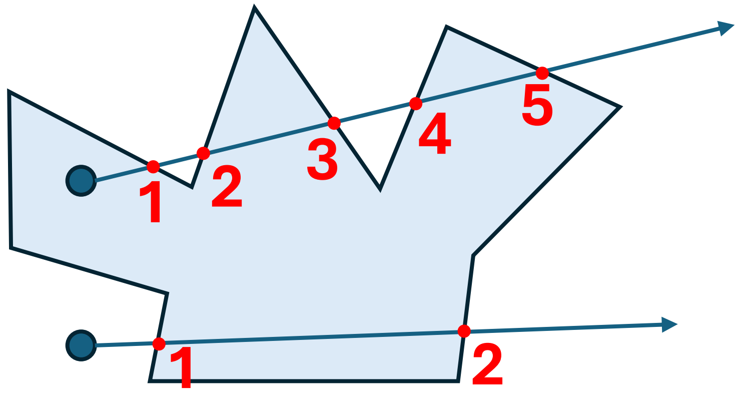


$\sum \theta_i = 0$



$\sum \theta_i = 2\pi$

Point in polygon problem



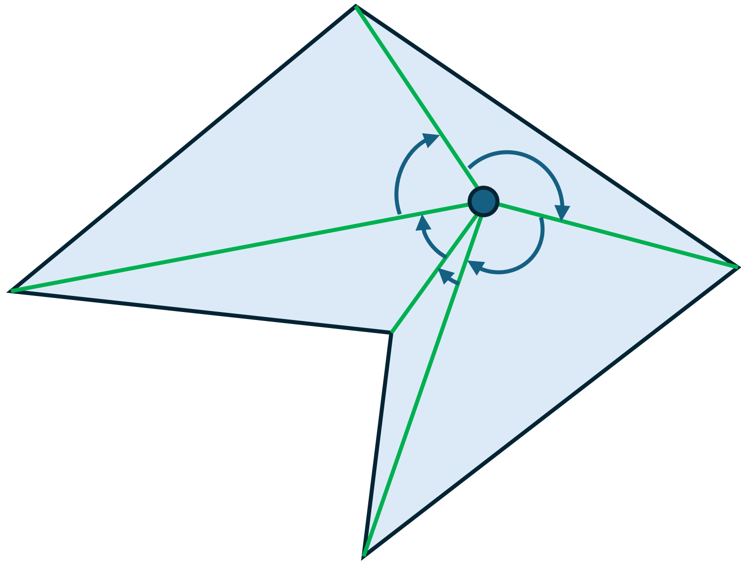
Caveats:

Ray casting:

- Intersection computation is prone to many corner cases (like what happens at intersections exactly at the vertices)

Winding number:

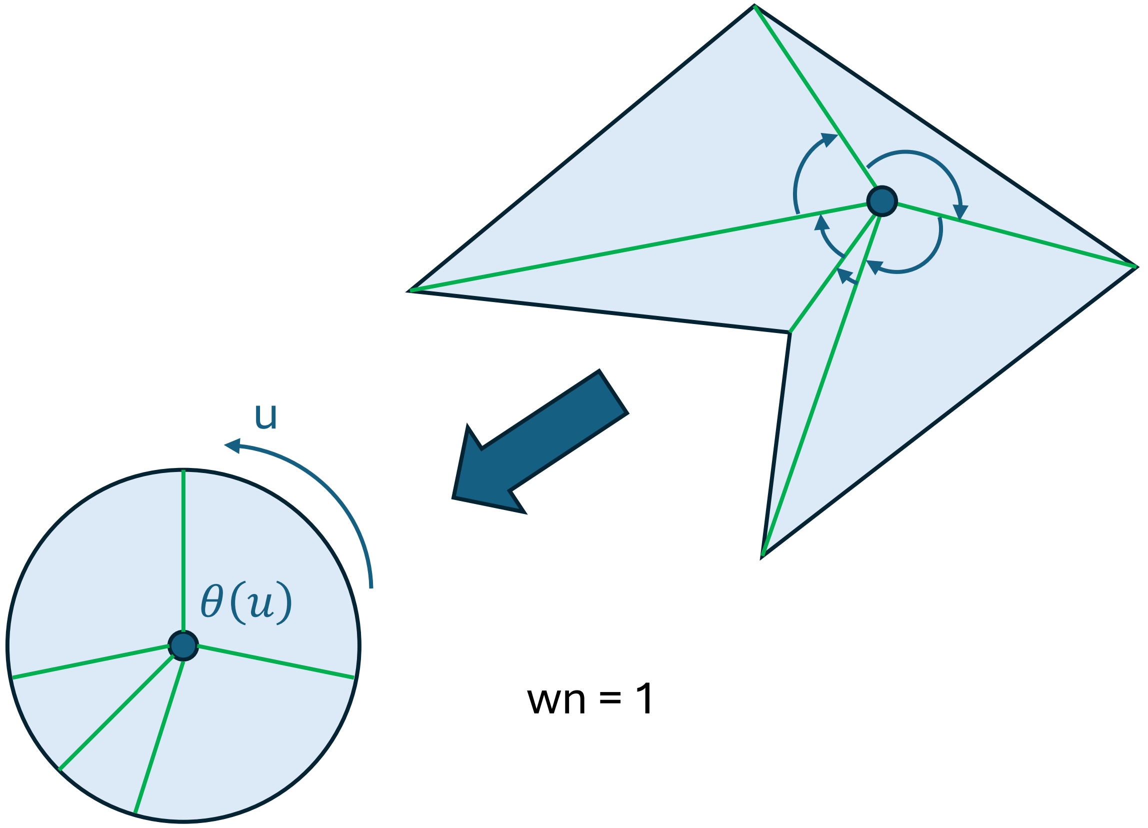
- Less corner case processing is needed



The winding number algorithm

Any closed curve C is homeomorphic to the unit circle, so we can use a polar coordinate description to define the winding number:

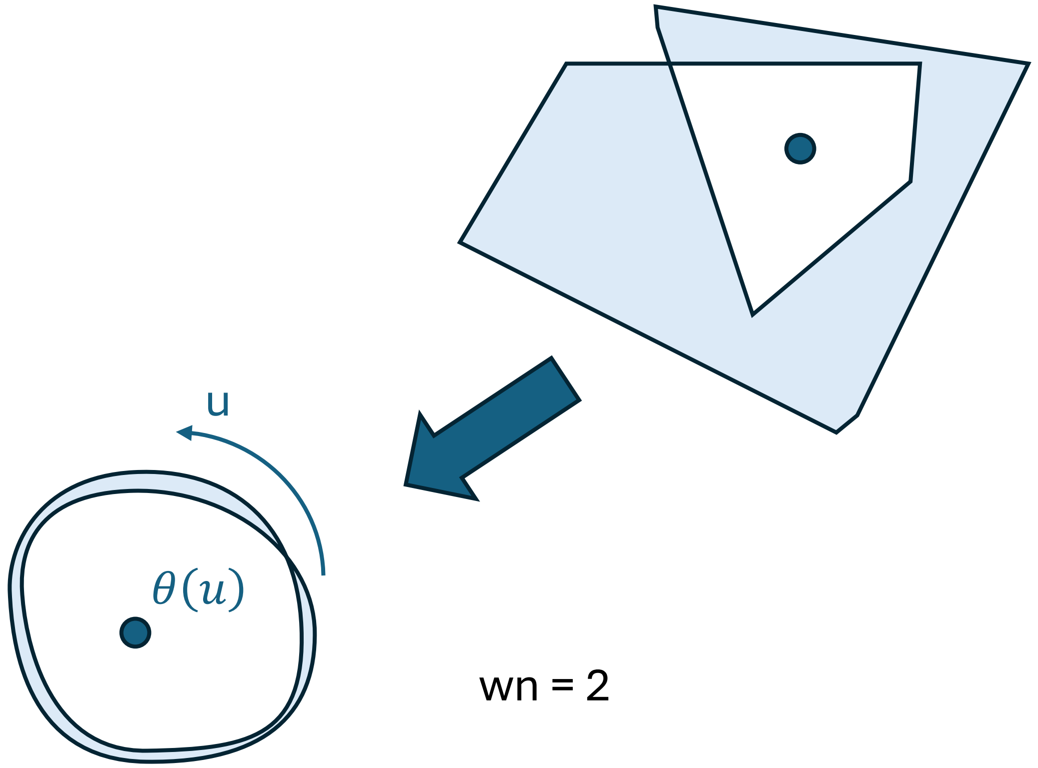
$$\mathrm{wn}(C) = \frac{1}{2\pi}\oint_C \, d\theta = \frac{1}{2\pi}\int_0^1 \theta(u)\mathrm{d}u$$



The winding number algorithm

Any closed curve C is homeomorphic to the unit circle, so we can use a polar coordinate description to define the winding number:

$$\mathrm{wn}(C) = \frac{1}{2\pi}\oint_C d\theta = \frac{1}{2\pi}\int_0^1 \theta(u)\mathrm{d}u$$

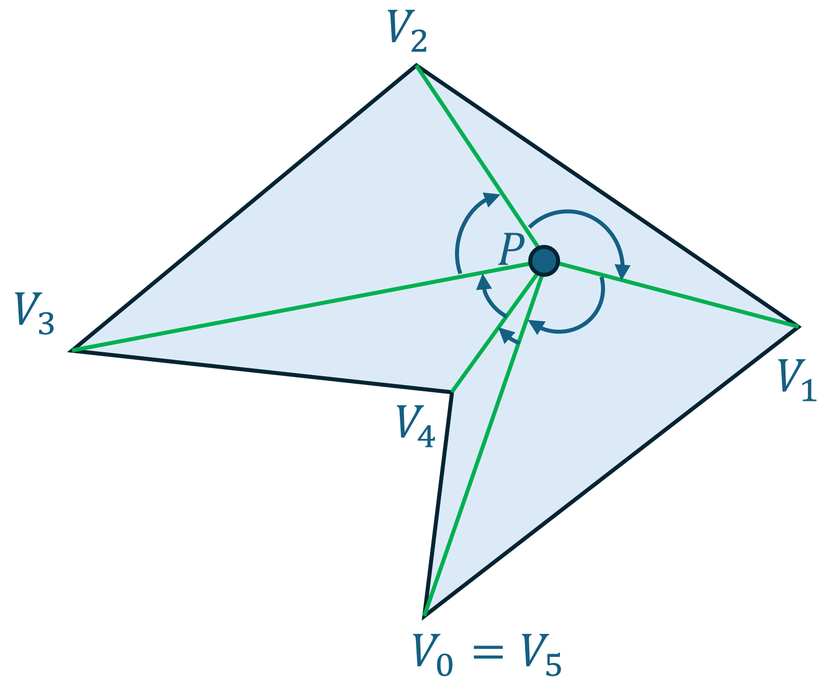


The winding number algorithm

For polygons of course, the formulas become more concrete:

$$\mathrm{wn}(C) = \frac{1}{2\pi}\oint_C d\theta = \frac{1}{2\pi}\int_0^1 \theta(u)\mathrm{d}u$$

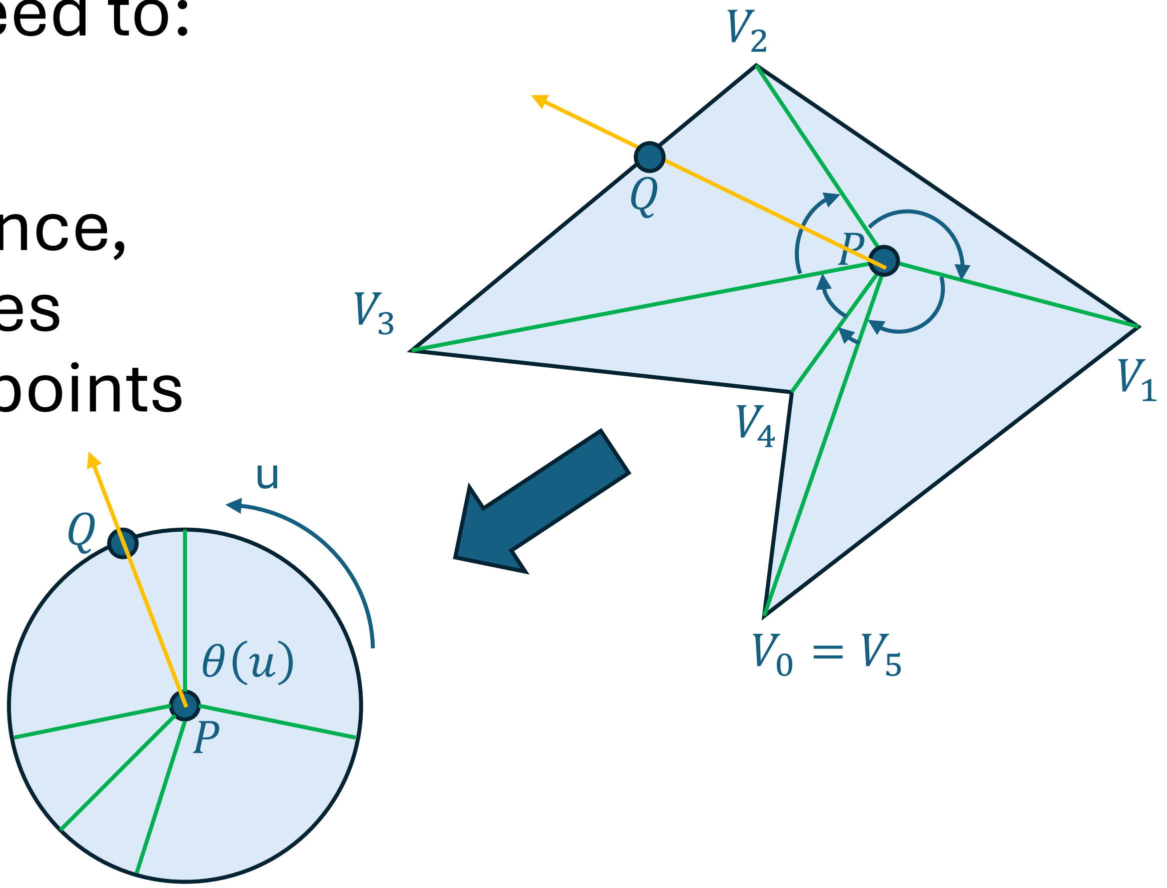
$$= \frac{1}{2\pi}\sum_{i=0}^{n-1}\theta_i = \frac{1}{2\pi}\sum_{i=0}^{n-1}\mathrm{acos}\left(\frac{(V_i - P)(V_{i+1} - P)}{|V_i - P|\,|V_{i+1} - P|}\right)$$



