Fast exact parallel 3D mesh intersection algorithm using only orientation predicates*

Salles V. G. Magalhães
Universidade Fed. de Viçosa
DPI, Campus da UFV
Viçosa, MG, Brazil 36570-000
salles@ufv.br

W. Randolph Franklin
Rensselaer Polytechnic Inst.
110 8th street
Troy, NY 12180
mail@wrfranklin.org

Marcus V. A. Andrade
Universidade Fed. de Viçosa
DPI, Campus da UFV
Viçosa, MG, Brazil 36570-000
marcus@ufv.br

ABSTRACT

We present an algorithm to compute the intersection of two 3D triangulated meshes. It has applications in GIS, CAD and Additive Manufacturing, and was developed to process big datasets quickly and correctly. The speed comes from simple regular data structures that parallelize very well. The correctness comes from using multiple-precision rational arithmetic to prevent roundoff errors and the resulting topological inconsistencies, and symbolic perturbation (simulation of simplicity) to handle special cases (geometric degeneracies). To simplify the symbolic perturbation, the algorithm employs only orientation predicates. This paper focuses on the challenges and solutions of the implementing symbolic perturbation. Our preliminary implementation has intersected two objects totalling 8M triangles in 11 elapsed seconds on a dual 8-core Xeon. The competing LibiGL took 248 seconds and CGAL took 2726 seconds. Our software is freely available for nonprofit research and education.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**; **Computer graphics**; **Shape modeling**;

KEYWORDS

GIS, 3D, Parallel Programming, Computational Geometry

ACM Reference format:
Salles V. G. Magalhães, W. Randolph Franklin, and Marcus V. A. Andrade. 2017. Fast exact parallel 3D mesh intersection algorithm using only orientation predicates. In *Proceedings of ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Redondo Beach, CA USA, Nov 2017 (SIGSPATIAL'17),* 11 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Computing intersections or overlays is important to GIS, CAD, Additive Manufacturing and computational geometry. While GIS usually deal with 2D data, there are several applications for 3D GIS. For example, while a 2D model could represent urban or hydrological networks, or the different kinds of soil in a region, a 3D model could model more complex features such as layers of soil in a mine, subway tunnels, buildings, etc. Computing intersections is an important operation often required by these systems. An example of application is to intersect polyhedra representing layers of soil with a polyhedron representing a section of the soil to be mined.

Although 3D models have been widely used, processing is still a challenge [10]. Due to the algorithm complexity caused by handling special cases, the necessity of processing big datasets, and floating point arithmetic errors, they note that software packages occasionally "fail to give a correct result, or they refuse to give a result at all". The likelihood of failure increases for bigger datasets.

An algorithm that occasionally fails might be acceptable. Nevertheless, an efficient, robust, and even exact, algorithm is especially important when it is a subroutine of another algorithm.

In previous works we have developed exact and efficient algorithms for processing 2D (polygonal maps) and 3D models (triangulated meshes). These algorithms employ a combination of five separate techniques to achieve both robustness and efficiency. Exact arithmetic is employed to completely avoid errors caused by floating point numbers. Special cases (geometric degeneracies) are treated using *Simulation of Simplicity* (SoS) [9]. The computation is performed using simple local information to make the algorithm easily parallelizable and to easily ensure robustness. Efficient indexing techniques with a uniform grid, and High Performance Computing (HPC) are used to mitigate the overhead of exact arithmetic.

In all these algorithms, our spatial data is represented using simple topological formats. The 2D maps are represented using sets of oriented edges where each edge contains the labels of the polygons on its positive and negative sides. In 3D, the meshes are represented using a set of oriented triangles and each triangle has the labels of the polyhedra on its positive and negative sides.

In this paper we will present 3D-EPUG-Overlay[1], an exact algorithm to intersect 3D triangulated meshes. This project is an extension of preliminary work previously published in Magalhães et al. [17], which presented the general idea of the algorithm. This paper presents an improved algorithm using only orientation operations. With this simplicity, implementing the complicated boolean expressions caused by special case 3D intersections is easier. We also discuss the challenges associated with the use of SoS to handle degeneracies.

*Produces the permission block, and copyright information

SIGSPATIAL'17, Nov 2017, Redondo Beach, CA USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

[1] Acronym for *Exact*, *Parallel* and *Uniform Grid*

2 RELATED WORK

While most of the available algorithms are inexact, recently some researchers have developed exact algorithms for performing boolean operations on meshes.

Hachenberger et al. [12] presented, and CGAL [5] implemented, an algorithm for computing exact booleans on Nef polyhedra (a Nef polyhedron is a finite sequence of complement and intersection operations on half-spaces). However, these algorithms have limitations such as poor performance due to the heavy-weight data structures associated with the representation [14, 21]. Another concern is that Nef Polyhedra are uncommon.

An example of application of the CGAL exact geometry in GIS is the SFCGAL [19] backend of the PostGIS DBMS. SFCGAL wraps wraps CGAL exact representation for 2D and 3D data, allowing PostGIS to perform exact geometric computation.

Bernstein and Fussell [4] also presented an intersection algorithm that tries to achieve robustness. Their basic idea is to represent the polyhedra using binary space partitioning (*BSP*) trees with fixed-precision coordinates. They mention that the main limitation is that the process to convert BSPs to widely used representations (such as meshes) is slow and inexact.

Recently Zhou et al. [21] presented an exact and parallel algorithm for performing booleans on meshes. The key of their algorithm 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 and, as result, their algorithm can also eliminate self-interesctions in the input meshes, repairing them. Finally, a classification step is applied to compute the resulting boolean operations.

Zhou et al. [21] employs exact predicates provided by CGAL. Furthermore, the triangle-triangle intersection computation is accelerated using CGAL's bounding-box-based spatial index.

3 DATA REPRESENTATION

The program input is a pair of meshes in 3D (E^3). Each mesh is a partition (triangulation) of the space into polyhedra. The polyhedra may have complex topologies, with holes and disjoint components. Polyhedra from the same mesh intersect only at common vertices, edges, and faces (triangles). Triangles from the same mesh intersect only at common vertices and edges. All triangles have a nonzero area and all tetrahedra a nonzero volume. (A mesh that violates those conditions can be cleaned up by splitting edges and retriangulating.) Vertices and polyhedra are labeled with ids.

Each mesh M is a set of triangles (a triangle soup) $\{t_i\}$. Each triangle $t = (a_0, a_1, a_2, q_+, q_-)$. (a_0, a_1, a_2) is the oriented list of triangle vertices, and (q_+, q_-) are the adjacent polyhedra on the positive and negative sides of t. Note that we store no global topology, such as face shells, because we do not need it. Our data structure simplicity is key, e.g., in facilitating parallelization.

Two types of vertices are processed by the algorithm: (1) input vertices occurring in the input meshes, and (2) intersection vertices resulting from intersections between an edge of one mesh and a triangle of the other. Similarly to the vertices, there are two types of triangles: input triangles and triangles from retesselation. The first one contains only input vertices while the second one may contain vertices generated from intersections and are created during the retesselation of input triangles.

A vertex from intersection is represented by the edge and the input triangle whose intersection generated it. Even though it is possible to compute the coordinates of these vertices using the information about how the they were generated, we have an additional data structure that caches these coordinates to avoid recomputation.

As it will be mentioned later, distinguishing input vertices from vertices generated from intersections is important for the implementation of the symbolic perturbation.

