Exact fast parallel intersection of large 3-D triangular meshes *(extended abstract)*

Salles V. G. Magalhães
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
salles@ufv.br

W. Randolph Franklin
Rensselaer Polytechnic Institute
Troy, NY, USA
mail@wrfranklin.org

Marcus V.A. Andrade
Universidade Federal de Viçosa
Minas Gerais, Brasil
marcus.ufv@gmail.com

ABSTRACT

We present 3D-EPUG-Overlay, a fast, exact, parallel, memory-efficient, algorithm for computing the intersection between two large 3-D triangular meshes with geometric degeneracies. Applications include CAD/CAM, CFD, GIS, and additive manufacturing. 3D-EPUG-Overlay combines 5 separate techniques: multiple precision rational numbers to eliminate roundoff errors during the computations; Simulation of Simplicity to properly handle geometric degeneracies; simple data representations and only local topological information to simplify the correct processing of the data and make the algorithm more parallelizable; a uniform grid to efficiently index the data, and accelerate testing pairs of triangles for intersection or locating points in the mesh; and parallel programming to exploit current hardware. 3D-EPUG-Overlay is up to 101 times faster than LibiGL, and comparable to QuickCSG, a parallel inexact algorithm. 3D-EPUG-Overlay is also more memory efficient. In all test cases 3D-EPUG-Overlay's result matched the reference solution. It is freely available for nonprofit research and education at https://github.com/sallesviana/MeshIntersection . The full version of this paper is being presented at the 2018 International Meshing Roundtable; it is currently online at https://project.inria.fr/imr27/files/2018/09/1035.pdf.

ACM Reference Format:
Salles V. G. Magalhães, W. Randolph Franklin, and Marcus V.A. Andrade. 2018. Exact fast parallel intersection of large 3-D triangular meshes *(extended abstract)*. In *FWCG abstracts.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The classic problem of intersecting two 3-D meshes has been a foundational component of CAD systems for some decades. However, as data sizes grow, and parallel execution becomes desirable, the classic algorithms and implementions now exhibit some problems.

1. *Roundoff errors.* Floating point numbers violate most of the axioms of an algebraic field, e.g., $(a + b) + c \neq a + (b + c)$. These arithmetic errors cause topological errors, such as causing a point to be seen to fall on the wrong side of a line. Those inconsistencies propagate, causing, e.g., nonwatertight models. Heuristics exist to ameliorate the problem, and they work, but only up to a point. Larger datasets mean a larger probability of the heuristics failing.
2. *Special cases (geometric degeneracies).* These include a vertex of one object incident on the face of another object. In principle, simple cases could be enumerated and handled. However, some widely available software fails.
3. Another problem is that current data structures are too complex for easy parallelization. Efficient parallelization prefers simple regular data structures, such as structures of arrays of plain old datatypes That disparages pointers, linked lists, and trees.

Some components of 3D-EPUG-Overlay have been presented earlier. PinMesh preprocesses a 3D mesh so that point locations can be performed quickly [24]. EPUG-Overlay overlays 2D meshes [23].

Background: Kettner et al [21] studied failures caused by roundoff errors in geometric problems. They also showed situations where epsilon-tweaking failed. Snap rounding arbitrary precision segments into fixed-precision numbers, Hobby [19], can also generate inconsistencies and deform the original topology. Variations attempting to get around these issues include de Berg et al [6], Hersberger [18], and Belussi et al [2]. Controlled Perturbation (CP), Melhorn [27], slightly perturbs the input to remove degeneracies such that the geometric predicates are correctly evaluated even using floating-point arithmetic. Adaptive Precision Floating-Point, Shewchuk [30], exactly evaluates predicates (e.g. orientation tests) using the minimum necessary precision.

Exact Geometric Computation (EGC), Li [22], represents mathematical objects using algebraic numbers to perform computations without errors. However this is slow.

One technique to accelerate algorithms based on exact arithmetic is to employ arithmetic filters and interval arithmetic, Pion et al [29], such as embodied in CGAL [4].

