Fast and Robust Mesh Arrangements using Floating-point Arithmetic

GIANMARCO CHERCHI, University of Cagliari, Italy
MARCO LIVESU, IMATI - CNR, Italy
RICCARDO SCATENI, University of Cagliari, Italy
MARCO ATTENE, IMATI - CNR, Italy

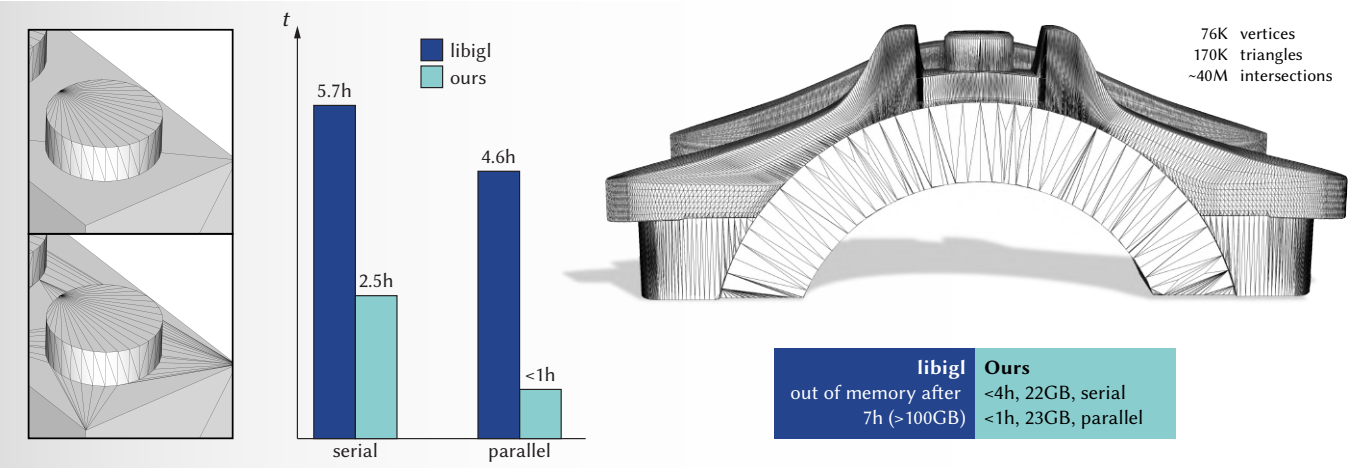


Fig. 1. We propose a novel method to robustly resolve mesh intersections (left). We can process the 4K meshes in Thingi10K [Zhou and Jacobson 2016] with at least one intersection at a fraction of the time required by prior methods, while better exploiting modern multi-core hardware (middle). Our method scales well on the most challenging model in the dataset, succeeding where previous methods fail due to excessive memory requirements (right). The model in the picture is excluded from the aggregated statistic due to the failure of libigl.

We introduce a novel algorithm to transform any generic set of triangles in 3D space into a well-formed simplicial complex. Intersecting elements in the input are correctly identified, subdivided, and connected to arrange a valid configuration, leading to a topologically sound partition of the space into piece-wise linear cells. Our approach does not require the exact coordinates of intersection points to calculate the resulting complex. We represent any intersection point as an unevaluated combination of input vertices. We then extend the recently introduced concept of *indirect predicates* [Attene 2020] to define all the necessary geometric tests that, by construction, are both exact and efficient since they fully exploit the floating-point hardware. This design makes our method robust and guaranteed correct, while being virtually as fast as non-robust floating-point based implementations. Compared with existing robust methods, our algorithm offers a number of advantages: it is much faster, has a better memory layout, scales well on extremely challenging models, and allows fully exploiting modern multi-core hardware with a parallel implementation. We thoroughly tested our method on thousands of meshes, concluding that it consistently outperforms prior art. We also demonstrate its usefulness in various applications, such as computing efficient mesh booleans, Minkowski sums, and volume meshes.

CCS Concepts: • **Computing methodologies** → **Mesh geometry models**; **Mesh models**.

Additional Key Words and Phrases: intersections, geometric predicates, mesh

1 INTRODUCTION

Modern Geometry Processing is increasingly calling for efficient and reliable tools for the realization of basic geometric operations. When compared with Computer Vision, where researchers and practitioners can count on a rich set of mature tools to process pixel matrices, Geometry Processing is still missing robust and efficient solutions, even for fundamental operations on triangle meshes.

In this paper we focus on a long-lasting problem in mesh processing: the robust handling of triangle intersections. Splitting mesh elements at their intersection points is at the basis of numerous higher level algorithms, such as the calculation of mesh booleans [Zhou

et al. 2016], sweepings of a 3D object along a guiding curve or surface [Hachenberger 2009], offsetting [Jung et al. 2003], and also the meshing of volumes whose bounding surfaces have defects [Hu et al. 2018]. All these methods share a basic need: partition the space into a set of topologically well formed cells, in order to unambiguously separate the interior from the exterior.

This partition into cells, or *mesh arrangement*, has been the subject of extensive research in previous years, and is notoriously regarded as a difficult yet computationally intensive problem, which often constitutes the biggest bottleneck for the aforementioned higher level algorithms.

The most critical part in the computation of a mesh arrangement is the representation of intersection points. To this end, we can classify the state of the art techniques in two broad classes: algorithms representing intersections exactly (i.e., using rational numbers to represent their coordinates) and algorithms that, on the contrary, rely on the standard floating-point encoding. Typically, algorithms in the former class give correctness and robustness guarantees at the expense of a significant loss of performances. In contrast, those in the latter category are efficient but may occasionally fail or produce inaccurate results. Despite some attempts have been made to speed up rationals (e.g. using lazy evaluation [The CGAL Project 2019]) , the performance gap with standard floats is still remarkable. Furthermore, these attempts do not allow for parallel implementations as the infrastructure needed to speed up the evaluation is not thread safe.

We propose the first method for mesh arrangements that is able to successfully combine the correctness guarantees secured by rational numbers with the efficiency typical of floating points. The core idea behind our technique is never to use coordinates of intersection points to make decisions during the algorithm. To do this, we classify all the possible intersections and represent them *implicitly* as suggested in [Attene 2020] (i.e., by storing references to the input primitives that generated them): their actual coordinates may be computed only during the last step of the pipeline, when there are no longer intersections. To guarantee accuracy and correctness, we define a comprehensive set of indirect predicates [Attene 2020] that can operate on all the possible implicit points in any configuration. Based on these basic tools, we introduce a mesh arrangement algorithm where crucial steps have been carefully designed to avoid the need of intersection point coordinates. The resulting mesh connectivity is provably the same as the one obtained by using rationals, but without the associated slowdown. As a bonus, our implicit intersection points and associated predicates are thread safe, and allow for an efficient fully parallel implementation of geometric algorithms.

We tested our implementation against the most challenging dataset available in literature, which consists in the 4408 intersecting meshes in Thingi10K [Zhou and Jacobson 2016]. We compared our method against the implementation provided by libigl [Panozzo and Jacobson 2014], which is based on fast rational numbers with lazy evaluation [The CGAL Project 2019]. Results confirm that we consistently outperform libigl, also greatly limiting the amount of hardware resources for the most challenging models (Figure 1).

A serial implementation of our method proved to be even faster than the parallelized version of libigl. A thorough analysis of our results is presented in Section 6, where we also demonstrate a number of applications that exploit our tool.

2 RELATED WORKS

We organize our review of the state of the art by first reporting about existing methods which are explicitly designed to resolve intersections and calculate mesh arrangements, and then describing techniques that depend on these low-level operations to produce their own results. Furthermore, we observe that many algorithms of this kind (including ours) need to cope with a non-trivial use of floating point arithmetic. Hence, we also summarize the main existing approaches to guarantee that algorithms are numerically robust.

2.1 Mesh Arrangements

Resolving intersections and self-intersections in explicitly represented polygon meshes is notoriously difficult. In tessellated CAD models, where manifold patches intersect at their borders and an approximation is tolerated, intersections can be solved with a localized approach [Bischoff and Kobbelt 2005]. For raw digitized models, triangles can be assumed to be small and local re-triangulation can be employed to replace self-intersections with valid configurations [Attene 2010]. Alternatively, vertex-based geometry can be converted to a plane-based representation [Bernstein and Fussell 2009] to enable the use of robust geometric predicates. This conversion requires a non trivial repairing but, under certain conditions, repairing can be avoided [Campen and Kobbelt 2010a]. Unfortunately, to meet these conditions the input may require a tricky clipping of edges and triangles that can introduce an additional rounding and new intersections. That is why more recent approaches rely on exact arithmetic to achieve this goal [Hu et al. 2018; Zhou et al. 2016], but the price to pay is a significant loss of performances. To the best of our knowledge, [de Magalhães et al. 2020] is the only existing algorithm that provides guarantees while being sufficiently fast though, unfortunately, it requires its input models to be manifold and closed to provide such guarantees while computing boolean operations. The restrictions they put on the input domain dramatically reduces the number of pathological cases to be managed in the classification and triangulation steps, making the core of the algorithm more straightforward. For this reason, they can compute explicit coordinates during the triangulation step without losing performance.

