SIGGRAPH ASIA 2020 VIRTUAL

Pre-recorded sessions:
From 4 December 2020

Live sessions:
10 – 13 December 2020

SA2020.SIGGRAPH.ORG
#SIGGRAPHAsia | #SIGGRAPHAsia2020

Fast and Robust Mesh Arrangements using Floating-point Arithmetic

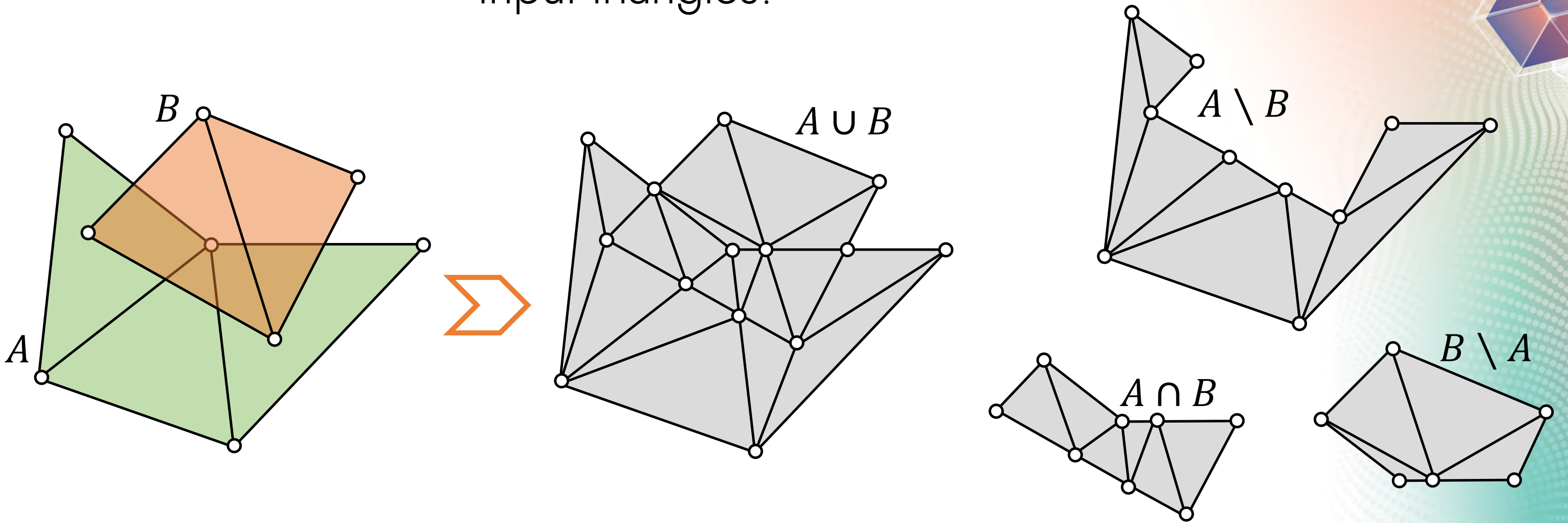
G. Cherchi [1], M. Livesu [2], R. Scateni [1], M. Attene [2]

