(Thesis)

EXACT AND PARALLEL INTERSECTION OF 3D TRIANGULAR MESHES

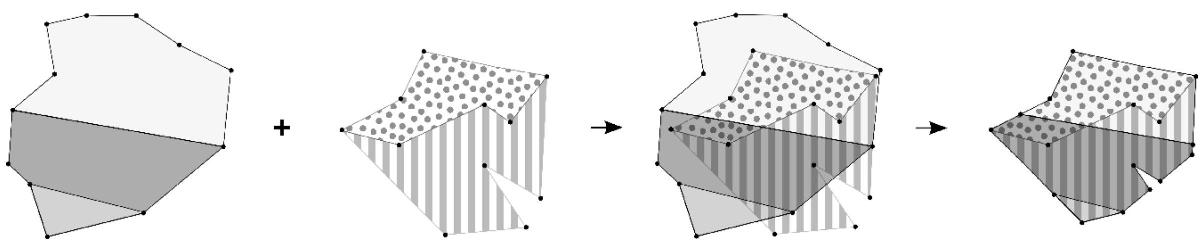
Salles Viana Gomes de Magalhães, PhD. Student
Prof. W Randolph Franklin, Supervisor

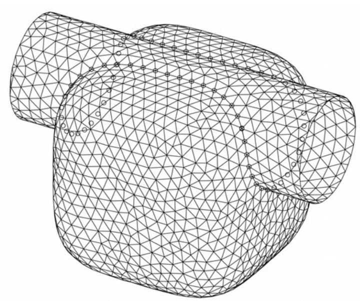
Rensselaer Polytechnic Institute, Troy NY USA
Federal University of Viçosa, MG, Brasil

ACM SIGSPATIAL, Redondo Beach, 2017-11-10

Map overlay

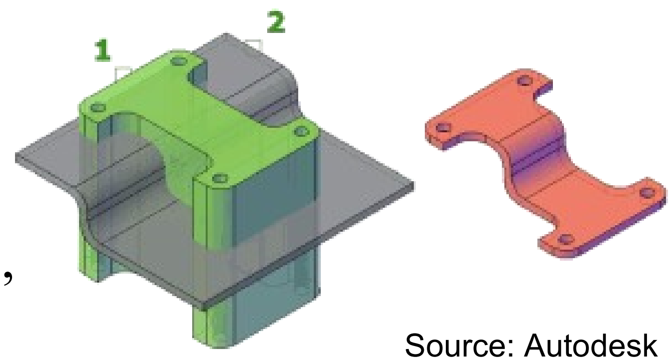
- Important in GIS/CAD/CAM
- Two vector maps are superimposed
- The intersection between polygons from the two maps is computed
- Several applications. Ex: counties and watersheds





source: wikipedia

- This problem extends to **3D** objects (triangulations)
- Example: intersection of CAD models, soil layers, etc



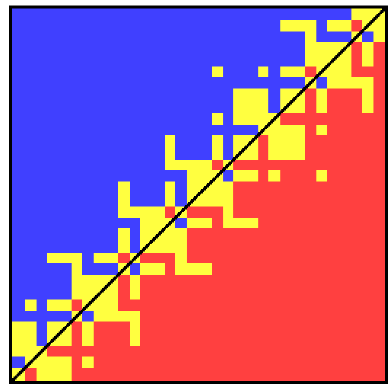
Source: Autodesk

Challenge

- Finite precision of floating point →roundoff errors

  1000000000.0+1.0-10000000000.0 = 0.0 *(wrong)*

  - Common techniques (snap rounding, epsilon tweaking, etc): no guarantee



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

  - More data & 3D→ bigger problem
  - Exactness and performance: very important – this function may be a small piece of a larger program

Our fast algorithms for large datasets

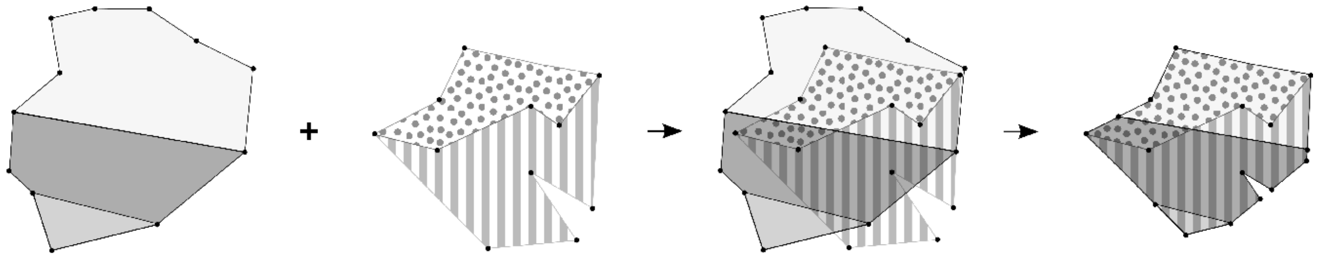
- ParCube – GPU parallel detection of cube-cube intersections
- 3D-EPUG-OVERLAY – 3D parallel map overlay
- NearptD – parallel nearest neighbor algorithm
- TiledVS – external memory viewshed computation.
- PinMesh – 3D point location
- UPLAN – path planning on road networks with polygonal constraints.
- Emflow – hydrography on massive external terrain
- EPUG-OVERLAY – 2D map overlay
- Grid-Gen – map simplification preserving topological relationships
- Parallel Multiple Observer Siting on Terrain
- RWFLOOD – hydrography on massive internal terrain
- UNION3 – volume of union of many cubes
- Connect – connected components of 1000^3 3D box of binary voxels
- TIN – incrementally triangulate 10000^2 terrain (update of (Franklin, 1973)).

We often combine 5 techniques

- Arbitrary precision rational numbers: no roundoff errors
- Simulation of Simplicity: handle special cases properly
- Minimize explicit topology: compact, parallelizable.
- Parallel programming: exploit current hardware
- Uniform grid: filter for probable intersections in parallel

EPUG-OVERLAY – 2D map overlay

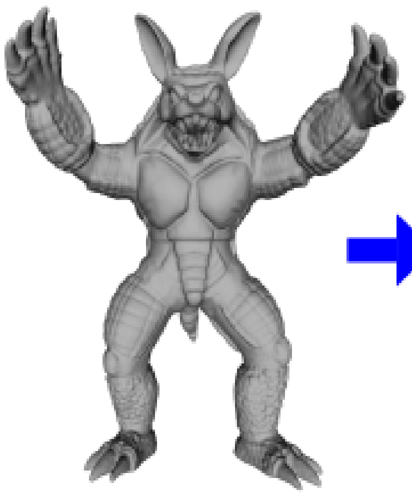
- **E**xact
- **P**arallel
- **U**niform **G**rid

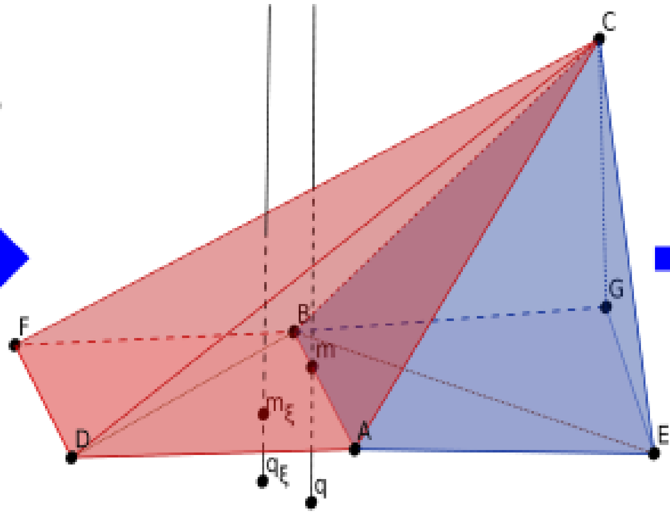


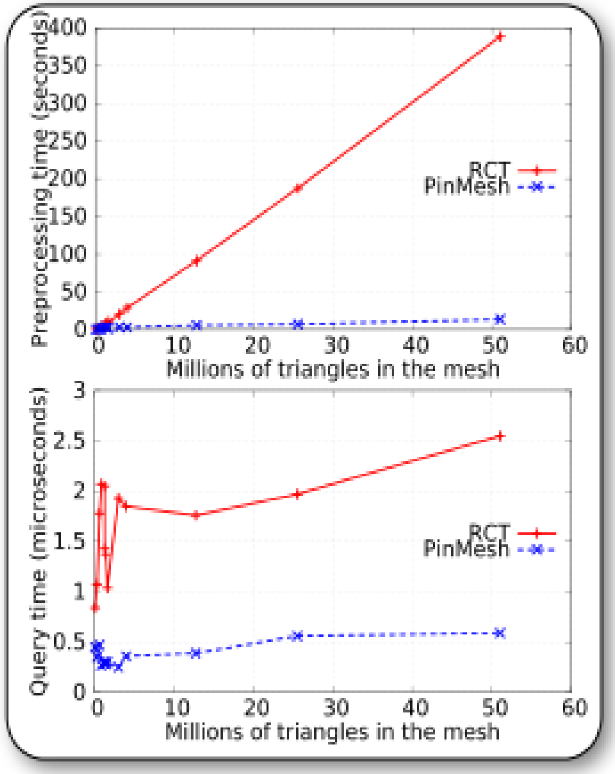
- Developed to evaluate our ideas
  - Exact
  - Efficient: 20x speedup if compared against GRASS GIS