2.2 Applications

Intersection resolution is a fundamental building block for advanced 3D modeling operations such as mesh booleans, Minkowski sums, offsetting, repairing, and volume meshing conforming to generic triangle sets.

CGAL [The CGAL Project 2019], probably the most commonly used library in Geometry Processing, supports boolean operations using *Nef polyhedra*. Explicit meshes may be converted to this specific volumetric representation to take advantage of these functionalities. Minkowski sums are similarly supported [Hachenberger 2009].

These approaches are robust and correct but slow due to the use of exact arithmetic. Since for boolean operations the intersection points are usually sparse and thus a minimal subset of the total number of vertices of the mesh, hybrid geometric kernels exhibit performance advantages [Attene 2017].

The plane-based representations introduced in [Sugihara and Iri 1990] and taken up in [Bernstein and Fussell 2009] and [Campen and Kobbelt 2010a] can be exploited to efficiently compute Minkowski sums and offset surfaces [Campen and Kobbelt 2010b]. If the input is closed and orientable, intersecting triangles can be classified depending on the boolean operation [Barki et al. 2015] to efficiently obtain the resulting mesh. When offsetting a mesh, if care is taken while tracking its outer surface, only a subset of the intersections need to be resolved [Jung et al. 2003].

Resolving intersections is crucial to repair raw meshes [Attene et al. 2013] and make them usable in broad application contexts. Approaches such as [Bernstein and Fussell 2009; Campen and Kobbelt 2010a] require to convert the mesh into other representations. This approach is not appropriate when surface attributes (e.g., textures or colors) must be preserved, and the split must operate directly on the triangles [Attene 2014]. The construction of a tetrahedral mesh starting from its unprocessed surface shell [Hu et al. 2019, 2018] is another problem that needs, first, to find and compute the possible intersections. A parallel line of recent work focuses on solving PDE-based geometry processing problems without addressing intersections at all. Sellan et al. [2019] allows to define smooth functions on arrangements of partially overlapping discrete volumes, and [Sawhney and Crane 2020] extended Monte Carlo rendering techniques to a variety of classical mesh processing tasks.

2.3 Numerical Approaches

Robust geometry processing often relies on geometric predicates that guarantee an exact program flow independently of round-off errors [Lévy 2016; Shewchuk 1997]. A typical predicate calculates the sign of an expression, typically a homogeneous polynomial (e.g., a determinant). Evaluating the expression using floating-point arithmetic may lead to an incorrect sign that, in turn, may quickly put an algorithm in an inconsistent state, cause infinite loops, or even lead to a crash [Li et al. 2005]. Arbitrary precision numbers [Fousse et al. 2007] solve the problem, but the slowdown is often unacceptable. Arithmetic filtering [Devillers and Pion 2003] is a more efficient alternative: the evaluation of the expression is in floating-point arithmetic but, along with it, an upper bound for the rounding error is computed. If the magnitude of the evaluated expression is larger than the error bound, its sign is correct. If not (i.e., the filter *fails*), the predicate is re-evaluated using arbitrary precision. If the failure rate is low enough, absolute precision rarely comes into play, and consequently, the slowdown is acceptable.

The error bound can be based on the expression only (static filtering), or it may use actual input variables (dynamic filtering). For static filters [Fortune and Van Wyk 1993], the error is pre-calculated, the run-time overhead is extremely low, but the failure rate is relatively high. Conversely, the error in dynamic filters [Brönnimann et al. 1998] is computed at each predicate call, leading to higher overhead and fewer failures. Semi-static filters [Meyer and Pion 2008] are a trade-off of the two approaches.

These techniques are correct as long as their input is exact. If the predicate input is affected by an error, guarantees are lost. Thus, if a predicate uses *intermediate constructions*, state of the art solutions rely on lazy exact evaluation [Pion and Fabri 2011]. Unfortunately, these solutions are far too slow when compared with floating-point implementations. For some algorithms, however, the construction itself can be embodied in the predicate's expression, and floating-point filtering can be used [Attene 2020]. Similarly to us, [Wang et al. 2020] exploits the indirect predicate concept, introducing two versions of the `orient3D` predicate where the first parameter is implicit and the other three are explicit. We do not make use of their predicates.

Building on the [Attene 2020] idea, in this paper we introduce the following novel contributions:

- a classification of the vertices in an arrangement based on their originating input elements (Sect. 4.1);
- a new set of *indirect* predicates that can operate on any vertex type and combination (Sect. 4.2, 4.3);
- an original mesh arrangement algorithm that fully exploits the new predicates outperforming state of the art both in speed and memory efficiency (Sect. 5).

3 PROBLEM STATEMENT AND POSITIONING

We consider a generic set of triangles T with no assumptions, and identify their *arrangement* in space, that is, a subdivision of the space in cells bounded by the input triangles. To explicitly represent the bounding surface of each cell we subdivide intersecting triangles, thus constructing new points to represent the intersections and connecting them to form the sub-triangles. An example is shown in Figure 2 . We achieve this result by first cleaning T from degenerate (null area) elements, and then resolving all the intersections on the remaining triangles, thus producing a modified set T'. Therefore, triangles in T' are not only geometrically coincident with T but also form a valid simplicial complex, meaning that they are either disjoint or connected through a shared sub-simplex (i.e., they have an edge or vertex in common). We can, eventually, compute the explicit geometry of each cell of the space partition by region growing on the simplicial complex generated in this way.

The most critical part of the computation of an arrangement is the representation of points of intersection. These points may not admit an exact floating-point representation. If this is the case, the simple solution of using the highest possible resolution representation, the `double` type, still introduces an approximation error that may put the algorithm in an inconsistent state, generating a partition of the space with a wrong topology, and even program crashes [Li et al. 2005]. State of the art approaches overcome this issue by using rational numbers, which are expressed as fractions between arbitrary length integer numbers and guarantee exact geometric tests. Doing computation with these numbers often requires to multiply them to compute a common denominator, generating a much bigger (and bit-wise longer) integer, and eventually leading to an explosion in the memory footprint and the amount of overhead necessary to perform even simple arithmetic operations.

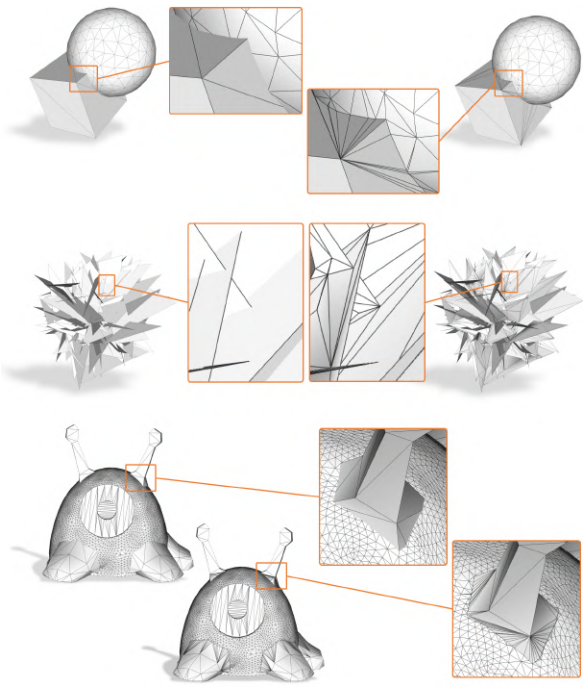


Fig. 2. The arrangement operation applied to some significant examples. Top to bottom: a simple case of a sphere intersecting a cube (110 intersections, 0.007 seconds); a randomly generated set of 100 intersecting triangles (5537 intersections, 1.06 seconds); the model 40509 of the Thingi10K [Zhou and Jacobson 2016] dataset (2712 intersections, 0.08 seconds).

Our main contribution is a novel method to robustly reconstruct the subdivided triangles T' without using the coordinates of intersection points. In our method, indeed, these points are implicitly represented in terms of the input geometric entities that generated them. Their relative positions, the program flow and the connectivity of T' are robustly determined by a novel set of exact geometric predicates. In other words, we can never lose or distort the information contained in the input set.

In Section 4, we first introduce our novel representation for implicit intersection points, as well as the set of geometric tests they support. In Section 5, we illustrate our pipeline for the computation of mesh arrangements.

4 REPRESENTATION AND PROCESSING OF IMPLICIT INTERSECTION POINTS

Our *implicit* intersection points keep the memory of the supporting planes and lines of the triangles and edges that generated them (Figure 3), and the vertices of the supporting entities are part of the input, hence exact. We can, thus, perform the computations avoiding any approximation error introduced by explicitly representing coordinates with floating-point numbers. At the same time, we can avoid paying the high computational cost of using arbitrarily precise rational numbers, as done in previous works [Attene 2017; Zhou et al. 2016].

