Rensselaer Polytechnic Institute, Troy NY USA
Universidade Federal de Vicosa, MG, Brasil

RPI

UFV

Exact fast parallel intersection of large 3-D triangular meshes

Salles V. G. de Magalhães, UFV/RPI
W Randolph Franklin, RPI
Marcus V. A. Andrade, UFV

- Our background: GIS, geometry, graphics, emphasizing fast parallel algorithms on large datasets.
- This is Salles's PhD work.

Contribution

- Intersect 3D meshes while
- Handling geometric degeneracies, including
  - Mesh with itself
  - Mesh with its translation
  - Mesh with its rotation
- With no roundoff errors
- Fast in parallel
- Economical of memory

Example: Intersection of two big meshes from AIM@SHAPE:

Ramesses: 1.7 million triangles x Neptune: 4 million triangles. 5.5 seconds on multicore Xeon.

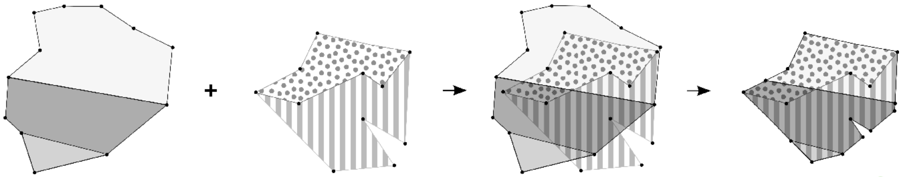
Five key techniques

•Arbitrary precision rational numbers: for exactness.
•Simulation of Simplicity: for ensuring all the special cases are properly handled.
•Simple data representation and local information: parallelization and correctness.
•Uniform grid: accelerate computation; quickly constructed in parallel.
•Parallel programming

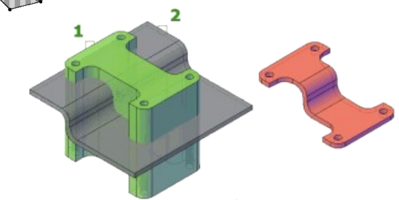
Hard part: making everything fit together.

Mesh intersection

- Polygonal map overlay/intersection: important CAD/GIS problem

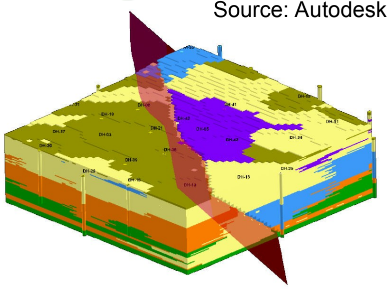


- 2D intersection also extends to 3D.



Source: Autodesk

- Applications: CAD, Additive Manufacturing, GIS, cross-interpolation after remeshing in CFD



- Our focus: 3D triangulated meshes

Source: Rockworks

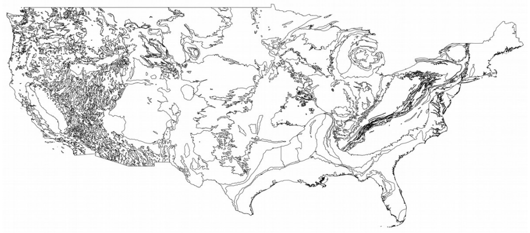
Initial step:

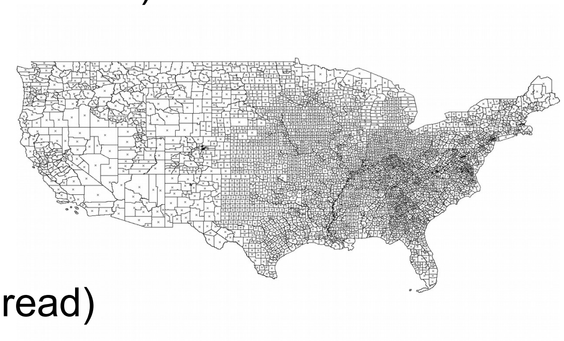
EPUG-Overlay: 2D planar graph overlay

Previous step, presented at 2015 ACM BIGSPATIAL

Biggest example:

- USWaterBodies: 21,652,410 vertices, 219,831 faces, with
- USBlockBoundaries: 32,762,740 vertices, 518,837 faces.
- (Images are of simpler similar datasets):



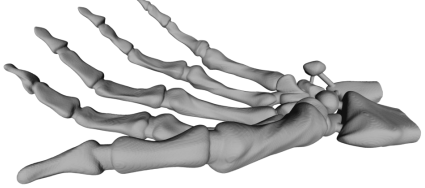


- Time (w/o I/O): 1342 secs (1 thread)
- 149 secs (16 cores, 32 threads).   9X parallel speedup

Another initial step:

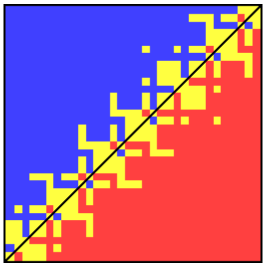
PINMESH: 3D point location

- Previous step, presented at 2016 Berlin Geometry Summit
- Uses rational numbers, Simulation of Simplicity, uniform grid, parallelism, simple data structures
- Biggest example: sample dataset with 50 million triangles.
  - Preprocessing: 14 elapsed seconds on 16-core Xeon.
  - Query time: 0.6 μs per point.
  - Some test datasets:



Problem: Roundoff errors

- Finite precision of floating point →roundoff errors.
  - Common techniques (snap rounding, epsilon tweaking, etc): no guarantee.