1 University of Cagliari, Italy

2 IMATI-CNR, Italy

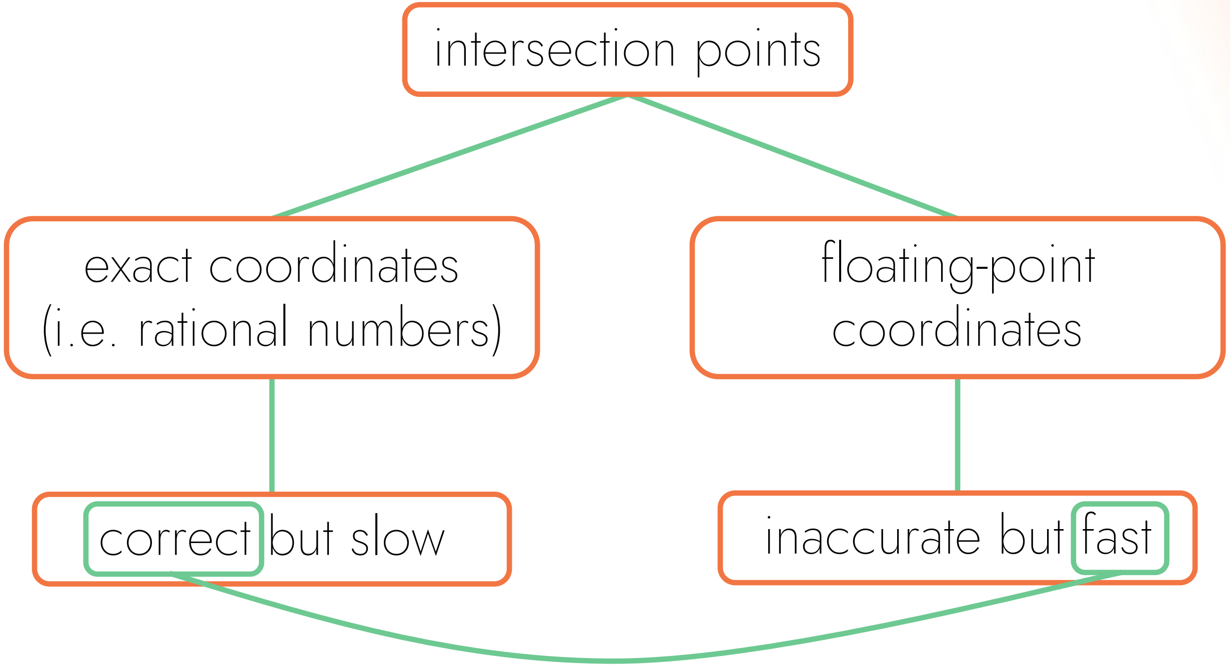
Mesh arrangements

Starting from a generic set of triangles with no assumptions (with self-intersections, degenerate, etc.) we want a subdivision of the space into topologically sound cells bounded by the input triangles.



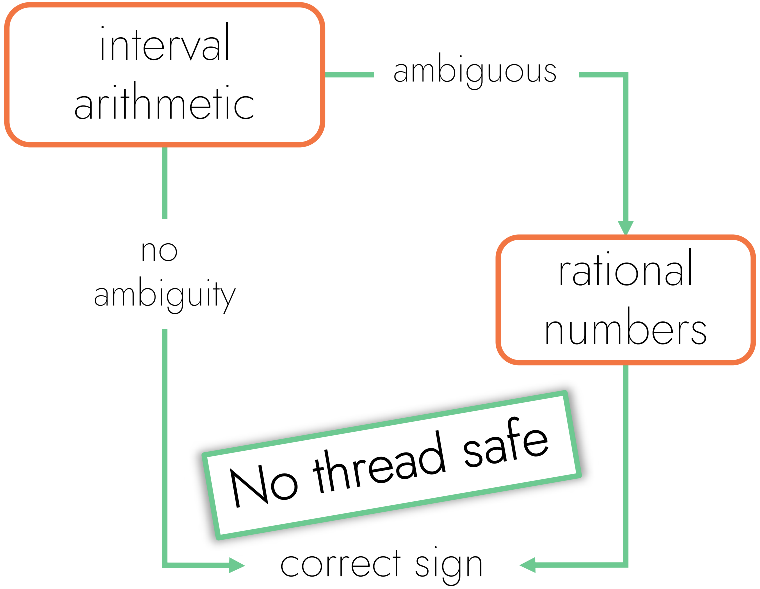
The main problem

Representing intersection points: 2 families of algorithms.



State of the Art

The CGAL solution: *lazy* evaluation



What we want?

- pure floating-point computation (3-8x faster than interval arithmetic)
- interval arithmetic as a second choice
- no rational numbers (we use floating-point hardware expansions)

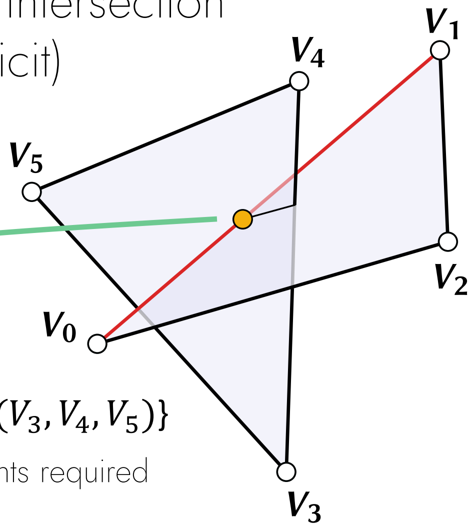
Contribution

Point representation

1. explicit point

$\{x, y, z\}$

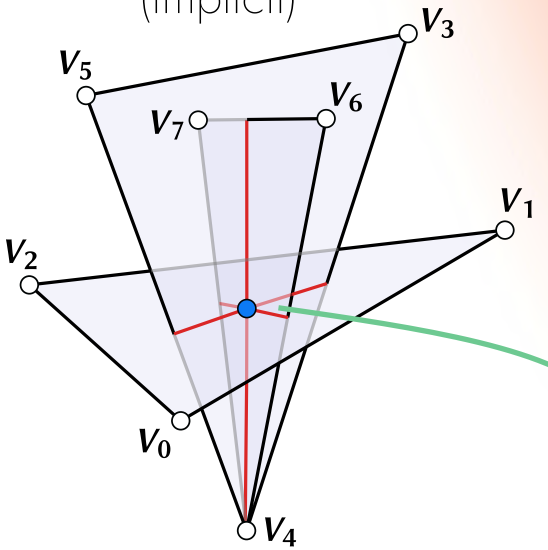
2. line-plane intersection (implicit)