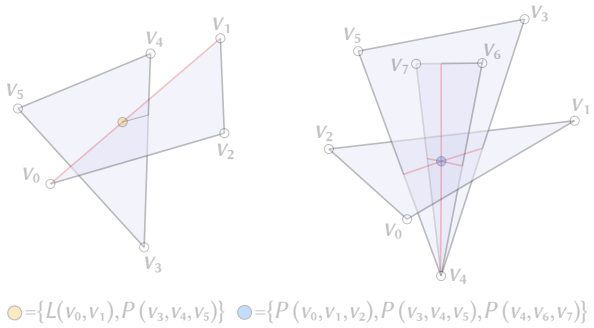


Fig. 3. On the left, five explicit points represent an implicit point at the intersection between two triangles (yellow dot): two defining the supporting line L of an edge (in red), and three defining the supporting plane P of the other triangle. On the right, a point at the intersection of three or more triangles (blue dot) requires nine explicit points, defining the supporting planes of three triangles.

But, using this strategy, we do not know the actual coordinates of the intersection points during the execution, and this implicit representation obliges us to define an entirely new framework inside which we can use these points for computation. In particular, creating a mesh arrangement requires the ability to cut each input polygon along its intersections with other polygons. Each polygon containing intersections becomes, via decomposition, a planar mesh where intersections (points and segments) are elements of the mesh (vertices and edges), and the union of all the faces is the original polygon.

Using an explicit representation, this is a relatively simple task. Still, in our framework, this requires to robustly discover the relative position of all the intersection points for existing mesh elements, and implicit points may describe the intersections and the involved mesh elements.

We devote this section to describe how to implement these core functionalities for sets of implicit points, as well as for hybrid sets made by both implicit and explicit (i.e., part of the input) points. This framework reprises most of the ideas on indirect predicates recently published by Attene [2020], which we extended as detailed in the remainder of this section.

4.1 Point representation

Our algorithm performs computations involving three different types of points, in terms of representation:

- **explicit** (i.e., input) points, for which exact floating-point coordinates are known a priori;
- points implicitly defined by the **intersection of two triangles**. We represent these points indirectly using **five** explicit points; two points to define the supporting line of one triangle edge, and three points to define the supporting plane of the other triangle (Figure 3, left);
- points implicitly defined by the **intersection of three or more triangles**. We represent these points indirectly with three triplets of explicit points, for a total of **nine**; each triplet

defines one of three linearly independent intersecting triangles (Figure 3, right);

Explicit points are the easiest to use for computation because they admit trivial (e.g., lexicographic) sorting, can be directly tested for coincidence, and are already supported by existing exact orientation predicates [Lévy 2016; Shewchuk 1997]. Conversely, these basic functionalities become tricky to implement for implicit points. How can we say that a point is lexicographically smaller than another point if we do not have their coordinates? Even a simple coincidence test becomes non trivial. Indeed, we note that the same point may be generated by intersecting different input elements, which means that its definition is not unique. E.g., for clusters of $n > 3$ triangles that intersect at a unique point, any possible triplet of non-coplanar triangles defines the same point, though with a different internal representation. Our key to implement these functionalities has been a set of novel geometric predicates that can operate on any combination of point types. Each of our predicates substitutes implicit point expressions within the expression of the predicate itself. Within this combined expression, all the variables are (exact) input values, and the predicate can always return an exact sign thanks to a clever multi-stage filtering as follows: first, all the calculations are done in floating point arithmetic. Along with the expression, a semi-static filter is also computed. If the magnitude of the evaluated expression is larger than the filter value, then its sign is guaranteed correct, and the process stops. If not, everything is recomputed using interval arithmetic. If the resulting interval does not contain the zero, the sign is correct and we stop. If not, we recompute everything using floating point expansions which always guarantee correctness. This guarantees absolute precision in the computation since every decision is based on the exact floating-point coordinates of the input points. It also guarantees efficiency because, for the large majority of cases, floating point calculations are precise enough and there is no need to switch to slower computation models.

The following subsections address orientation, sorting, and coincidence problems of implicit points.

4.2 Point orientation

The computation of a mesh arrangement spends most of the time performing two operations (Fig. 4(c)):

- triangle subdivision - intersection points are inserted to split an input triangle;
- conformality enforcement - the subdivided triangle is locally retriangulated so as to make intersection segments become (unions of) triangulation edges.

At their very core, the two operations are based on the same simple geometric test: the computation of the relative position, on a plane, of a point and a line. This operator is called `orient2d` and – for points having explicit floating-point coordinates – is usually available in any geometry processing kernel. Our threefold representation of points requires us to be able to reliably perform orientation tests with any possible combination of the points described in Section 4.1. No available geometric toolkit can do this.

Even though our points are embedded in $\mathbb{R}^3$, orienting a point about a given line is intrinsically a two-dimensional problem, and requires to express the coordinates of the points in a 2D frame. The rotation that brings the points in a canonical 2D frame (e.g., the xy plane) would require applying a rotation matrix that may introduce round-off errors, possibly poisoning the result of the orientation test. Absolute precision can be achieved by applying orthogonal projections, which only require to drop a coordinate and therefore do not introduce imprecision in the other two coordinates. Deciding which coordinate to drop is tricky because axis-aligned triangles may project to segments; hence, triplets of points that are not collinear in 3D may become collinear when projected in the 2D plane. In our application, we robustly compute the coordinate to drop by analyzing normal orientation.

Our algorithm considers one triangle t at a time and must resolve orientation predicates on its plane. Let t have vertices v_i, v_j, v_k, we consider the normal vector $\vec{n} = (v_j - v_i) \times (v_k - v_i)$. When calculating the orientation of three points on t, we just drop the coordinate associated to the biggest component between $|\vec{n}_x|$, $|\vec{n}_y|$, and $|\vec{n}_z|$. This guarantees that the triangle will not become degenerate under the orthogonal projection we defined. Computing $\vec{n}$ with standard floating-point operations may become unreliable for almost degenerate triangles. Even worse, the entries of $\vec{n}$ might not even be representable in the machine's floating-point system, making it practically impossible to compute it exactly. Luckily, in our algorithm we only need to detect its biggest component, which we can robustly do as described in the following subSection 4.2.1. Once we are able to project triangles in the 2D space, we just need to generalize the `orient2d` operator to handle any possible combination of explicit and implicit points (Sec. 4.2.2).

4.2.1 Robust orthogonal projection. The vector $\vec{n}$ can be computed using floating point arithmetic but, if the triangle is nearly degenerate, the magnitude of its largest component $n_i \in \{|\vec{n}_x|, |\vec{n}_y|, |\vec{n}_z|\}$ might be smaller than the numerical error. In this case, we can no longer guarantee that such a component is non-zero as it should be.

We implemented a semi-static filter to make sure that n_i is far enough from zero. Such a filter, as calculated by [Attene 2020] for the expression of n_i, is $\varepsilon_n = 8.88395\,10^{-16}\delta^2$, where δ is the maximum magnitude among the nine coordinates of v_i, v_j and v_k. If n_i is larger than ε_n, we can safely use it to proceed. Otherwise, we recalculate it exactly using expansion arithmetic [Attene 2020].

One might argue that, instead of computing the triangle normal, we can simply drop one of the three coordinates, and then verify whether the so-projected triangle becomes degenerate (i.e. its three vertices become collinear). If so, the coordinate to be dropped must be changed. Despite simpler to implement, this method may easily create near-degenerate projections even for well shaped triangles, ultimately requiring more switches to expansion arithmetic, with a consequent slowdown.

4.2.2 Generalized 2D orientation. In the easiest case where all the three points are explicit, any standard `orient2d` predicate can be used (i.e. by selecting two of the three coordinates). If input points are both explicit and implicit, but implicit points are all of the same kind, indirect versions of the predicate can be implemented as described in [Attene 2020]. Herewith we need to mix explicit and different sorts of implicit points.

We have implemented three versions of the `orient2d` predicate, one for each of the three possible orthogonal projections (XY, YZ,

ZX). Any such predicate is basically a wrapper that, first, determines the type of the three input points according to the classification given in Section 4.1 and, second, evaluates a proper expression depending on these types. Any input point can be Explicit (E), a Line-plane intersection (L), or an intersection of Three independent planes (T), which leads to a total of 27 possible combinations (EEE, EEL, ELE, LET, ...) of input points. Fortunately, the permutation of input points has a predictable outcome (e.g. `orient2d(a,b,c) = -orient2d(c,b,a)`), which reduces the number of necessary expressions to nine, plus the standard `orient2d` representing the EEE case.

Herewith, we describe how the expression for `orient2d_XY_LTE` can be derived. We refer to Appendix A for the other eight versions. The input parameters for this predicate are one Line-plane intersection (L), one intersection of Three independent planes (T), and one Explicit point (E). Points are projected on the xy plane.