Source: Kettner et al., Classroom examples of robustness problems in geometric computations

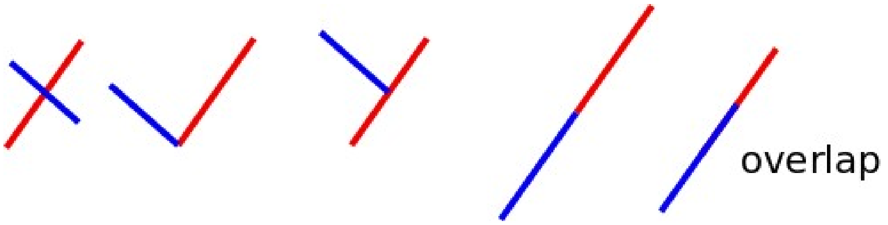
- Big amount of data & 3D→ increase problem.
- Exactness and performance: very important (e.g. guaranteed subroutine)

Our solution: Rational numbers

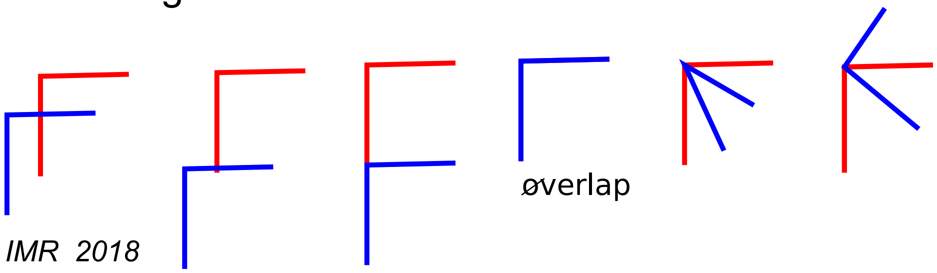
Each component of each coordinate is a ratio of integers

- No rounding or finite precision errors.
- Each integer: array of groups of digits
- Uses GMPXX
- Rationals double in size with each operation: 2/3+4/5=22/15
- However depth of computation tree is small
- Problem: GMPXX liberally constructs new objects on heap
- (Have these packages been used on large examples before?)
- Heap: superlinear time in number of objects, and parallel hostile.
- We minimize heap constructions.
- Increased execution time is tolerable.

Problem: Geometric degeneracies



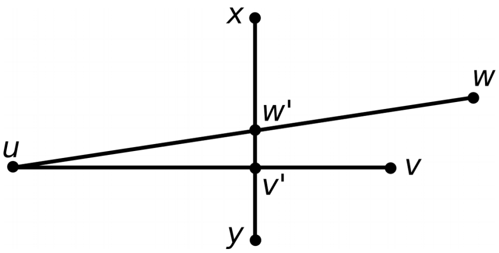
- Ad-hoc enumerating special cases is error-prone.
- How many ways can two line segments intersect?
- That must be usable for comparing two polylines.
- Local rules must lead to a globally consistent result.
- Existing programs can get complicated cases wrong.
- Need a general solution.



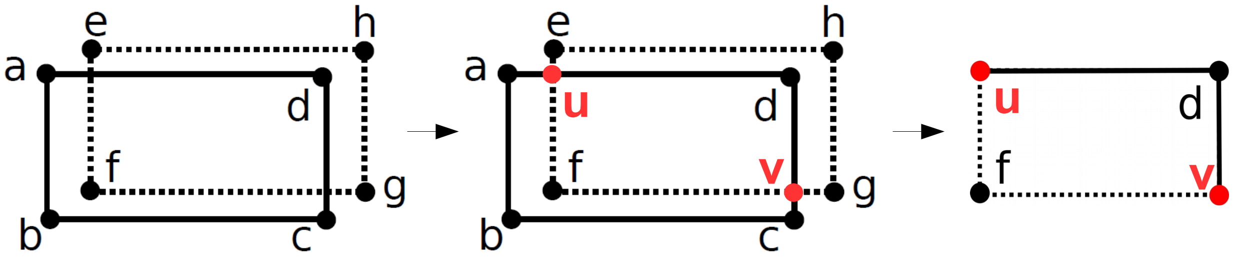
Simulation of Simplicity

- Edelsbrunner and Mücke
- Simple and efficient general purpose technique.
- Globally consistent
- Basic idea: if points are perturbed, the degeneracies in geometrical problems will disappear and do not need to be treated.
- Points are perturbed using orders of infinitesimals ε^i
- Infinitesimal: indeterminate (code simulates the *effect* of the infinitesimals – we do not actually use specific infinitesimals).

Global consistency (*uw*, *uv* were coincident):
- w' is on the positive side of uv
- w' is closer to x than v' is



Simulation of Simplicity, ctd



Implementing SoS

- In a predicate:
  - No coincidence → unperturbed result = perturbed result ≠ 0
  - Coincidence → unperturbed result = 0, unperturbed result ≠ 0

- For performance:
  - Two versions of each predicate:
    - One developed for efficiency (standard algorithms from literature)
    - One for simplicity (using as few predicates as possible).

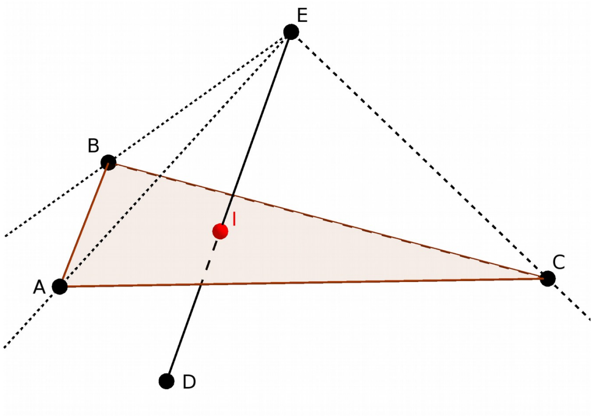
    - The simpler version: used when a coincidence is detected.
    - Consequence: implement SoS only in few predicates.

