

Exact fast parallel intersection of large 3-D triangular meshes

Salles Viana Gomes de Magalhães, W. Randolph Franklin and Marcus Vinícius Alvim Andrade

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>.

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

Salles Viana Gomes de Magalhães
Universidade Federal de Viçosa (MG) Brasil, e-mail: sallesviana@gmail.com

W. Randolph Franklin
Rensselaer Polytechnic Institute, Troy NY, USA 12180, e-mail: mail@wrfranklin.org

Marcus V.A. Andrade
Universidade Federal de Viçosa (MG) Brasil, e-mail: marcus.ufv@gmail.com

execution becomes desirable, the classic algorithms and implementations 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. There are a few reasons.
 - a. The number of special cases grows exponentially with the dimension. In 2-D, when intersecting an infinite line l with a polygon, (at least) the following cases occur with respect to the line's intersection with a finite edge e of the polygon: l crosses e 's interior, l is coincident with e , and l is incident on a vertex v of e , and the other edge e' incident on v is either coincident with l , on the same side of l as e , or on the opposite side of l as e . In 3-D, the problem is much worse, so that a complete enumeration may be infeasible.
 - b. One technique is to reduce the number of cases by combining them. E.g., when comparing point p against line l , the three cases of *above*, *on*, and *below* may be compressed into two: *above or on* and *below*. The problem is to do this in a way that results in higher level functions that call this as a component executing correctly. E.g., does intersecting two polylines where a vertex of one is coincident with a vertex of the other still work?
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. If the platform is an Nvidia GPU, then warps of 32 threads are required to execute the same instruction (or be idle). Ideally, the data used by adjacent threads is adjacent in memory. That disparages pointers, linked lists, and trees.

Some components of 3D-EPUG-OVERLAY have been presented earlier. PIN-MESH preprocesses a 3D mesh so that point locations can be performed quickly [31]. EPUG-OVERLAY overlays 2D meshes [30].

Background: Kettner et al [27] studied failures caused by roundoff errors in geometric problems. They also showed situations where epsilon-tweaking failed. (That uses an ϵ tolerance to consider two values x and y to be equal if $|x - y| \leq \epsilon$.) Snap rounding arbitrary precision segments into fixed-precision numbers, Hobby [24], can also generate inconsistencies and deform the original topology. Variations attempting to get around these issues include de Berg et al [6], Hersberger [23], and Belussi et al [2]. Controlled Perturbation (CP), Melhorn [34], 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 [39], exactly evaluates predicates (e.g. orientation tests) using the minimum necessary precision.

Exact Geometric Computation (EGC), Li [29], represents mathematical objects using algebraic numbers to perform computations without errors. E.g., $\sqrt{2}$ can be represented exactly as the pair $(x^2 - 2, [1, 2])$, interpreted as the root of the polynomial $x^2 - 2$ that lies in the interval $[1, 2]$. However this is slow. Even determining the sign of an expression such as $\sqrt{\sqrt{\sqrt{5}+1} + \sqrt{\sqrt{5}-1}} - \sqrt[4]{2\sqrt{5}+4}$ is nontrivial. (Answer: 0.)

One technique to accelerate algorithms based on exact arithmetic is to employ arithmetic filters and interval arithmetic, Pion et al [38], such as embodied in CGAL [4]. Arithmetic operations are applied to the intervals. If the sign of the exact result can be inferred based on the sign of the bounds of the interval, its value is returned. Otherwise, the predicate is re-evaluated using exact arithmetic.

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 [33], and then perform the overlay using the converted data. This approach has some advantages: first, the volumetric model can be created using any precision, and so, if the application does not demand a high precision, this algorithm can be used to compute a fast approximation of the overlay. Furthermore, it is trivial to perform a robust overlay of volumetric representations. However, the volumetric representation is usually not exact, and the overlay results are usually approximate. Also, oblique surfaces cannot be represented exactly, which impacts fluid flow and visualization. Also, the data structure size tends to grow at least quadratically with the desired resolution. Pavić et al. [37] present an efficient algorithm for performing this kind of overlay.

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 [35] use octrees, and Yongbin et al [41] use Oriented Bounding Boxes trees (OBBs) to intersect triangulations. Although those algorithms do not use approximations, robustness cannot be guaranteed because of floating point errors. For example, Feito et al [11] use a tolerance to process floating-point numbers, but this is error-prone.