PinMesh – 3D point location

- Preprocess 3D mesh to perform point queries
  - Exact and efficient (up 27 times faster than RCT, an inexact competing method) point location
  - Subproblem of the mesh overlay



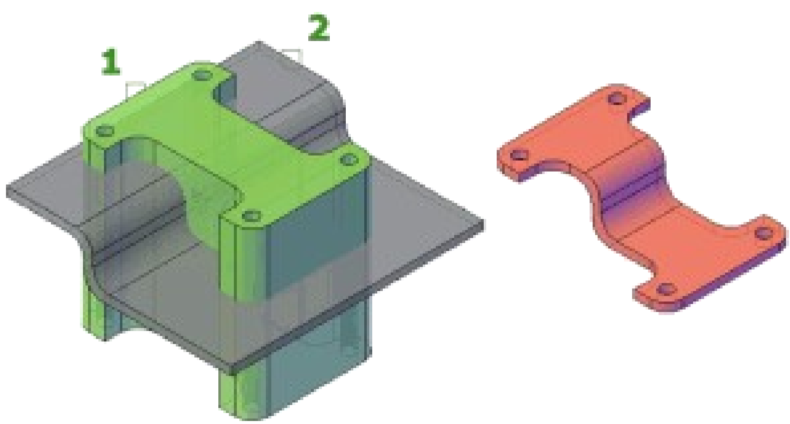




3D-EPUG-OVERLAY

Current work

- 3D mesh intersection
- Techniques + experience from PinMesh and EPUG-OVERLAY → 3D-EPUG-OVERLAY



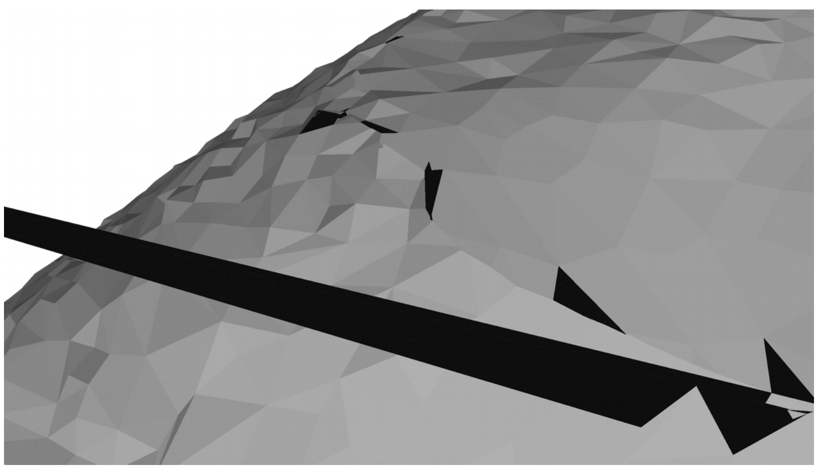
source: Autodesk

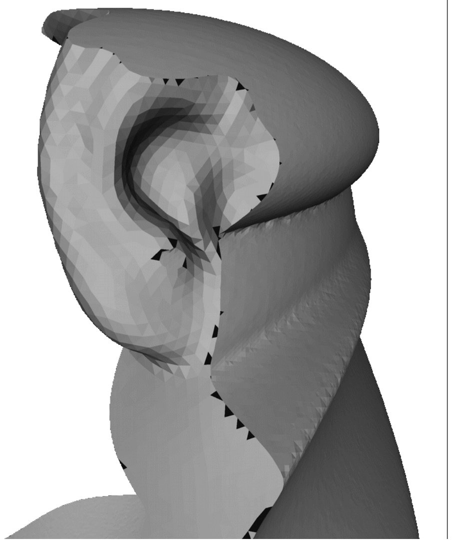
Related work

- Approximate algorithms:
  - Example: voxelization
- Nef Polyhedra/CGAL:
  - Exact, sequential, slow
  - For Nef Polyhedra
    - Polyhedron: sequence of complement and intersection of half-spaces
    - Challenge: convert data

Related work - QuickCSG

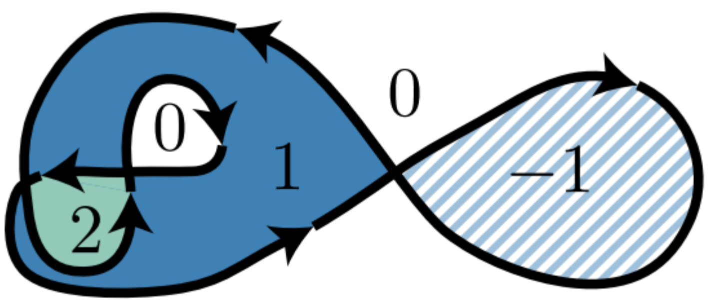
- QuickCSG:
  - Recent
  - Designed to be very fast: no special cases, floating-point, parallel
  - User can **try** to avoid special cases: numeric perturbation
  - Error-prone





Related work - LibiGL

- Zhou's algorithm (LibiGL):
  - Very recent
  - Parallel and relatively fast
  - Uses CGAL (example: bounding-box for triangle-triangle intersection)
  - Key idea: use of winding number in mesh representation
    - Merge meshes + resolve self-intersections

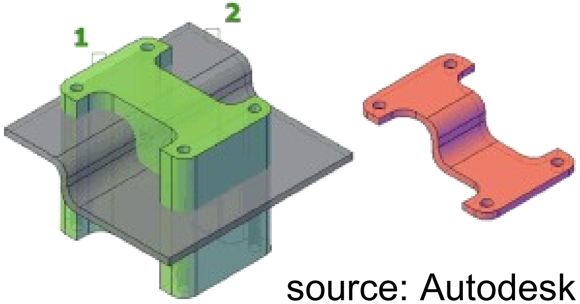


Winding numbers (source: Zhou et al. [77])

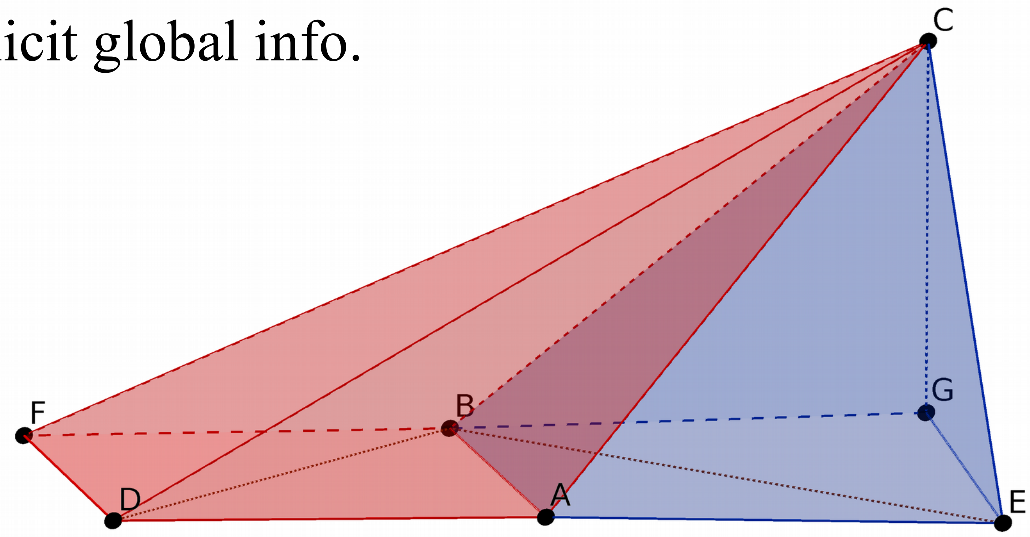
Our data representation

- Intersection: pair of meshes
- Each mesh: set of polyhedra (usually one polyhedron) that partition space.

- Mesh representation
  - Set of triangles, plus
  - Information about positive and negative sides
  - No explicit global info.