To start with, we consider the line-plane intersection point $\mathbf{p}_L$ defined as the intersection of an edge $(\mathbf{q}_1, \mathbf{q}_2)$ and a triangle $(\mathbf{r}, \mathbf{s}, \mathbf{t})$ [Attene 2020]. Therefore:

$$\mathbf{p}_L = \left(\frac{\lambda_{Lx}}{d_L}, \frac{\lambda_{Ly}}{d_L}, \frac{\lambda_{Lz}}{d_L} \right)$$

where

$$d_L = \begin{vmatrix} \mathbf{q}_1 - \mathbf{q}_2 \\ \mathbf{s} - \mathbf{r} \\ \mathbf{t} - \mathbf{r} \end{vmatrix}, \quad n = \begin{vmatrix} \mathbf{q}_1 - \mathbf{r} \\ \mathbf{s} - \mathbf{r} \\ \mathbf{t} - \mathbf{r} \end{vmatrix}$$
$$\lambda_{Lx} = d_L q_{1x} + n q_{2x} - n q_{1x}$$
$$\lambda_{Ly} = d_L q_{1y} + n q_{2y} - n q_{1y}$$
$$\lambda_{Lz} = d_L q_{1z} + n q_{2z} - n q_{1z}$$

Note that $\mathbf{p}_L$ is undefined if $d_L = 0$, which happens if e and t are parallel or degenerate.

Let $\mathbf{p}_T$ be defined by the intersection of three triangles (v_1, v_2, v_3), (w_1, w_2, w_3), (u_1, u_2, u_3). Therefore:

$$\mathbf{p}_T = \left(\frac{\lambda_{Tx}}{d_T}, \frac{\lambda_{Ty}}{d_T}, \frac{\lambda_{Tz}}{d_T} \right), d_T = \begin{vmatrix} n_v \\ n_w \\ n_u \end{vmatrix}$$

where

$$\lambda_{Tx} = \begin{vmatrix} p_v & n_{vy} & n_{vz} \\ p_w & n_{wy} & n_{wz} \\ p_u & n_{uy} & n_{uz} \end{vmatrix}$$
$$\lambda_{Ty} = \begin{vmatrix} n_{vx} & p_v & n_{vz} \\ n_{wx} & p_w & n_{wz} \\ n_{ux} & p_u & n_{uz} \end{vmatrix}$$
$$\lambda_{Tz} = \begin{vmatrix} n_{vx} & n_{vy} & p_v \\ n_{wx} & n_{wy} & p_w \\ n_{ux} & n_{uy} & p_u \end{vmatrix}$$

$$p_v = \mathbf{n_v} \cdot \mathbf{v_1}, \; p_w = \mathbf{n_w} \cdot \mathbf{w_1}, \text{ and } p_u = \mathbf{n_u} \cdot \mathbf{u_1}.$$

and

$$n_v = (v_2 - v_1) \times (v_3 - v_2)$$
$$n_w = (w_2 - w_1) \times (w_3 - w_2)$$
$$n_u = (u_2 - u_1) \times (u_3 - u_2)$$

If $d_T = 0$, $\mathbf{p}_T$ is undefined.

The expression for `orient2d_XY_LTE`$(\mathbf{p}_L, \mathbf{p}_T, \mathbf{p}_E)$ can be obtained by substituting the expressions of $\mathbf{p}_L$ and $\mathbf{p}_T$ in the expression $(p_{Tx} - p_{Lx})(p_{Ey} - p_{Ly}) - (p_{Ty} - p_{Ly})(p_{Ex} - p_{Lx})$ of the standard `orient2d` predicate as follows:

$$\frac{(d_L\lambda_{Tx} - d_T\lambda_{Lx})(d_L p_{Ey} - \lambda_{Ly}) - (d_L\lambda_{Ty} - d_T\lambda_{Ly})(d_L p_{Ex} - \lambda_{Lx})}{d_L^2 d_T}$$

In this rewriting the predicate value is expressed as a fraction of two homogeneous polynomials on input coordinates. The sign of this fraction can be exactly calculated as described in [Attene 2020] using the following semi-static filters for floating point calculation of d_L, d_T, and for the numerator Δ:

$$\varepsilon_{dL} = 4.884981308350689\,10^{-15}\delta_L^3$$
$$\delta_L = \max\{|q_{1x}|, |q_{1y}|, |q_{1z}|, |q_{1x} - q_{2x}|, |q_{1y} - q_{2y}|, |q_{1z} - q_{2z}|, |s_x - r_x|, |s_y - r_y|, |s_z - r_z|, |t_x - r_x|, |t_y - r_y|, |t_z - r_z|, |q_{1x} - r_x|, |q_{1y} - r_y|, |q_{1z} - r_z|\}$$

$$\varepsilon_{dT} = 8.7041485130612 34\,10^{-14}\delta_T^6,$$
$$\delta_T = \max\{|v_{1x}|, |v_{1y}|, |v_{1z}|, |u_{1x}|, |u_{1y}|, |u_{1z}|, |w_{1x}|, |w_{1y}|, |w_{1z}|, |v_{3x} - v_{2x}|, |v_{3y} - v_{2y}|, |v_{3z} - v_{2z}|, |v_{2x} - v_{1x}|, |v_{2y} - v_{1y}|, |v_{2z} - v_{1z}|, |w_{3x} - w_{2x}|, |w_{3y} - w_{2y}|, |w_{3z} - w_{2z}|, |w_{2x} - w_{1x}|, |w_{2y} - w_{1y}|, |w_{2z} - w_{1z}|, |u_{3x} - u_{2x}|, |u_{3y} - u_{2y}|, |u_{3z} - u_{2z}|, |u_{2x} - u_{1x}|, |u_{2y} - u_{1y}|, |u_{2z} - u_{1z}|\}$$

$$\varepsilon_\Delta = 2.1849581172128 75\,10^{-10}\delta_\Delta^{14},$$
$$\delta_\Delta = \max\{|p_{Ex}|, |p_{Ey}|, \delta_L, \delta_T\}$$

Our predicate implementation calculates d_L, d_T and Δ in this order, while verifying that their magnitudes are all larger than their respective filters. If this is true, the sign of the fraction is determined by combining the signs of the numerator Δ and d_T (d_L may be ignored for this as it appears as a square in the denominator), and the result is guaranteed correct. Otherwise, as soon as one of these values is too close to zero, an ambiguity occurs and the floating point calculation is stopped. In this latter case, everything is recomputed using interval arithmetic and, if this is ambiguous too, we eventually revert to floating point expansions, which always guarantee correctness.

4.3 Point sorting

Given an edge $e(v_a, v_b)$, and the list of vertices that partition it in sub-segments $V(e) = \{v_0, v_1, ..., v_n\}$, if V is sorted such that $v_a < v_0 < v_1 < ... < v_n < v_b$, then the splitting process is much simpler and efficient. In fact, the first point splits edge e and, for any $i > 0$, the segment containing v_i is always (v_{i-1}, v_b), hence no explicit point in segment test must be performed to find it, as

would be required for an unsorted list of splitting points. We exploit point sorting to organize intersection points that are incident to input triangle edges. As for the `orient2d` case, sorting is trivial for explicit points, though less trivial for implicit points.

As we do for the generalized 2D orientation, we have implemented an indirect version of the `pointCompare(a,b)` predicate to determine if a point a is lexicographically smaller than, equal to, or larger than another point b. `pointCompare(a,b)` relies on the subsequent evaluation of `pointCompare_on_X()`, `pointCompare_on_Y()` and `pointCompare_on_Z()` predicates, which compare the first, second and third coordinates respectively. Herewith we describe how to implement `pointCompare_on_X_LT()`, while we point the reader to Appendix B for the other combinations.

`pointCompare_on_X_LT`$(p_L,\ p_T)$ returns the sign of $p_{Lx} - p_{Tx}$. With reference to Section 4.2.2, $p_{Lx} = \lambda_{Lx}/d_L$ and $p_{Tx} = \lambda_{Tx}/d_T$. The predicate expression can be written as:

$$\frac{\Delta}{d_L d_T} = \frac{d_T\lambda_{Lx} - d_L\lambda_{Tx}}{d_L d_T}$$

And the floating point filter for Δ is:

$$\varepsilon_\Delta = 4.3213800593466941\,10^{-12}\delta_\Delta^{10},$$
$$\delta_\Delta = \max\{\delta_L, \delta_T\}$$

5 MESH ARRANGEMENTS

After introducing the set of numerical tools to process mixed explicit and implicit points, we present the algorithm for the computation of a mesh arrangement. Our method is conceptually no different from previous approaches (e.g., [Attene 2014; Zhou et al. 2016]), but the implicit representation of intersection points necessitates a careful re-design of every single step.

Figure 4 illustrates our pipeline. In the first stage we remove zero area elements and detect pairwise triangle intersections. This step identifies all the new segments and the majority of the intersection points to be inserted in T to secure mesh conformity (Section 5.1). In the second step we process each triangle $t \in T$ separately, inserting all the previously identified intersection points (Section 5.2) and segments (Section 5.3). Conflicts between segments may reveal new intersection points involving more than two input triangles, and therefore not identified at the previous step. With this, the full set of intersection points is determined. Coplanar triangles may intersect not only at shared points or edges, but also share polygonal pockets, requiring special attention (Section 5.4). Finally, the last two (optional) steps of the pipeline comprise: converting the coordinates of the intersection points from implicit to explicit form for downstream applications (Section 5.6); and export an explicit piece-wise representation of the arrangement cells (Section 5.5).