$\{L(V_0, V_1), P(V_3, V_4, V_5)\}$

5 explicit points required

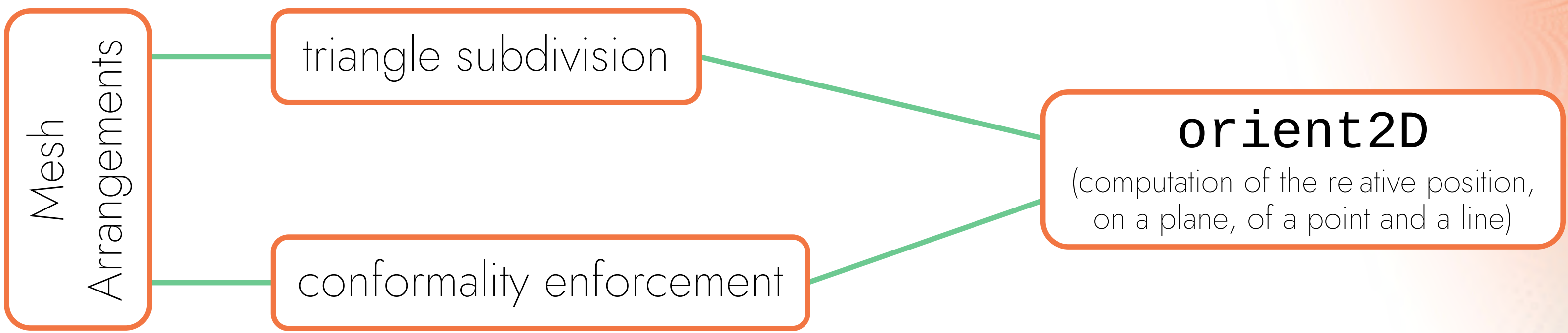
3. three planes intersection (implicit)



$\{P(V_0, V_1, V_2), P(V_3, V_4, V_5), P(V_6, V_7, V_8)\}$

9 explicit points required

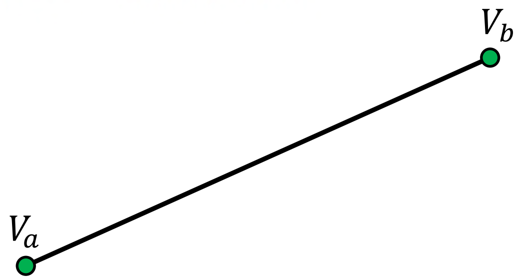
Point orientation



2D problem:

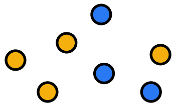
- **robustly** compute triangle normal orientation
- **orthogonal** projection of the elements
- **generalized** 2D orientation (indirect predicates, based on [Attene 2020])
  - works with a mix of explicit and implicit points

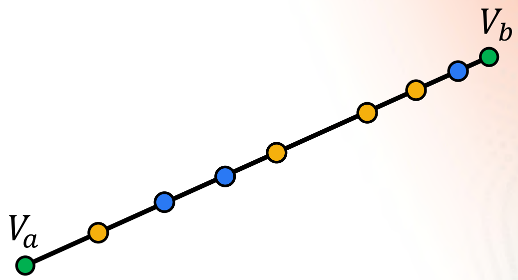
Point sorting




```
pointCompare(a,b)
```
(determines if a is smaller, equal to or larger than b)

implicit points in $e(V_a, V_b)$



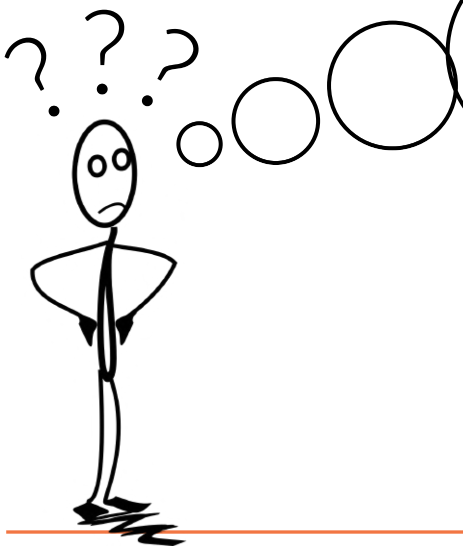


2D problem:

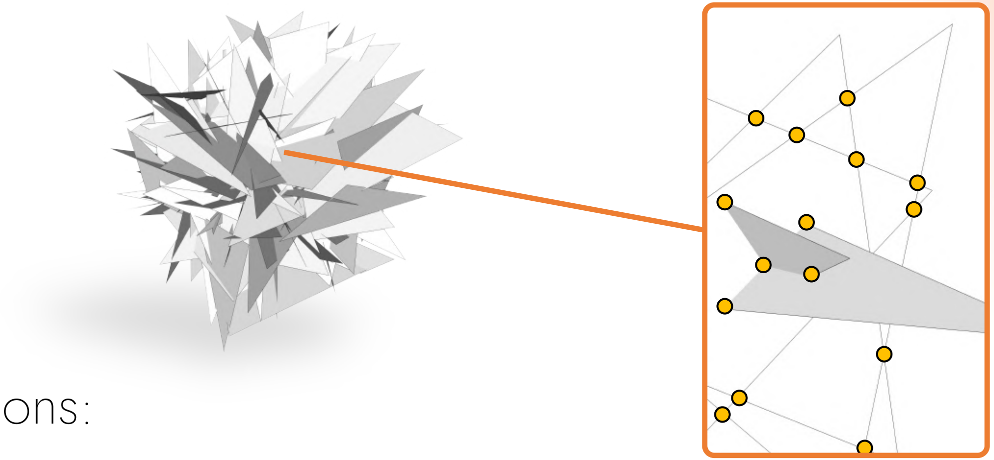
- **generalized** point comparator
- indirect predicates working with a mix of explicit and implicit points