source: Autodesk

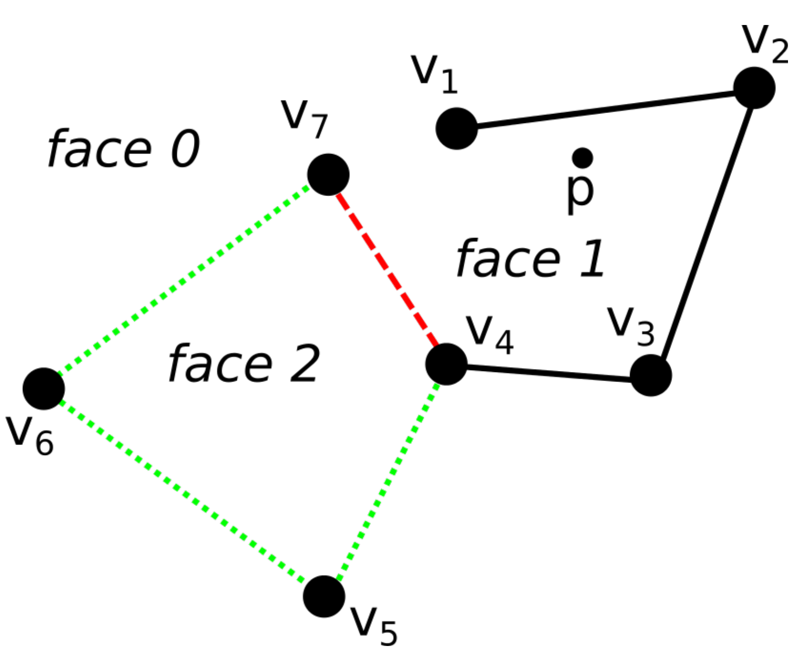


ABC:
- Positive: blue
- Negative: red

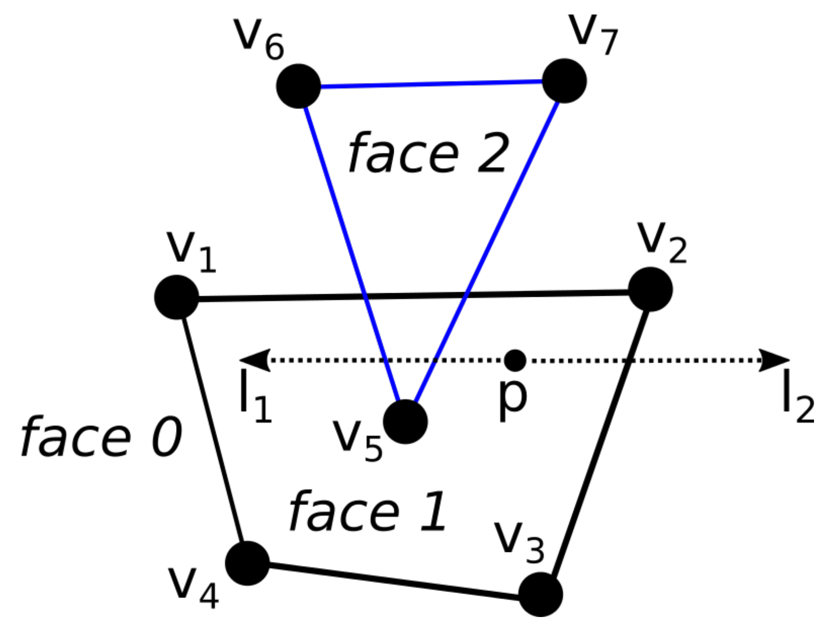
ABD:
- Positive: red
- Negative: outside

Data representation

- Mesh restriction: should be “valid”
  - watertight
  - consistent



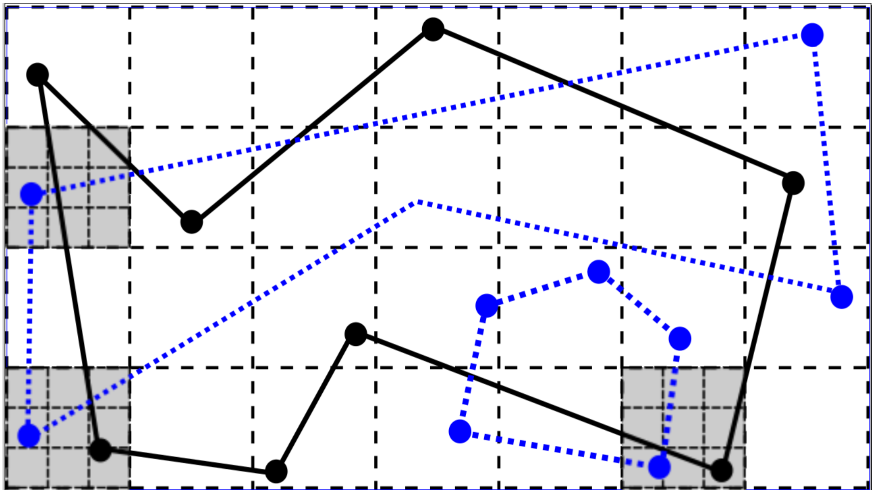
Non watertight mesh (2D)



Self-intersecting mesh (2D)

Indexing the data

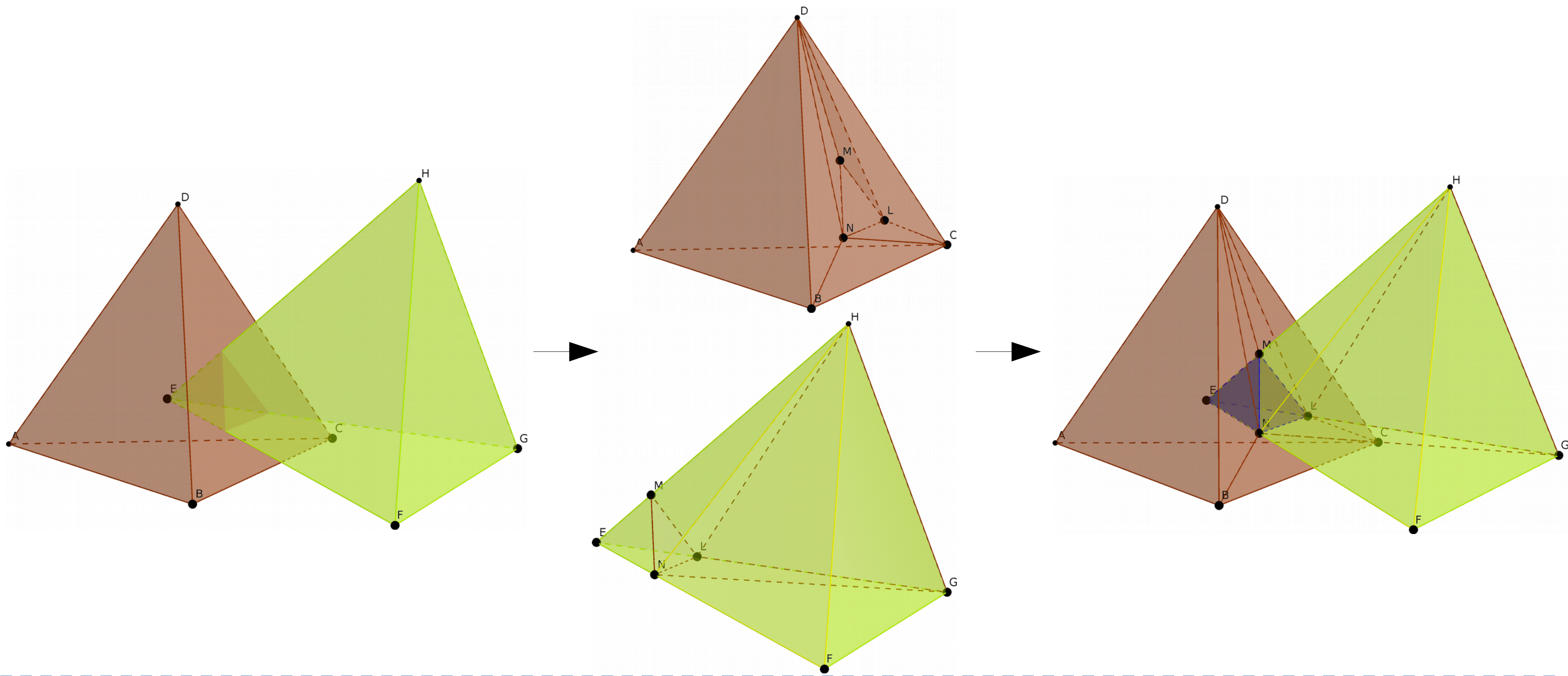
- We employ a 2-level 3D uniform grid.
  - Employed for detecting intersections and point location.
  - Coding shortcut: Insert a 3D triangle into the cells that *its bounding box* intersects. That is many more cells than necessary (asymptotically superlinear).
  - That shortcut motivates the 2 levels.



Example: detecting black-blue intersections (2D)

Algorithm summary

- Detect intersections between the two meshes
- Retesselate intersecting triangles
- Classify the triangles, both non-intersecting and retesselated.

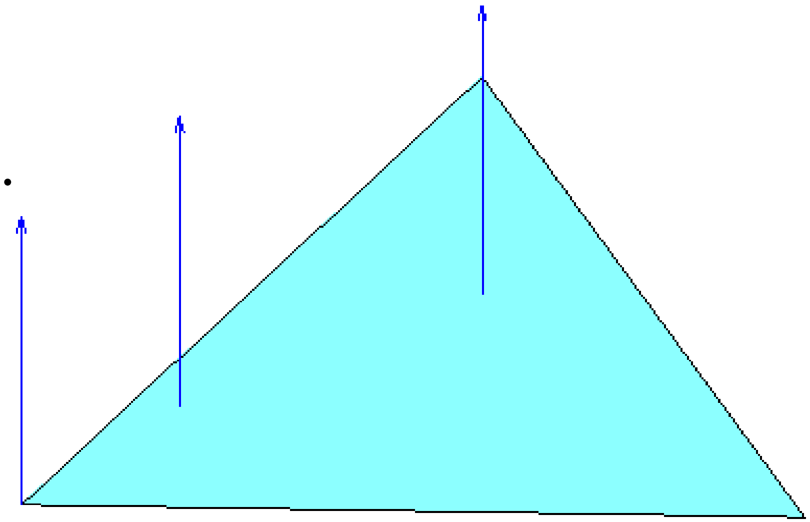


Rational numbers

- Motivation: no roundoff errors.
- Each number is stored as a ratio of two integers
- E.g., 1/3+2/5 = 11/15
- C++ operators are overloaded to do this
- Each operation doubles the number of digits
- Numerator and denominator are arrays of groups of digits
- Doubling is acceptable if depth of computation tree is small
- Packages like gmp++ mostly work
- Big problem: frequent allocations on global heap
- That's slow for many objects and for multithreading.
- Solution: code to minimize allocations and use a better allocator.
- Execution time penalty: small integer factor
- Combine with interval arithmetic ([lo,hi]) for speed
- [.30,.35] + [.48,.52] = [.78..87]

Simulation of Simplicity (SoS)

- Reduces the number of special cases.
- Point vs line? *Above*, *on*, or *below*.
- Combine *on* case into *above*?
- Solution must handle higher level functions correctly
- e.g., Pnpoly (Franklin, 1970) : test point inclusion in polygon by running ray up from point and counting intersections with edges.
- How many intersections when vertex is on ray?
- Much worse: ray vs polyhedron
- Sos: move ray slightly to right.
- Then no ray—vertex intersections.