Implementing SoS

- Challenge:
  - If a vertex of mesh 0 has coordinates (x,y,z), what is its perturbed coordinate? Ans: (x,y,z)
  - If a vertex of mesh 1 has coordinates (x,y,z), what is its perturbed coordinate? Ans: $(x+\varepsilon, y+\varepsilon^2, z+\varepsilon^3)$
  - If a vertex generated by an intersection of a triangle with an edge has coordinates (x,y,z), what is its perturbed coordinate?
    - Ans: ???

→ store these coordinates implicitly

→ process implicit coordinates in the predicates



Problem: Coincidence detection

- Point location:
  - Preprocess a 2D partition, then
  - Locate query point.

    Ideal time: linear preprocess, constant query.
- Intersection culling:
  - Given a large set of small elements (2D edges, 3D triangles),
  - Find the small set of pairs that might intersect.

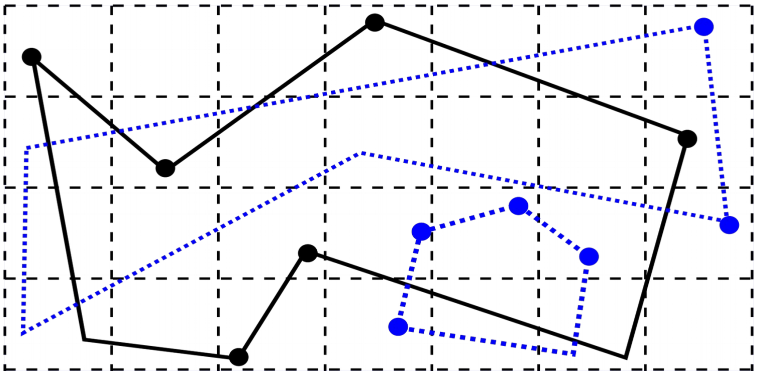
    Ideal time: linear in input plus output sizes.
  - Current solution: quadtrees, octrees, sweep lines
    - Construction doesn’t parallelize well; time too high.

Solution: uniform grid

- Select integer G depending on statistics of data. 4<=G<=1000
- Superimpose GxG or GxGxG grid on data.
- For each of N input elements (line segment, triangle, …), record which grid cells it intersects.
- Use a ragged array (read twice, write once).
- Desired G: constant average number of elements per cell, independent of N.
- Equivalent to grid cell size proportional to element size.
- Time to process one element: constant.
- Total preprocessing time: linear.

Uniform Grid - query

- Insert edges in grid cells (edge may be in several cells).
- For each grid cell *c*, compute intersections in *c*.
- 3D version is analogous
- Provably efficient for I.i.d. input
- Experimentally more efficient on irregular data than octrees



4x7 uniform grid.
Blue map: 8 edges
Black map: 16 edges

Uniform grid time analysis (2D)

- Assume: input elements are uniform I.i.d.
- Average number of elements per cell is constant, by construction.
- Number of elements per grid cell has a Poisson dist.
- Mean of the square of number of elements per cell is constant.
- Time to test all pairs of elements in one cell is constant.
- Expected number of actual intersections per cell is constant.
- Total query time is linear in number of actual intersections.
- Still true for non I.i.d. data satisfying a Lipschitz condition.

Uniform grid time variant

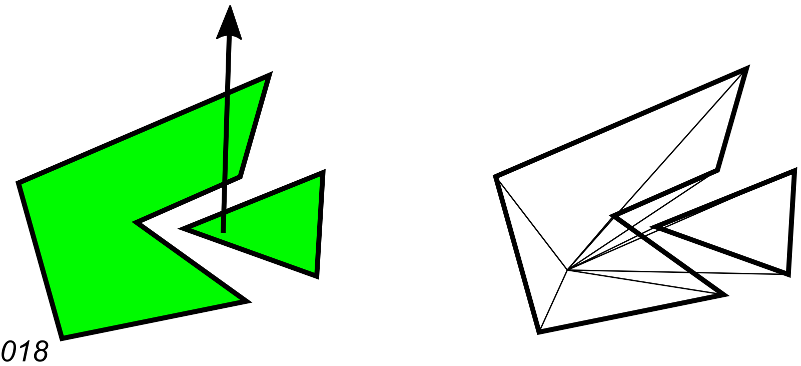
- Exactly and quickly computing the grid cells an oblique 3D triangle intersects was too tedious, so:
- Enclose the triangle with a minimal box and intersect that with the grid, however:
- That finds too many cells, especially when the cells are small, so:
- Use a two-level grid for cells with many elements.
- We tested 3-level grids and octrees (because that’s what everyone wants to know).
- They’re slower.

Problem: complex data structures don’t parallelize well

- Especially on SIMT machines like Nvidia GPUs.
- Prefer arrays of Plain Old Datatypes.
- Disparage pointers, links, trees.
- This work uses multicore Xeons and OpenMP, but we have other algorithms implemented on Nvidia GPUs, and would like to transition this one also.