Current freely available implementations: One technique for overlaying 3-D polyhedra is to convert the data to a volumetric representation (voxelization), perhaps stored as an octree, Meagher [26], and then perform the overlay using the converted data. For exactly computing overlays, a common strategy is to use indexing to accelerate operations such as computing the triangle-triangle intersection. For example, Franklin [12] uses a uniform grid to intersect two polyhedra, Feito et al [11] and Mei et al [28] use octrees, and Yongbin et al [32] use Oriented Bounding Boxes trees (OBBs) to intersect triangulations.

Another algorithm that does not guarantee robustness is QuickCSG, Douze et al [9], which is designed to be extremely efficient. QuickCSG employs parallel programming and a *k-d*-tree index to accelerate the computation. However, it does not handle special cases (it assumes vertices are in general position), and does not handle the numerical non-robustness from floating-point arithmetic, Zhou et al [33]. To reduce errors caused by special cases, QuickCSG allows

Fall Workshop in Computational Geometry, Nov 2018, NYC
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the user to apply random numerical perturbations to the input, but this has no guarantees.

Although small errors may sometimes be acceptable, they accumulate if several inexact operations are performed in sequence. This gets even worse in CAD and GIS where it is common to compose operations. For use when exactness is required, Hachenberger et al [17] presented an algorithm for computing the exact intersection of Nef polyhedra.

Bernstein et al [3] presented an algorithm that tries to achieve robustness in mesh intersection by representing the polyhedra using binary space partitioning (*BSP*) trees with fixed-precision coordinates. It can intersect two such polyhedra by only evaluating fixed-precision predicates. However, in 3D, the BSP representation often has superlinear size, because the partitioning planes intersect so many objects. Also, converting BSPs back to more widely used representations (such as triangular meshes) is slow and inexact.

Recently, Zhou [33] presented an exact and parallel algorithm for performing booleans on meshes. The key is to use the concept of winding numbers to disambiguate self-intersections on the mesh. That algorithm is freely available and distributed in the LibiGL package, Jacobson et al [20]. Its implementation employs CGAL's exact predicates. The triangle-triangle intersection computation is also accelerated using CGAL's bounding-box-based spatial index. LibiGL is not only exact, but also much faster than Nef Polyhedra. However, it is still slower than fast inexact algorithms such as QuickCSG.

2 OUR TECHNIQUES

Our solution to the above problems combines the following five techniques.

Big rational numbers: Representing a number as the quotient of two integers, each represented as an array of groups of digits, is a classic technique. The fundamental limitation is that the number of digits grows exponentially with the depth of the computation tree. Our relevant computation comprises comparing the intersection of two lines defined by their endpoints against a plane defined by three vertices. So, this growth in precision is quite tolerable.

The implementation challenges are harder. Many C++ implementations of new data structures automatically construct new objects on a global heap, and assume the construction cost to be negligible. That is false for parallel programs processing large datasets. Constructing and destroying heap objects has a superlinear cost in the number of objects on the heap. Parallel modifications to the heap must be serialized. Therefore we carefully construct our code to minimize the number of times that a rational variable needs to be constructed or enlarged. This includes minimizing the number of temporary variables needed to evaluate an expression. Furthermore, we use interval arithmetic as a filter to determine when evaluation with rationals is necessary.

Simulation of Simplicity: Simulation of Simplicity (SoS), Edelsbrunner et al [10], addresses the problem that, "sometimes, even careful attempts at capturing all degenerate cases leave hard-to-detect gaps", Yap [31]. Figure 1 is a challenging case. It consists of two pyramids with central vertices incident at a common vertex v. v is non-manifold and is on 8 faces, 4 from each pyramid. It is not easy to determine which of the 8 faces should intersect the ray that would be run up from v in order to locate v. In the subproblem of point location, RCT gets this point location case wrong; PinMesh is correct because of SoS, Magalhães et al [24]. SoS symbolically perturbs coordinates by adding infinitesimals of different orders. The result is that there are no longer any coincidences, e.g., three points are never collinear.