Special cases

- $p(x,y,z) \rightarrow p_\varepsilon(x+i\varepsilon, y+i\varepsilon^2, z+i\varepsilon^3) \rightarrow$ coincidences eliminated
  - i=0 or 1 (which input dataset is this?)
  - A vertex of one mesh is never on the plane of a triangle of the other mesh (→ intersection of triangles is never a point)
  - Edges from different meshes do not intersect → edges will only intersect interior of triangles
  - Triangles from different meshes are never coplanar
  - Etc

- Example of consequence: intersection of two 3D triangles is always an edge

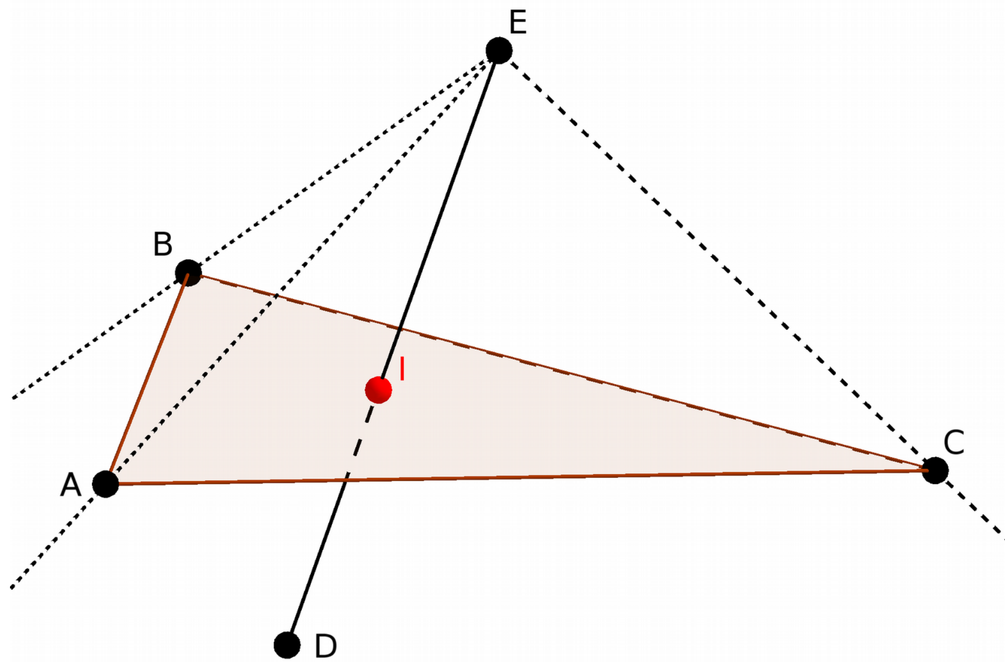
Implementing SoS

- Don't actually implement infinitesimal math.
- Instead: rewrite geometric predicates to have that effect.
- $(a+\varepsilon^{i} < b+\varepsilon^{j}) \rightarrow ((a<b \mid (a==b) \,\&\, (i>j))$
- Leads to incrutable source code.
- Computation can be initially done with the rational coordinates. If coincidence is detected → consider the infinitesimals → good performance
- Challenge: too many predicates!
- Solution → use a small set of predicates

Orientation predicates

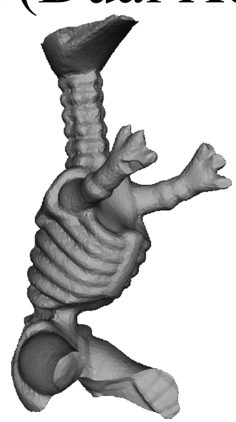
- The algorithm was completely implemented using orientation predicates (except for the indexing) → SoS only in the orientation predicate.

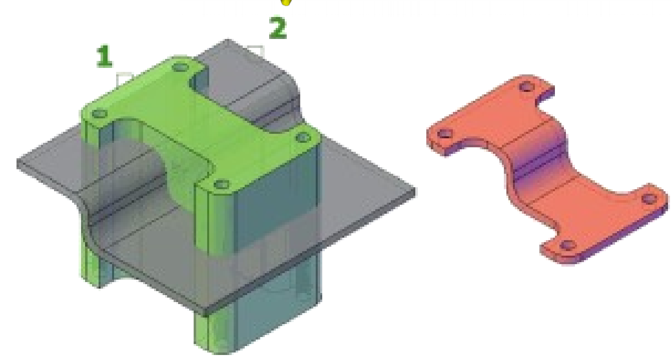
- Example: detect intersection of two triangles
  - → detect intersections between edges and a triangle
  - → 5 orientations for each edge-triangle test (Segura and Feito, 2001)



Experiments

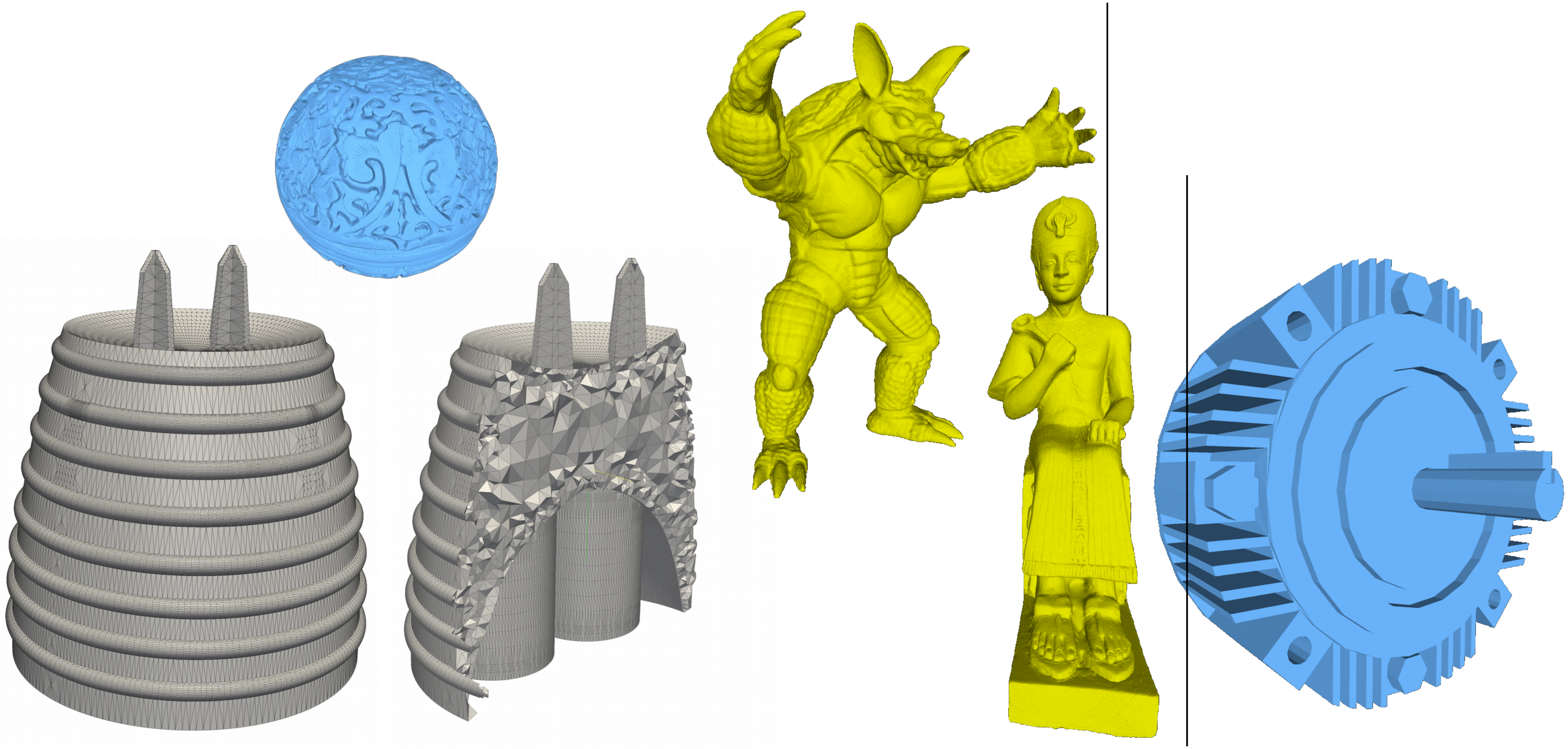
- Algorithm designed to be parallel:
  - Little data dependency, simple representation
- Implemented using OpenMP
- Compiled with g++ -O3, using Tcmalloc
- All times in seconds
- Machine:
  - 16-Core workstation (Dual Xeon E5-2687)
  - 128 GB of RAM
  - Ubuntu Linux