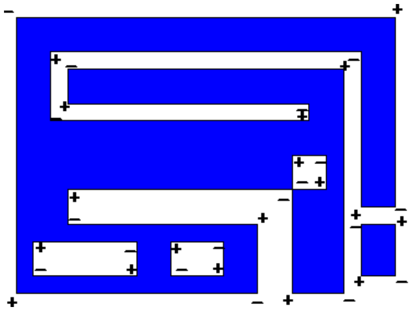
Minimal explicit topology

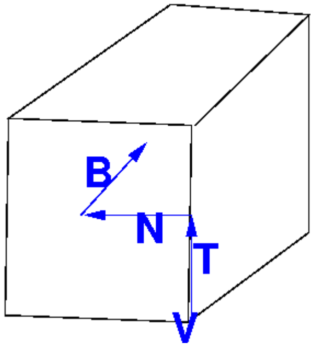
- What’s the simplest representation that permits a solution?
- To test whether a point is in a polygon, the set of unoriented edges suffices.
- Multiple disconnected nested components ok.
- No global connectivity info is stored.
- To compute mass properties, set of oriented edges suffices.



Minimal explicit topology 2

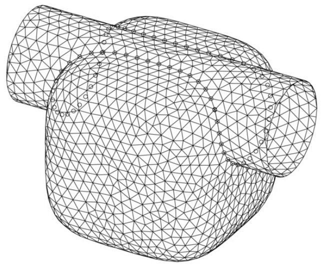
- What's the simplest representation that permits a solution?
- Area of a rectilinear polygon:
- $A = \Sigma_i\, s_i\, x_i\, y_i$
- Volume of a general polyhedron:
- $M = -1/6\ \Sigma\ V.T\ \ V.N\ \ V.B$
- V: vertex position
- T: unit tangent along edge; N: unit normal in face plane; B: unit normal into interior
- Sum over all incidences
- E.g. 48 for cube.
- No loops, shells, edges, faces stored.

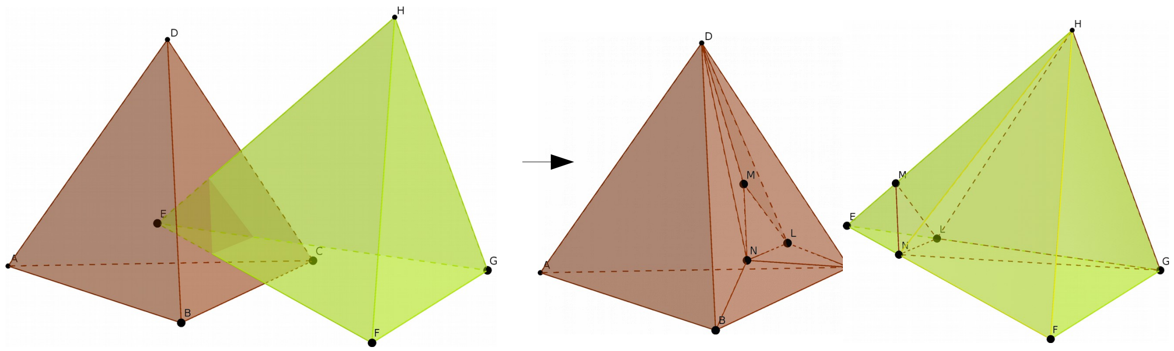




3D-EPUG-OVERLAY Algorithm

- Input: 2 meshes, either polyhedron surfaces or tetrahedrulated.
- Internal data structure: set of triangles.
- Insert triangles into uniform grid.
- Find triangle-triangle intersections.
- Retesselate intersecting faces.
- Classify new triangles.

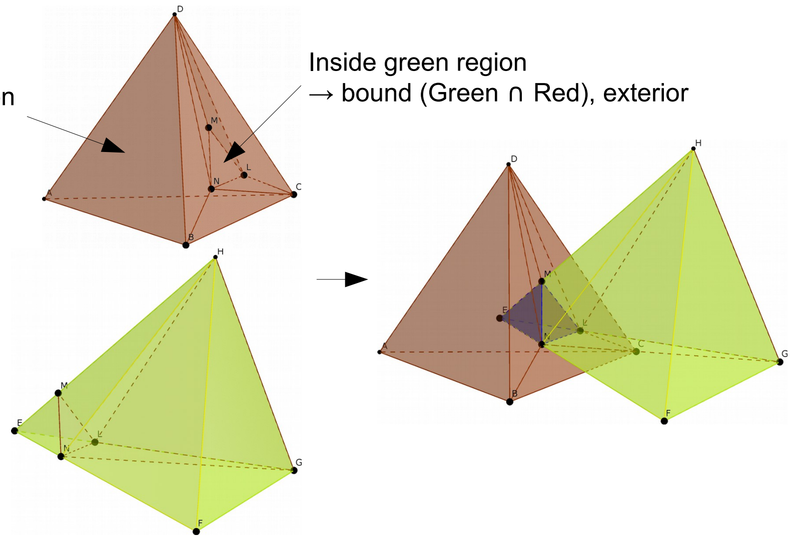




source: wikipedia

Triangle classification

- How to locate a triangle?
  - Simple and fast solution: point location (PinMesh)