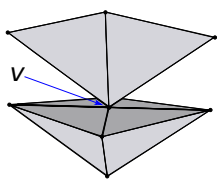

Figure 1: Difficult test case for 3-D point location.

Minimal topology: A sufficient representation of a 3-D mesh comprises the following: (a) the array of vertices, (v_i), where each $v_i = (x_i, y_i, z_i)$. (b) the array of tetrahedra or other polyhedra, t_i, used solely to store properties such as density, and (c) the array of augmented oriented triangular faces (f_i), where $f_i = (v_{i1}, v_{i2}, v_{i3}, t_{i1}, t_{i2})$. The tetrahedron or polyhedron t_{i1} is on the positive side of the face $f_i = (v_{i1}, v_{i2}, v_{i3})$; t_{i2} on the negative. It is unnecessary to store any further relations, such as from face to adjacent face, from vertex to adjacent face, edge loops, or face shells.

Note that there are no pointers or lists; we need only several structures of arrays. If the tetrahedra have no properties, then the tetrahedron array does not need to exist, so long as the tetrahedra, which we are not storing explicitly, are consistently sequentially numbered. The point is to minimize what types of topology need to be stored.

Uniform grid: The uniform grid, Akman et al [1], Franklin et al [13–15] is used as an initial cull so that, when two objects are tested for possible intersection, then the probability of intersecting is bounded below by a positive number. Therefore, the number of pairs of objects tested for intersection that do not actually intersect is linear in the number that do intersect. Thus the expected execution time is linear in the output size.

A careful concrete implementation of this abstraction is critical. We tested several choices; details are in Magalhães [7]. We also tested an octree, but our uniform grid implementation is much faster. We also used a second level grid for some cells. This allowed us to use an approximation to determine which faces intersected each cell: enclosing oblique faces with a box and then marking all the cells intersecting that box, which is more cells than necessary.

3 3-D MESH INTERSECTION

3D-EPUG-Overlay exactly intersects 3-D meshes. Its input is two triangular meshes M_0 and M_1. Each mesh contains a set of 3-D triangles representing a set of polyhedra. The output is another mesh where each represented polyhedron is the intersection of a polyhedron from M_0 with another one from M_1. The key is the combination of five techniques described later. Extra details are in Magalhães et al [7, 8, 23–25].

Data representation: The input is a pair of triangular meshes in 3-D (E^3). Both meshes must be watertight and free from self-intersections. The polyhedra may have complex and nonmanifold topologies, with holes and disjoint components. The two meshes may be identical, which is an excellent stress test, because of all the degeneracies.

There are two types of output vertices: input vertices, and intersection vertices resulting from intersections between an edge of one mesh and a triangle of the other. Similarly, there are two types of output triangles: input triangles and triangles from retesselation. The first contains only input vertices while the second may contain vertices generated from intersections created during the retesselation of input triangles. An intersection vertex is represented by an edge and the intersecting triangle. For speed, its coordinates are cached when first computed.

Retesselation of faces that were split was implemented with orientation predicates, Magalhães [7], which reduced to implementing 164 functions. A Wolfram Mathematica script was developed to create the code for all the predicates.

Experiments: 3D-EPUG-Overlay was implemented in C++ and compiled using g++ 5.4.1. For better parallel scalability, the gperftools Tcmalloc memory allocator [16], was employed. Parallel programming was provided by OpenMP 4.0, multiple precision rational numbers were provided by GNU GMPXX and arithmetic filters were implemented using the Interval_nt number type provided by CGAL for interval arithmetic. The experiments were performed on a workstation with 128 GiB of RAM and dual Intel Xeon E5-2687 processors, each with 8 physical cores and 16 hyper-threads, running Ubuntu Linux 16.04. We evaluated 3D-EPUG-Overlay, by comparing it against three state-of-the-art algorithms:

1. *LibiGL* [33], which is exact and parallel,
2. *Nef Polyhedra* [4], which is exact, and
3. *QuickCSG* [9], which is fast and parallel, but not exact, and does not handle special cases.

Our experiments showed that 3D-EPUG-Overlay is fast, parallel, exact, economical of memory, and handles special cases.