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 [42]. To reduce errors caused by special cases, QuickCSG allows the user to apply random numerical perturbations to the input, but this has no guarantees.

Even if an algorithm using floating-point arithmetic can intersect two specific meshes consistently (i.e., without creating topological impossibilities or crashing), some output coordinates may not be exactly representable as floating-point numbers.

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 [21] presented an algorithm for computing the exact intersection of Nef polyhedra. A Nef polyhedron is a finite sequence of complement and intersection operations on half-spaces. Although dating from the 1970s, only in the 2000s were concrete algorithms developed, and then embodied into CGAL [4]. One application is the SFCGAL [36] backend to the PostGIS DBMS. SFCGAL wraps the CGAL exact representation for 2-D and 3-D data, allowing PostGIS to perform exact geometric computations. Although these algorithms are exact, they are slow, Leconte et al [28]. Also, in most cases, the data must be converted into the Nef format.

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 *BSP*s back to more widely used representations (such as triangular meshes) is slow and inexact.

Recently, Zhou [42] 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. Their algorithm first constructs an arrangement with the two (or more) input meshes, and then resolves the self-intersections in the combined mesh by retesselating the triangles such that intersections happen only on common vertices or edges. The self-intersection resolution eliminates not only the triangle-triangle intersections between triangles of the different input meshes, but also between triangles of the same mesh. As a result, their algorithm can also eliminate self-intersections in the input meshes, repairing them. Finally, a classification step is applied to compute the resulting boolean operations.

That algorithm is freely available and distributed in the LibiGL package, Jacobson et al [25]. 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 [40]. 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 [31].

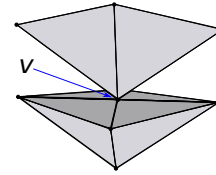


Fig. 1 Difficult test case for 3-D point location.

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. A positive infinitesimal, ϵ , is smaller than any positive real number (but greater than zero). That violates the Archimedean property for the real field, but we don't need this independent axiom. A second order infinitesimal, ϵ^2 , is smaller than any first order infinitesimal. Etc. Linear combinations of reals and infinitesimals work. In 1-D, SoS can be realized by indexing all the input variables of both input objects, and then modifying them thus:

$$x_i \rightarrow x_i + \epsilon^{(2^i)} \quad (1)$$

A coordinate's perturbation depends on its index. The efficient implementation of SoS is to examine its effects on each predicate and then to recode the predicate to have the same effect, but without the need to use infinitesimals. E.g., $x_i \leq x_j$ becomes $(x_i < x_j) \vee ((x_i = x_j) \wedge (i > j))$.

Because each type of predicate needs to be analyzed and recoded, we construct our algorithm to use only one type of predicate: the sign of a 3×3 determinant, or equivalently the order of 4 vertices in 3-D.

Minimal topology: The minimal explicit topology required for computing some property of an object depends on the desired property. E.g., testing for point location in a polygon requires only the set of unordered edges. That is still true for multiple and nested components. For computing the area and other mass properties, the set of

ordered edges (where we know each edge’s inside side) suffices. Alternatively, the set of vertices and their neighborhoods is sufficient, Kankanhalli et al [26], Franklin [14]. That means for each vertex, knowing its location, the directions of the two adjacent edges, and which adjacent sector is inside. A sufficient representation of a 3-D mesh comprises the following:

1. the array of vertices, (v_i) , where each $v_i = (x_i, y_i, z_i)$.
2. the array of tetrahedra or other polyhedra, t_i , used solely to store properties such as density, and
3. 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, 17] 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. Frey and George [18] contains a comprehensive description of spatial data structures.

Our basic algorithm goes as follows.

1. Choose a positive integer g for the grid resolution as a function of the statistics of the input data. Typically, $10 \leq g \leq 1000$. The goal is for each grid cell, as described later, to have a constant number of intersections with input objects.
2. Superimpose a 3-D grid of $g \times g \times g$ cells on the input data.
3. Each cell will contain an abstract data structure of the set of input objects intersecting it. Call it the *cell intersection set*.
4. For each input object, determine which cells it intersects, and insert it into each of those cells’ sets.

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.

OpenMP: Because the data structures are simple and the algorithms are regular, they are easily parallelizable with OpenMP to run on a multicore Intel Xeon. This should also parallelize well on an NVIDIA GPU, as we have done for other algorithms, Hedin et al [22], Franklin et al [16].