Outside green region
→ not in the output

Inside green region
→ bound (Green ∩ Red), exterior

Experiments

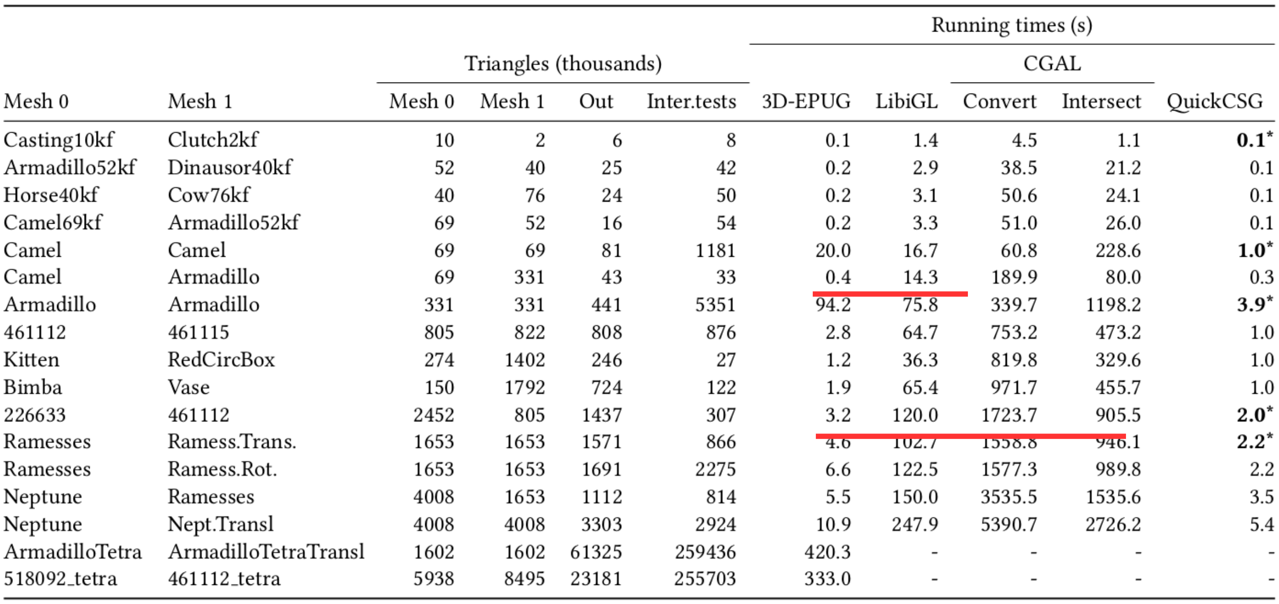
- Algorithm implemented in C++.
  - OpenMP (parallel) + GMPXX (exact coordinates)

- Experiments on a workstation
  - Dual Intel Xeon E5-2687 processors, 8 cores, 2 threads/core
  - 128 GB of RAM.
  - Ubuntu Linux 16.04.

- Comparison with:
  - LibiGL: recent, exact, parallel and resolves self-intersections.
  - CGAL Nef Polyhedra: exact
  - QuickCSG: fast, parallel, but may fail (floating-point errors/do not handle special cases).

Experiments

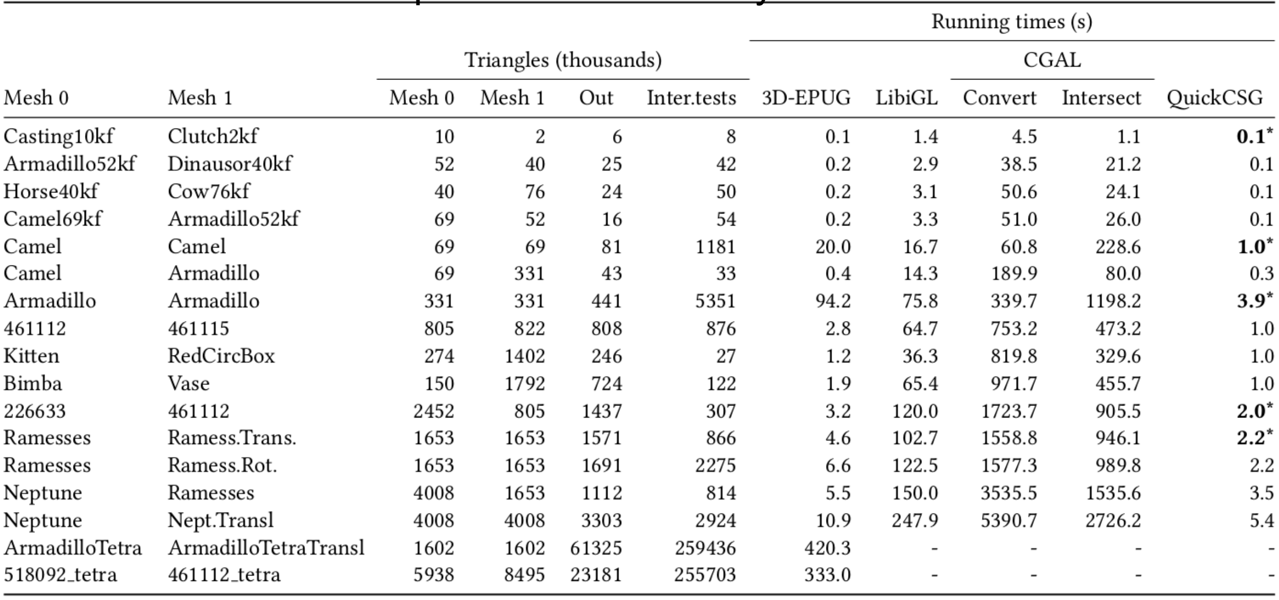
- Up to 37x faster than LibiGL
- Up to 281x faster than CGAL (935x including conversion)
- Slightly slower than LibiGL when a mesh is intersected with itself: too many SoS calls (non-optimized, future work).