Experiments were performed with a variety of non self-intersecting and watertight meshes. The datasets and lengthy results are detailed in the full paper.

We compared 3D-EPUG-Overlay against other three algorithms. 3D-EPUG-Overlay was up to 101 times faster than LibiGL. The only test cases where the times spent by LibiGL were similar to the times spent by 3D-EPUG-Overlay were during the computation of the intersections of a mesh with itself (even in these test cases 3D-EPUG-Overlay was still faster than LibiGL). In this situation, the intersecting triangles from the two meshes are never in general position, and thus the computation has to frequently trigger the SoS version of the predicates, which we haven't not optimized yet. In the future, we intend to optimize this.

However, LibiGL also repairs meshes (by resolving self-intersections) during the intersection computation, which 3D-EPUG-Overlay does not attempt.

Because of the overhead of Nef Polyhedra and since it is a sequential algorithm, CGAL was always the slowest. When computing the intersections, 3D-EPUG-Overlay was up to 1,284 times faster than CGAL. The difference is much higher if the time CGAL spends converting the triangular mesh to Nef Polyhedra is taken into consideration: intersecting meshes with 3D-EPUG-Overlay was up to 4,241 times faster than using CGAL to convert and intersect the meshes.

While 3D-EPUG-Overlay was faster than QuickCSG in most of the test cases (mainly the largest ones), in others QuickCSG was up to 20% faster than 3D-EPUG-Overlay. The relatively small performance difference between 3D-EPUG-Overlay and an inexact method (that was specifically designed to be very fast) indicates that 3D-EPUG-Overlay presents good performance allied with exact results. Besides reporting errors during some experiments QuickCSG also failed in some situations where errors were not reported.

Finally, we also performed experiments with tetra-meshes. Each tetrahedron in these meshes is considered to be a different object and, thus, the output of 3D-EPUG-Overlay is a mesh where each object represents the intersection of two tetrahedra (from the two input meshes). These meshes are particularly hard to process because of their internal structure, which generates many triangle-triangle intersections. For example, during the intersection of the *Neptune* with the *Neptune translated* datasets (two meshes without internal structure), there are 78 thousand pairs of intersecting triangles and the resulting mesh contains 3 million triangles. On the other hand, in the intersection of *518092_tetra* (a mesh with 6 million triangles and 3 million tetrahedra) with *461112_tetra* (a mesh with 8 million triangles and 4 million tetrahedra) there are 5 million pairs of intersecting triangles and the output contains 23 million triangles.

To the best of our knowledge, LibiGL, CGAL and QuickCSG were not designed to handle meshes with multi-material and, thus, we couldn't compare the running time of 3D-EPUG-Overlay against them in these test cases.

We also evaluated the peak memory usage of each algorithm. 3D-EPUG-Overlay was: almost always smaller than LibiGL, with the difference increasing as the datasets became larger; smaller than QuickCSG in every case where QuickCSG returned the correct answer; and much smaller than CGAL. A typical result was the intersection of Neptune (4M triangles) with Ramesses (1.7M triangles): 3D-EPUG-Overlay used 2.6GB, LibiGL used 6.7GB, and CGAL 84GB. The largest example that 3D-EPUG-Overlay processed, 518092Tetra (6M triangles) with 461112Tetra (8.5M triangles) used 43GB. Magalhães [7] contains detailed results.

Correctness evaluation: 3D-EPUG-Overlay was developed on a solid foundation (i.e., all computation is exact and special cases are properly handled using Simulation of Simplicity) in order to ensure correctness. However, perhaps its implementation has errors? Therefore, we performed extensive experiments comparing it against LibiGL (as a reference solution). We employed the Metro tool, Cignoni et al [5], to compute the Hausdorff distances between the meshes being compared. Metro is widely employed, for example, to evaluate mesh simplification algorithms by comparing their results with the original meshes.