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, 30, 31, 32].

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.

Implementing intersection with simple geometric predicates: To simplify the implementation of the symbolic perturbation, we developed two versions of each geometric function. The first one focused on efficiency, and was implemented based on efficient algorithms available in the literature. The second one focused on simplicity, and was implemented using as only a few orientation predicates.

The idea is that, during the computation, the first version of each function is called. If a special case is detected, then the second version is called. In order to make sure the special cases are properly handled we only need to implement the perturbation scheme on these predicates.

The mesh intersection algorithm: This computation uses only local information. The algorithm has 3 basic steps and a uniform grid is employed to accelerate the computation:

1. Intersections between triangles of one mesh and triangles of the other mesh are detected and the new edges generated by the intersection of each pair of triangles are computed.

2. A new mesh containing the triangles from the two original meshes is created and the original triangles are split (retesselated) at the intersection edges. I.e., if a pair of triangles in this resulting mesh intersect, then this intersection will happen necessarily on a common edge or vertex.
3. A classification is performed: triangles that shouldn't be in the output are removed and the adjacency information stored in each triangle is updated to ensure that the new mesh will consistently represent the intersection of the two original ones.

A two-level 3-D uniform was employed in 3D-EPUG-OVERLAY to accelerate two important steps of the algorithm: the detection of intersections between pairs of input triangles, and the point location algorithm employed in the triangle classification.

After computing the intersections between each pair of triangles, the next step is to split the triangles where they intersect, so that after this process all the intersections will happen only on common vertices or edges. After the intersections between the triangles are computed, the triangles from one mesh that intersect triangles from the other one are split into several triangles, creating meshes M'_0 and M'_1 .

Retessellation 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 [20], 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* [42], 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.

Datasets: Experiments were performed with a variety of non self-intersecting and watertight meshes; see Figure 2 and Table 1. The ones with suffix *tetra* were tetrahedralized with GMSH [19]. The sources of the data are as follows: Barki (Clutch2kf, Casting10kf, Horse40kf, Dinosaur40kf, Armadillo52kf, Camel69kf, Cow76kf), AIM@SHAPE (Camel, Bimba, Kitten, RedCircBox, Ramesses, Vase, Neptune), Stanford (Armadillo), Thingi10K (461112, 461115, 226633), Thingi10k+GMSH

Table 1 Test datasets.

Mesh	Verts ($\times 10^3$)	Tris ($\times 10^3$)	Polys ($\times 10^3$)	Mesh	Verts ($\times 10^3$)	Tris ($\times 10^3$)	Polys ($\times 10^3$)
Clutch2kf	1	2	-	Casting10kf	5	10	-
Horse40kf	20	40	-	Dinausor40kf	20	40	-
Armadillo52kf	26	52	-	Camel	35	69	-
Camel69kf	35	69	-	Cow76kf	38	76	-
Bimba	75	150	-	Kitten	137	274	-
Armadillo	173	346	-	461112	403	805	-
461115	411	822	-	RedCircBox ^a	701	1403	-
Ramesses	826	1653	-	Ramesses Rot.	826	1653	-
Ramesses Tran.	826	1653	-	Vase	896	1793	-
226633	1226	2452	-	Neptune	2004	4008	-
Neptune Tran.	2004	4008	-	914686Tetra	66	605	281
68380Tetra	107	1067	506	Armad.Tetra ^b	340	3377	1602
Arm.Tet. ^b Tran.	340	3377	1602	518092Tetra	603	5938	2814
461112Tetra	842	8495	4046				

* meshes with the suffix Tetra have been tetrahedralized; * Rot. and Tran. mean, respectively, that the mesh has been rotated or translated; ^a Red Circular Box; ^b tetrahedralized version of the Armadillo mesh.

(914686Tetra, 68380Tetra, 518092Tetra, 461112Tetra), and Stanford+GMSH (Armad.Tetra).

Table 2 presents the pairs of meshes used in the intersection experiments, the number of input triangles, the number of triangles in the resulting meshes and the uniform grid size.