						Running times (s)				
		Triangles (thousands)						CGAL		
Mesh 0	Mesh 1	Mesh 0	Mesh 1	Out	Inter.tests	3D-EPUG	LibiGL	Convert	Intersect	QuickCSG
Casting10kf	Clutch2kf	10	2	6	8	0.1	1.4	4.5	1.1	**0.1***
Armadillo52kf	Dinausor40kf	52	40	25	42	0.2	2.9	38.5	21.2	0.1
Horse40kf	Cow76kf	40	76	24	50	0.2	3.1	50.6	24.1	0.1
Camel69kf	Armadillo52kf	69	52	16	54	0.2	3.3	51.0	26.0	0.1
Camel	Camel	69	69	81	1181	20.0	16.7	60.8	228.6	**1.0***
Camel	Armadillo	69	331	43	33	0.4	14.3	189.9	80.0	0.3
Armadillo	Armadillo	331	331	441	5351	94.2	75.8	339.7	1198.2	**3.9***
461112	461115	805	822	808	876	2.8	64.7	753.2	473.2	1.0
Kitten	RedCircBox	274	1402	246	27	1.2	36.3	819.8	329.6	1.0
Bimba	Vase	150	1792	724	122	1.9	65.4	971.7	455.7	1.0
226633	461112	2452	805	1437	307	3.2	120.0	1723.7	905.5	**2.0***
Ramesses	Ramess.Trans.	1653	1653	1571	866	4.6	102.7	1558.8	946.1	**2.2***
Ramesses	Ramess.Rot.	1653	1653	1691	2275	6.6	122.5	1577.3	989.8	2.2
Neptune	Ramesses	4008	1653	1112	814	5.5	150.0	3535.5	1535.6	3.5
Neptune	Nept.Transl	4008	4008	3303	2924	10.9	247.9	5390.7	2726.2	5.4
ArmadilloTetra	ArmadilloTetraTransl	1602	1602	61325	259436	420.3	-	-	-	-
518092_tetra	461112_tetra	5938	8495	23181	255703	333.0	-	-	-	-

Experiments

- Up to 3x slower than QuickCSG (tests without reported failures), but exact.
    - * → QuickCSG failed and reported failure
    - If a failure is not reported → result may still have errors



						Running times (s)				
		Triangles (thousands)						CGAL		
Mesh 0	Mesh 1	Mesh 0	Mesh 1	Out	Inter.tests	3D-EPUG	LibiGL	Convert	Intersect	QuickCSG
Casting10kf	Clutch2kf	10	2	6	8	0.1	1.4	4.5	1.1	**0.1***
Armadillo52kf	Dinausor40kf	52	40	25	42	0.2	2.9	38.5	21.2	0.1
Horse40kf	Cow76kf	40	76	24	50	0.2	3.1	50.6	24.1	0.1
Camel69kf	Armadillo52kf	69	52	16	54	0.2	3.3	51.0	26.0	0.1
Camel	Camel	69	69	81	1181	20.0	16.7	60.8	228.6	**1.0***
Camel	Armadillo	69	331	43	33	0.4	14.3	189.9	80.0	0.3
Armadillo	Armadillo	331	331	441	5351	94.2	75.8	339.7	1198.2	**3.9***
461112	461115	805	822	808	876	2.8	64.7	753.2	473.2	1.0
Kitten	RedCircBox	274	1402	246	27	1.2	36.3	819.8	329.6	1.0
Bimba	Vase	150	1792	724	122	1.9	65.4	971.7	455.7	1.0
226633	461112	2452	805	1437	307	3.2	120.0	1723.7	905.5	**2.0***
Ramesses	Ramess.Trans.	1653	1653	1571	866	4.6	102.7	1558.8	946.1	**2.2***
Ramesses	Ramess.Rot.	1653	1653	1691	2275	6.6	122.5	1577.3	989.8	2.2
Neptune	Ramesses	4008	1653	1112	814	5.5	150.0	3535.5	1535.6	3.5
Neptune	Nept.Transl	4008	4008	3303	2924	10.9	247.9	5390.7	2726.2	5.4
ArmadilloTetra	ArmadilloTetraTransl	1602	1602	61325	259436	420.3	-	-	-	-
518092_tetra	461112_tetra	5938	8495	23181	255703	333.0	-	-	-	-

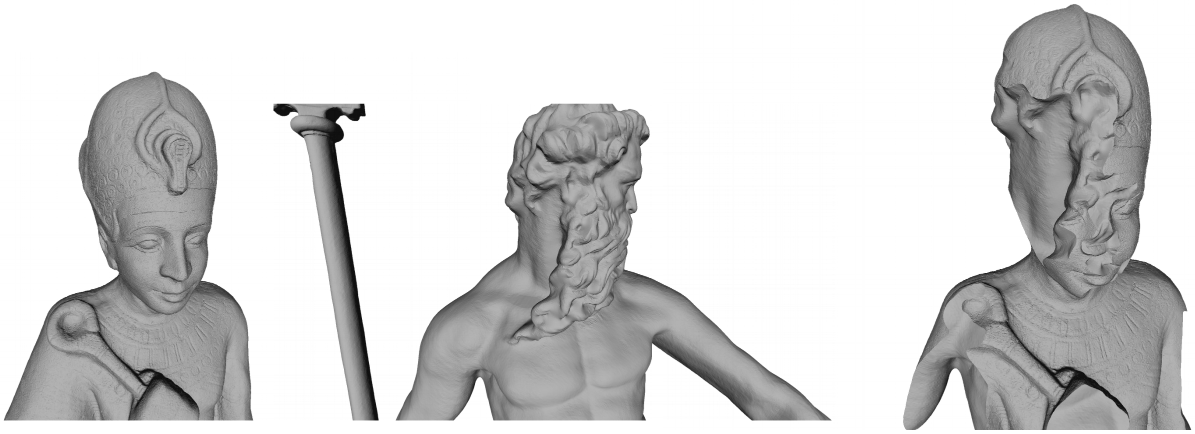
Experiments

- Can process meshes with millions of triangles in few seconds.
- Can handle tetra-meshes (461112_tetra: 8 M triangles, 4 M tets).
- Memory efficient: Neptune vs Neptune translated: 3D-EPUG: 5GB of RAM, LibiGL: 22.5GB, CGAL: 110GB, QuickCSG: 4.5GB