Mesh Arrangements

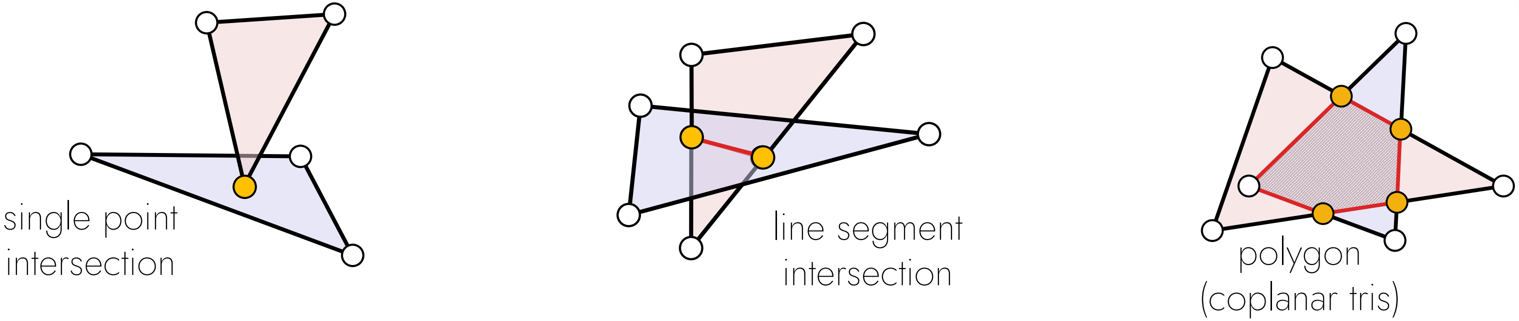
How can we compute the Mesh Arrangements ***without*** *the* ***coordinates*** *of* ***intersection points****?*



Intersection localization

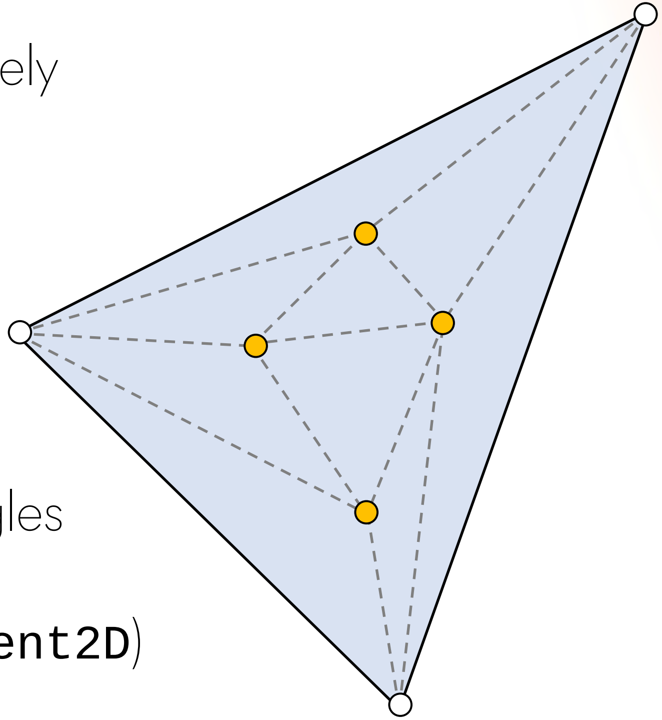


3 possible intersections:



Splitting triangles

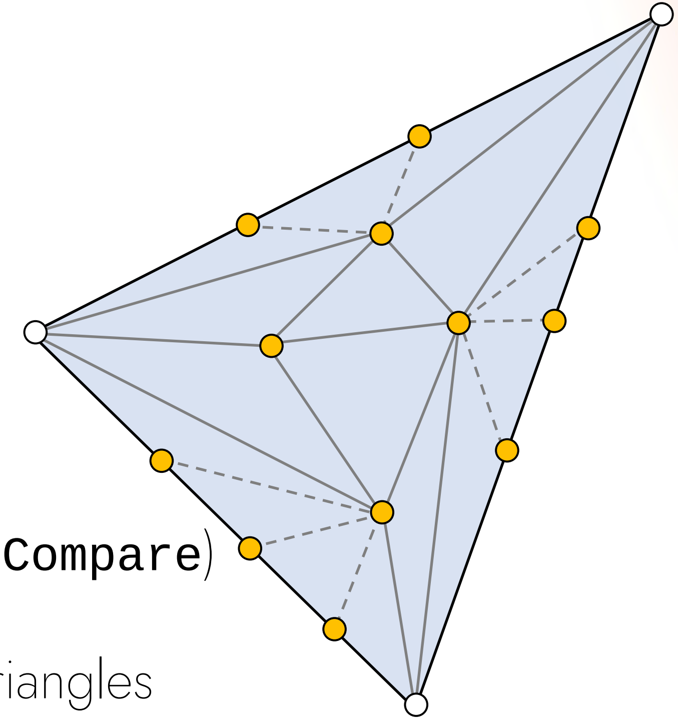
Each triangle is processed separately



- single triangle split in sub-triangles
- **exact** point-in-triangle test (`orient2D`)