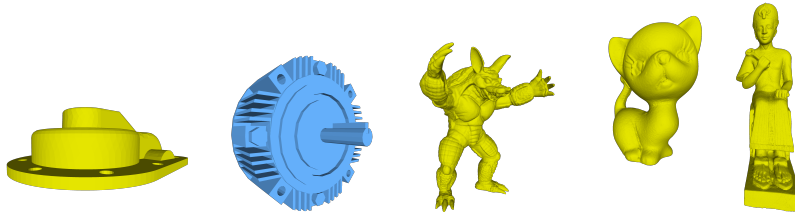
**Fig. 2** Some test meshes.

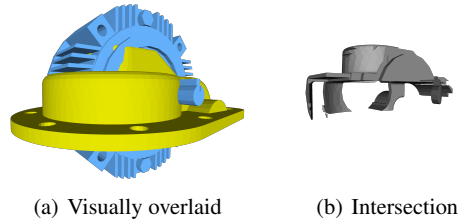
Figure 3 shows one test, which took 0.2 seconds.

Arithmetic filters and other optimizations: To evaluate the effect of different optimizations, we profiled two key steps: the creation of the uniform grid and the detection of intersections between pairs of triangles. These experiments were performed with the Neptune and Neptune translated meshes using a uniform grid with first level resolution 64^3 and second level resolution 16^3 . We evaluated various ver-

Table 2 Pairs of meshes intersected.

M_0	M_1	# triangles ($\times 10^3$)			Grid size	
		M_0	M_1	Out	G_1	G_2^a
Casting10kf	Clutch2kf	10	2	6	64	2
Armadillo52kf	Dinausor40kf	52	40	25	64	4
Horse40kf	Cow76kf	40	76	24	64	4
Camel69kf	Armadillo52kf	69	52	16	64	4
Camel	Camel	69	69	81	64	4
Camel	Armadillo	69	331	43	64	4
Armadillo	Armadillo	331	331	441	64	8
461112	461115	805	822	808	64	8
Kitten	RedCircBox	274	1402	246	64	8
Bimba	Vase	150	1792	724	64	8
226633	461112	2452	805	1437	64	8
Ramesses	RamessesTrans	1653	1653	1571	64	16
Ramesses	RamessesRotated	1653	1653	1691	64	16
Neptune	Ramesses	4008	1653	1112	64	16
Neptune	NeptuneTrans	4008	4008	3303	64	16
68380Tetra	914686Tetra	1067	605	9393	64	2
ArmadilloTetra	ArmadilloTetraTran.	3377	3377	61325	64	4
518092Tetra	461112Tetra	5938	8495	23181	64	4

^a resolution of the first level grid, second level grid.

**Fig. 3** Intersecting Casting10kf with Clutch2kf

sions of the algorithm, and observed that the biggest impacts on speed were caused by using a good allocator, by using an interval arithmetic filter for the rational computations, and by coding the rational arithmetic expressions to minimize memory allocations.

The importance of the uniform grid: This accelerates the detection of pairs of triangles that intersect. To evaluate this idea, we compared it against an implementation using the CGAL method for intersecting dD Iso-oriented Boxes. Both algorithms are exact and employ arithmetic filters with interval arithmetic. Indeed, this CGAL algorithm is employed by LibiGL to accelerate the triangle-triangle intersection detection step of its mesh intersection method.

The CGAL method is sequential, and employs a hybrid approach composed of a sweep-line and a streaming algorithm to detect intersections between pairs of Axis Aligned Bounding Boxes. Thus, pairwise intersections of triangles can be detected by filtering the pairs of intersecting bounding-boxes, and then testing the triangles for intersection. Since the CGAL exact kernel was not thread-safe, even the triangle-triangle intersection tests were performed sequentially. Since our uniform grid was designed to be parallel, we evaluated it using 32 threads.

Table 3 presents these comparative experiments, performed on 6 pairs of meshes. The number of intersections detected is not necessarily the same in the two algorithms because our algorithm implements Simulation of Simplicity. E.g., co-planar triangles never intersect.

CGAL is better at culling pairs of non-intersecting bounding-boxes and so performs fewer intersection tests. However, since the uniform grid is lightweight and parallelizes well, its pre-processing time is much smaller (up to 134 times faster, which is much more than the degree of parallelism), and this difference is never recaptured. Indeed, except for the intersection of the Armadillo with itself, even if CGAL took 0 seconds to detect the intersections the total time spent by the uniform grid method would still be smaller.