4 ROUNDOFF ERRORS

The finite precision of floating point computations cause floating arithmetic to violate most of the axioms of the real number field. E.g., associativity under addition is not always true because $(10^{-20} + 1)$ rounds to 1, so $(10^{-20} + 1) - 1 \neq 10^{-20} + (1 - 1)$. Such arithmetic roundoff errors can cause the sign of a determinant to be evaluated wrong, resulting in a boolean predicate being wrong. Therefore, whether two edges intersect can be determined wrongly. Worse, a transformation that ought to preserve such incidence properties, such as translations, rotations, or scalings, might erroneously change how the predicate is evaluated.

These arithmetic errors can lead to topological errors and impossibilities causing an algorithm to produce inconsistent output. Although the chance of any one roundoff error being problematic may be quite small, as datasets get larger, the probability of a problem somewhere also gets larger. Our solution is to compute within a field of rational numbers, so that there are no roundoff errors. Rationals are slower than floats, but our algorithm is fast enough that this is not a problem.

5 GEOMETRIC DEGENERACIES

Another common source of error in geometric algorithms are the special cases (geometric degeneracies). Algorithms are usually described considering they will process non-degenerate input. However, during the actual implementation, degenerate data have to be considered. Degeneracies increase the number of cases that must be considered. E.g., when comparing point q against line l, there are now three cases: q may be (above / on / below) l instead of only two (above / below). By itself, this would not be bad, except that this predicate may be a component of a larger one. Perhaps q is a vertex of a piecewise straight line m with vertices qpr, and we wish to know how m intersects l. There are now more special cases. Next, consider the intersection of the piecewise straight line with vertices $a_0a_1a_2$ with the piecewise straight line with vertices $b_0b_1b_2$. Perhaps $a_1 = b_1$, or a_0a_1 is collinear with b_1b_2, and so on...

As mentioned by Yap [20], "sometimes, even careful attempts at capturing all degenerate cases leave hard-to-detect gaps". Properly handling special cases is a challenge mainly in geometric problems such as the mesh intersection, that depend on several subproblems, each one with its own special cases.

A technique for handling special cases is Simulation of Simplicity (SoS) [9], a general-purpose symbolic perturbation technique. It is

based on the idea that if the geometric objects are perturbed with infinitesimals, then the degeneracies disappear.

SoS uses a symbolic perturbation by a power of an indeterminate infinitesimal ϵ. Its mathematical formalization extends some exactly computable field, such as rationals, by adding orders of infinitesimals. Floating point numbers cannot be the base because of roundoff errors. The infinitesimal is an indeterminate. It has no meaning apart from the rules for how it combines. All positive first-order infinitesimals are smaller than the smallest positive number. All positive second-order infinitesimals are smaller than the smallest positive first-order infinitesimal, and so on. All this is consistent, satisfying the axioms of an abstract algebra field.

The result of SoS is that degeneracies are resolved in a way that is globally consistent, which is crucial for the development of algorithms relying on multiple geometric predicates. Figure 1 illustrates the importance of the consistency: suppose edges uv and uw are collinear. If after a perturbation, w is on the positive side of uv, then a predicate that computes the orientation of w with respect to uv should return "positive". At the same time, a predicate that verifies if w' is closer to x than v' is should return "true", otherwise the results of these two predicates would be inconsistent.

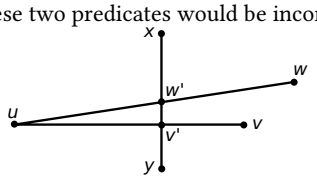


Figure 1: Importance of global consistency and determining if a segment is below another one.

To implement the algorithm using SoS, the perturbation scheme must satisfy three properties. (1) The perturbed dataset must be non-degenerate, (2) it must retain the non-degenerate properties of the original one and (3) the computational overhead of processing it instead of the original one must be small.

Edelsbrunner illustrated the application of SoS by implementing an orientation predicate and employing it to solve the convex hull problem. Initially, the predicate is applied to the non-perturbed input points and the result can be returned if it is non-degenerate (because of the second property). If a degeneracy is detected, a function that evaluates the predicate considering the perturbed dataset is applied. This is efficient because there is no overhead if the input is non-degenerate and because the use of different orders of infinitesimals allows the computation of the predicate without evaluating all the terms in the expressions.

5.1 The symbolic perturbation

In Edelsbrunner [9], the geometric objects processed by the convex hull algorithm are the input vertices. In the 2D version of the problem, the j-th coordinate of each vertex was perturbed by translating it by $\epsilon^{2^{i-j}}$. As shown by the authors, even if all the points were originally coincident, after the perturbation no three points would be collinear, which completely eliminates degeneracies for the convex hull problem.

In contrast, for the mesh intersection problem, we assume that the individual meshes are not degenerate, and coincidences happen only when they are processed together. For example, two triangles from the same mesh only intersect at common edges or vertices but the interior of a triangle from one mesh can intersect a co-planar triangle from the other mesh, which represents a special case of the intersection algorithm.

Allowing an individual mesh M to be degenerate is challenging because it is hard to develop a perturbation scheme that ensures M will be consistent after the perturbation. For example, M could contain two triangles where all six vertices coincide. Even though a perturbation scheme such as the one employed by Edelsbrunner [9] for the 3D orientation would ensure no two vertices will coincide, there is no guarantee that the information about the tetrahedra on each side of the triangles will be consistent after the perturbation.

For the rest of this paper the following notation will be employed. The two input meshes will be represented by M_0 and M_1 and the corresponding perturbed meshes will be, respectively, $M_{0\epsilon}$ and $M_{1\epsilon}$. All the perturbed geometric objects (vertices, edges and triangles) will be followed by an ϵ subscript (for example, vertex v_ϵ is the perturbed version of vertex v).

We propose the following perturbation scheme for the 3D mesh intersection problem. (1) Do not modify mesh 0. (2) Translate each vertex of mesh 1 equally by the vector $(\epsilon, \epsilon^2, \epsilon^3)$, i.e., $v_{j\epsilon} = (v_{jx} + \epsilon, v_{jy} + \epsilon^2, v_{jz} + \epsilon^3)$. Now, no vertex from $M_{0\epsilon}$ can coincide with any vertex from $M_{1\epsilon}$.

The rest of this Section will present lemmas describing properties of the perturbed meshes. The proofs will be omitted because of space limitations.

Lemma 5.1. *If a_ϵ is a vertex from mesh i and t_ϵ is a triangle from mesh $1-i$, then a_ϵ and t_ϵ are not coplanar.*

Lemma 5.2. *Given an edge $e_{1\epsilon} = a_\epsilon b_\epsilon$ ($a_\epsilon \neq b_\epsilon$), i.e., the endpoints are a_ϵ and b_ϵ, from mesh i and another edge $e_{2\epsilon} = c_\epsilon d_\epsilon$ ($c_\epsilon \neq d_\epsilon$) from mesh $1-i$ such that $e_{1\epsilon}$ and $e_{2\epsilon}$ are not parallel, then $e_{1\epsilon}$ and $e_{2\epsilon}$ do not intersect.*

Lemma 5.3. *Given two distinct vertices a_ϵ and b_ϵ from mesh i and another vertex c_ϵ from mesh $1-i$, then a_ϵ, b_ϵ and c_ϵ are not collinear.*

Lemma 5.4. *If an edge e_ϵ from a mesh i intersects a triangle t_ϵ from mesh $(1-i)$, then this intersection happens in the interior of t_ϵ.*

Lemma 5.5. *If e_ϵ is an edge from mesh i and t_ϵ is a triangle from mesh $1-i$, then e_ϵ and t_ϵ are not coplanar.*

Lemma 5.6. *If $t_{i\epsilon}$ and $t_{(1-i)\epsilon}$ are triangles belonging to, respectively, meshes i and $1-i$, then $t_{i\epsilon}$ and $t_{(1-i)\epsilon}$ are not co-planar.*

While these lemmas could be still valid for some other perturbation schemes, they are not valid for all perturbations. For example, if mesh M_1 was translated by $(\epsilon, \epsilon, \epsilon)$ instead of $(\epsilon, \epsilon^2, \epsilon^3)$, then the following three points would be collinear: $a_\epsilon = (0, 0, 0)$, $b_\epsilon = (1, 1, 1)$ and $c_\epsilon = (0 + \epsilon, 0 + \epsilon, 0 + \epsilon)$ (where a_ϵ and b_ϵ are in mesh $M_{0\epsilon}$ and c_ϵ is in mesh $M_{1\epsilon}$).