Splitting edges

Each original edge is split independently on each triangle

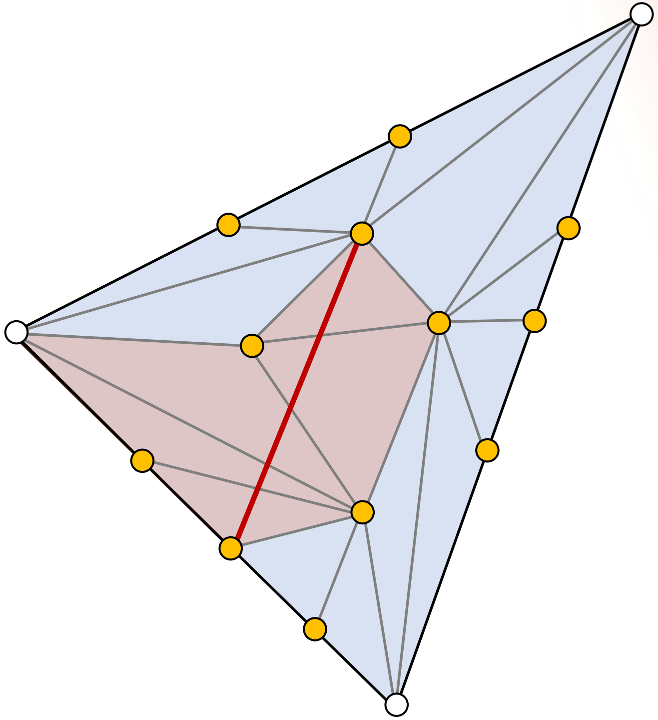


- points on edge sorted (`pointCompare`)
- adjacent triangles split in sub-triangles

Adding intersection segments

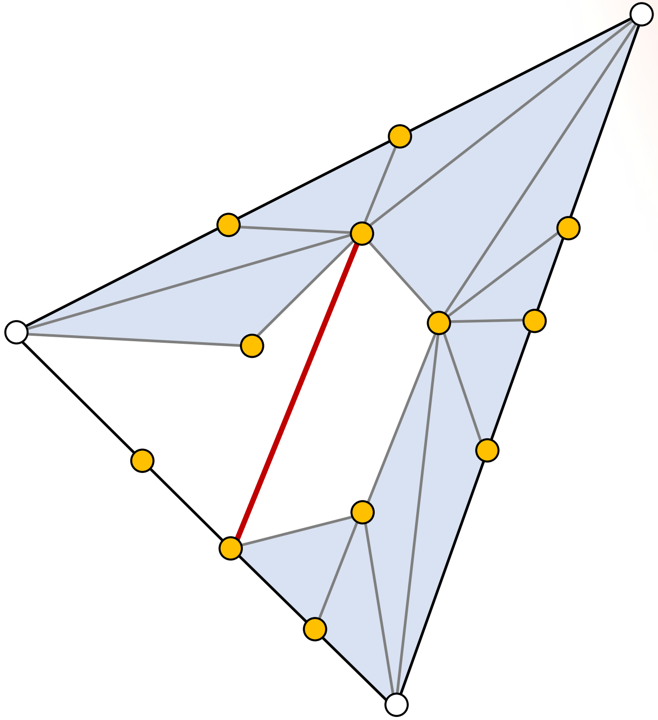
Intersection segments are defined by two intersecting triangles

- we select the triangles intersecting the segment



Adding intersection segments

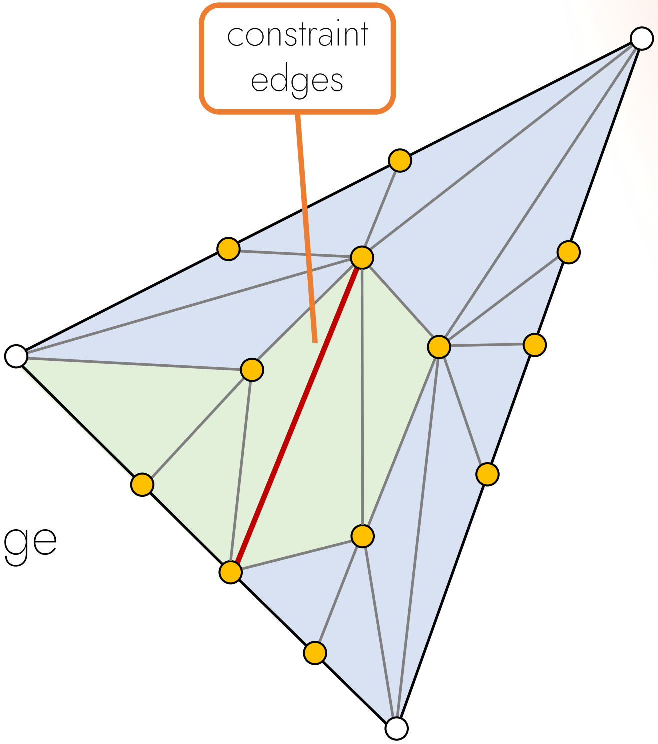
Intersection segments are defined by two intersecting triangles