The only situation where the intersection detection time was much larger than the pre-processing time was in the intersection of the Armadillo mesh with itself. In this situation the uniform grid was still faster than CGAL for two reasons: first, the number of intersection tests performed by the two methods was similar. Second, the intersection computation done by the uniform grid method is performed in parallel.

The worst performance for both methods happened during the intersection of the Armadillo mesh with itself. There are many coincidences (co-planar triangles being tested for intersection, triangles intersecting other triangles on the edges, etc). These coincidences lead to arithmetic filter failures (because the result of some of the orientation predicates is 0 and, thus, the intervals representing these results are likely to have different signs for their bounds), which lead to exact computations with rationals. Furthermore, coincidences lead to the use of SoS predicates (which we have not optimized yet) when using the uniform grid.

Comparing 3D-EPUG-OVERLAY to other methods: We compared 3D-EPUG-OVERLAY against other three algorithms using the pairs of meshes presented in Table 2. The resulting running times (in seconds, excluding I/O) are presented in Table 4. Since the CGAL exact intersection algorithm deals with Nef Polyhedra, we also included the time it spent converting the triangulating meshes to this representation and to convert the result back to a triangular mesh (it often takes more time to convert the dataset than to compute the intersection). Both times are reported to let the user choose.

We can see that 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

Table 3 Comparing the times (in seconds) for detecting pairwise intersections of triangles using CGAL (sequential) versus using a uniform grid (parallel).

CGAL							
		# faces ($\times 10^3$)		# int. ^a	Int.tests ^b	Time (s)	
M_0	M_1	M_0	M_1 ($\times 10^3$)	($\times 10^3$)	($\times 10^3$)	Pre.proc. ^c	Inter. ^d
Camel	Armadillo	69	331	3	14	0.32	0.01
Armadillo	Armadillo	331	331	4611	5043	1.27	259.23
Kitten	RedC.Box ^e	274	1402	3	13	2.33	0.01
226633	461112	2452	805	23	128	7.18	0.08
Ramesses	Ram.Tran. ^f	1653	1653	36	237	12.38	0.17
Neptune	Nept.Tran. ^g	4008	4008	78	647	36.24	0.47
Uniform grid							
		# faces ($\times 10^3$)		# int. ^a	Int.tests ^b	Time (s)	
M_0	M_1	M_0	M_1 ($\times 10^3$)	($\times 10^3$)	($\times 10^3$)	Pre.proc. ^c	Inter. ^d
Camel	Armadillo	69	331	3	33	0.06	0.02
Armadillo	Armadillo	331	331	50	5351	0.25	63.80
Kitten	RedC.Box ^e	274	1402	3	27	0.08	0.02
226633	461112	2452	805	23	307	0.16	0.05
Ramesses	Ram.Tran. ^f	1653	1653	36	866	0.16	0.10
Neptune	Nept.Tran. ^g	4008	4008	78	5087	0.27	0.35

^a number of intersections detected; ^b number of intersection tests performed;

^c pre-processing time; ^d time spent testing pairs of triangles for intersection;

^e Red Circular Box; ^f Ramesses Translated; ^g Neptune Translated.

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 the experiments detached with a * in

Table 4 Times, in seconds, spent by different methods for intersecting pairs of meshes. QuickCSG reported errors during the intersections whose times are flagged with *. The tetrahedral mesh tests (last three rows) used only 3D-EPUG-OVERLAY.

M_0	M_1	Time (s)				
		3D-Epug	CGAL			QuickCSG
			LibiGL	Convert ^a	Intersect ^b	
Casting10kf	Clutch2kf	0.2	1.3	4.2	1.1	0.1*
Armadillo52kf	Dinosaur40kf	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.Tran. ^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	-	-	-	-

^a time converting the meshes to CGAL Nef Polyhedra;

^b time intersecting the Nef Polyhedra; ^c Ramesses Translated; ^d Ramesses Rotated;

^e Neptune Translated; ^f 68380Tetra; ^g 914686Tetra; ^h ArmadilloTetra;

ⁱ ArmadilloTetra Translated.

Table 4, QuickCSG also failed in some situations where errors were not reported (this will be detailed later).

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.

Let $e(p, S)$ be the minimum Euclidean distance between the point p and the surface S . [5] defines the one sided distance $E(S_1, S_2)$ between two surfaces S_1 and S_2 as: $E(S_1, S_2) = \max_{p \in S_1} e(p, S_2)$. The Hausdorff distance between two surfaces S_1 and S_2 is the maximum between $E(S_1, S_2)$ and $E(S_2, S_1)$. The Metro implementation employs an approximation strategy that samples points on the surface of the meshes in order to estimate the Hausdorff distance. In all experiments we employed the default parameters (where 10 points are sampled per face).