						Running times (s)				
		Triangles (thousands)						CGAL		
Mesh 0	Mesh 1	Mesh 0	Mesh 1	Out	Inter.tests	3D-EPUG	LibiGL	Convert	Intersect	QuickCSG
Casting10kf	Clutch2kf	10	2	6	8	0.1	1.4	4.5	1.1	**0.1***
Armadillo52kf	Dinausor40kf	52	40	25	42	0.2	2.9	38.5	21.2	0.1
Horse40kf	Cow76kf	40	76	24	50	0.2	3.1	50.6	24.1	0.1
Camel69kf	Armadillo52kf	69	52	16	54	0.2	3.3	51.0	26.0	0.1
Camel	Camel	69	69	81	1181	20.0	16.7	60.8	228.6	**1.0***
Camel	Armadillo	69	331	43	33	0.4	14.3	189.9	80.0	0.3
Armadillo	Armadillo	331	331	441	5351	94.2	75.8	339.7	1198.2	**3.9***
461112	461115	805	822	808	876	2.8	64.7	753.2	473.2	1.0
Kitten	RedCircBox	274	1402	246	27	1.2	36.3	819.8	329.6	1.0
Bimba	Vase	150	1792	724	122	1.9	65.4	971.7	455.7	1.0
226633	461112	2452	805	1437	307	3.2	120.0	1723.7	905.5	**2.0***
Ramesses	Ramess.Trans.	1653	1653	1571	866	4.6	102.7	1558.8	946.1	**2.2***
Ramesses	Ramess.Rot.	1653	1653	1691	2275	6.6	122.5	1577.3	989.8	2.2
Neptune	Ramesses	4008	1653	1112	814	5.5	150.0	3535.5	1535.6	3.5
Neptune	Nept.Transl	4008	4008	3303	2924	10.9	247.9	5390.7	2726.2	5.4
ArmadilloTetra	ArmadilloTetraTransl	1602	1602	61325	259436	420.3	-	-	-	-
518092_tetra	461112_tetra	5938	8495	23181	255703	333.0	-	-	-	-

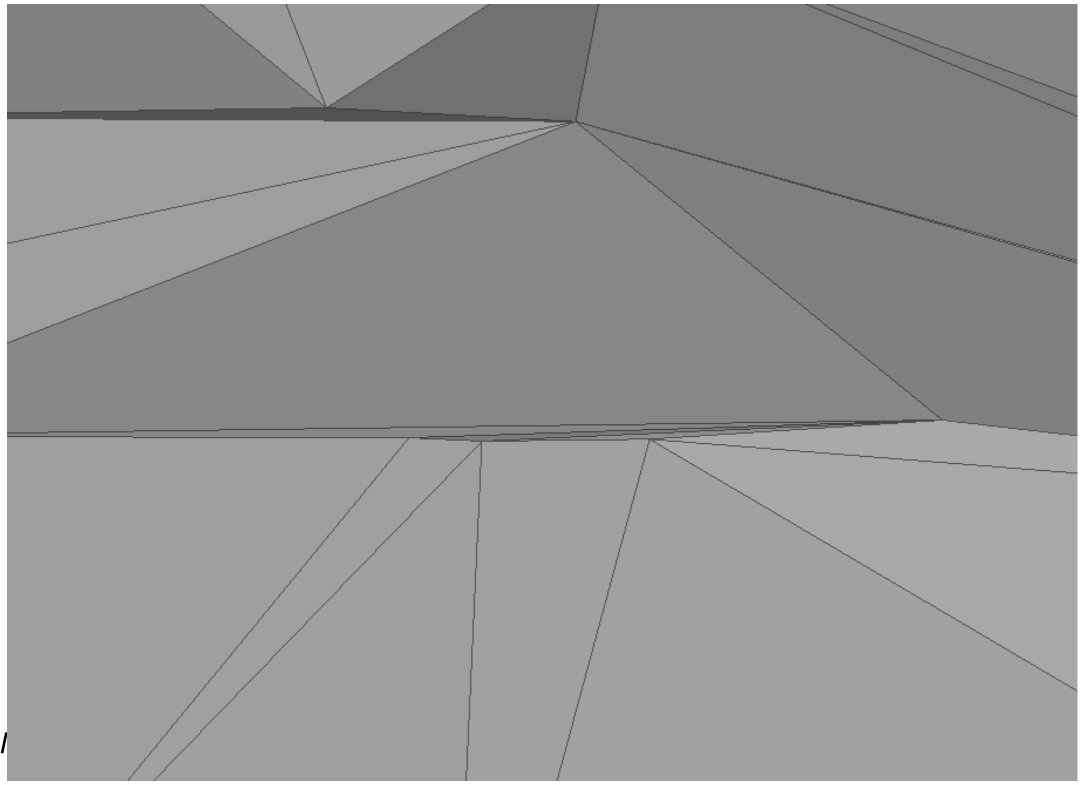
Example of result

- Intersection of two big meshes from AIM@SHAPE:
  - Ramesses: 1.7 million triangles
  - Neptune: 4 million triangles
  - 5.5 seconds