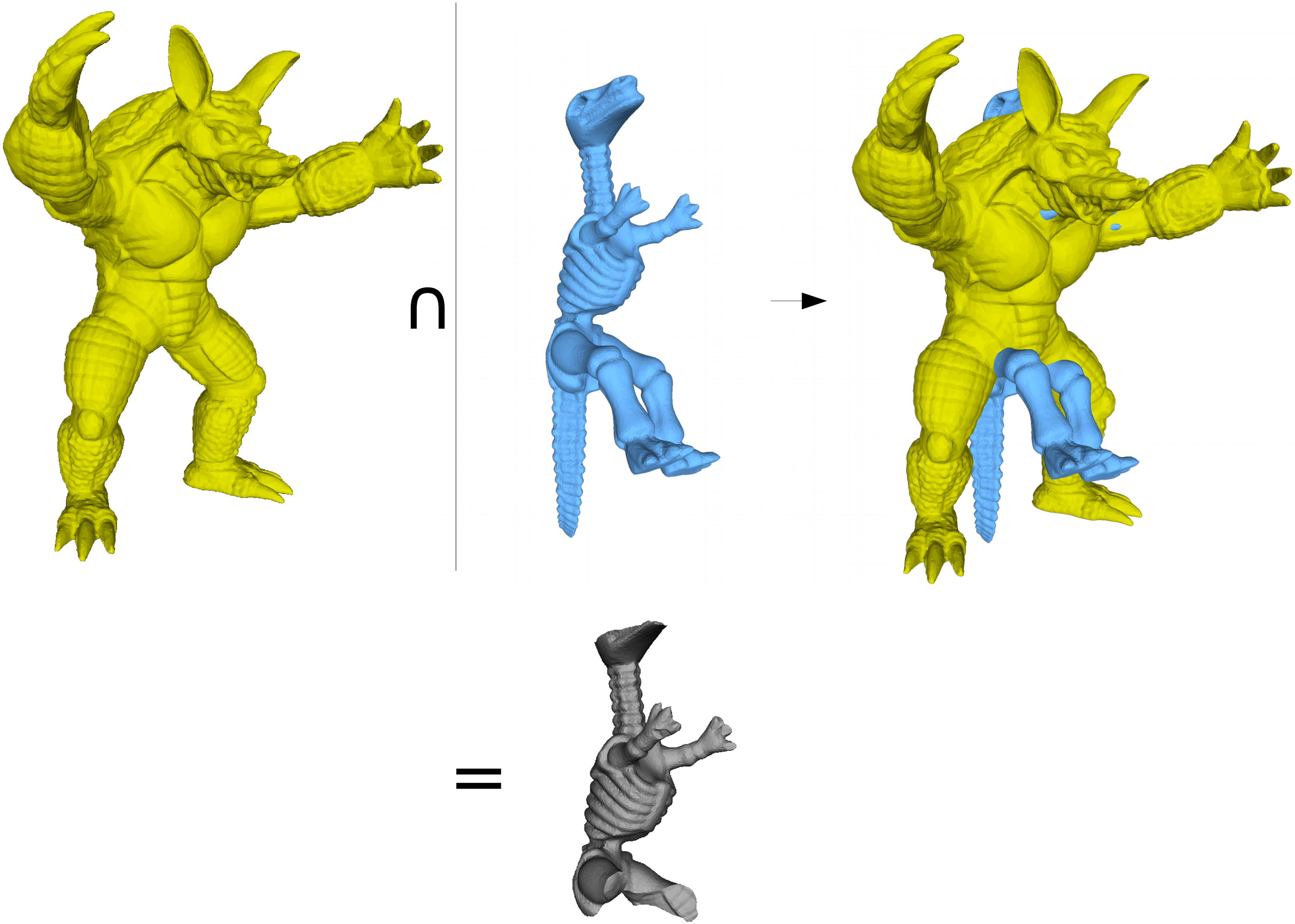




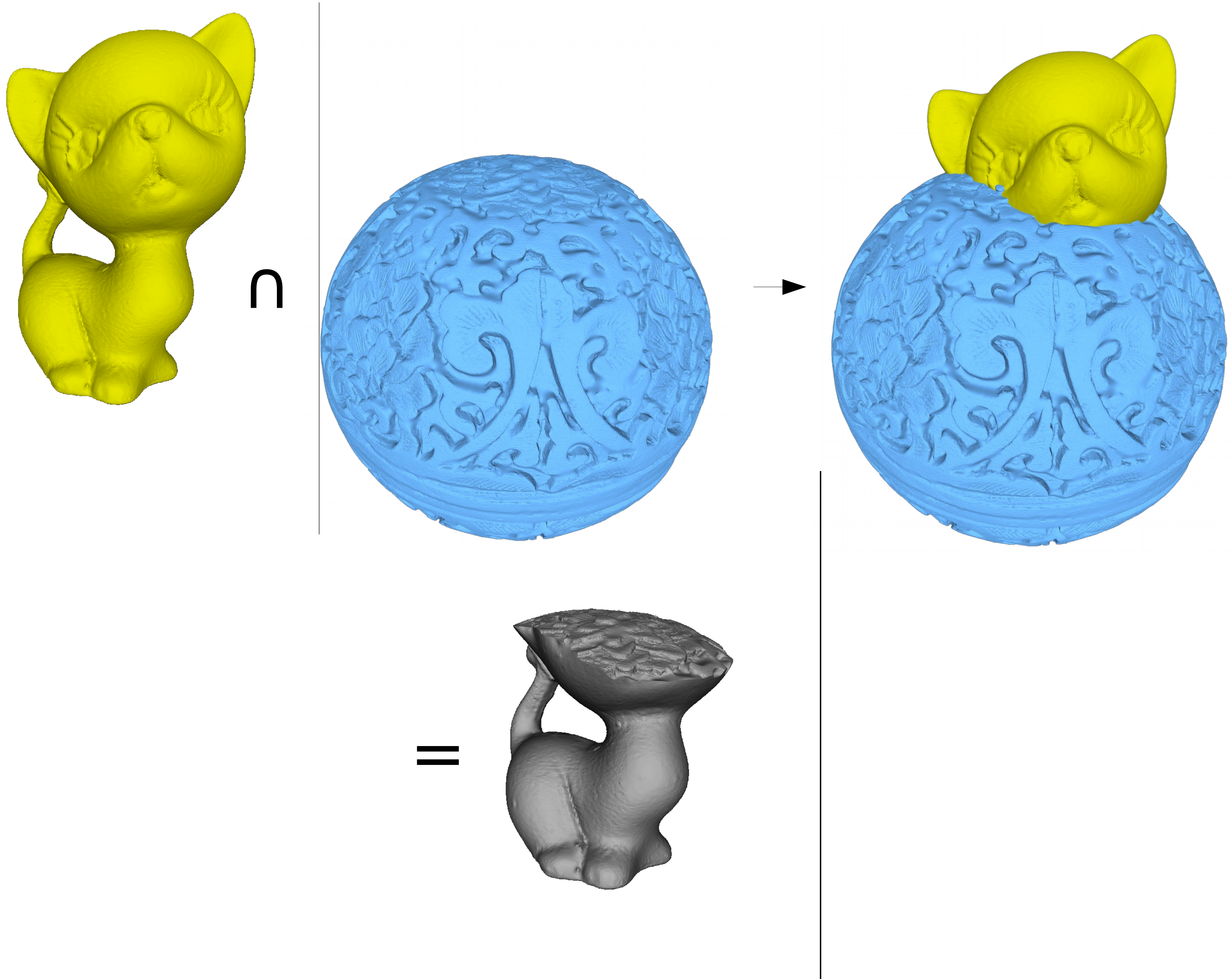
Datasets

- Datasets from 4 sources
- Meshes with up to 4 million triangles
- Tetra meshes with up to 8 million triangles/4 million tetrahedra