Since Metro is not exact (all the computation is performed using double variables), we use the distance between meshes only as evidence that our implementation is correct. 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.

Figure 4 shows the intersection of Bimba and the Vase. The first part is the complete overlay mesh, as computed by 3D-EPUG-OVERLAY. The second is a detail of an error-prone output region, computed correctly by 3D-EPUG-OVERLAY. The third part shows the same region computed by QuickCSG. Note the errors along the edges.

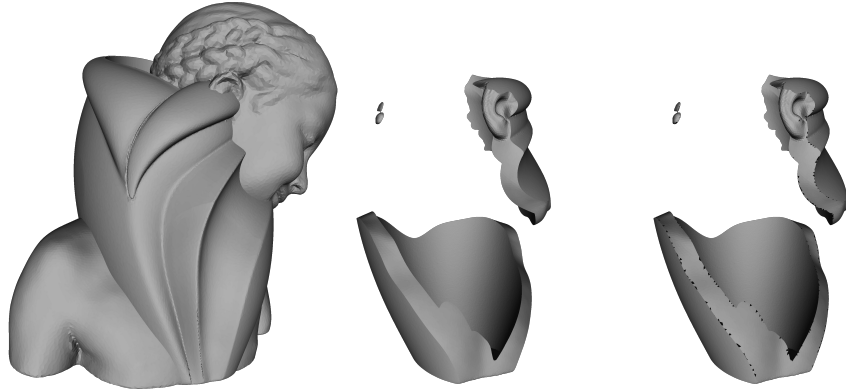


Fig. 4 Intersection of the Bimba and Vase meshes computed by 3D-EPUG-OVERLAY and QuickCSG, showing only 3D-EPUG-OVERLAY computing a region correctly.

Figure 5 (a) presents a zoom in the output of QuickCSG for the intersection of the Ramesses dataset with Ramesses Translated: some triangles are oriented incorrectly. These errors may be created either by floating-point errors or because QuickCSG doesn't handle the coincidences.

To mitigate this later problem, QuickCSG provides options where the user can apply a random perturbation in the input dataset. In contrast to the symbolic perturbations of Simulation of Simplicity, these numerical perturbations are not guaranteed to work, and the user has to choose a maximum range. A too-small range may not eliminate all errors while a too-big range may modify the mesh too much. Figures 5 displays the results from perturbations with maximum range 10^{-1} , and 10^{-6} . None of these perturbations removed all errors and the bigger perturbation (10^{-1}) even added undesirable artifacts to the output. Similar problems in QuickCSG have been reported by [42].

Rotation invariance: We also validated 3D-EPUG-OVERLAY by verifying that its result does not change when the input meshes are rotated. I.e., a pair of meshes were rotated around the same point, intersected, and the resulting mesh was rotated back. To ensure exactness, we chose a rotation angle with rational sine and cosine. We evaluated all the pairs in Table 2. For each pair, we performed a rotation around the x axis and, then, a rotation around the y axis (the origin was defined as the center of the joint bounding-box of the two meshes). We chose rotation angle θ such that $\sin \theta = 400/10004$ and $\cos \theta = 9996/10004$. $\theta \approx 2.29$ degrees.

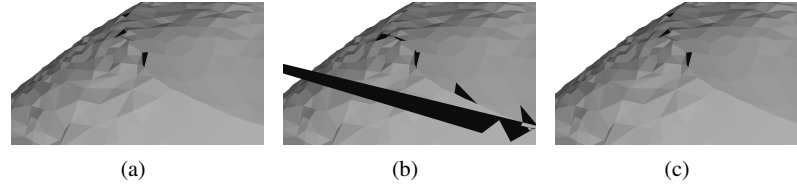


Fig. 5 Detail of the intersection of Ramesses with Ramesses Translated generated by QuickCSG using different ranges for the numerical perturbation: (a) no perturbation, (b) 10^{-1} , and (c) 10^{-6} .

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 each mesh from Table 2 with a rotated version of itself. This is a notoriously difficult case for CAD systems because the large number of intersections and small triangles. Each mesh M was rotated around the center of its bounding-box using the above θ , and intersected with its original version, using both LibiGL and 3D-EPUG-OVERLAY. 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.