6 IMPLEMENTING INTERSECTION WITH SIMPLE GEOMETRIC PREDICATES

To simplify the implementation of the symbolic perturbation, we developed two versions of each geometric function employed in the mesh intersection algorithm. 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 few geometric predicates as possible.

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. As will be seen, the second version of the functions was implemented with only orientation predicates.

The main advantage of having the orientation predicates as base for other functions is the simplicity of the implementation. It is usually much easier to implement symbolic perturbation with orientation predicates than with more complicated functions, such as using barycentric coordinates to detect if a point is in a triangle.

Since the focus of this paper is the implementation of the symbolic perturbation scheme, details about the first version of the geometric functions will not be presented. Furthermore, as mentioned before, the first versions of the functions are used only for speed. Our algorithm can be correctly implemented without them.

6.1 Orientation predicates

From Edelsbrunner [9], the orientation of a sequence of $d+1$ points in E^d is "either negative or positive - unless the $d+1$ points lie on a common hyperplane, in which it is undefined". The orientation is

$$orientation(p_0, p_1, .., p_d) = sgn\begin{pmatrix}\begin{vmatrix} p_{00} & p_{01} & \cdots & p_{0(d-1)} & 1 \\ p_{10} & p_{11} & \cdots & p_{1(d-1)} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ p_{d0} & p_{d1} & \cdots & p_{d(d-1)} & 1 \end{vmatrix}\end{pmatrix}$$

where p_{ij} is the j-th coordinate of point i and sgn is the signum function that returns -1 if the argument is negative, 1 if it is positive and 0 if it is 0 (coincidence).

As it will be shown later, the predicates employed by the 3D mesh intersection algorithm will be the 1D, 2D and 3D orientation. These predicates will be adapted to handle the perturbed points, remaining consistent with the perturbation. E.g,, because of Lemma 5.1 $orientation(a_\epsilon, b_\epsilon, c_\epsilon, d_\epsilon)$ will never be 0 if $a_\epsilon b_\epsilon c_\epsilon$ is a triangle from one mesh and d_ϵ is a vertex of the other mesh.

As mentioned in the section on symbolic perturbation above, not all coincidences are eliminated by the perturbations. For example, $orientation(a_\epsilon, b_\epsilon, c_\epsilon, d_\epsilon)$ may be 0 if all the points are from the same mesh. However, as it will be shown, in the few functions where these coincidences may happen, the behavior of the algorithm is well defined and the coincidence does not propagate to other steps of the algorithm.

7 THE MESH INTERSECTION ALGORITHM

The computation is performed using only local information stored in the individual triangles. That is, the triangles from one mesh are intersected with the triangles from the other one. Then a new mesh containing the triangles from the two original meshes is created, and the original triangles are split at the intersection points. I.e., if a pair of triangles in this new mesh intersect, then this intersection will happen necessarily on a common edge or vertex. Finally, 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.

7.1 Intersecting triangles

For speed, a uniform grid is employed to cull the number of pairs of triangles that need to be tested for intersection. For each uniform grid cell, the intersections between pairs of triangles from the two triangulations are computed. Triangles that do not occur in the same cell are not tested.

More specifically, a two-level 3D uniform grid is employed to accelerate the computation. That is, the grid will be created by inserting in its cells triangles from both meshes $M_{0\epsilon}$ and $M_{1\epsilon}$. Then, for each grid cell c, the pairs of triangles from both meshes in c are intersected. If the resolution of the uniform grid is chosen such that the expected number of triangles per grid cell is a constant K, then it is expected that each triangle will be tested for intersection with the other K triangles in its grid cell. Thus, the expected total number of intersection tests performed will probably be linear in the size of the input maps. The exception would be if there were a superlinear number of triangle pairs very close to each other. This does not occur in real datasets, which satisfy a form of Lipshitz condition bounding the maximum density of elements. Also, experiments with uniform grids in various applications show excellent performance on real data [2, 11, 16].

Since the cells do not influence each other, the process of intersecting the triangles can be trivially parallelized: the grid cells can be processed in parallel by different threads using a parallel programming API such as OpenMP.

7.1.1 Implementation with orientation predicates. Let t_0 and t_1 be two triangles from, respectively, meshes M_0 and M_1. Assume $t_{0\epsilon}$ and $t_{1\epsilon}$ intersect and, w.l.o.g, let e_ϵ be an edge of one of the triangles $t_{i\epsilon}$ that intersects $t_{(1-i)\epsilon}$. Since, because of Lemma 5.1, no vertex of e_ϵ can be on the plane of $t_{(1-i)\epsilon}$, it is clear that the intersection will necessarily happen in the interior of e_ϵ. Furthermore, e_ϵ will intersect the interior of $t_{(1-i)\epsilon}$ (Lemma 5.4)

Since $t_{0\epsilon}$ and $t_{1\epsilon}$ cannot be co-planar and edges intersect only the interior of triangles, the intersection of $t_{0\epsilon}$ and $t_{1\epsilon}$ will always be an edge $u_\epsilon v_\epsilon$ (with $u_\epsilon \neq v_\epsilon$), where u_ϵ and v_ϵ are vertices generated from the intersection of the interior of one of the triangles and the interior of one edge of the other triangle.

Vertices u_ϵ and v_ϵ can be computed by testing the intersection of the six combinations of edges from one triangle against the other one. The number of intersections detected will be either zero (when the triangles do not intersect) or two (when they do intersect).

As showed by [18], the intersection between an edge e_ϵ and a triangle can be detected by computing five 3D orientations. For example, if both vertices of e_ϵ the are on the same side of the triangle, then they do not intersect. Therefore, the intersection computation can be implemented employing only orientation predicates.

Since intersections are computed only between input triangles, in this step the only necessary predicate is the 3D orientation of input vertices. The vertices from intersection are created in this step of the algorithm, and since their coordinates are stored implicitly as a pair (edge, triangle), no further computation is necessary.

Furthermore, since there will be no coincidence between edges of one perturbed mesh and triangles of the other perturbed mesh, the 3D orientation predicates will never return 0.

7.2 Retesselating the triangles

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. When a triangle is split, the labels of its two bounding objects will be copied to the new triangles.

Figure 2 presents an example of intersection computation. In Figure 2(a), we have two meshes representing two tetrahedra with one region in each one: the brown mesh (mesh M_0) bounds the exterior region and region 1 while the yellow mesh (mesh M_1) bounds the exterior region and region 2.