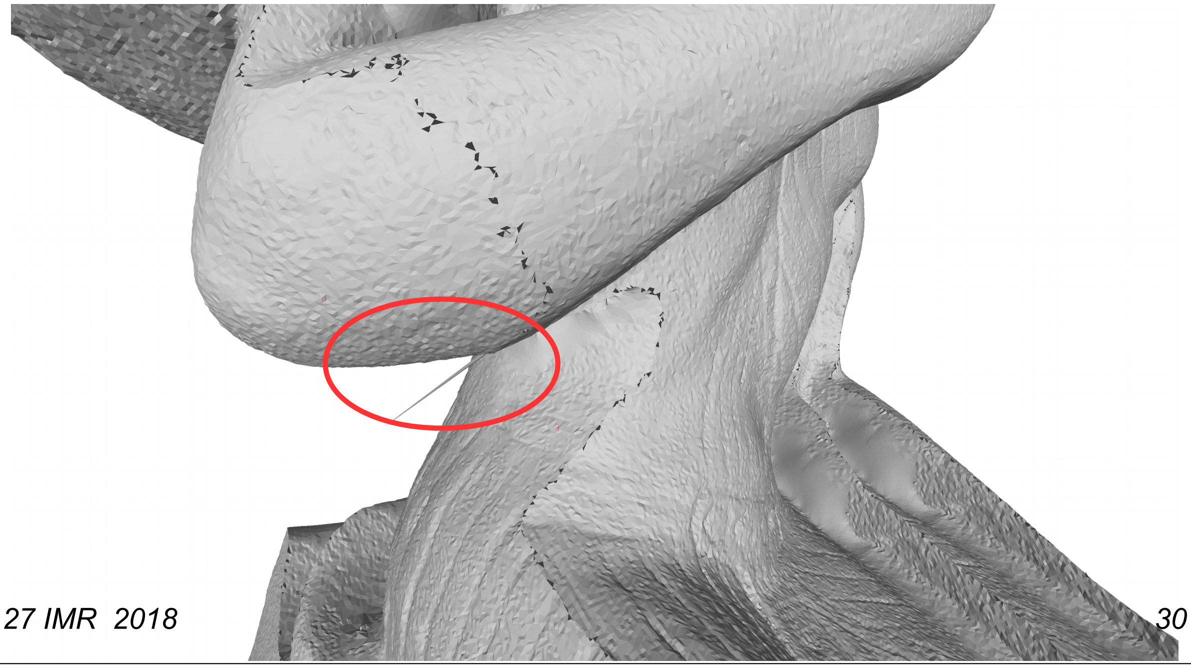
Example of result

- Hard to process triangles → roundoff errors



Example of result

- QuickCSG: Ramesses vs Ramesses translated.
- No error reported, yet several failures
  - To mitigate: use <u>numerical</u> perturbation
  - Does not work always (figure: max perturbation = 10^{-1})



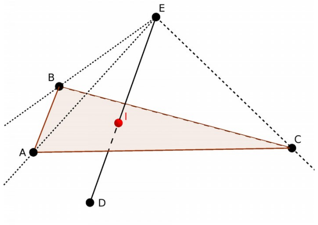
Correctness evaluation

- Compare output to LibiGL using Metro to measure Hausforff distances.
- We passed; QuickCSG sometimes failed.
- Visual inspection, ditto.
- Rotation invariance: rotate meshes (by angles with rational sin and cos), intersect, rotate back, compare to nonrotated intersection.
- No difference.
- Intersect meshes with rotated versions of themselves.
    - Big stress test, can crash CAD systems
    - Our result is identical to LibiGL.

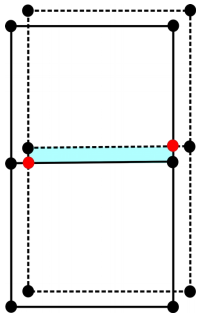
Limitations

- Non watertight or other bad input causes bad output. (Some other systems patch some things automatically.)
- Converting output from rationals to floats might require snapping to maintain topology. However this sort of ill conditioned data will probably cause worse problems for any system computing in floats. We, at least, compute a correct result expressed in rationals.
- SoS output might need regularizing to delete generated objects with zero volume etc. That’s doable.
- Some projective geometry-like configurations can not be converted to rational coordinates (because a cross ratio is irrational) but floats are also inexact, and these probably don’t occur in meshing.
- These techniques do not extend to curved objects.
- This prototype is not yet nearly optimized.

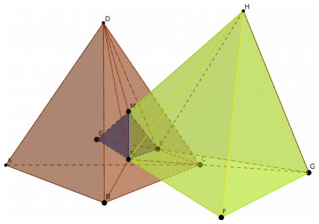
Conclusions

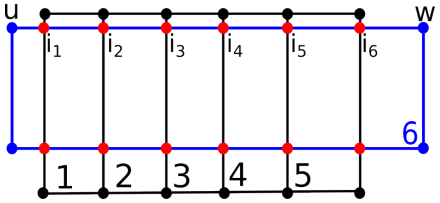


- 3D-EPUG-OVERLAY
  - Exact with rationals
  - Handles geometric deneracies with SoS
  - Minimal explicit topology → simple compact data structures
  - Parallel on multicore shared memory Xeons with OpenMP
  - Uniform grid to filter possible coincidences fast.



- Future work:
  - Improve performance (mainly of SoS calls)
  - Grid oblique triangles optimally
  - Test much larger datasets
  - Use similar ideas for other problems




u
w
i_1
i_2
i_3
i_4
i_5
i_6
6
1
2
3
4
5