In every test, the difference between 3D-EPUG-Overlay and LibiGL was reported as 0. In some situations the difference between LibiGL and CGAL was a small number (maximum 0.0007% of the diagonal of the bounding-box). We guess this is because the exact results are stored using floating-point variables, and different strategies are used to round the vertices to floats and write them to the

text file. QuickCSG, on the other hand, generated errors much larger than CGAL: in the worst case, the difference between QuickCSG output and LibiGL was 0.13% of the diagonal of the bounding-box). Magalhães [7] contains detailed results.
Visual inspection: We also visually inspected the results using MeshLab. Even though small changes in the coordinates of the vertices cannot be easily identified by visual inspection (and even the program employed for displaying the meshes may have roundoff errors), topological errors (such as triangles with reversed orientation, self-intersections, etc) often stand out.

Even when QuickCSG did not report a failure, results were frequently inconsistent, with open meshes, spurious triangles or inconsistent orientations.
Rotation invariance: We also validated 3D-EPUG-Overlay by verifying that its result does not change when the input meshes are rotated. In all the experiments Metro reported that the resulting meshes were equal (i.e., the Hausdorff distance was 0.000000) to the corresponding ones obtained without rotation. In addition, we intersected several meshes with a rotated version of themselves. This is a notoriously difficult case for CAD systems because the large number of intersections and small triangles. In every experiment the Hausdorff distance between the two outputs was 0.000000. That is, we can quickly process cases that can crash CAD systems.

Summary: 3D-EPUG-Overlay is an algorithm and implementation to intersect a pair of 3D triangular meshes. It is simultaneously the fastest, free from roundoff errors, handles geometric degeneracies, parallelizes well, and is economical of memory. The source code, albeit research quality, is freely available for nonprofit research and education at https://github.com/sallesviana/MeshIntersection . We have extensively tested it for errors; we encourage others to test it. It is a suitable subroutine for larger systems such as 3D GIS or CAD systems. Computing other kinds of overlays, such as union, difference, and exclusive-or, would require modifying only the classification step. We expect that 3D-EPUG-Overlay could easily process datasets that are orders of magnitude larger, with hundreds of millions of triangles. Finally, 3D-EPUG-Overlay has not nearly been fully optimized, and could be made much faster.

REFERENCES

[1] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and the uniform grid data technique. *Computer Aided Design*, 21(7):410–420, 1989.

[2] A. Belussi, S. Migliorini, M. Negri, and G. Pelagatti. Snap rounding with restore: An algorithm for producing robust geometric datasets. *ACM Trans. Spatial Algorithms and Syst.*, 2(1):1:1–1:36, Mar. 2016.

[3] G. Bernstein and D. Fussell. Fast, exact, linear booleans. *Eurographics Symp. on Geom. Process.*, 28(5):1269–1278, 2009.

[4] Cgal, Computational Geometry Algorithms Library. https://www.cgal.org (retrieved Sept 2018).

[5] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Comput. Graph. Forum*, 17(2):167–174, June 1998.

[6] M. de Berg, D. Halperin, and M. Overmars. An intersection-sensitive algorithm for snap rounding. *Computational Geometry*, 36(3):159–165, Apr. 2007.

[7] S. V. G. de Magalhães. *Exact and parallel intersection of 3D triangular meshes.* PhD thesis, Rensselaer Polytechnic Institute, 2017.

[8] S. V. G. de Magalhães, W. R. Franklin, M. V. A. Andrade, and W. Li. An efficient algorithm for computing the exact overlay of triangulations. In *25th Fall Workshop on Computational Geometry*, U. Buffalo, New York, USA, 23-24 Oct 2015. (extended abstract).

[9] M. Douze, J.-S. Franco, and B. Raffin. *QuickCSG: Arbitrary and faster boolean combinations of n solids.* PhD thesis, Inria-Research Centre, Grenoble–Rhône-Alpes, France, 2015.

[10] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM TOG*, 9(1):66–104, 1990.

[11] F. Feito, C. Ogayar, R. Segura, and M. Rivero. Fast and accurate evaluation of regularized boolean operations on triangulated solids. *Computer-Aided Design*, 45(3):705 – 716, 2013.