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'. The only triangle from mesh M_0 that intersects mesh M_1 is BCD. Since BCD intersects three triangles from M_1, it was split into seven triangles when M_0' was created. Similarly, each of the three triangles from M_1 intersecting M_0 was split into three smaller triangles.

Before retesselating each triangle t_ϵ, the original edges of t_ϵ that intersect other triangles are split at the intersection points. This process is performed by sorting the intersection points along the edge based on their distance from one of the end vertices of the edge. Then, a planar graph G is created to represent the original non-intersecting edges of t_ϵ, the intersecting edges of t_ϵ split at the intersection points, and the edges generated by intersecting t_ϵ with the other mesh. Since this process is performed on the plane, t_ϵ is first projected onto a plane ($x = 0$, $y = 0$ or $z = 0$) that it is not nearly perpendicular to.

The retesselation of t_ϵ is performed using two strategies. First, if G contains only one connected component, then the algorithm presented in [13] is employed to extract the faces of G. In the set of faces from G, each face is triangulated using the ear-clipping algorithm [8], that has a time complexity quadratic in the number of vertices in the face. That time is acceptable because the expected size of a face is small. If it were a problem, more efficient polygon triangulation algorithms exist, using as little as linear time in the face size, at the cost of considerable complexity.

Second, if G contains multiple connected components, a trivial incremental algorithm is employed to triangulate G. A trivial implementation has a $\theta(n^4)$ complexity, where n is the number of vertices in G. The reasons why we decided to employ this simpler algorithm are two: the graphs should be relatively small in practice and, according to preliminary experiments, disconnected graphs happen rarely if compared to connected ones.

7.2.1 Implementation with orientation predicates. Assume t_ϵ is a triangle from mesh $M_{i\epsilon}$ and T is the set of triangles from mesh $M_{(1-i)\epsilon}$ intersecting t_ϵ. We will show how to retesselate t_ϵ employing only orientation predicates.

As mentioned before, during the retesselation the vertices are projected onto a plane ($x = 0$, $y = 0$ or $z = 0$) with which t_ϵ is non-perpendicular. For simplicity, unless otherwise noted all orientation operations will be performed using the projected vertices.

Splitting edges at intersection points. Let $e_\epsilon = a_\epsilon b_\epsilon$ be an edge of t_ϵ. In this step, the vertices generated from the intersection of e_ϵ with triangles from $M_{(1-i)\epsilon}$ are sorted based on their distance to a_ϵ. The required geometric predicate to perform this operation is a comparison predicate that verifies if a vertex is closer to a_ϵ than another vertex is.

Let $v_{1\epsilon}$ and $v_{2\epsilon}$ be two vertices generated from the intersection of e_ϵ, with respectively, $t_{1\epsilon}'$ and $t_{2\epsilon}'$ of mesh $M_{(1-i)\epsilon}$. Since both $v_{1\epsilon}$ and $v_{2\epsilon}$ are on e_ϵ, there would be a coincidence on the distance of these points to a_ϵ iff $v_{1\epsilon}$ and $v_{2\epsilon}$ coincide. But, they cannot coincide because otherwise the interior of $t_{1\epsilon}'$ and $t_{2\epsilon}'$ would intersect (which cannot happen since $t_{1\epsilon}'$ and $t_{2\epsilon}'$ are from the same mesh). The predicate to decide which vertex is closer to a_ϵ can be easily implemented by applying a 3D orientation to the non-projected (3D) vertices: $v_{1\epsilon}$ is closer to a_ϵ than $v_{2\epsilon}$ is iff $v_{1\epsilon}$ and a_ϵ are on the same side of $t_{2\epsilon}'$.

Figure 3 illustrates this. Since both a_ϵ and $v_{1\epsilon}$ are on the positive side of $t_{2\epsilon}' = d_\epsilon e_\epsilon f_\epsilon$, then $v_{1\epsilon}$ is closer to a_ϵ than $v_{2\epsilon}$ is.

Face extractions. The only geometric predicate required by the polygon extraction algorithm is the one that sorts pairs of edges by their polar angle around a shared vertex. Given two edges $e_{1\epsilon} = u_\epsilon v_\epsilon$ and $e_{2\epsilon} = u_\epsilon w_\epsilon$, if v_ϵ and w_ϵ are in the same quadrant (assuming that u_ϵ is the origin) then the polar angle of $e_{1\epsilon}$ is smaller than the polar angle of $e_{2\epsilon}$ iff the 2D orientation of u_ϵ, v_ϵ and w_ϵ is positive. If they are in different quadrants, then comparing their polar angle is trivial.

To determine in which quadrant a vertex v_ϵ is considering w_ϵ as the origin, it is necessary to evaluate the signum of each coordinate of the vector $w_\epsilon v_\epsilon$, which is equivalent to computing the 1D orientation of w_ϵ and v_ϵ for the corresponding coordinate.

During the retesselation of $t_\epsilon \in M_{i\epsilon}$ the edges $e_{1\epsilon} = u_\epsilon v_\epsilon$ or $e_{2\epsilon} = u_\epsilon w_\epsilon$ can be either one of the original edges of t_ϵ (possibly split) or one edge generated by the intersection of t_ϵ with a triangle t_ϵ' from $M_{(1-i)\epsilon}$.

If, say, $e_{1\epsilon}$ is one of the original edges of t_ϵ, then its polar angle cannot be equal to $e_{2\epsilon}$'s polar angle otherwise t_ϵ would be degenerate, or $e_{1\epsilon}$ and t_ϵ' would be co-planar. The first and second situations would happen if, respectively, $e_{2\epsilon}$ was an edge of t_ϵ or an edge generated from an intersection.

Now, suppose $e_{1\epsilon}$ and $e_{2\epsilon}$ are edges generated from intersections. Since both edges are on t_ϵ, they are generated from the intersection of t_ϵ with other triangles $t_{1\epsilon}'$ and $t_{2\epsilon}'$ of mesh $M_{(1-i)\epsilon}$. Since the interior of edges generated from intersections is always in the interior of the triangles that generated them, $e_{1\epsilon}$ and $e_{2\epsilon}$ cannot have the same polar angle otherwise the intersection of these two edges would have a common point and, thus, $t_{1\epsilon}'$ and $t_{2\epsilon}'$ would intersect in their interior (which cannot happen since both triangles are in the same mesh). Therefore, there will be no coincidence in the predicate to compare pairs of edges by their polar angle when the perturbed input is processed.

An attentive reader may notice that the symbolic perturbation may modify the result of the comparison predicate for a non-degenerate input: if before the perturbation an edge has polar angle exactly 0, then after the perturbation this edge may be in the first or forth quadrants. However, this modification does not affect the face extraction algorithm since the objective of sorting the edges is to extract the wedges and the algorithm works properly as long as the list is sorted in a cyclic order.

Ear clipping. The ear-clipping algorithm employs only two geometric predicates: one that verifies if a vertex is an ear (convex) and one that verifies if a vertex is outside of a triangle.

Again, these two operations can be performed using 2D orientations. Given two oriented edges $u_\epsilon v_\epsilon$ and $v_\epsilon w_\epsilon$ of a face, v_ϵ is convex iff $orientation(u_\epsilon, v_\epsilon, w_\epsilon)$ is positive. Also, given a triangle $t_\epsilon = a_\epsilon b_\epsilon c_\epsilon$, we can determine if v_ϵ is inside t_ϵ by evaluating the orientation of v_ϵ w.r.t. the edges $a_\epsilon b_\epsilon$, $b_\epsilon c_\epsilon$ and $c_\epsilon a_\epsilon$.