5.1 Intersections: localization and assessment

The goal of the algorithm's first step is to detect, for each non-degenerate triangle $t \in T$, the list of triangles intersecting it, and generate the corresponding list of intersection points and segments. A triangle could potentially intersect all the other elements of T. Therefore, in the worst-case scenario the detection of intersections has quadratic complexity. In practice the number of intersections is often much smaller, and involve just a small amount of elements. However, in specific applicative scenarios (e.g., when converting a smooth CAD model to a piece wise linear mesh) the automatic tessellation can contain long and skinny triangles traversing a large number of mesh elements. These pathological cases could be very close to the worst-case scenario, and contain several millions of intersections. Our method scales well also on these pathological cases (Figure 1).

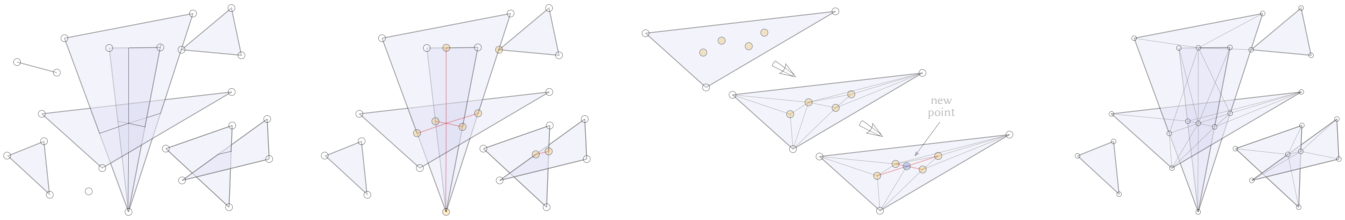
To speed up the detection process, we employ spatial data structures. We first fill a Kd Tree with all the non-degenerate input triangles. We start from the input bounding box and, at each step, consider the cell containing most triangles and split it in two equal halves along its longest direction. We stop when all the cells contain less than a given number of triangles (10K in our implementation) or when a maximum number of cells has been created (10K). For the sake of efficiency, we consider a triangle to be in a cell if its bounding box intersects the cell. At the end of this process, for each cell we perform an all-with-all triangle-triangle intersection check [Guigue and Devillers 2003]. At this early stage, all the points involved are explicit, therefore accurate intersection tests can be performed using standard orientation predicates [Shewchuk 1997].

Detection of intersection points and edges. As depicted in Figure 5, two intersecting triangles $t_i, t_j \in T$ may share a single point, a line segment, or a polygon. If t_i and t_j share three common vertices, they are coincident, and we keep only one of them. If they share a common edge, they can either form a valid simplicial complex or intersect at a shared polygonal pocket; if they share zero or one common vertex all types of intersections (point, edge, polygon) are possible. For each of these cases, we perform intersection tests on the sub simplices forming the triangles (points vs. segments, points vs. triangles, segments vs. segments, segments vs. triangles) and fill a map of the intersections. For each input triangle, the map contains the points and segments to insert into it to secure mesh conformity. Each newly generated intersection point is in implicit form as detailed in Section 4.1, and appended to the point list. Note that triangles may also intersect without defining any new point. For instance, this happens when one vertex of t_i lies inside t_j (or on one of its edges). In such a case, t_i requires no splitting, and we add the offending vertex of t_i in the list of elements that split either t_j or one of its edges. Triangles sharing a polygonal pocket are marked as coplanar and processed separately, as detailed in Section 5.4.

5.2 Adding intersection points

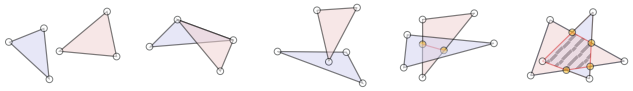
We now embed all the intersection points found at the previous step in the connectivity, by splitting the elements of the input set T. This process translates into breaking each triangle and each edge that contains at least one intersection point. There is a specific order to perform these operations that keeps the process safe. Splitting edges first would change the set of triangles, damaging the map between input triangles and intersection points.

Splitting triangles. Given a triangle $t \in T$ and an intersection point p strictly contained in it, we refine t by creating three sub-triangles formed by connecting each of its vertices with p. We iteratively split triangles containing multiple intersection points until all such



(a) Input (b) Detect all the intersections (points and edges) (c) Process each triangle separately inserting points first, then inserting edges (d) Output

Fig. 4. An illustrated summary of our pipeline: we start from a generic set of triangles, possibly containing disjoint, degenerate, and intersecting elements (a). We remove triangles with null area and process the remaining items pairwise, detecting intersection points (yellow) and segments (red) (b). We then process each triangle separately, adding intersection points first, and putting intersection segments in the tessellation afterward (c). Notice that conflicts between intersection segments may reveal new intersection points (blue) generated by more than two input triangles, that could not be detected in the previous step. The mesh is locally refined to accommodate the new element. We eventually recombine all sub-triangles, producing a valid intersection-free simplicial complex (d).



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

Fig. 5. Two triangles of the input set can: be entirely disjoint (a); form a valid simplicial complex (b); intersect at a single point (c); intersect at a segment (d); be coplanar and share a convex polygonal portion (e). The first two cases do not require processing. The latter three need to split some elements into sub-triangles to ensure mesh conformity. Newly generated points are yellow, intersection segments are red. Notice that for the (c,d,e) cases, we do not exhaustively report all the possible combinations. An intersection point can lie along an edge; an intersection segment can cross the perimeter of the triangle; a polygonal pocket can be a polygon with three to six vertices.

points are embedded in the simplicial complex. Note that while the first intersection point splits t, all the subsequent points split a sub-triangle of t, that we need to identify beforehand. We efficiently find the triangle containing each intersection point by keeping a tree data structure rooted at the input triangle. Each tree node has two or three children depending on the number of sub triangles: three if the parent splits at an interior point, two if it splits at a border point. Once we find a sub-triangle $t' \subset t$, we first test p against its edges. If an edge containing p is found, we split the two triangles $t', t'' \subset t$ incident to it, obtaining four new sub triangles and two new children in the data structure for both t' and t''. If p is strictly inside t', we split it into three sub-triangles, appending three children in the tree data structure. The point in triangle test is exact: we perform it checking the orientation of the splitting point p with respect to the three oriented edges forming the triangle t. If the sign is always ≥ 0 or ≤ 0, then p is either inside or at the perimeter of t. Note that while t has three explicit vertices, the triangles down in the tree have from zero to three implicit vertices. Therefore, to perform the point in triangle test we must use our novel orientation predicates defined in Section 4.2.

Splitting edges. If an intersection point p belongs to the interior of an input edge e, we split e at p and thus refine all its incident triangles by creating two sub-triangles for each. As for the triangle case, edges containing multiple intersection points raise the problem of finding – for each such point – the sub-edge $e' \subset e$ that contains it. We sort intersection points from one endpoint of e to the other, and process them in the resulting order. Doing so, we can optimally locate the containing edge for each splitting point, avoiding explicit tests. For details on the sorting procedure refer to Section 4.3.

5.3 Adding intersection segments

Adding all the intersection points in the complex does not yet guarantee mesh conformity. The endpoints of the segments defined by the intersection of two triangles are, until now, treated independently and may not be endpoints of an edge in the triangulation. In this step of the pipeline we transform intersection segments into mesh edges. Once inserted, these edges are marked as *constrained*, meaning that they cannot be removed from the mesh, otherwise conformity between intersecting elements will be lost again.

Given the endpoints v_i, v_j of an intersection segment to be added in the complex, the procedure for segment insertion works as follows (see the top line of Figure 6 for a pictorial illustration of the process): starting from v_i and following the mesh topology, we identify the sequence of edges $E = \{e_0, e_1, ..., e_n\}$ intersected by the segment (v_i, v_j). All the triangles incident to such edges are then removed, generating a void in the mesh. The edge (v_i, v_j) is then added to the mesh, halving the void in two polygons. Each polygon contains v_i, v_j and the endpoints of edges in E lying to the left (and to the right) of (v_i, v_j). The polygon may also contain some dangling edge, which must be included in the triangulation (Figure 8). This procedure is equivalent to the one presented in [Shewchuk and Brown 2015], and is guaranteed to work when the inserted segments do not intersect.

In our case, conflicts between intersection segments are possible, and correspond to configurations in the input where three or more triangles intersect at a common point. We improved the algorithm for segment insertion by handling the two possible cases depicted

in the bottom part of Figure 6. Namely, the case where the segment being inserted intersects a previously inserted segment at an inner point (bottom left), and the case where the intersection coincides with one endpoint (bottom right).