- we remove the selected triangles creating two voids in the mesh

Adding intersection segments

Intersection segments are defined by two intersecting triangles

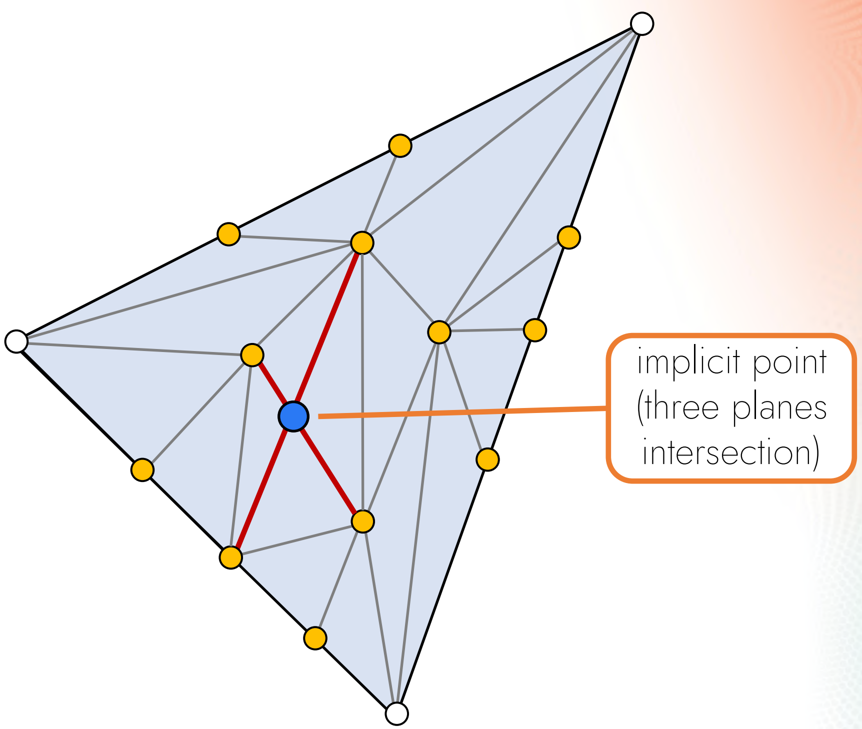


- we triangulate the pockets including the segment as an edge in the mesh

- the segment is marked as constraint edge

Adding intersection segments

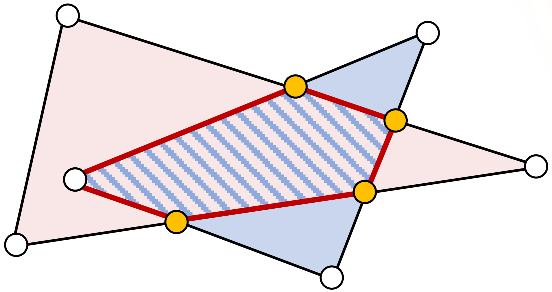
If a constraint segment intersect a previously inserted intersecting segment…



- each constraint edge is defined by two intersecting triangles
- a new **implicit point** of type 3