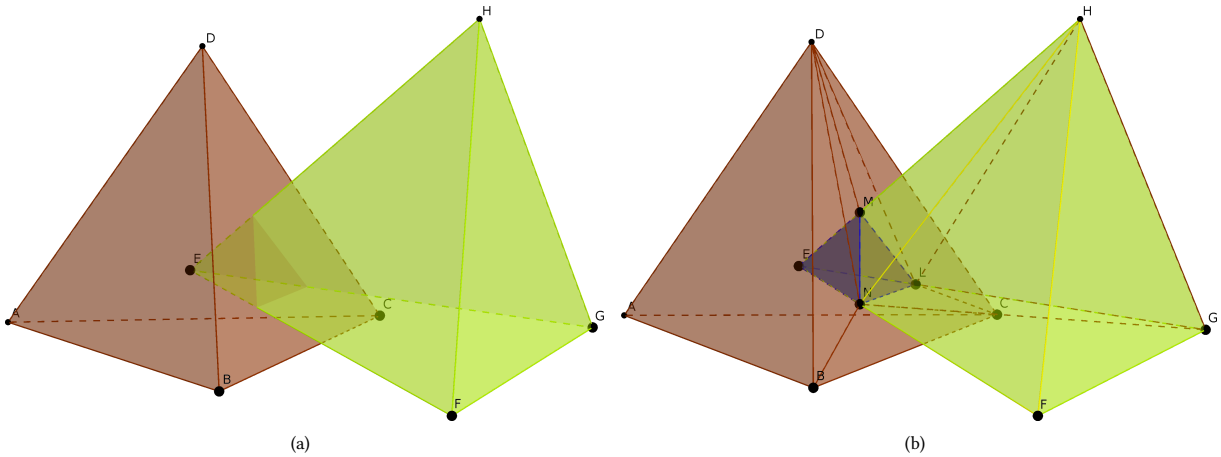


Figure 2: Computing the intersection of two tetrahedra. Source: [17]

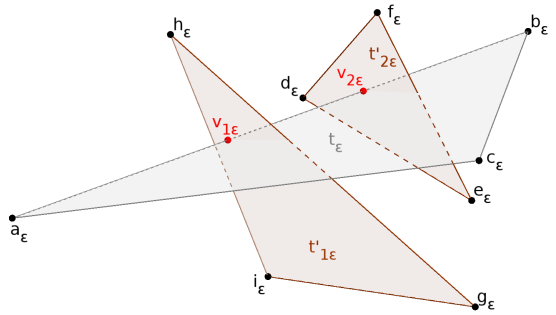


Figure 3: Sorting the vertices along an edge

During the ear-clipping some coincidences may happen even after the perturbation. Since all vertices of the same mesh are translated using the same perturbation, three vertices from the intersection u_ϵ, v_ϵ and w_ϵ may be collinear. Even though this coincidence may happen, they do not affect the ear-clipping algorithm since if a vertex v_ϵ and its two neighbors are collinear, v_ϵ will not be considered an ear. Also, the algorithm assumes the point in triangle algorithm always returns true if a vertex is on an edge.

One could argue that even these simple coincidences should be completely eliminated. However, we believe they do not negatively affect the algorithm since they happen only in a lower level predicate, and do not propagate to higher level functions. Furthermore, they cannot be completely removed without violating mathematical properties: for example, the point of intersection of an edge with a triangle will always be collinear with the endpoints of the edge.

Triangulating disconnected subdivisions. Disconnected subdivisions are triangulated using only one kind of geometric predicate, which verifies if any two edges $e_{1\epsilon} = u_\epsilon v_\epsilon$ and $e_{2\epsilon} = k_\epsilon w_\epsilon$ intersect other than at their endpoints.

Since all the vertices have unique coordinates, two edges will intersect at their endpoints iff they share a vertex. 2D orientation predicates can be employed to detect if $e_{1\epsilon}$ and $e_{2\epsilon}$ intersect at their interior. If $e_{1\epsilon}$ and $e_{2\epsilon}$ are collinear, an intersection in the interior of these edges can be detected by projecting them onto one of the Cartesian axes and verifying if the intervals defined by this projection intersect at their interior. A 1D orientation predicate can be employed to perform this verification.

7.3 Classifying triangles

After the intersections are detected and all the triangles that intersect other triangles are split at the intersection points, two new meshes $M'_{0\epsilon}$ and $M'_{1\epsilon}$ are created such that each new mesh $M'_{i\epsilon}$ will have the following two kinds of triangles:

- Triangles from the original mesh: if triangle t_ϵ from $M_{i\epsilon}$ did not intersect the interior of a triangle from the other mesh, then t_ϵ will be in $M'_{i\epsilon}$.
- New triangles: if triangle t_ϵ from $M_{i\epsilon}$ intersects the interior of one or more triangles from the other mesh, then t_ϵ will be partitioned into smaller triangles that will be inserted into $M'_{i\epsilon}$.

It is clear that each mesh $M'_{i\epsilon}$ will exactly represent the same regions that $M_{i\epsilon}$ represents. Thus, computing the intersection between $M'_{i\epsilon}$ and $M'_{(1-i)\epsilon}$ is equivalent to computing the intersection of $M_{i\epsilon}$ with $M_{(1-i)\epsilon}$. However, $M'_{i\epsilon}$ and $M'_{(1-i)\epsilon}$ are easier to process: since the triangles from one mesh intersect with the triangles of the other one only in common vertices or edges, then each triangle t_ϵ from $M'_{i\epsilon}$ will be completely inside a region from $M'_{(1-i)\epsilon}$. Suppose a triangle t_ϵ from $M'_{i\epsilon}$ bounds regions R_a and R_b and is completely inside region R_c from mesh $M'_{(1-i)\epsilon}$. When $M'_{i\epsilon}$ is intersected with $M'_{(1-i)\epsilon}$, t_ϵ will be in the resulting mesh and it will bound regions $R_a \cap R_c$ and $R_b \cap R_c$.

Therefore, the process of classifying the triangles to create the output mesh consists in processing each triangle t_ϵ from the meshes $M'_{i\epsilon}$ $(i = 0, 1)$, determining in what region of $M'_{(1-i)\epsilon}$ t_ϵ is and, then, updating the information about the regions t_ϵ bounds such that the resulting mesh is consistent.

If a triangle t_ϵ is in the exterior of the other mesh, in the resulting mesh the two regions t_ϵ bounds will be the exterior region and, thus, they can be ignored and not stored in the output mesh.

Figure 2(b) illustrates the classification step. All the intersections happen at common edges, and the only triangle from M'_0 that is completely inside region 2 (of M'_1) is triangle LMN. Since LMN bounds region 1 and the exterior region in M'_0, in the resulting intersection LMN will bound region $1 \cap 2$ and the exterior region. All the other triangles from M'_0 are in the exterior region of M'_1 and, thus, they will only bound the exterior region in the resulting

intersection (therefore, they will be ignored when the output mesh is computed). Similarly, in M'_0 the only triangles that are inside region 1 of M'_0 are triangles EMN, ELM and ELN.