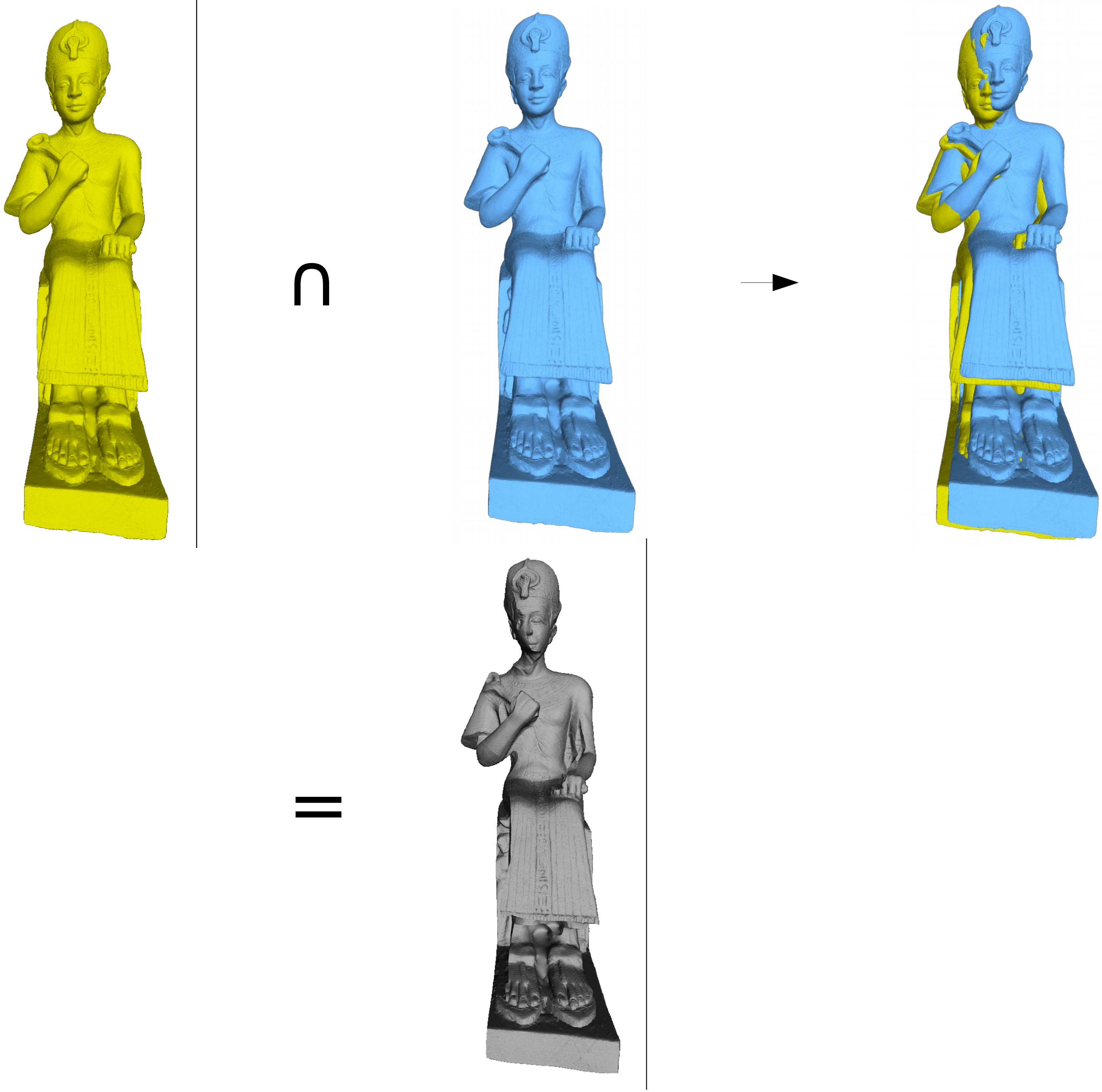




RPI - Rensselaer Polytechnic Institute



RPI - Rensselaer Polytechnic Institute



Experiments

- First set of experiments: two key techniques for performance:
  - Arithmetic filtering: accelerate rationals
  - Uniform grid: easily parallelizable
  - This also shows that the uniform grid can efficiently process data that much worse than uniform random, which would have coincidences only with probability 0.
  - We also experimented with various grid concrete realizations.
  - Very bad: linked list or STL vector for each cell.
  - Ragged array is much better.
    - String together in one array all the cells' contents.
    - A dope vector points to start of each cell's contents.

Arithmetic filtering

- Makes using rationals faster.

- Arithmetic filtering → rationals: not always necessary
  - Basic idea: associate floating-point approximations to each number
  - Evaluate predicates (determinants) with the approximation
  - If signal can be trusted → use it
  - Otherwise, recompute exactly

Uniform grid much faster than CGAL

CGAL

		# faces ($\times 10^3$)		# int.[a]	Int.tests[b]	Time (s)	
Mesh 0	Mesh 1	Mesh 0	Mesh 1	($\times 10^3$)	($\times 10^3$)	Pre.proc.[c]	Inter.[d]
Camel	Armadillo	69	331	3	14	0.32	0.01
Armadillo	Armadillo	331	331	4,611	5,043	1.27	259.23
Kitten	RedC.Box[e]	274	1,402	3	13	2.33	0.01
226633	461112	2,452	805	23	128	7.18	0.08
Ramesses	Ram.Tran.[f]	1,653	1,653	36	237	12.38	0.17
Neptune	Nept.Tran.[g]	4,008	4,008	78	647	36.24	0.47

Uniform grid

		# faces ($\times 10^3$)		# int.[a]	Int.tests[b]	Time (s)	
Mesh 0	Mesh 1	Mesh 0	Mesh 1	($\times 10^3$)	($\times 10^3$)	Pre.proc.[c]	Inter.[d]
Camel	Armadillo	69	331	3	33	0.06	0.02
Armadillo	Armadillo	331	331	50	5,351	0.25	63.80
Kitten	RedC.Box[e]	274	1,402	3	27	0.08	0.02
226633	461112	2,452	805	23	307	0.16	0.05
Ramesses	Ram.Tran.[f]	1,653	1,653	36	866	0.16	0.10
Neptune	Nept.Tran.[g]	4,008	4,008	78	5,087	0.27	0.35

Choosing the grid resolution

- Parameter: grid resolution
- Number of expected pairs of triangles: *np*

$$np = \frac{n_0 \, x \, n_1}{G_1^3 \, x \, G_2^3}$$

$$G_1 \, x \, G_2 = \sqrt[6]{\frac{n_0 \, x \, n_1}{np}}$$

- Experiments: *np* to a small constant:
  - 0.00001 (regular meshes) or 0.1 (internal structure)

→ Good performance (broad optimum)

Choosing the grid resolution

Mesh 0: Ramesses (2M triangles), Mesh 1: Ramesses.rot.[h] (2M triangles)								
Grid Size[a]	Pairs of triangles ($\times 10^3$) Grid[b]	Unique[c]	Inter.[d]	Memory (GB)	Time (s) Grid[e]	Inter.[f]	Class.[g]	Total
16,8	94,302	90,616	60	4.59	0.11	6.20	0.72	7.72
16,16	22,585	19,852	60	2.31	0.11	1.52	0.59	2.88
32,8	22,585	19,852	60	2.27	0.12	1.41	0.58	2.79
16,32	8,287	5,748	60	1.92	0.13	0.76	0.53	2.13
32,16	8,287	5,748	60	1.84	0.13	0.77	0.61	2.18
64,8	8,287	5,748	60	1.79	0.13	0.74	0.57	2.10
16,64	5,486	2,275	60	3.08	0.27	0.67	0.57	2.21
32,32	5,486	2,275	60	2.44	0.19	0.59	0.57	1.98
64,16	**5,486**	**2,275**	**60**	**2.08**	**0.18**	**0.61**	**0.60**	**2.11**
32,64	7,365	1,240	60	7.00	0.74	0.90	0.55	2.90
64,32	7,365	1,240	60	4.08	0.42	0.70	0.60	2.42
64,64	18,899	865	60	19.26	2.31	1.80	0.52	5.29

Comparing against other methods

		Time (s)				
				CGAL		
Mesh 0	Mesh 1	3D-Epug	LibiGL	Convert[a]	Intersect[b]	QuickCSG
Casting10kf	Clutch2kf	0.2	1.3	4.2	1.1	**0.1***
Armadillo52kf	Dinausor40kf	0.1	3.0	38.0	21.5	0.1
Horse40kf	Cow76kf	0.1	3.2	51.1	24.2	0.1
Camel69kf	Armadillo52kf	0.1	3.2	54.3	25.7	0.1
Camel	Camel	13.9	18.0	62.7	230.6	**0.9***
Camel	Armadillo	0.2	11.7	189.9	80.0	0.3
Armadillo	Armadillo	67.0	88.1	339.7	1,198.2	**4.1***
461112	461115	0.8	58.9	753.2	473.2	1.1
Kitten	RedCircBox	0.3	28.6	819.8	329.6	1.1
Bimba	Vase	0.6	58.0	971.7	455.7	1.1
226633	461112	0.9	96.0	1,723.7	905.5	**2.2***
Ramesses	Ram.Transl.[c]	1.3	93.0	1,558.8	946.1	**2.4***
Ramesses	Ram.Rot.[d]	2.1	122.0	1,577.3	989.8	2.4
Neptune	Ramesses	1.2	118.1	3,535.5	1,535.6	4.1
Neptune	Nept.Tran.[e]	2.7	220.2	5,390.7	2,726.2	6.1
68380Tet.[f]	914686Tet.[g]	51.3	-	-	-	-
Armad.Tet.[h]	Arm.Tet.Tran.[i]	263.3			-	
518092Tetra	461112Tetra	136.6			-	

Exact, parallel

Exact, sequential

Inexact, parallel

Comparing against other methods

Mesh 0	Mesh 1	Time (s) 3D-Epug	LibiGL	CGAL Convert[a]	CGAL Intersect[b]	QuickCSG
Casting10kf	Clutch2kf	0.2	1.3	4.2	1.1	**0.1***
Armadillo52kf	Dinausor40kf	0.1	3.0	38.0	21.5	0.1
Horse40kf	Cow76kf	0.1	3.2	51.1	24.2	0.1
Camel69kf	Armadillo52kf	0.1	3.2	54.3	25.7	0.1
Camel	Camel	13.9	18.0	62.7	230.6	**0.9***
Camel	Armadillo	0.2	11.7	189.9	80.0	0.3
Armadillo	Armadillo	67.0	88.1	339.7	1,198.2	**4.1***
461112	461115	0.8	58.9	753.2	473.2	1.1
Kitten	RedCircBox	0.3	28.6	819.8	329.6	1.1
Bimba	Vase	0.6	58.0	971.7	455.7	1.1
226633	461112	0.9	96.0	1,723.7	905.5	**2.2***
Ramesses	Ram.Transl.[c]	1.3	93.0	1,558.8	946.1	**2.4***
Ramesses	Ram.Rot.[d]	2.1	122.0	1,577.3	989.8	2.4
Neptune	Ramesses	1.2	118.1	3,535.5	1,535.6	4.1
Neptune	Nept.Tran.[e]	2.7	220.2	5,390.7	2,726.2	6.1
68380Tet.[f]	914686Tet.[g]	51.3	-			-
Armad.Tet.[h]	Arm.Tet.Tran.[i]	263.3	-			-
518092Tetra	461112Tetra	136.6				-