Coplanar triangles

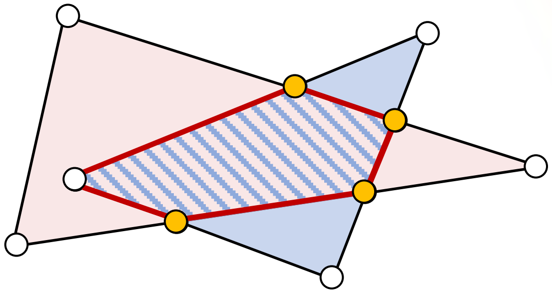
Each triangle is processed separately



Coplanar triangles

Each triangle is processed separately

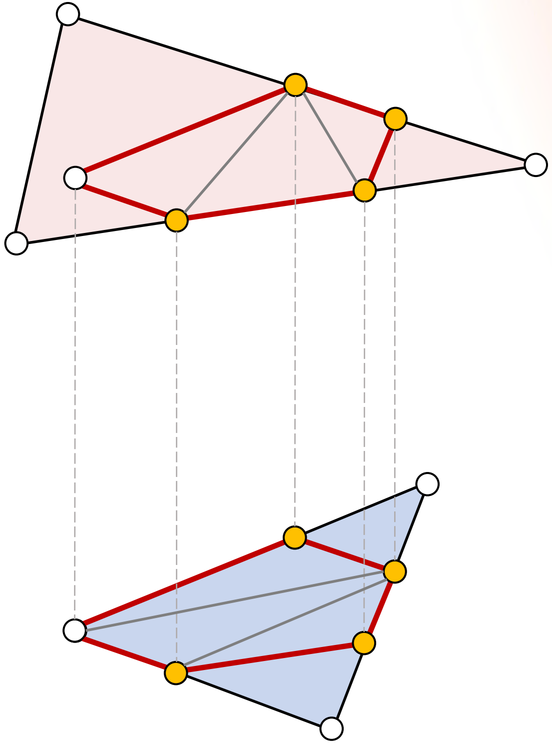
- we keep track of coplanar pockets



Coplanar triangles

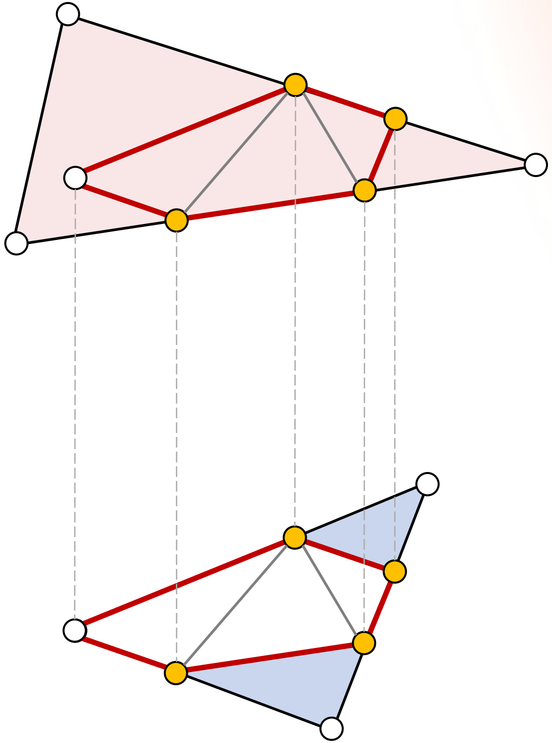
Each triangle is processed separately

- we tessellate each pocket separately



Coplanar triangles

Each triangle is processed separately

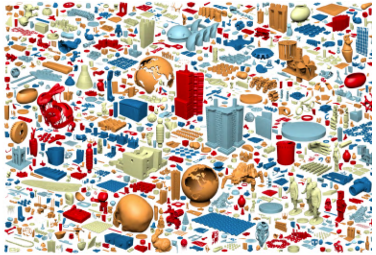


- we use **only one** tessellation for triangles sharing the same pocket

Results

Results

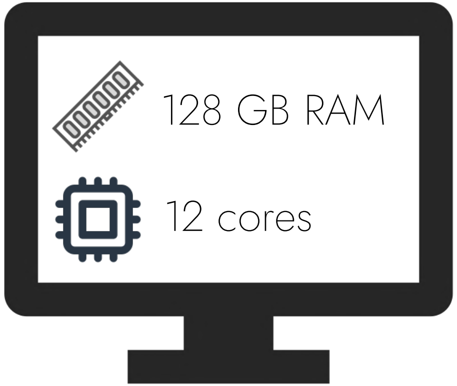
Tests on:



Thingi10K dataset

[Zhou and Jacobson 2016]

- 1000 models
- 4407(+1) models with self intersections



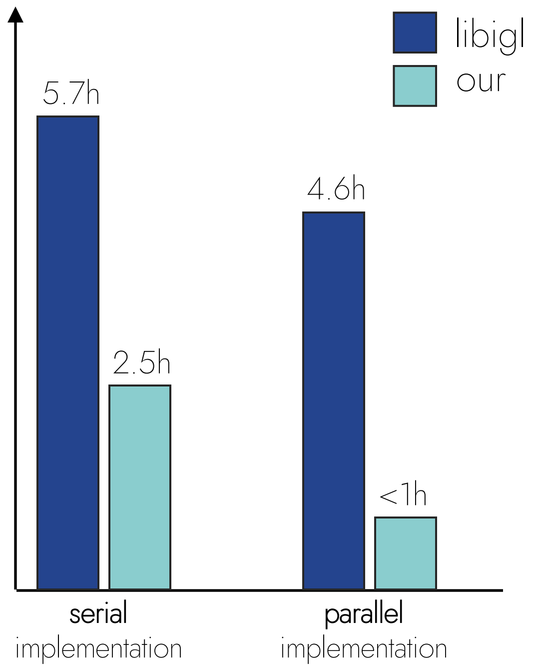
ImatiSTL [Attene 2017] + **CinoLib** [Livesu 2019]

VS

libigl [Panozzo and Jacobson 2014] + **CGAL**

(lazy evaluation)

Comparisons



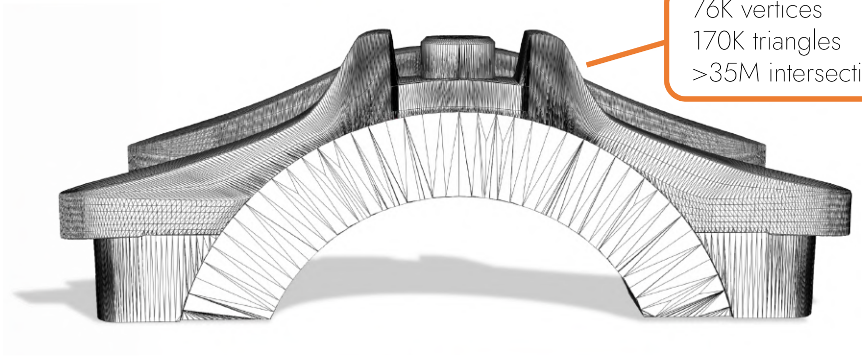
Serial version:
we run faster in 99% of the models