The process of locating triangles of one mesh in the other one can be performed using a point location algorithm and a flood-fill algorithm. Suppose triangles of mesh $M'_{i\epsilon}$ are being located. If two adjacent triangles t_ϵ and t'_ϵ share an edge that was not generated by an intersection with $M'_{(1-i)\epsilon}$, than these triangles are in the same region of $M'_{(1-i)\epsilon}$. If t_ϵ and t'_ϵ share an edge that was generated by an intersection with triangle t''_ϵ of mesh $M'_{(1-i)\epsilon}$, then t_ϵ and t'_ϵ are in different regions of the other mesh. Since the regions t''_ϵ bound are known, it is possible to determine the location of t_ϵ and t'_ϵ once the location of at least one of these two triangles is known. For example, if t''_ϵ bounds region 0 on its positive side and 1 on the negative side and it is known that t_ϵ is in region 0 of $M'_{(1-i)\epsilon}$, then t'_ϵ can only be in region 1.

Thus, the location of all triangles in each connected component of triangles can be performed by locating one of the triangles as a seed and, then, using a traversal algorithm to locate the others. We use as seed a triangle containing an input vertex. Since the location of an input vertex is the same of the triangles containing it, the seed is located by locating one of its input vertices. This process is performed using the PinMesh [15] point location algorithm that, besides being able to perform queries in expected constant time, uses the same index we employ for indexing the triangles.

7.3.1 Implementation with orientation predicates. The triangles from one mesh are located in the other one by employing PinMesh to locate triangles with input vertices and, then, using a flood-fill algorithm to assign the location of the other triangles. Thus, we only have to show that PinMesh can be implemented using only orientation predicates.

PinMesh performs only 3 geometric operations [15]:

- $isOnProj(t_\epsilon, q_\epsilon)$: given a triangle t_ϵ and a query point q_ϵ, decide if the projection of q_ϵ onto the plane passing through t_ϵ is on the interior of t_ϵ.
- $isAbove(t_\epsilon, q_\epsilon)$: given a query point q_ϵ and a triangle t_ϵ such that $isOnProj(t_\epsilon, q_\epsilon)$ is true, decide if the projection of q_ϵ onto t_ϵ is above q_ϵ, i.e., the triangle is above the point.
- $isBelow(t_\epsilon, t'_\epsilon, q_\epsilon)$: given two triangles t_ϵ and t'_ϵ directly above a query point q_ϵ, decide if the z component of the projection of q_ϵ onto t_ϵ is smaller than the z component of the projection of q_ϵ onto t'_ϵ.

The first operation is the point in triangle test and, as mentioned earlier in the section on ear clipping, it can be implemented using three 2D orientation predicates. The operation $isAbove(t_\epsilon, q_\epsilon)$ can be implemented by deciding on which side of t_ϵ q_ϵ lies and verifying if t_ϵ's normal has a positive or negative z-component (which can computed by verifying if the 2D orientation of the 3 vertices of t_ϵ is positive considering t_ϵ is projected onto $z = 0$).

Finally, $isBelow(t_\epsilon, t'_\epsilon, q_\epsilon)$ can also be implemented using orientation: let v_ϵ be the vertex generated from the intersection of t_ϵ with a vertical edge passing through q_ϵ. $isBelow(t_\epsilon, t'_\epsilon, q_\epsilon)$ will be true iff the (3D) orientation of q_ϵ with respect to t'_ϵ is equal to the orientation of v_ϵ with respect to t'_ϵ (i.e., if both q_ϵ and v_ϵ are on the same side of t'_ϵ). This predicate can be implemented by creating the vertex v_ϵ as a dummy vertex from the intersection and applying the 3D orientation predicate.

Figure 1 also illustrates an analogous process in 2D: to determine if the segment uv intersects a vertical line passing through y at a lower point than uw intersects, a dummy vertex v' can be created as the intersection of uv with yx, where x is an arbitrary point above y. Since v' and y are on the negative side of uw, then v' is lower than w'.

PinMesh is employed to query an input vertex of one mesh against the other original mesh and, thus, only input vertices are processed. Also, PinMesh employs the same perturbation scheme employed in this paper and, thus, there will be no coincidence during the point location queries.

8 IMPLEMENTING SYMBOLIC PERTURBATION

Since all geometric operations can be implemented using only orientation predicates, the symbolic perturbation needs to be implemented only for these predicates. Because of the determinants' regularity, orientation predicates can be easily adapted to process perturbed points, Edelsbrunner [9].

However, there is a challenge in the mesh intersection problem: the predicates will have not only to handle input vertices (with real or rational coordinates), but also vertices generated from intersections. Since the coordinates of a vertex generated from an intersection is a function of five input points (two points defining an edge of one mesh and three points defining a triangle of the other mesh) and these points are perturbed, then the orientation has to be modified to handle these points.

As shown above in the section describing the mesh intersection algorithm, the 3D orientation will only be computed using, as arguments, three input vertices and another vertex that may be either an input vertex or a vertex from the intersection. Thus, at least two versions of the 3D orientation will have to be implemented.

For the 2D orientation, on the other hand, any of the three parameters may be either an input vertex or a vertex generated by an intersection. Thus, eight versions of the orientation predicate will be required. Since the orientation is computed using a determinant, the order of the parameters may be modified as long as the signum of the result is negated for each parameter swap. Then the number of functions actually written can be reduced by sorting the parameters by their type (input vertex or vertex from intersection).

During the evaluation of a predicate where at least one of the parameters p_ϵ is a vertex from the intersection, the coordinates of this vertex need to be computed. By definition, p_ϵ is represented by the points $t_{0\epsilon} = t_0 + (i\epsilon, i\epsilon^2, i\epsilon^3)$, $t_{1\epsilon} = t_1 + (i\epsilon, i\epsilon^2, i\epsilon^3)$ and $t_{2\epsilon} = t_2 + (i\epsilon, i\epsilon^2, i\epsilon^3)$ of the triangle t_ϵ and $e_{0\epsilon} = e_0 + ((1-i)\epsilon, (1-i)\epsilon^2, (1-i)\epsilon^3)$ and $e_{1\epsilon} = e_1 + ((1-i)\epsilon, (1-i)\epsilon^2, (1-i)\epsilon^3)$ of the edge e_ϵ, where i is the id (that may be either 0 or 1) of the mesh containing t_ϵ and p_ϵ is the intersection of t_ϵ with e_ϵ.

Each coordinate of p_ϵ will be an ϵ-expression with degree 3, where each coefficient is a function of the input vertices and of id. The coefficient of degree 0 represents the corresponding coordinate of the intersection point if the dataset was not perturbed.

Once the determinant to compute the signum of the orientation is evaluated, the resulting ϵ-expression will have maximum degree 6, where the coefficient of degree 0 represents the result that would

be obtained if the meshes were not perturbed. The signum of the determinant will be the signum of the non-vanishing coefficient of smaller degree, or 0 if all coefficients vanish.

Given an orientation predicate $orientation(p_{0\epsilon}, p_{1\epsilon}, ..., p_{d\epsilon})$, there are only two possibilities for the perturbation of each vertex $p_{j\epsilon}$. If $p_{j\epsilon}$ is an input vertex, then $p_{j\epsilon}$ will be either perturbed (translated) if it belongs to mesh 1 or it will not be translated if it belongs to mesh 0. If $p_{j\epsilon}$ is a vertex from the intersection, then either the edge that generated $p_{j\epsilon}$ or the triangle that generated $p_{j\epsilon}$ will be translated by $(\epsilon, \epsilon^2, \epsilon^3)$. Since the number of combinations is relatively small, we decided to generate a different predicate to handle each combination instead of evaluating the perturbations at runtime.