In the first case, we insert a new implicit vertex in the mesh, splitting the mesh edge containing it. Since each constrained edge is defined by the intersection of two triangles, their intersection arises at a point shared between three (or more) triangles (Figure 3, right). To create the point we attempt to find three linearly independent triangles that define the nine input points required to construct an implicit vertex. If we fail, we are in a coplanar case, and we proceed as described in Section 5.4. After the point insertion, both cases can be handled in the same way, which consists of re-calling the segment insertion procedure on the resulting sub-segments. Iterating this procedure allows handling any possible configuration in the input. We list the complete pseudo-code for segment insertion in Algorithm 1.

ALGORITHM 1: addSegment(v_{beg}, v_{end})

Input: the endpoints v_{beg}, v_{end} of a segment to be inserted in the mesh (the points are already part of the mesh)
Output: a re-triangulation of the portion of mesh intersected by (v_{beg}, v_{end}), having the input segment as edge

```
if (v_beg, v_end) is already a mesh edge then
    return;
end
P_l = {v_beg, v_end};
P_r = {v_end, v_beg};
v = v_beg;
t = a triangle incident to v_beg and intersected by (v_beg, v_end);
e = edge (e_0, e_1) opposite to v in t;
while v ≠ v_end do
    if e_0 ∈ (v_beg, v_end) then
        addSegment(v_beg,e_0);
        addSegment(e_0,v_end);
        return;
    end
    if e_1 ∈ (v_beg, v_end) then
        addSegment(v_beg,e_1);
        addSegment(e_1,v_end);
        return;
    end
    if e is an edge constrained then
        create new point v_new = e ∩ (v_beg, v_end);
        split edge e at point v_new;
        addSegment(v_beg,v_new);
        addSegment(v_new,v_end);
        return;
    else
        if e_0 is at the left of (v_beg, v_end) then
            append(P_l, e_0);
            append(P_r, e_1);
        else
            append(P_l, e_1);
            append(P_r, e_0);
        end
    end
    t = triangle opposite to t along e;
    v = vertex opposite to e in t;
    e = edge incident to v and t, and intersecting (v_beg, v_end);
end
triangulate(P_l);
triangulate(P_r);
return;
```

5.4 Coplanar triangles

Coplanar triangles pose additional challenges to our pipeline and require special handing, for two main reasons: (i) we cannot implicitly represent intersection points between two coplanar triangles using the supporting plane of one triangle and the supporting line of one edge of the other triangle, because all the points defining these entities are not linearly independent and therefore their intersection is undefined; (ii) triangles may intersect at a shared polygon, creating pockets with up to six sides. Splitting each triangle independently generates multiple tessellations for these shared pockets, which can be either identical or conflicting and thus require post-processing. (Figure 7).

Representation of intersection points: to obtain a proper implicit representation of intersection points between coplanar triangles, we use an auxiliary tetrahedron centered at the origin of the coordinate reference system. In the coplanar case intersections always occur along the perimeters of each triangle. Given two conflicting triangles and their intersecting edges, we represent the endpoints of these intersections with the extrema of one edge and a triangle obtained by pairing the second edge with one of the vertices of the auxiliary tetrahedron, so that the generated triangle is not coplanar with the first edge (Figure 9). Since the vertices of the tetrahedron are linearly independent, if the two edges are not collinear at least one such vertex is guaranteed to exist. Note that in the collinear case the intersection would occur at the original endpoints of the edges, therefore there would be no necessity to generate a new implicit vertex.

Tessellation of shared pockets: Since we process and refine every triangle separately, coplanar triangles that intersect at a shared pocket will produce multiple tessellations for the same geometry piece. If the triangulation of the pocket is the same in all its occurrences, duplicate triangles will be generated. Conversely, if each coplanar triangle will generate a different tessellation for the same pocket, conflicting triangles endowing new intersections will be generated. Since we cannot guarantee that the same polygonal pocket will always receive the same triangulation, we proceed as follows: we keep a global map of the pockets generated by coplanar triangles, which are uniquely identified by the sorted sequence of their corners. When a new coplanar triangle is processed, we identify its pockets and check for their existence on the map. If the pocket is not in the map, this is its first occurrence, we add its tessellation to T', and we insert the list of corners in the pocket map. If the map already contains the pocket, we already have its triangulation, and

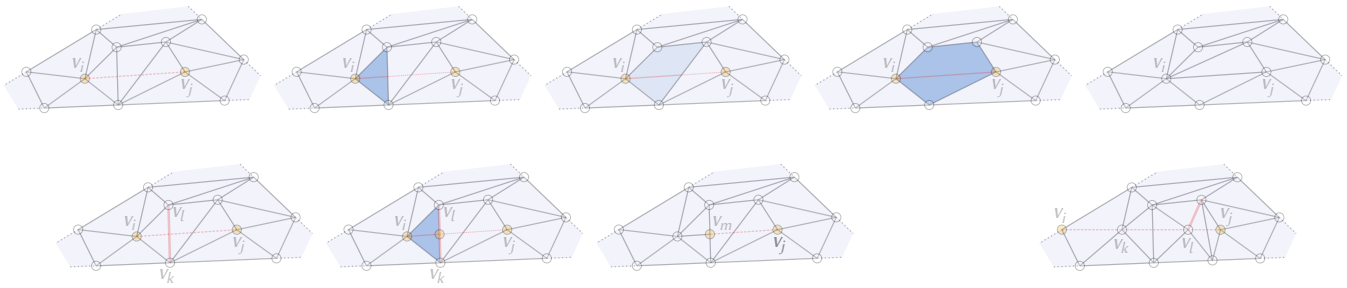


Fig. 6. Top line: insertion of the constrained segment (v_i, v_j). All the crossed triangles are progressively identified, starting from v_i and following the mesh connectivity. Their removal leaves a hole split into two halves by the new edge. We triangulate each half separately with simple earcut. Bottom line, on the left three figures: the segment to be inserted may intersect some pre-existing constrained edge. The intersection point between (v_i, v_j) and (v_k, v_l) (in red) could not be detected by testing triangles for intersection pairwise. We add the new point v_m to the mesh and then split (v_k, v_l) in (v_k, v_m) and (v_m, v_l). Afterward we triangulate the pocket traversed by (v_i, v_m), and we recall the segment insertion with (v_m, v_j). Bottom right figure: segment (v_i, v_j) intersects two vertices, one (v_l) belongs to a constrained edge (in blue), the other (v_k) does not. The insertion performs progressive split of (v_i, v_j) for any intersected vertex using the strategy explained before.

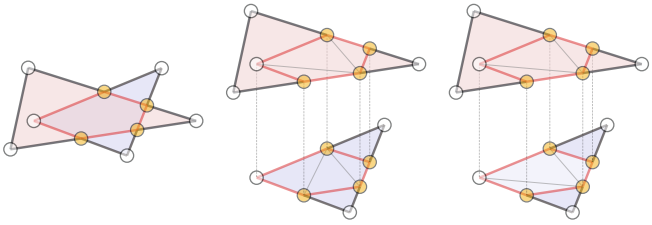


Fig. 7. Coplanar triangles may overlap at a polygonal pocket with up to six sides (left, red). Since we process each triangle separately, the tessellations of the same pocket may differ (middle). We keep track of coplanar pockets, and eventually, we use only one tessellation on all the triangles sharing the same pocket (right).

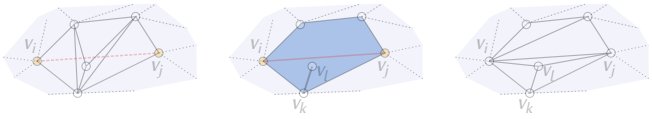


Fig. 8. The insertion of the segment (v_i, v_j) requires to remove five mesh triangles, leaving a dangling edge ,(v_k, v_l), that does not intersect it. The dangling edge may be a previously inserted constrained edge; therefore, we cannot remove it from the tessellation. We, thus, store the lower polygonal hole as the vertex chain $\{v_i, v_j, v_k, v_l, v_k\}$, repeating vertex v_k twice. The triangulator ignores the degenerate triangle v_k, v_l, v_k, producing a valid triangulation that preserves the edge (v_k, v_l).

therefore we discard the newly generated triangles (Figure 7). The detection of a pocket is performed with a simple region growing approach. Given an input triangle, we first insert all its intersection points and edges. Then, we start from each sub-triangle and expand to edge-adjacent triangles if the shared edge is not an intersection segment. All clusters with all boundary edges that are intersection segments identify a pocket, which we then test as described. Note that we apply this routine only to those triangles that were considered coplanar during the intersection detection (Section 5.1).

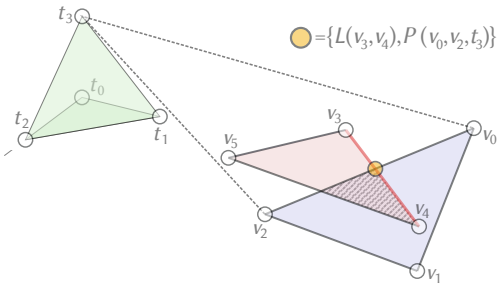


Fig. 9. Points defined by the intersection of coplanar triangles (yellow) cannot be implicitly defined using only triangle vertices, because they are not linearly independent. We define them using the supporting line of one of the edge (v_3, v_4), in red, and a plane passing through the endpoints of the other edge (v_0, v_2) and one vertex (t_3) of an auxiliary tetrahedron (in green). For any possible plane containing two intersecting triangles, there always exists at least one vertex of the tetrahedron not coplanar with them.