Parallel version:
we run faster in 94% of the models

Our serial implementation is faster than parallel libigl in 63% of the models

Serial libigl is faster than our serial in 31 small models and in 1 model with 1.7M of intersections of type 3.
We are faster in parallel-vs-parallel version

Challenging models



76K vertices
170K triangles
>35M intersections

ours serial: <4h (22GB)
ours parallel: <1h (23GB)

libigl: out of memory after 7h (>100GB)

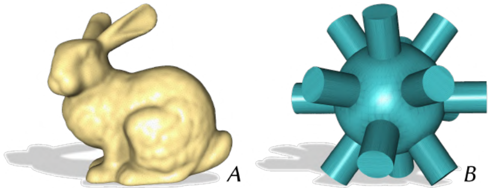
ID	Int.	Timing		Memory		Ratio	
		Ours	libigl	Ours	libigl	time	mem
252784	2,074,680	104.66	1,162.34	2,471.65	10,654.76	9.00%	23.20%
101633	1,712,644	868.46	1,378.00	1,947.55	6,408.16	63.02%	30.39%
55928	1,160,227	87.67	764.80	1,092.00	4,398.07	11.46%	24.83%
1368052	1,034,695	120.08	916.09	4,395.86	9,112.31	13.11%	48.24%
498461	463,958	18.68	157.37	568.86	2,266.13	11.87%	25.10%
338910	434,923	7.74	186.62	528.58	2,109.12	4.15%	25.06%
252785	403,159	24.25	219.81	519.88	1,932.96	11.03%	26.90%
498460	352,430	12.02	130.41	504.64	1,768.93	9.22%	28.53%
242236	239,831	49.96	206.31	1,137.13	1,466.49	24.22%	77.54%
242237	239,644	49.11	201.83	1,129.47	1,470.90	24.33%	76.79%

10 most challenging models

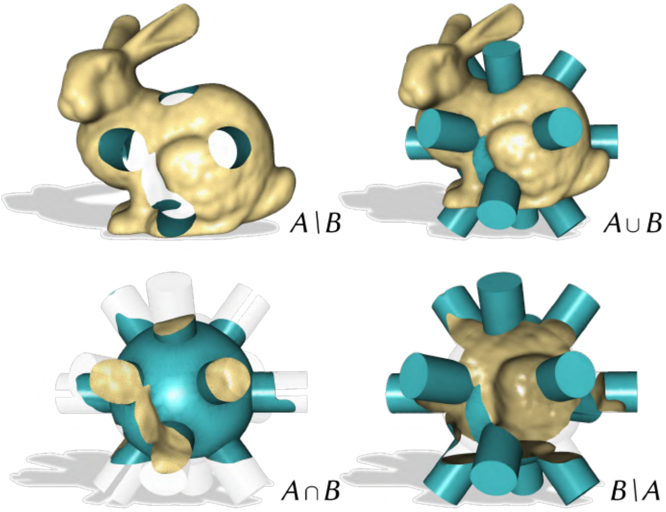
time ratio: 9% - 63% (avg 18%)

mem ratio: 23% - 77% (avg 39%)

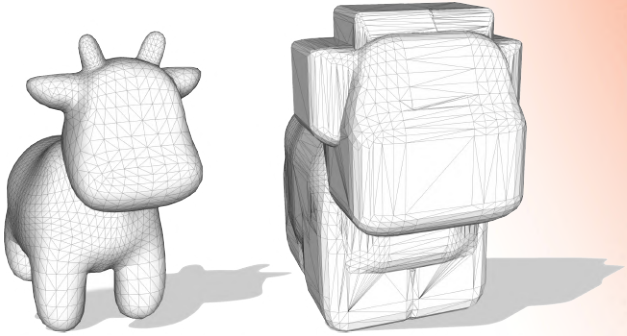
Applications



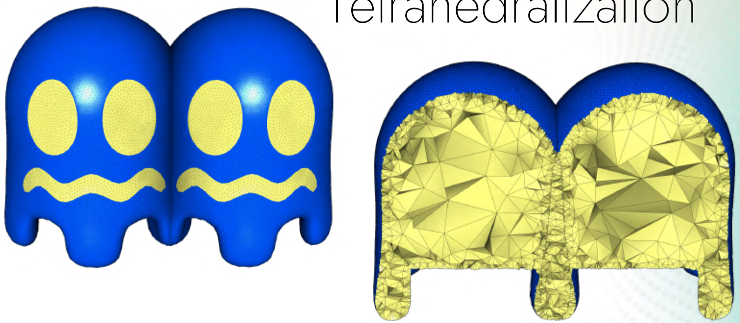
Booleans



Sweeping,
Minkowski Sums



Tetrahedralization



Conclusions

Code is available!

A novel algorithm for **robust** and **efficient** mesh arrangements computation

scan me!

github.com/gcherchi/FastAndRobustMeshArrangements

Future works

- Conversion from implicit to explicit point: **Snap rounding** problem
- Extension of the input to segments, points and generic polygons
- In-Circle indirect predicate -> constrained Delaunay triangulation
- Re-engineering of code and parallel version improvement

Thanks!