For the 2D orientation, for example, there will be eight possible combinations of perturbation schemes used in the three parameters. Furthermore, each of the three parameters may be either input vertices or vertices from the intersection, which results in eight combinations for the parameter types (since the parameters may be sorted, the actual number of combinations that have to be implemented is four). Finally, since the orientation 2D deals with 3D points projected onto one of the planes $x = 0$, $y = 0$ or $z = 0$, one version of the predicate will have to be implemented for each plane. Thus the total number of functions will be 96. For the 3D orientation, at least three of the parameters will be input vertices and, thus, the number of predicates that will be implemented is 32. Finally, in the 1D orientation the number of predicates will be 36.

One may argue that implementing 164 functions is a hard (and error-prone) task. However, all the functions are very regular and a program to generate them automatically can be implemented. Indeed, during the implementation of this mesh intersection algorithm a Wolfram Mathematica script was developed and the code for all the predicates was created automatically by the script.

9 EXPERIMENTS

3D-EPUG-Overlay was implemented in C++ and compiled using g++. Parallel programming was provided by OpenMP and multiple precision rational number were provided by GNU GMPXX. All the experiments were performed on a workstation with dual Intel Xeon E5-2687 processors, each with 8 physical cores, each core able to run 2 threads using the Intel Hyper-threading technology. The workstation has 128 GiB of RAM and runs Ubuntu Linux 16.04.

We evaluated 3D-EPUG-Overlay, by comparing it against three state of the art intersection algorithms: the exact and parallel method developed by Zhou et al. [21] and distributed in LibiGL, the exact algorithm for Nef Polyhedra available in CGAL, and the fast and parallel algorithm available in QuickCSG [7]. Even though QuickCSG is not an exact algorithm and does not process special cases [7], we compared 3D-EPUG-Overlay against it to verify how our exact algorithm compared with a fast approximate algorithm.

Experiments were performed with a variety of meshes downloaded from 3 datasets. All these meshes are non self-intersecting and watertight. Meshes whose names are numbers were downloaded from the Thingi10k dataset [22] (the number represents the id of the mesh). The ones with the sufix *kf* were obtained from the dataset provided by Barki [3]. All the other meshes were downloaded from the AIM@SHAPE-VISIONAIR Shape Repository [1]. Observe that some meshes are available in different datasets with different resolutions (e.g., there is a version of *Armadillo* with 331 thousand triangles and another one with 52 thousand).

Also, some of the meshes (the ones with sufix *tetra*) were tetrahedralized using GMSH. For example, *ArmadilloTetra* is the tetrahedralization of the *Armadillo* mesh.

Table 1 presents the running-times (in seconds, excluding I/O) of the algorithms during the processing of 17 pairs of meshes. Since the CGAL exact intersection algorithm deals with Nef Polyhedra, we also included the time it spent converting the triangulating meshes to this representation (it often takes more time to convert the dataset than to compute the intersection).

3D-EPUG-Overlay was up to 37 times faster than LibiGL. The only test cases where LibiGL was slightly faster than 3D-EPUG-Overlay was during the computation of the intersections of a mesh with itself. This can be explained because in this situation the intersecting triangles from the two meshes are never in general position and, thus, the computation have to frequently trigger the SoS version of the predicates (that were not optimized yet) in order to evaluate them considering the perturbed meshes. As future work we intend to optimize these functions in order to be faster even in unusual situations where the special cases happen frequently.

It is worth mentioning that, as mentioned in section 2, differently from other algorithms, LibiGL also repairs meshes (by resolving self-intersections) during the overlay computation. Since in the representation employed by 3D-EPUG-Overlay (where each triangle stores the ids of the two polyhedra it bounds) self-intersecting meshes are ambiguous, we do not attempt to repair them (as mentioned before, experiments were performed only with non self-intersecting meshes).

Because of the overhead associated with the Nef Polyhedra and since it is a sequential algorithm, CGAL was always the slowest one. When computing the intersections, 3D-EPUG-Overlay was up to 281 times faster than CGAL. The difference is much higher if the time CGAL spends converting the triangulated mesh to Nef Polyhedra is taken into consideration.

Except for the self-intersection experiments (where QuickCSG reported errors), QuickCSG was up to 3 times 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 a reasonable performance. As it will be mentioned later, QuickCSG also failed in situations where errors have not been reported.

Finally, to intersect meshes containing more than one material (objects with unique identification numbers), we 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, there are 78 thousand pairs of intersecting triangles, while 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. To the best of our knowledge, LibiGL, CGAL and QuickCSG were not designed to handle meshes with multi-material.

Besides being fast, 3D-EPUG-Overlay is also memory efficient. For example, during the intersection of the Neptune dataset with Nept.Translated it used 5GB of RAM, while LibiGL used 22.5 GB, CGAL used 110 GB and QuickCSG used 4.5 GB.

9.1 Comparing the results

To validate the accuracy of the outputs generated by 3D-EPUG-Overlay, we compared them with reference solutions obtained by LibiGL. This comparison was performed using the Hausdorff distance reported by the sampling tool Metro [6]. For all datasets Metro reported a distance 0 between LibiGL's results and ours.

We also employed tools such as MeshLab and CGAL to perform validations on the meshes (example: by checking if they are closed, properly oriented, etc.). The resulting meshes passed all validation tests.

As mentioned before, QuickCSG reported failures during the intersection of several meshes. Furthermore, even during some intersections where errors have not been reported the output results were frequently inconsistent, presenting open meshes, inconsistent orientations, etc. Others researchers have reported similar errors in QuickCSG and in other inexact algorithms such as the commercial package Maya [3, 21].

For example, Figure 4 presents a zoom in the output of QuickCSG for the intersection of the Ramesses dataset with Ramesses Translated. As can be seen, 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 employed by 3D-EPUG-Overlay (that are conceptual and use indeterminate infinitesimals), these numerical perturbations are not guaranteed to work and the user has to chose the maximum range for them. A too small range may not eliminate all errors while a too big range may modify the mesh too much. Figures 4 (b) and (c) and (d) display the result when perturbations 10^{-1}, 10^{-3} and 10^{-6} were employed. As it can be seen, none of these perturbations removed all errors and the bigger perturbation (10^{-1}) even created undesirable artifacts.

9.2 Other considerations

Even though all the computation is performed exactly, common file formats for 3D objects such as OFF represent data using floating-point numbers. The process of converting the rational output into floating-point numbers may introduce errors since not all rationals can be represented exactly as floating-point numbers. Approaches to solve this problem include avoiding the conversion (i.e., always employing multiple-precision rationals in the representations) or using heuristics such as the one presented by Zhou et. al. [21].

A limitation of the use of symbolic perturbation technique is that the results are consistent considering the perturbed dataset, not necessarily considering the original one. 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).

An interesting direction for future work is to develop an algorithm for cleaning the perturbed output dataset when the user does not want a perturbed output, generating an unperturbed mesh that is consistent with the non-degenerate features of the perturbed one. Another alternative is to process the output mesh using only algorithms that are aware of the symbolic perturbation (i.e., the pipeline of algorithms processing a dataset should consider the perturbed coordinates during the computations).

10 CONCLUSION AND FUTURE WORK

We have presented an algorithm to intersect a pair of 3D triangulated meshes. Except for the indexing, we showed that all the geometric functions employed by the algorithm can be expressed using 1D, 2D and 3D orientation predicates.

Implementing the symbolic perturbation is simplified since fewer lower level predicates are used, and only these predicates have to directly deal with the perturbation. Since the perturbations are not expected to modify the results of the functions when there is no coincidence, one can implement the higher level functions using possibly faster strategies and only employ the implementation using orientations when a coincidence is detected.