Since, in practice, coplanar intersecting triangles seldom occur, the overhead of this procedure is negligible.

5.5 Explicit arrangements

When all the self-intersections are fixed, we can calculate the arrangement cells by region growing following [Attene 2018]. Note that this requires to radially sort incident triangles around a non-manifold edge, which can be done by extending the `orient3d` predicate to implicit points as explained in [Attene 2020]. Alternatively, if an explicit volumetric representation is required, the simplicial complex can be passed to any standard tetrahedral meshing algorithm (e.g., [Si 2015]). Tetrahedra can eventually be clustered to form the arrangement cells.

5.6 Conversion to explicit coordinates

The coordinates of intersection points must be converted from implicit to explicit form to make our results available for downstream

applications. It is possible to do this precisely if the calling application supports rational numbers, but this is rare in practice. Performing this operation in floating-point while ensuring that there is no introduction of new degenerate elements or intersections is an extremely tricky – yet still open – problem, referred to as *snap rounding* [Devillers et al. 2018]. In practice, current implementations (including ours) use exact predicates to generate the combinatorial structure of the mesh and naively compute intersection points by solving small linear systems in double precision, finding the floating-point numbers that are closest to the precise intersection coordinates.

On our benchmark dataset with more than 4k models, naively snapping coordinates to double-precision leads to valid (intersection and degeneracy free) simplicial complexes in 85.2% of the cases. In case some error occurred, we share here a heuristic that fixes the problem in the vast majority of the cases. In short, we cast the so generated coordinates from double to single precision, and then run our algorithm again, resolving the newly generated intersections. The idea is that casting to a lesser precision number snaps offending points in double precision into the same point in single precision, which can then be converted again to a double-precision number after resolving the intersections. We applied this heuristic in all the cases where the naive rounding failed, and we managed to fix 96% of the models. If iterated, this heuristic shows success in 99.95% of the cases experimented in [Zhou et al. 2016].

6 RESULTS AND APPLICATIONS

We implemented our tool in C++, using ImatiSTL [Attene 2017] for intersection detection and CinoLib [Livesu 2019] for mesh processing. ImatiSTL was compiled in *Fast* mode to disable hybrid arithmetic which we do not use. To grant full reproducibility, upon acceptance we will release both the source code and the data of all our tests to the public domain.

Similarly to recent tools for robust geometry processing we demonstrate the capabilities of our algorithm on a notoriously difficult benchmark, which comprises all the meshes in Thingi10K [Zhou and Jacobson 2016] that contain at least one intersection. Overall, it amounts to 4408 models, some of which are well known for being extremely challenging, and for pushing the demand of CPU and memory resources way above the limits of commodity hardware (Figure 1). For the sake of a fair comparison, we have removed the model in Figure 1 from our dataset, that hence is made of 4407 models. For validation, we compared the meshes generated with our method with the output of libigl (counting the number of verties, edges and triangles), verifying that these numbers always coincide.

Prior to this work, the state of the art in the field can be considered to be the use of arbitrary precision arithmetic to explicitly represent intersection points. Specifically, we compared against the publicly available implementation contained in libigl [Panozzo and Jacobson 2014] (`igl::copyleft::cgal::remesh_self_intersections`), which uses CGAL [The CGAL Project 2019] to represent rational numbers. Note that while CGAL itself is just a wrap to an external library for the pure rational part, its internal infrastructure allows for a fast (*lazy*) evaluation, which essentially undertakes the computations using fast interval arithmetic and switches to the costly evaluation of the rational only when ambiguities occur. To the best of our knowledge, this infrastructure makes CGAL's rationals the most performing in the field of robust computing. On the negative sides, this infrastructure is not thread safe, which requires to encapsulate any piece of code that uses CGAL's numbers into a critical section, making the parallel code only slightly faster than its serial counterpart.

In a sense, CGAL's lazy evaluation is similar in spirit with what we propose in this work, but with two important differences: (i) in our case interval arithmetic is a second choice; our first computational model is pure floating point arithmetic, which is from three to eight times faster than interval arithmetic [Brönnimann et al. 1998]; (ii) we do not count on costly rationals as backup strategy, but rather squeeze the power of the floating point hardware a bit more by using expansions to evaluate predicates on our implicit points. Not only this makes us faster in serial mode, but also allows to better exploit modern multi-core hardware, evaluating predicates on our implicit points in parallel.

Our experimental setup is as follows: we compiled two versions of our algorithm and libigl's code, one single threaded and one parallel (libigl's code is natively parallel in the intersection tests. We edited that part in order to make also a serial version of it). We then batch processed all the 4407 models in our dataset with all four programs on a cluster equipped with 12 cores and 128GB or RAM, keeping trace of running times and memory consumption.

In Figure 10 we compare our running times with libigl. As far as the serial implementation is concerned, our method runs faster in 99.3% of the models. Considering the parallel code, we are faster in 94.7% of the models, also greatly reducing the overall running time, and processing the entire dataset in less than one hour. Interestingly, due to the limits in the parallelization of codes that use CGAL's numbers, our serial method results faster than parallel libigl in the 63.6% of the cases, also with a lower global running time across the whole dataset (2.5 hours against 4.6 hours). These numbers clearly indicate that our implicit constructions allow for a major speedup when compared with prior art, also scaling better on multi-core architectures. The figure also shows that in some cases libigl is faster than our method: in the serial-vs-serial experiment this happens on 31 small models requiring very short running times which may easily fluctuate from run to run due to external factors. There is one noticeable exception though, where the serial version of libigl is faster than our method (model ID: 101633 – 2350secs for us, 1350secs for libigl). In this case, nearly all of the 1.7M implicit points are intersections of three or more planes, which means that our segment insertion algorithm heavily enters recursion. Note that on the parallel-vs-parallel libigl does not improve, and we become faster on this model too (second line in Table 1). In the aforementioned table we restrict our analysis on the ten most challenging models in the dataset where both methods could converge (i.e., the ones with the highest number of intersections, excluding the bridge in Figure 1). Our method is faster at processing all such models, running in a fraction of their time (from 9% to 63%, average 18.1%), also exhibiting similar gains in the memory footprint (from 23.2% to 77.54% of their memory, on average 38.6%). Our ability to be more effective at handling the hard cases is further demonstrated in Figure 11, where we group models based on their running times. The first group of

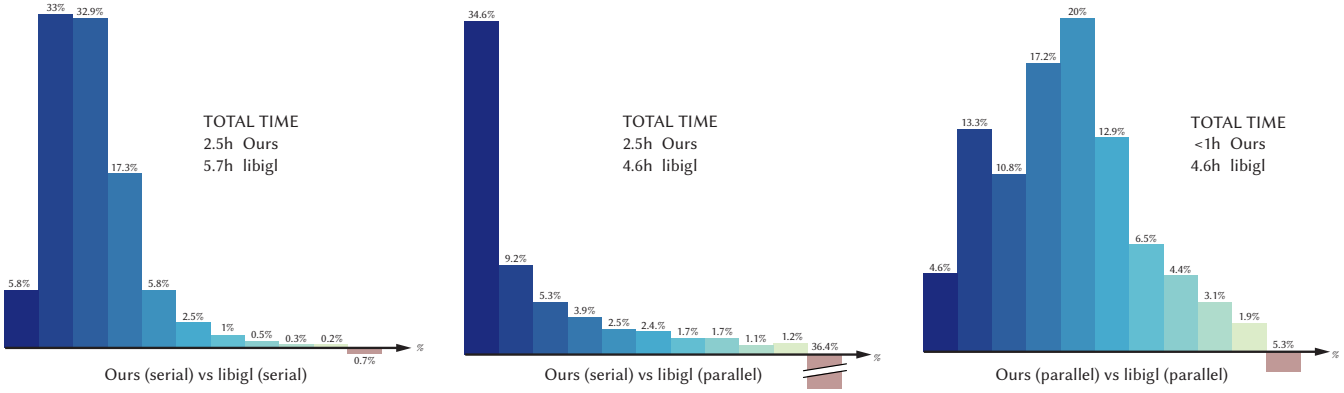


Fig. 10. Comparison with libigl on the 4K intersecting models present in Thingi10K [Zhou and Jacobson 2016], excluded the model shown in Figure 1. Statistics can be read as follows: the ten bars in shades of blue count the models in the dataset where our method run in a fraction of the time required by libigl (from left to right: (0%, 10%], (10%, 20%], . . . , (90%, 100%] of their running time). The negative bar reports on the number of times libigl was faster than us: with the exception of one pathological case, these negative bars collect mostly small models requiring very short running times. Our method runs faster in more than 99% of the cases in serial mode, and in more than 94% of the cases in parallel mode. Moreover, our serial implementation runs faster than parallel libigl in more than 60% of the cases.