[12] W. R. Franklin. Efficient polyhedron intersection and union. In *Proc. Graphics Interface*, pages 73–80, Toronto, 1982.

[13] W. R. Franklin. Adaptive grids for geometric operations. *Cartographica*, 21(2–3):161–167, Summer – Autumn 1984. monograph 32–33.

[14] W. R. Franklin, N. Chandrasekhar., M. Kankanhalli, M. Seshan, and V. Akman. Efficiency of uniform grids for intersection detection on serial and parallel machines. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proc. Computer Graphics International'88)*, pages 288–297. Springer-Verlag, 1988.

[15] W. R. Franklin, C. Narayanaswami, M. Kankanhalli, D. Sun, M.-C. Zhou, and P. Y. Wu. Uniform grids: A technique for intersection detection on serial and parallel machines. In *Proceedings of Auto Carto 9: Ninth International Symposium on Computer-Assisted Cartography*, pages 100–109, Baltimore, Maryland, 2-7 April 1989.

[16] S. Ghemawat and P. Menage. TCMalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html (retrieved on 13 Nov 2016), 15 Nov 2015.

[17] P. Hachenberger, L. Kettner, and K. Mehlhorn. Boolean operations on 3d selective nef complexes: Data structure, algorithms, optimized implementation and experiments. *Comupt. Geom.*, 38(1):64–99, Sept. 2007.

[18] J. Hershberger. Stable snap rounding. *Comput. Geom.*, 46(4):403–416, May 2013.

[19] J. D. Hobby. Practical segment intersection with finite precision output. *Comput. Geom.*, 13(4):199–214, 1999.

[20] A. Jacobson, D. Panozzo, et al. *libigl: A Simple C++ Geometry Processing Library*, 2016. http://libigl.github.io/libigl/ (Retrieved on 18 Oct 2017).

[21] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. *Comput. Geom. Theory Appl.*, 40(1):61–78, May 2008.

[22] C. Li. *Exact geometric computation: theory and applications,*. PhD thesis, Department of Computer Science, Courant Institute - New York University, January 2001.

[23] S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li. Fast exact parallel map overlay using a two-level uniform grid. In *4th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial)*, Bellevue WA USA, 3 Nov 2015.

[24] S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li. PinMesh – Fast and exact 3D point location queries using a uniform grid. *Computer & Graphics Journal, special issue on Shape Modeling International 2016*, 58:1–11, Aug. 2016. (online 17 May). Awarded a reproducibility stamp, http://www.reproducibilitystamp.com/.

[25] S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, W. Li, and M. G. Gruppi. Exact intersection of 3D geometric models. In *Geoinfo 2016, XVII Brazilian Symposium on GeoInformatics*, Campos do Jordão, SP, Brazil, Nov. 2016.

[26] D. J. Meagher. Geometric modelling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, June 1982.

[27] K. Mehlhorn, R. Osbild, and M. Sagraloff. Reliable and efficient computational geometry via controlled perturbation. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *ICALP (1)*, volume 4051 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2006.

[28] G. Mei and J. C. Tipper. Simple and robust boolean operations for triangulated surfaces. *CoRR*, abs/1308.4434, 2013.

[29] S. Pion and A. Fabri. A generic lazy evaluation scheme for exact geometric computations. *Sci. Comput. Program.*, 76(4):307 – 323, Apr. 2011.

[30] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discret. & Comput. Geom.*, 18(3):305–363, Oct. 1997.

[31] C. K. Yap. Symbolic treatment of geometric degeneracies. In M. Iri and K. Yajima, editors, *System Modelling and Optimization: Proc. 13th IFIP Conference*, pages 348–358. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[32] J. Yongbin, W. Liguan, B. Lin, and C. Jianhong. Boolean operations on polygonal meshes using obb trees. In *ESIAT 2009*, volume 1, pages 619–622. IEEE, 2009.

[33] Q. Zhou, E. Grinspun, D. Zorin, and A. Jacobson. Mesh arrangements for solid geometry. *ACM Trans. Graph.*, 35(4):39:1–39:15, July 2016.