We showed that the symbolic perturbation eliminates the special cases and, thus, the algorithm can handle correctly all valid inputs. Since all the special cases are properly handled by the perturbation and all computation is exact, the algorithm can exactly compute the intersection of any valid input.

The algorithm was designed to parallelize very well. Since no global topology needs to be maintained, the individual triangles of the two meshes can be processed individually in parallel. The process is mostly a series of map-reduce operations. Therefore our implementation can build upon any of many existing, well constructed, parallel tools.

A preliminary version of this algorithm was implemented in C++ using OpenMP to parallelize the computation and GMPXX to provide exact arithmetic. 3D-EPUG-Overlay was up to 37 times faster than the parallel and exact algorithm available in LibiGL and up to 281 times faster than the exact algorithm available in CGAL. This is comparable to the performance of QuickCSG (a very fast, but inexact, algorithm).

This excellent performance allied with its robustness makes 3D-EPUG-Overlay suitable to be used as a subroutine in larger systems such as 3D GIS or CAD systems.

While we decided to focus on the problem of computing intersections, 3D-EPUG-Overlay can be trivially adapted to compute other kinds of overlays (such as union, difference, exclusive-or, etc). Indeed, the only required modification is in the classification step.

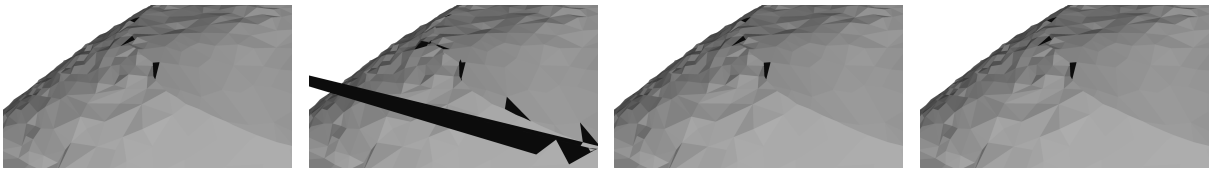
Future work includes developing an algorithm to consistently remove the symbolic perturbation of the outputs and improve the performance of the SoS version of the predicates.

ACKNOWLEDGMENTS

This research was partially supported by FAPEMIG, CAPES (Ciencia sem Fronteiras - grant 9085/13-0), CNPq, and NSF under grant IIS-1117277.

Table 1: Running-times (I/O not included) of the 4 algorithms to process the meshes. Column *Out* and *Inter. tests* present, respectively, the number of triangles in the output meshes and the number of intersection tests performed by 3D-EPUG-Overlay. QuickCSG reported errors during the intersections whose times are flagged with *.

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



(a) (b) (c) (d)

Figure 4: Detail of the intersection of Ramesses with Ramesses Translated generated by QuickCSG using different ranges for the numerical perturbation: no perturbation (a), 10^{-1} (b), 10^{-3} (c) and 10^{-6} (d)

REFERENCES

[1] AIM@SHAPE-VISIONAIR Shape Repository 2017. AIM@SHAPE-VISIONAIR Shape Repository. http://visionair.ge.imati.cnr.it// (accessed on Jun-2017). (2017).

[2] V. Akman, Wm. Randolph Franklin, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. 1989. Geometric Computing and the Uniform Grid Data Technique. *Comput. Aided Design* 21, 7 (1989), 410–420.

[3] Hichem Barki, Gael Guennebaud, and Sebti Foufou. 2015. Exact, robust, and efficient regularized Booleans on general 3D meshes. *Computers & Mathematics with Applications* 70, 6 (2015), 1235–1254.

[4] Gilbert Bernstein and Don Fussell. 2009. Fast, exact, linear booleans. *Eurographics Symposium on Geometry Processing* 28, 5 (2009), 1269–1278. https://doi.org/10.1111/j.1467-8659.2009.01504.x

[5] CGAL 2016. Cgal, Computational Geometry Algorithms Library. (2016). http://www.cgal.org (accessed on Apr-2017).

[6] Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. 1998. Metro: Measuring error on simplified surfaces. In *Computer Graphics Forum*, Vol. 17. Wiley Online Library, 167–174.

[7] Matthijs Douze, Jean-Sébastien Franco, and Bruno Raffin. 2015. *QuickCSG: Arbitrary and faster boolean combinations of n solids.* Ph.D. Dissertation. Inria-Research Centre Grenoble–Rhône-Alpes; INRIA.

[8] David Eberly. 2008. Triangulation by ear clipping. *Geometric Tools* (2008).

[9] Herbert Edelsbrunner and Ernst Peter Mücke. 1990. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics (TOG)* 9, 1 (1990), 66–104.

[10] F.R. Feito, C.J. Ogayar, R.J. Segura, and M.L. Rivero. 2013. Fast and accurate evaluation of regularized Boolean operations on triangulated solids. *Computer-Aided Design* 45, 3 (2013), 705 – 716. https://doi.org/10.1016/j.cad.2012.11.004

[11] Wm. Randolph Franklin, Narayanaswami Chandrasekhar, Mohan Kankanhalli, Manoj Seshan, and Varol Akman. 1988. Efficiency of uniform grids for intersection detection on serial and parallel machines. In *New Trends in Computer Graphics (Proc. Computer Graphics International'88)*, Nadia Magnenat-Thalmann and D. Thalmann (Eds.). Springer-Verlag, 288–297.

[12] Peter Hachenberger, Lutz Kettner, and Kurt Mehlhorn. 2007. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, optimized implementation and experiments. *Computational Geometry* 38, 1 (2007), 64–99.

[13] X.Y. Jiang and H. Bunke. 1993. An optimal algorithm for extracting the regions of a plane graph. *Pattern Recognition Letters* 14, 7 (1993), 553 – 558. https://doi.org/10.1016/0167-8655(93)90104-L

[14] Cyril Leconte, Hichem Barki, and Florent Dupont. 2010. Exact and Efficient Booleans for Polyhedra. Citeseer.

[15] Salles V.G. Magalhães, Marcus V.A. Andrade, W. Randolph Franklin, and Wenli Li. 2016. PinMesh - Fast and exact 3D point location queries using a uniform grid. *Computers & Graphics* 58 (2016), 1 – 11. https://doi.org/10.1016/j.cag.2016.05.017 Shape Modeling International 2016.

[16] Salles V. G. Magalhães, Marcus V. A. A. Andrade, W. Randolph Franklin, and Wenli Li. 2015. Fast exact parallel map overlay using a two-level uniform grid. In *Proc. of the 4th ACM Bigspatial (BigSpatial '15)*. ACM, New York, NY, USA.

[17] Salles V. G. Magalhães, Marcus V. A. A. Andrade, W. Randolph Franklin, Wenli Li, and Maurıcio G. Gruppi. 2016. Exact intersection of 3D geometric models. (2016), 44 – 55.

[18] Rafael J Segura and Francisco R Feito. 2001. Algorithms to test ray-triangle intersection. Comparative study. (2001).
[19] SFCGAL 2017. SFCGAL. (2017). http://www.sfcgal.org/ (accessed on Jun-2017).
[20] Chee-Keng Yap. 1988. *Symbolic treatment of geometric degeneracies*. Springer Berlin Heidelberg, Berlin, Heidelberg, 348–358. https://doi.org/10.1007/BFb0042803
[21] Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh Arrangements for Solid Geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016).
[22] Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).