The winding number algorithm

In reality, we do not want to compute inverse trigonometric functions, but fortunately we don’t need to:

Based on the unit circle correspondence, We only need to track, how many times traversal passes the ray intersection points on the boundary!

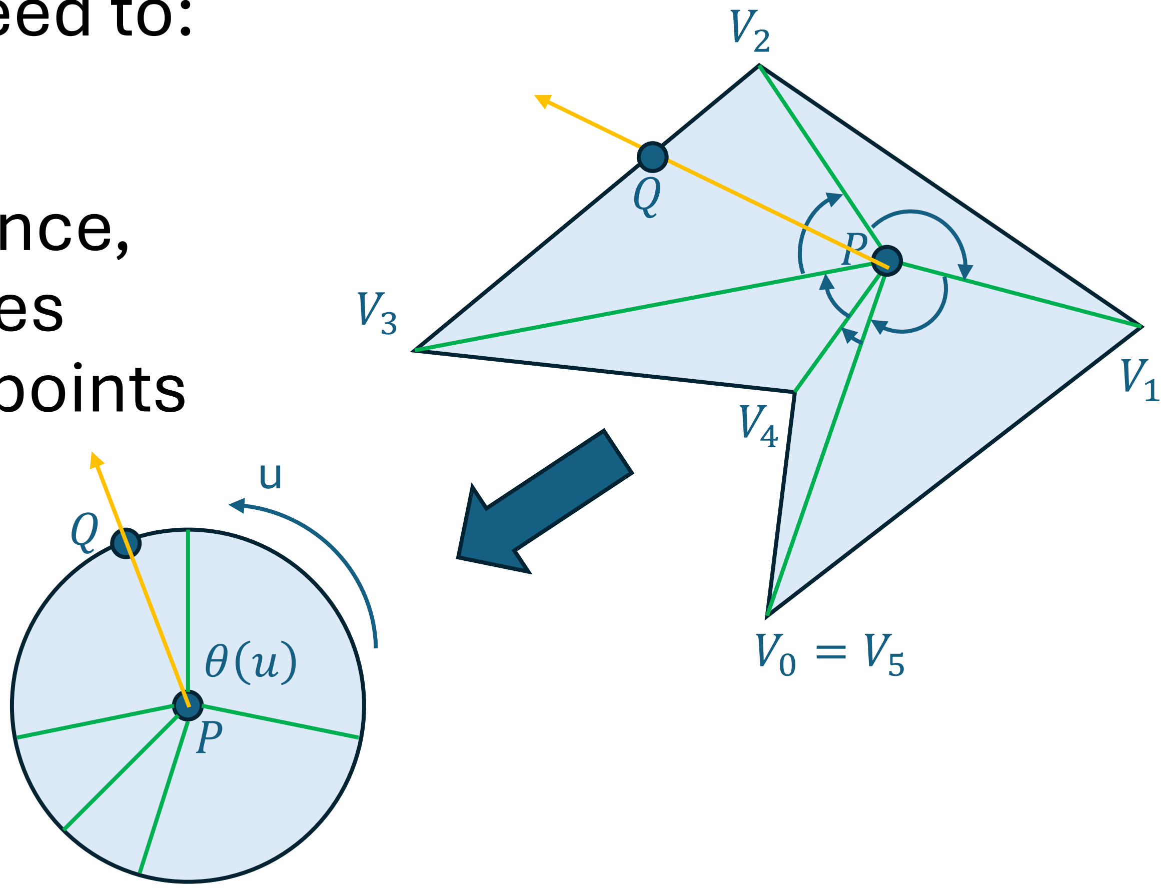


The winding number algorithm

In reality, we do not want to compute inverse trigonometric functions, but fortunately we don't need to:

Based on the unit circle correspondence,
We only need to track, how many times traversal passes the ray intersection points on the boundary!

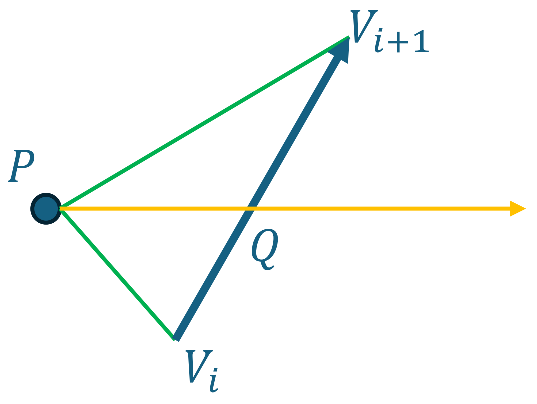
This gets awkwardly similar
to the ray intersection alg…



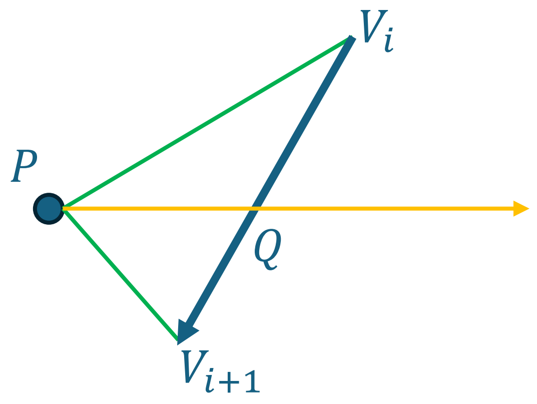
The winding number algorithm

But we do not need the intersections, we just need a sign, if an edge is being crossed in one direction, or the other

also, we can choose any ray direction, so let's choose a horizontal one:

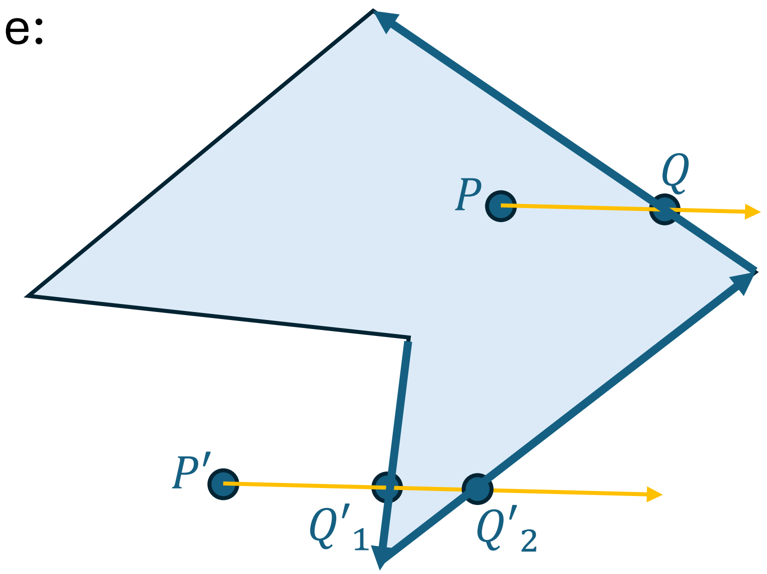


Upward crossing
$wn \mathrel{+}= 1$



Downward crossing
$wn \mathrel{-}= 1$

This relative orientation is obtained by computing the signed area of the $V_iV_{i+1}P$ triangle



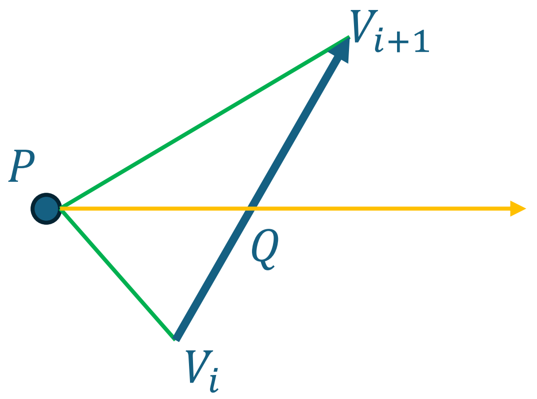
wn(P) = 1

wn(P’) = (-1)+(+1) = 0

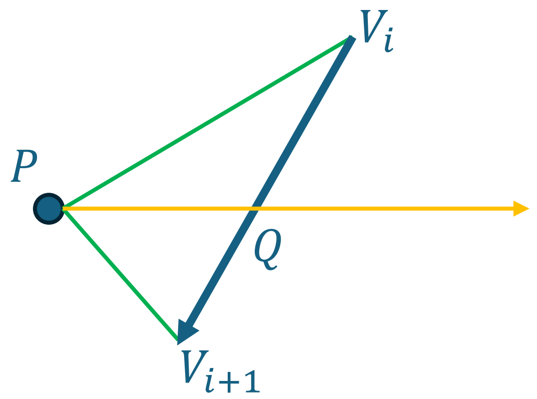
The winding number algorithm

But we do not need the intersections, we just need a sign, if an edge is being crossed in one direction, or the other

also, we can choose any ray direction, so let's choose a horizontal one:

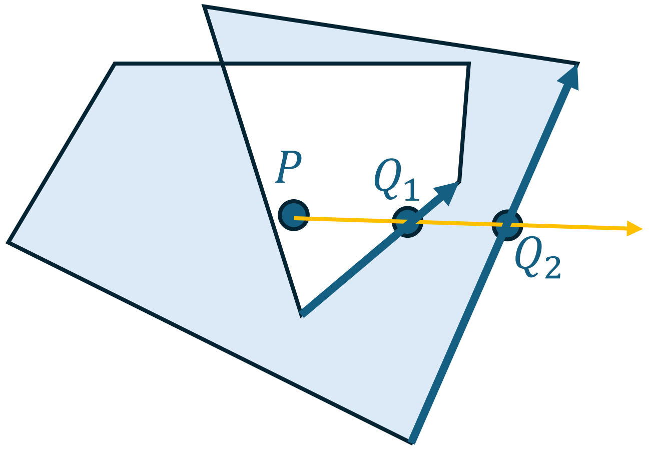


Upward crossing
$wn \mathrel{+}= 1$



Downward crossing
$wn \mathrel{-}= 1$

This relative orientation is obtained by computing the signed area of the $V_iV_{i+1}P$ triangle



wn(P) = +2

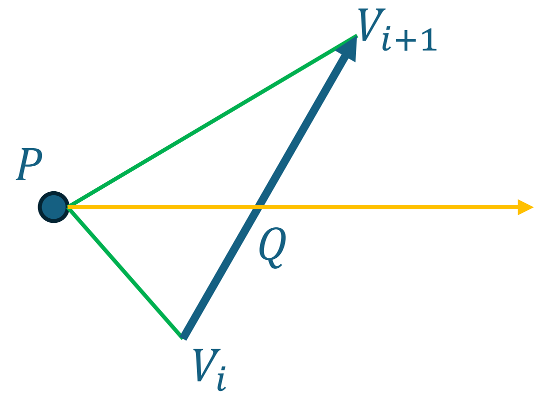
The winding number algorithm

The signed area of a triangle can be obtained by the determinant:

$$A = \frac{1}{2}\begin{vmatrix} 1 & x_0 & y_0 \\ 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \end{vmatrix}$$

That can also be written:

$$A = \frac{1}{2}\left((x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)\right)$$



The winding number algorithm

Putting everything together, the following algorithm computes the winding number wn of point P for a polygon given by a list of vertices using a horizontal ray:

```
wn = 0

Loop over edges
  Let V0, V1 be the end points of the current edge

  if(V1.y <= P.y)
  {
    if(V0.y > P.y && area(V1, V0, P) > 0){ wn += 1 }
  }
  else if(V0.y <= P.y && area(V1, V0, P) < 0){ wn -= 1 }
```

The winding number algorithm

Problem: We need to test millions of pixels

The naïve algorithm tests every pixel with every edge,
this is clearly inefficient

What can we do?