Limitations: Even though the computations are performed exactly, common file formats for 3D objects such as OFF represent data using floating-point numbers. Converting 3D-EPUG-OVERLAY’s rational output into floats may introduce errors since most rationals cannot be represented exactly. Possible solutions include avoiding the conversion (i.e., always employing multiple-precision rationals in the representations), or using heuristics such [42] to try to choose floats for each coordinate so that the approximate output will not only be similar to the exact one, but also it will not present topological errors.

A limitation of symbolic perturbation is that the results are consistent considering the perturbed dataset, not necessarily considering the original one [10]. Thus, if the perturbation in the mesh resulting from the intersection is ignored, the unperturbed mesh may contain degeneracies such as triangles with area 0 or polyhedra with volume 0 (these polyhedra would have infinitesimal volume if the perturbation was not ignored). More details are in [7].

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. Polygon properties calculated from the vertex neighborhoods. In *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pages 110–118, 1987.
15. 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.
16. W. R. Franklin and S. V. G. Magalhães. Parallel intersection detection in massive sets of cubes. In *Proceedings of BigSpatial’17: 6th ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*, Los Angeles Area, CA, USA, 7-10 Nov 2017.
17. 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.
18. P. J. Frey and P. George. *Mesh Generation: Application to Finite Elements, Second Edition*. ISTE Ltd. and Wiley, 2010.
19. C. Geuzaine and J.-F. Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. for Numerical Methods in Eng.*, 79(11):1309–1331, May 2009.

20. 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.
21. P. Hachenberger, L. Kettner, and K. Mehlhorn. Boolean operations on 3d selective nef complexes: Data structure, algorithms, optimized implementation and experiments. *Comput. Geom.*, 38(1):64–99, Sept. 2007.
22. D. Hedin and W. R. Franklin. Nearptd: A parallel implementation of exact nearest neighbor search using a uniform grid. In *Canadian Conference on Computational Geometry*, Vancouver Canada, Aug. 2016.
23. J. Hershberger. Stable snap rounding. *Comput. Geom.*, 46(4):403–416, May 2013.
24. J. D. Hobby. Practical segment intersection with finite precision output. *Comput. Geom.*, 13(4):199–214, 1999.
25. A. Jacobson, D. Panozzo, et al. *libigl: A Simple C++ Geometry Processing Library*, 2016. <http://libigl.github.io/libigl/> (Retrieved on 18 Oct 2017).
26. M. Kankanhalli and W. R. Franklin. Area and perimeter computation of the union of a set of iso-rectangles in parallel. *J. Parallel Distrib. Comput.*, 27(2):107–117, June 1995.
27. 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.
28. C. Leconte, H. Barki, and F. Dupont. Exact and Efficient Booleans for Polyhedra. Technical Report RR-LIRIS-2010-018, LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2/École Centrale de Lyon, Oct. 2010. (Retrieved on 19 Oct 2017).
29. C. Li. *Exact geometric computation: theory and applications*. PhD thesis, Department of Computer Science, Courant Institute - New York University, January 2001.
30. 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.
31. 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/>.
32. S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, W. Li, and M. G. Grupp. Exact intersection of 3D geometric models. In *GeoInfo 2016, XVII Brazilian Symposium on Geoinformatics*, Campos do Jordão, SP, Brazil, Nov. 2016.
33. D. J. Meagher. Geometric modelling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, June 1982.
34. K. Mehlhorn, R. Osbald, 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.
35. G. Mei and J. C. Tipper. Simple and robust boolean operations for triangulated surfaces. *CoRR*, abs/1308.4434, 2013.
36. Oslandia and IGN. *SFCGAL*, 2017. <http://www.sfcgal.org/> (Retrieved on 19 Oct 2017).
37. D. Pavić, M. Campen, and L. Kobbelt. Hybrid booleans. *Comput. Graph. Forum*, 29(1):75–87, Jan. 2010.
38. S. Pion and A. Fabri. A generic lazy evaluation scheme for exact geometric computations. *Sci. Comput. Program.*, 76(4):307 – 323, Apr. 2011.
39. J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discret. & Comput. Geom.*, 18(3):305–363, Oct. 1997.
40. 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.
41. 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.
42. 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.