Meshes with many polyhedra: natural for our method

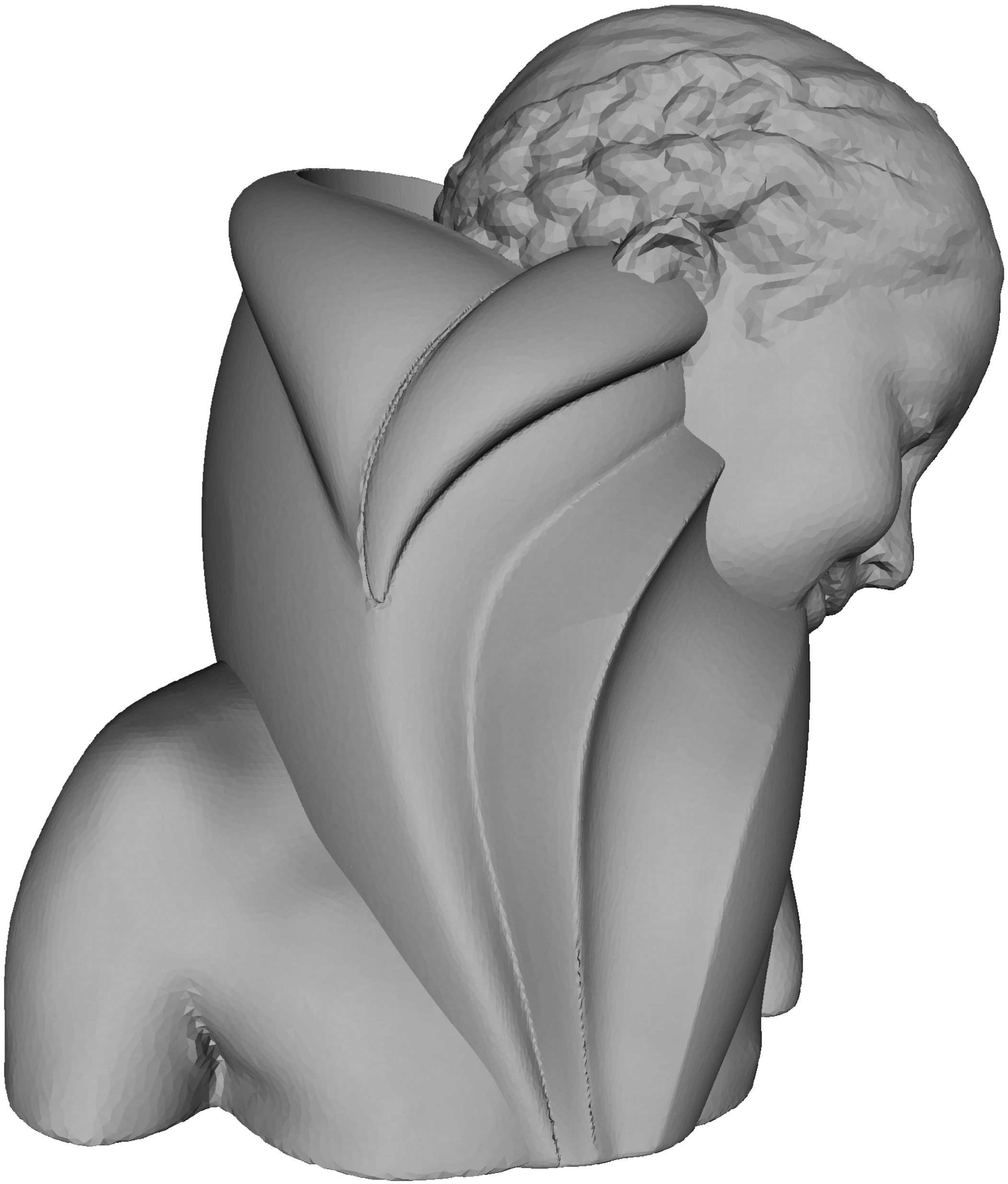
Correctness evaluation

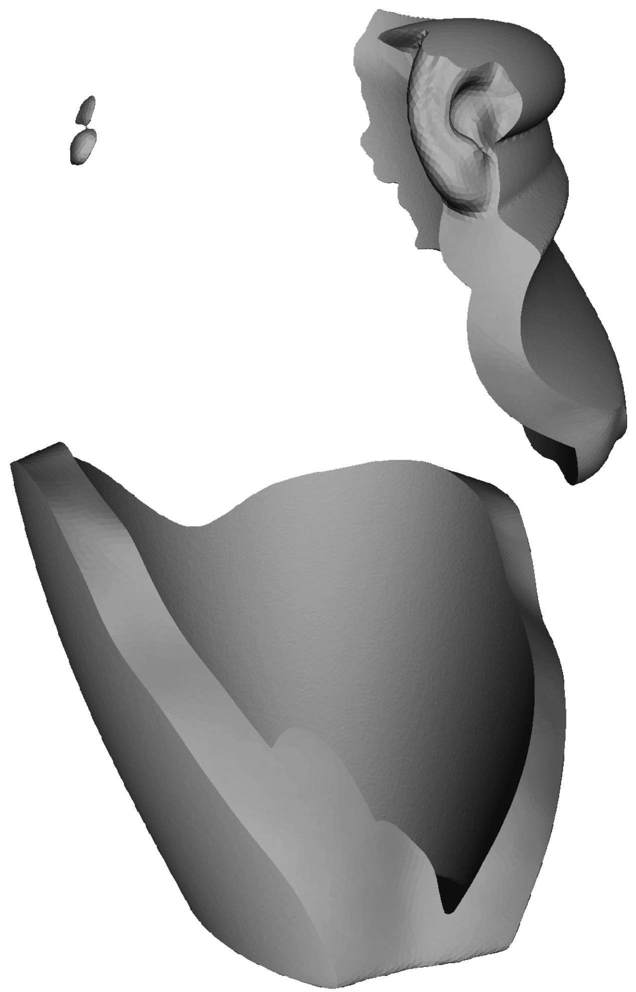
- 3D-EPUG-OVERLAY
  - Solid foundation: SoS + rationals
  - We showed: special cases
  - Correct algorithm → Bug-free implementation ?
- Evaluation:
  - Metro: Hausdorff distance
    - $max(E(S_1,S_2), E(S_2,S_1))$
    - Evidence of correctness: I/O, FP errors in Metro
  - Compared against LibiGL
  - Visual inspection
  - Rotation experiments: mesh ∩ rotated mesh, rotated mesh ∩ rotated mesh

Hausdorff distances vs LibiGL

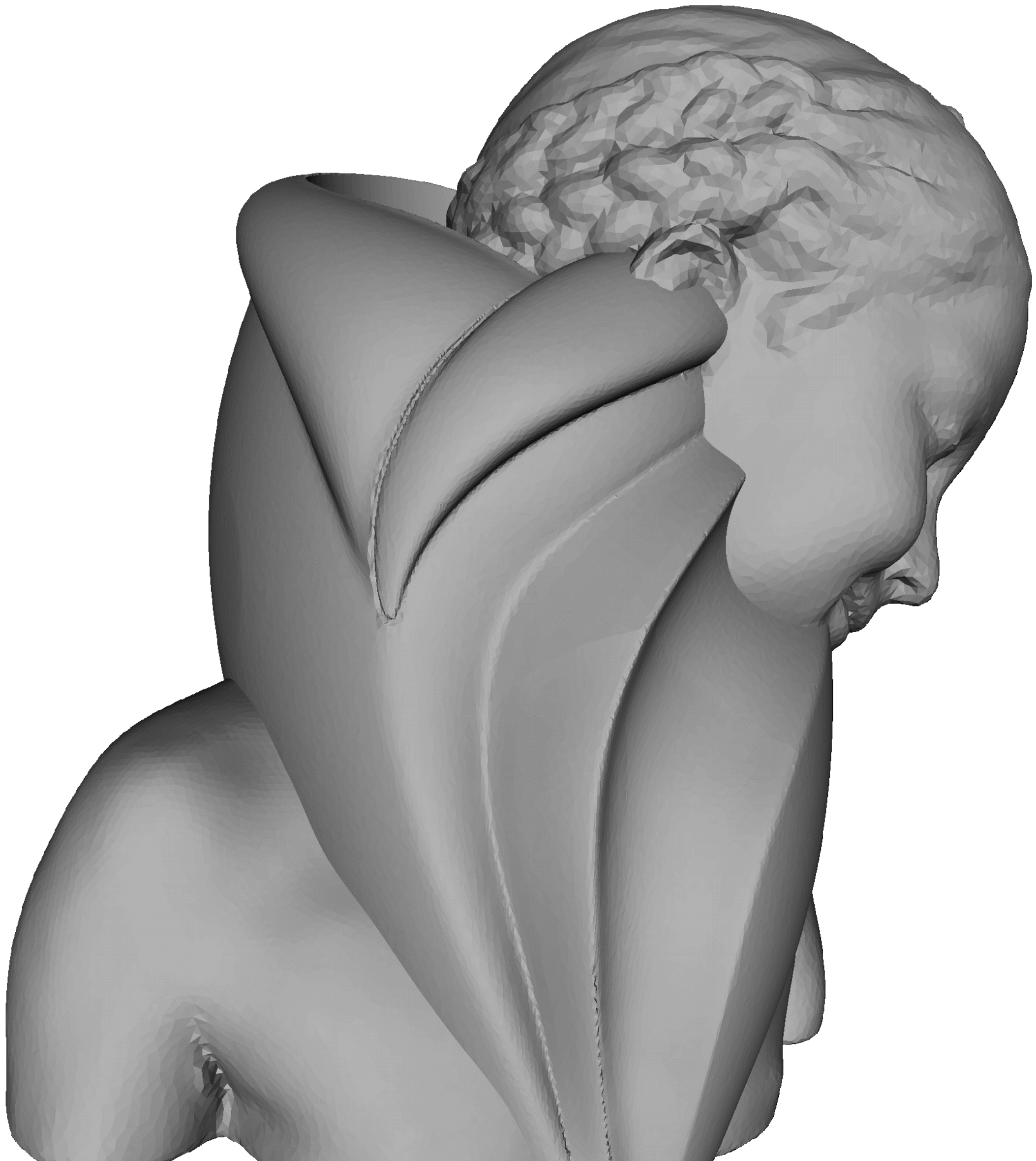
		Difference (%)		
Mesh 0	Mesh 1	3D-Epug	CGAL	QuickCSG
Casting10kf	Clutch2kf	0.0000	0.0000	-
Armadillo52kf	Dinausor40kf	0.0000	0.0001	0.1181
Horse40kf	Cow76kf	0.0000	0.0001	0.0490
Camel69kf	Armadillo52kf	0.0000	0.0001	0.1254
Camel	Camel	0.0000	0.0000	-
Camel	Armadillo	0.0000	0.0001	0.1121
Armadillo	Armadillo	0.0000	0.0000	-
461112	461115	0.0000	0.0002	0.0119
Kitten	RedCircBox	0.0000	0.0005	0.1020
Bimba	Vase	0.0000	0.0001	0.0847
226633	461112	0.0000	0.0003	-
Ramesses	Ram.Transl. [a]	0.0000	0.0007	-
Ramesses	Ram.Rot. [b]	0.0000	0.0007	0.0465
Neptune	Ramesses	0.0000	0.0007	0.0386
Neptune	Nept.Transl. [c]	0.0000	0.0004	0.0149