The winding number algorithm

Problem: We need to test millions of pixels

The naïve algorithm tests every pixel with every edge,
this is clearly inefficient

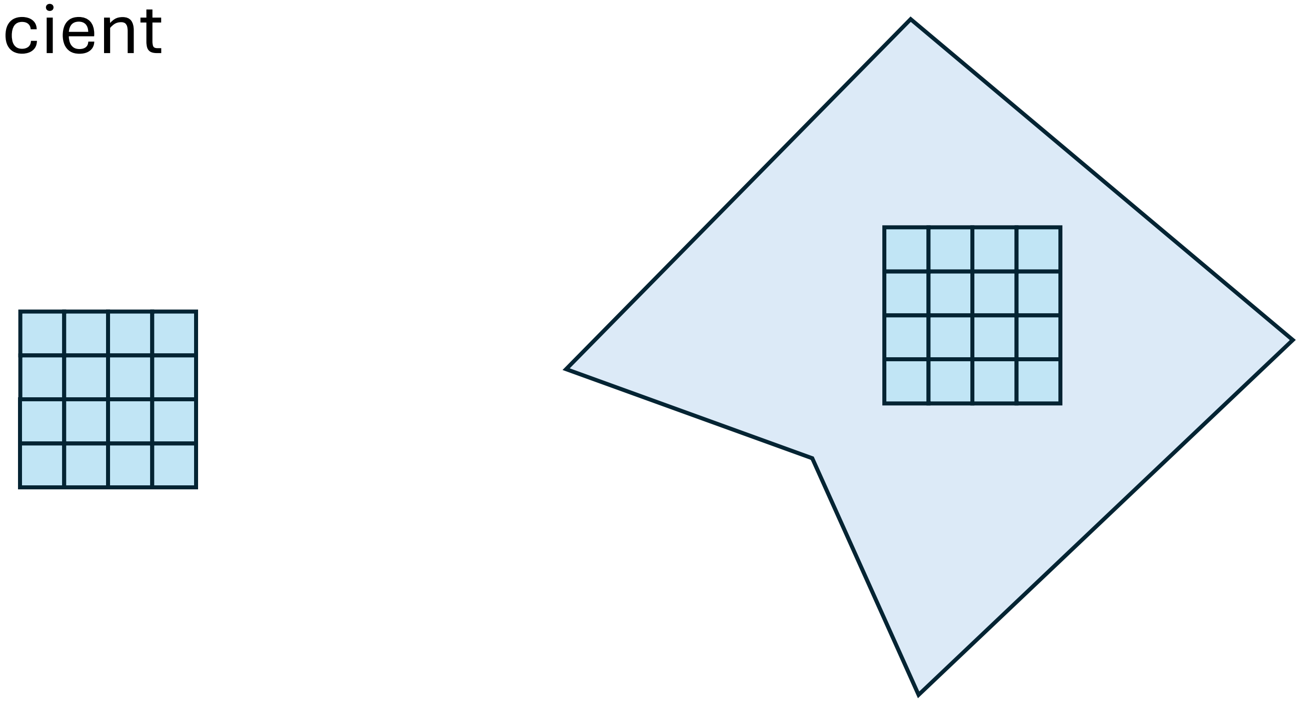
What can we do?

As usual: acceleration structures / early reject

The winding number algorithm - optimizations

Acceleration structures / early reject:

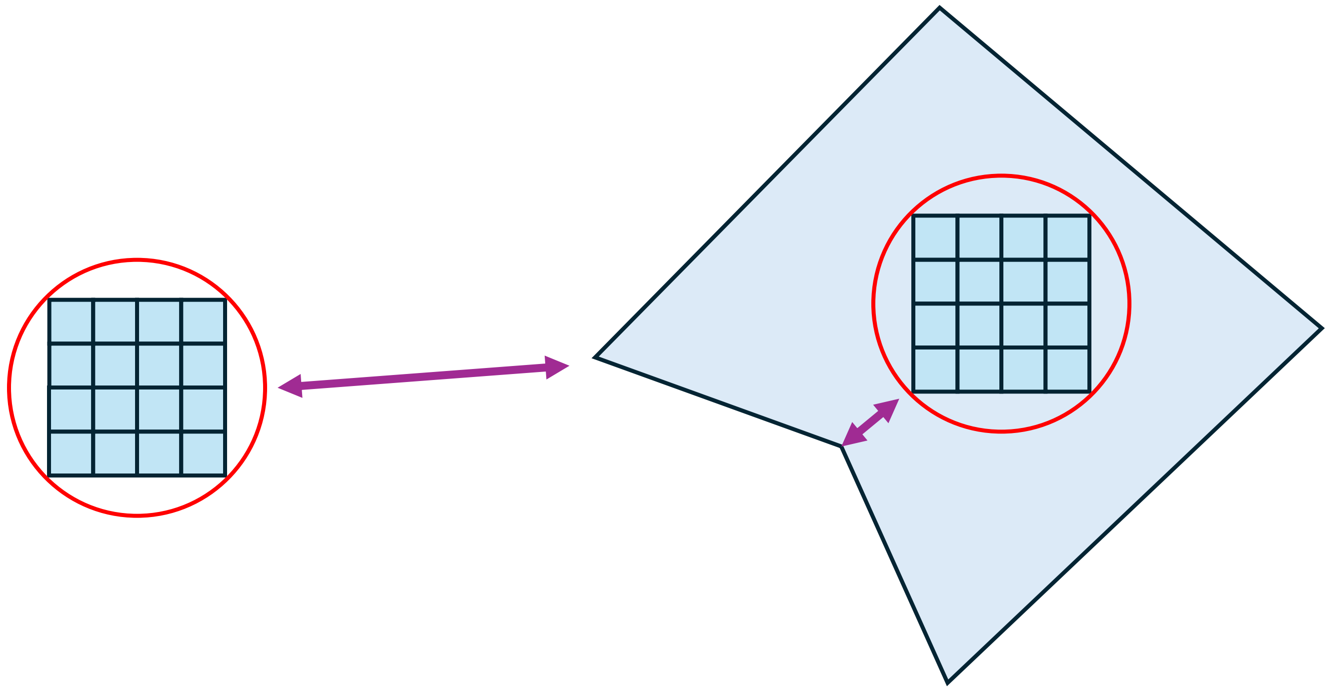
If we have a tile of pixels that is sufficiently far away from the polygon boundaries, it is either entirely inside, or outside, so checking a single point is sufficient



The winding number algorithm - optimizations

Acceleration structures / early reject:

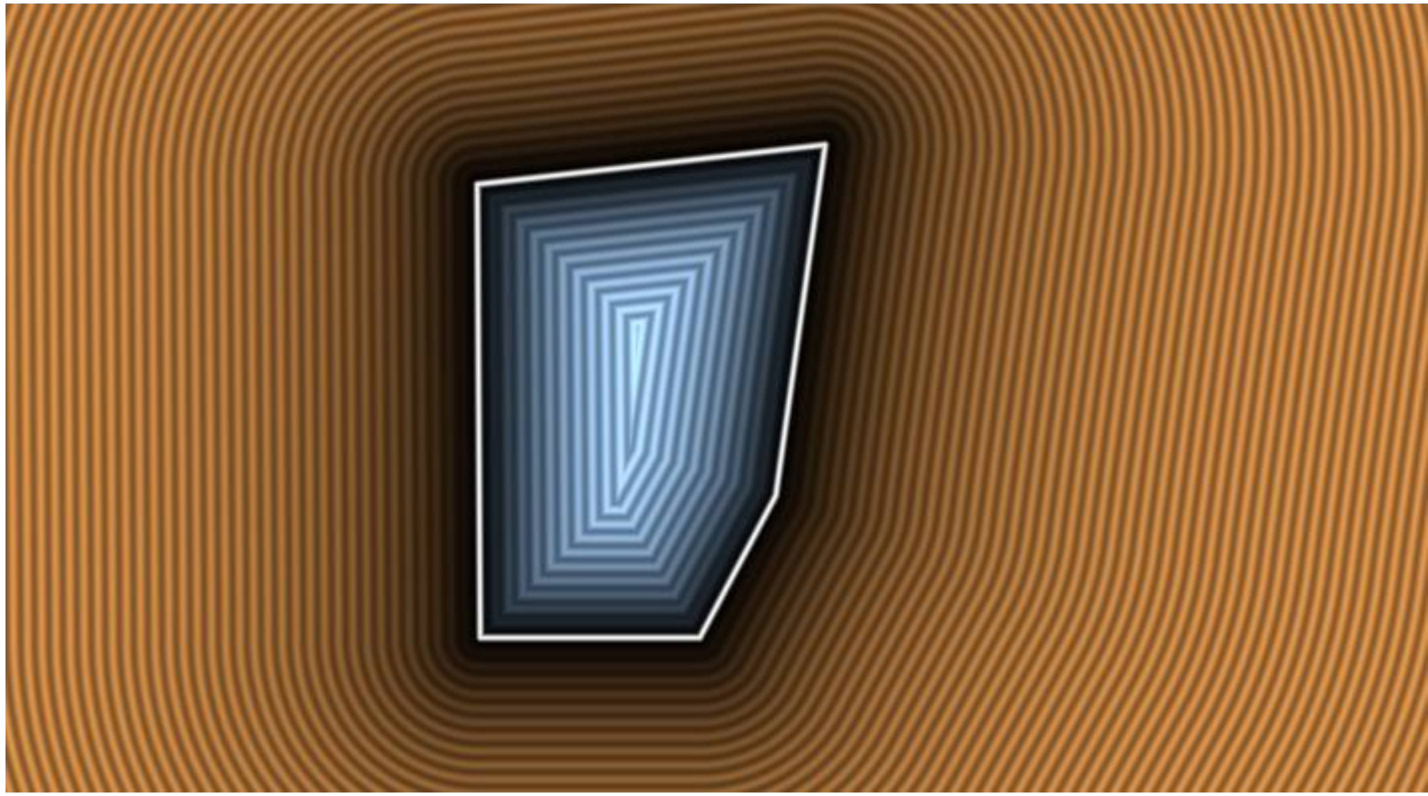
We could compute the minimal distance of the polygon edges to the radius of the circumscribed circle of a square tile:



The winding number algorithm - optimizations

Signed Distance Fields: an easy to compute function that measures distance from a shape
(+) outside
(-) inside

See Inigo Quilez's webpage for formulas and articles

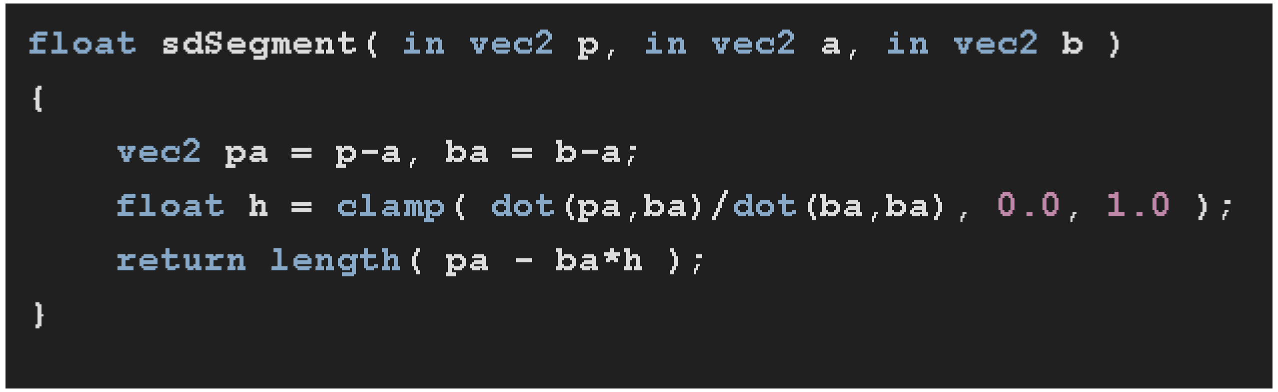


Shadertoy – SDF arbitrary 2D polygon by wagyx

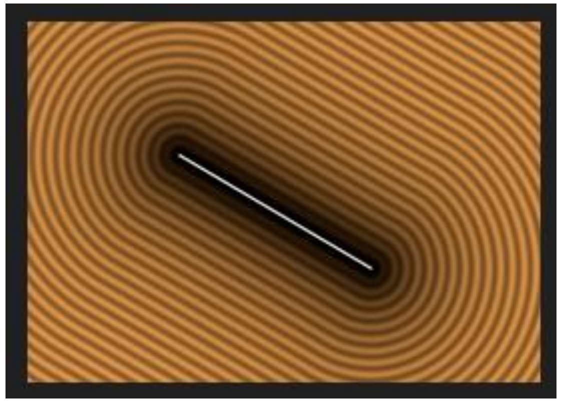
The winding number algorithm - optimizations

Signed Distance Fields: an easy to compute function that measures distance from a shape

We could take the distance function of segments, and compute the minimum of them



```
float sdSegment( in vec2 p, in vec2 a, in vec2 b )
{
    vec2 pa = p-a, ba = b-a;
    float h = clamp( dot(pa,ba)/dot(ba,ba), 0.0, 1.0 );
    return length( pa - ba*h );
}
```



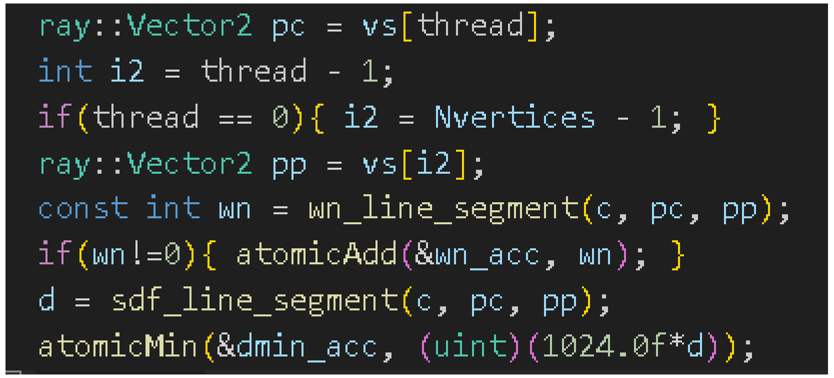
https://iquilezles.org/articles/distfunctions2d

The winding number algorithm - optimizations

The key insight is the following:

Both the winding number contributions and the line segment SDFs can be computed in parallel for all the edges!

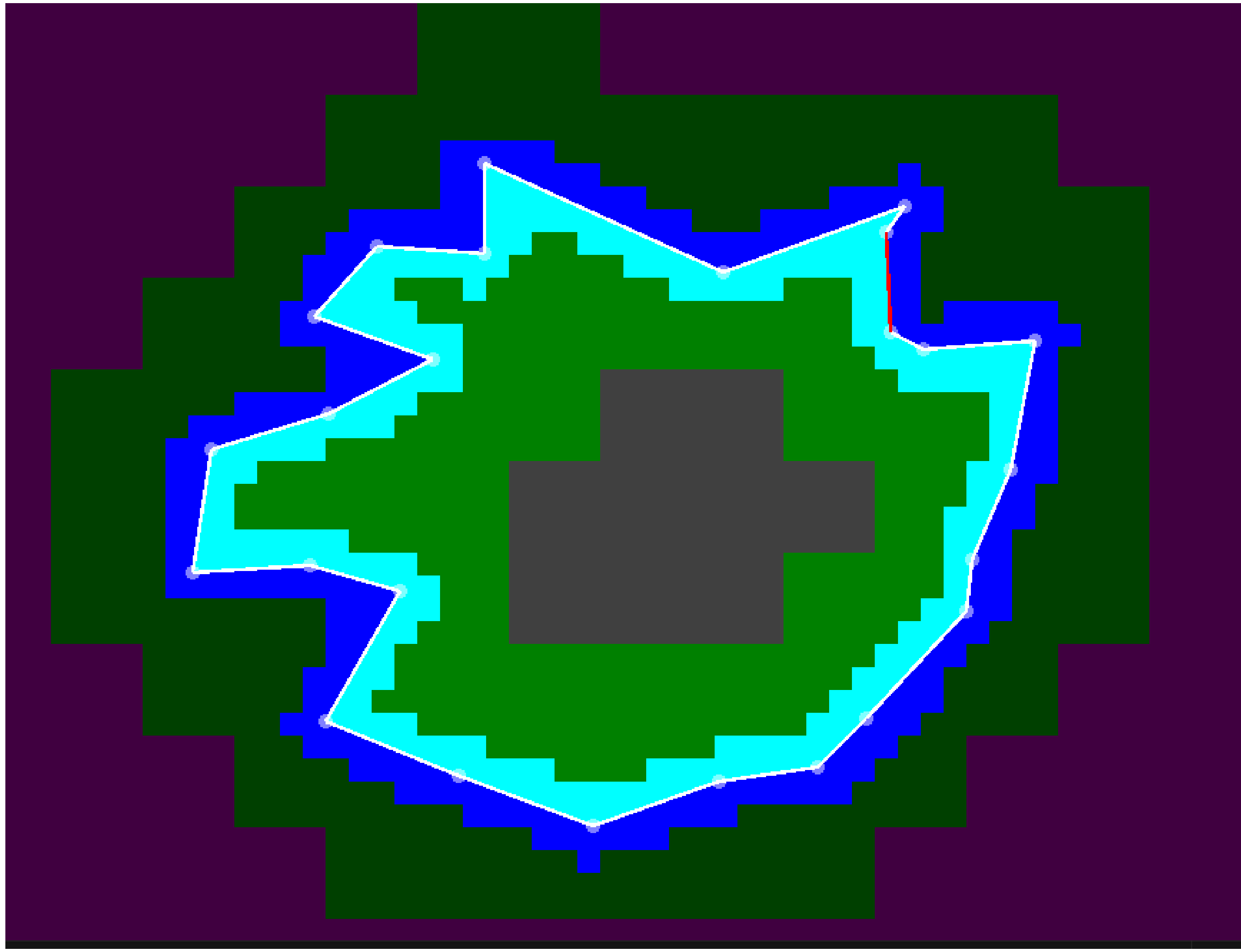
Our threads in a thread block can each pick a segment and compute these simple formulas for the edge’s contribution, and we just need to accumulate these partial results.



```
ray::Vector2 pc = vs[thread];
int i2 = thread - 1;
if(thread == 0){ i2 = Nvertices - 1; }
ray::Vector2 pp = vs[i2];
const int wn = wn_line_segment(c, pc, pp);
if(wn!=0){ atomicAdd(&wn_acc, wn); }
d = sdf_line_segment(c, pc, pp);
atomicMin(&dmin_acc, (uint)(1024.0f*d));
```