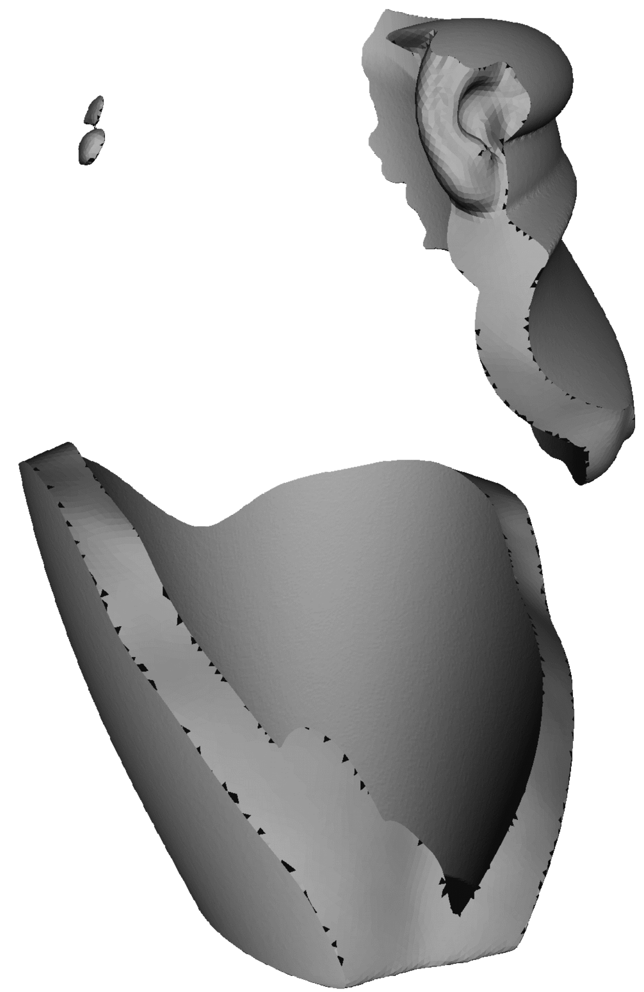
Visual inspection – 3D-EPUG-OVERLAY





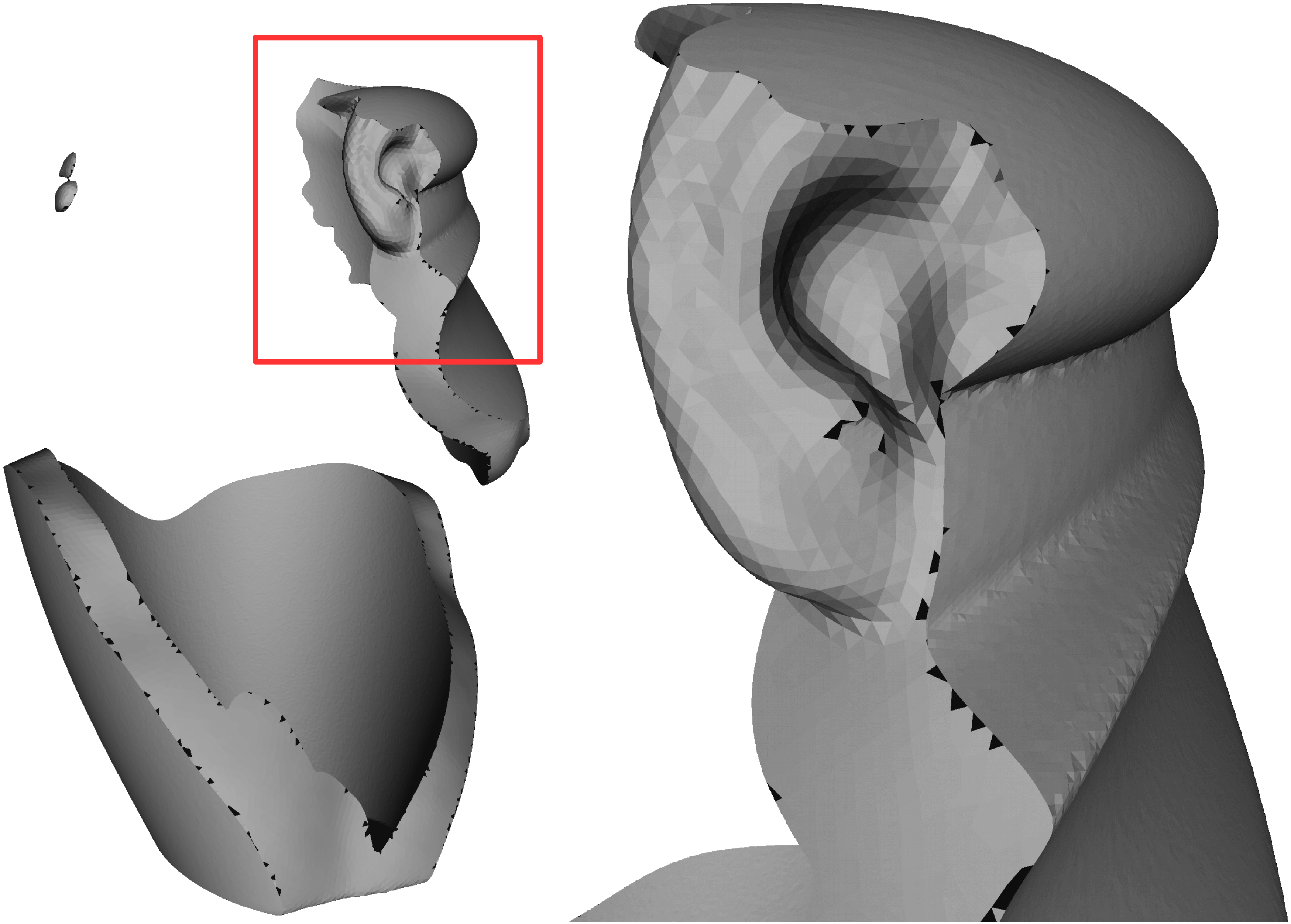
Visual inspection – QuickCSG





RPI - Rensselaer Polytechnic Institute

Visual inspection – QuickCSG



Conclusions

- Careful implementation → 3 exact and efficient algorithms
- Two preliminary algorithms
- EPUG-OVERLAY:
  - Faster than GRASS GIS inexact method
  - Exact
- PinMesh:
  - Up to 27x faster than RCT
  - Exact

RPI - Rensselaer Polytechnic Institute

Conclusions

- Main result: 3D-EPUG-OVERLAY
- Exact: rationals and SoS
  - Results matched reference solution
- Fast: uniform grid, parallel, simple representation, intervalU
  - Up to 101x times faster than LibiGL (also parallel)
  - Up to 1.284x/4,241x times faster than CGAL
  - Faster than QuickCSG (parallel/inexact/no special cases) in most of test cases
  - Parallel → better usage of computers
- Fast and exact → good for applications like CAD/GIS (interactivity & exactness)

Future work

- Algorithms developed in sequence → use 3D-EPUG-OVERLAY improvements in other methods
- Implement other CSG operations (easy)
- Create a CGAL kernel with SoS → use CGAL algorithms (example: Delaunay)
- Improve performance of SoS predicates
- Develop strategies for choosing the grid resolution (ex: recreate grid until good resolution)
- Strategies for removing the perturbation from the output