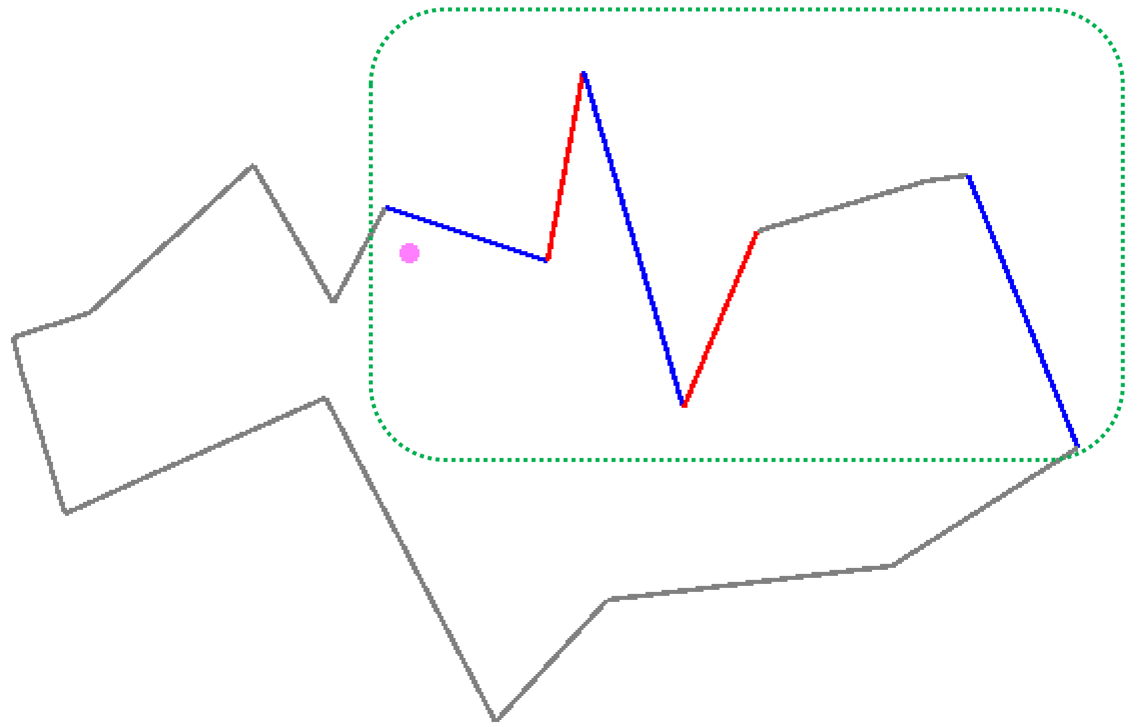
The winding number algorithm - optimizations

With large enough tiles, we can reject large areas, but close to the edges we still need to handle all edges for all pixels…



The winding number algorithm - optimizations

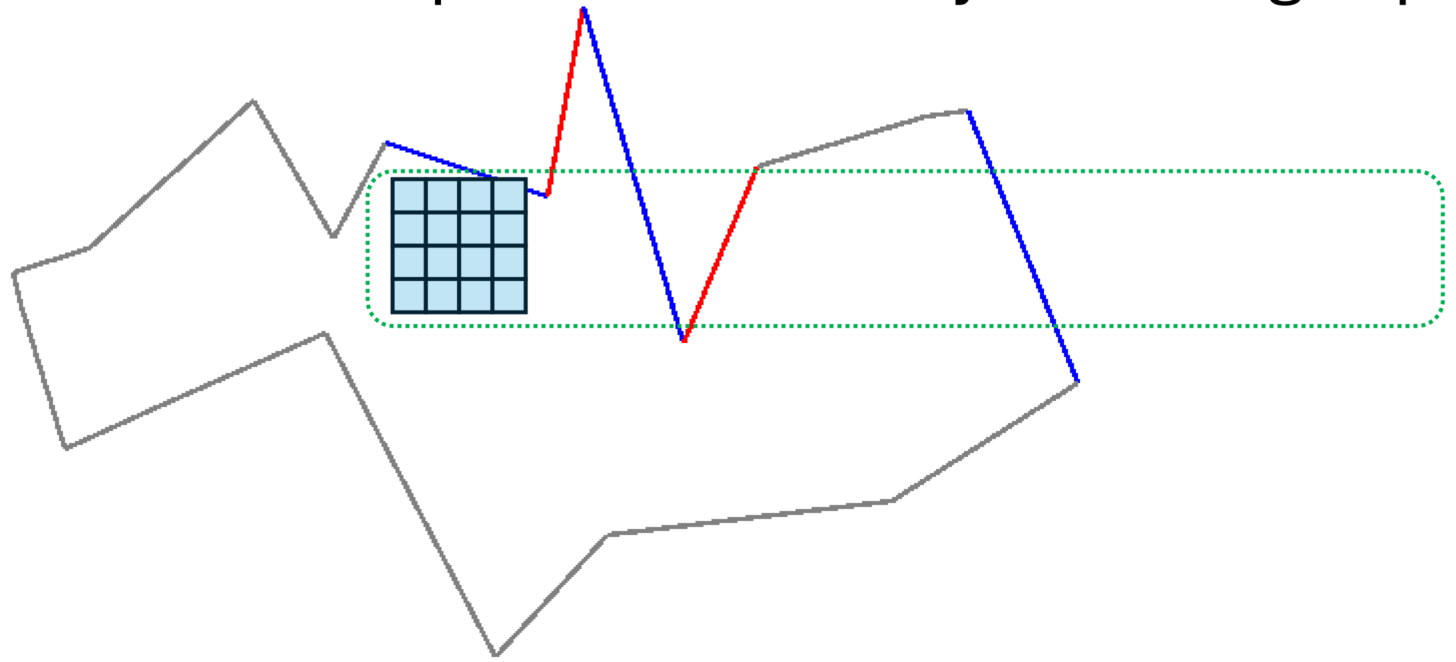
It turns out, that only those edges contribute non-zero to the winding number, that are to the right of the point
and whose end’s y values are on two sides of the point’s y coords.



The winding number algorithm - optimizations

So, we can filter all the edges (again can be done in parallel) if they are relevant to a particular tile or not:

Usually this narrows down the pixel tests to only a few edges per tile



Results

Let’s compare the point in&out rasterization timings (poly: 126 edges):

	RTX 4080 Super 4.15 Mpx	GTX 1650 Mobile 2.07 Mpx
Naïve:	0.456 ms	2.09 ms
Tiled:	0.065 ms	0.32 ms
Edge filtered:	0.033 ms	0.22 ms
Tiled+Edge filtered:	0.030 ms	0.10 ms
	30% of ref mem BW	68% of ref mem BW
Reference: vct clear:	0.01 ms	0.068 ms

15x-20x speed-up

Summary

- When porting algorithms to the GPU, first consider the high-level algorithms, and how they can be parallelized efficiently
- Even without micro-optimizations, very large gains are possible over naïve implementations
- Fast image processing codes are in dire need all over the world for various applications

Hungarian GPGPU Community on Discord

- Meeting place for all who are involved in, or would like to learn more of GPGPU technologies
- All fields: Graphics, Compute, Machine Learning, ...
- All APIs: CUDA, OpenCL, Vulkan, OpenGL, ...
- All vendors: AMD, Intel, NVIDIA, ...
- All levels: Students, Teachers, Professionals, Scientists, ...
- All topics: Programming, Bugs, Optimizations, Teaching, Trends, Events, News, ...

Magyar GPGPU Közösség
Discord szerver
https://discord.gg/eX8uHsmUwd