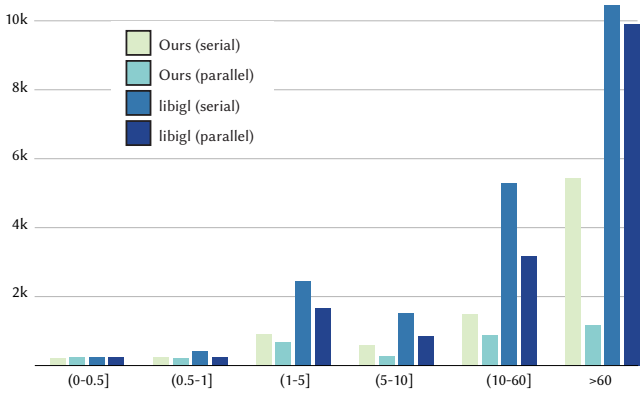


Fig. 11. Cumulative elapsed time w.r.t. the difficulty of the input. We measure difficulty of a model as the time spent by an algorithm to detect and resolve all its intersections. Based on this measure, we cluster our dataset in six subsets, each containing all the models whose processing time belongs to a specific range. Each group of four bars represents the total time spent by our reference algorithms to process the entire subset. All times are in seconds.

four bars represents the total time spent to process all the easiest models (i.e., those that terminate in less than 0.5 seconds each). The second group is the time spent for slightly harder models (i.e., those with running time in between 0.5 and 1 seconds), and so on. It is evident that, as input models become harder, our gain w.r.t. the state of the art becomes more significant.

Bottlenecks. In Figure 12 we show an aggregated statistic on the time spent by our algorithm in each step of the pipeline. These numbers refer to the serial implementation. As can be noticed, the algorithm spends most of its time (44.3%) adding intersection segments into the triangulation. The second most time consuming step is the generation of the Kd Tree to speed up the computation of intersections (24.2%), followed by the classification and generation of implicit intersection points (13.8%). Despite these numbers may suggest that the edge insertion is the bottleneck of our code, we found out that in 3914 cases out of 4407 the computation of the Kd Tree was the most time consuming step in the pipeline. In 320 cases the bottleneck was the segment insertion, and in the remaining 173 cases the point insertion. We conclude that for models having a moderate amount of intersections, the refinement of the geometry is practically inexpensive, and most of the computation is devoted to locate the actual intersections points. For challenging models exhibiting a huge amount of intersections, the actual refinement dominates.

Filtering. Filter values are calculated as in [Attene 2020], with FP rounding set to $+\infty$ to be conservative. We counted the number of calls of each of our predicates, and the number of filter failures. Failure rates are higher for predicates involving more implicit points because more arithmetic operations (and hence more rounding) are required for the evaluation. Nonetheless, these *more difficult* predicates are also more rarely called. E.g., for model #105693, `orient2d_EEE` is called 3102360 times, and the FP filter fails 71 times (0.002%), whereas `orient2d_TTT` is never called. On model #66112, `orient2d_TTT` is called twice, the FP filter fails both times, whereas the interval filter never fails.

6.1 Applications

Mesh arrangements are the basic ingredient for a number of shape processing tools. In this section we exploit our algorithm to showcase a subset of them. While most of these tools are often regarded as difficult to realize and time consuming to execute, their complexity is largely linked with the difficulties of resolving intersections between mesh elements. To this end, our fast and memory efficient computation of mesh arrangements directly impacts both the

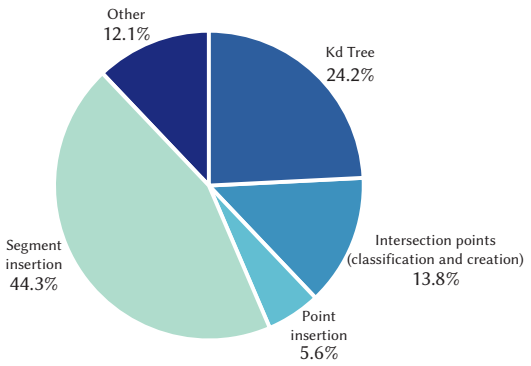


Fig. 12. Pie chart reporting how the total running time of our method distributes across the various steps of the pipeline. These data refer to the time spent processing our entire test dataset. Adding intersection segments into the triangulation is by far the most dominant step, followed at a distance by the construction of the spatial data structure that supports triangle-triangle intersection queries.

Table 1. We compare here running time and memory footprint for the parallel versions of our method and libigl [Panozzo and Jacobson 2014]. We consider the ten models in Thingi10k [Zhou and Jacobson 2016] with the highest number of intersections. For each model we report timing, memory footprint, and time and memory ratio (Ours/libigl). Times are in seconds, memory occupancy in Megabytes.

ID	Int.	Timing		Memory		Ratio	
		Ours	libigl	Ours	libigl	time	mem
252784	2,074,680	104.66	1,162.34	2,471.65	10,654.76	9.00%	23.20%
101633	1,712,644	868.46	1,378.00	1,947.55	6,408.16	63.02%	30.39%
55928	1,160,227	87.67	764.80	1,092.00	4,398.07	11.46%	24.83%
1368052	1,034,695	120.08	916.09	4,395.86	9,112.31	13.11%	48.24%
498461	463,958	18.68	157.37	568.86	2,266.13	11.87%	25.10%
338910	434,923	7.74	186.62	528.58	2,109.12	4.15%	25.06%
252785	403,159	24.25	219.81	519.88	1,932.96	11.03%	26.90%
498460	352,430	12.02	130.41	504.64	1,768.93	9.22%	28.53%
242236	239,831	49.96	206.31	1,137.13	1,466.49	24.22%	77.54%
242237	239,644	49.11	201.83	1,129.47	1,470.90	24.33%	76.79%

memory footprint and running times, broadening the use of these constructions on bigger and more complex datasets that were previously out of reach due to limits in the hardware resources or computational time.

Mesh booleans. Computing boolean operations between 3D shapes explicitly represented by a triangle mesh is useful in a variety of applications, including CSG, fabrication and design [Garg et al. 2016; Muntoni et al. 2018; Yao et al. 2017]. Given two watertight triangle meshes A, B, computing a boolean between them amounts to: (i) resolve all their intersections; (ii) determine what triangles of A are located inside B, and vice-versa; (iii) filter the triangles depending on the boolean of choice (e.g., $A \cup B$ is the set of all triangles of A that are external to B, plus all the triangles of B that are external to A). We implemented a simple boolean kernel that resolves (i) with our algorithm, and uses winding numbers for (ii). Note that since our input is a generic triangle soup, the algorithm naturally allows variadic boolean operations that simultaneously involve an arbitrary number of meshes [Zhou et al. 2016]. Furthermore, substituing classical winding numbers with their *generalized* version [Jacobson et al. 2013], also booleans between meshes that do not unambiguously enclose a solid are possible. Figure 14 shows various results obtained with our tool. In this simple example, our detection and solution of self-intersections required 0.23 seconds (the same task with libigl took 2.04 seconds).

Sweeping, Minkowski Sums. Providing an explicit representation of the volume occupied by an object being swept along a guiding path or surface is of interest in various fields of computer graphics and engineering, such as CNC machining, path planning, and collision detection, to name a few. In particular, the translational sweeping of a shape (*structural element*) on another is called *Minkowski sum*, whereas the translational and optionally rotational sweeping along a given path is simply called *sweeping*. In both cases, it is relatively easy to generate a superset of the boundary of the swept volume by replicating each mesh element and extruding it along the wanted direction, but the complexity of the so generated mesh and the amount of intersections it contains makes the generation of a clean boundary representation extremely hard. To demonstrate the capabilities of our algorithm we implemented a tool to perform Minkowski sums. We naively replicate each mesh element, filtering out part of the intersections as described in [Campen and Kobbelt 2010b], and resolving the remaining ones with our tool. We eventually extract the boundary using generalized winding numbers [Jacobson et al. 2013]. A result obtained with this pipeline is depicted in Figure 15, where our arrangement step takes 0.81 seconds, as opposed to the 2.37 seconds of libigl.

Tetrahedralization. Generating a volumetric mesh from a given boundary representation is a common need in applied sciences (e.g. to solve PDEs). The majority of the tools that generate a tetrahedralization of a given piece-wise linear complex (PLC) do not handle imperfections such as self-intersections or degenerate elements, which must be resolved in pre-processing [Si 2015]. Despite some recent tools have proven robust against imperfect inputs [Hu et al. 2018], these methods do not guarantee conformity w.r.t. the input complex, and may therefore completely miss semantic features that were incorporated in the elements of the PLC (Figure 13). Our method allows to resolve all possible intersections, generating a refined PLC which can be turned into a tetrahedral mesh that precisely conforms to all the input features, both geometric and semantic. In our current implementation we support only PLCs made of triangles. General poygonal facets could be trivially incorporated by triangulating them in pre-processing. Considering also isolated points and lines would require an extension of our algorithm to this class of inputs, which we plan to release in the near future. With our current implementation we spent 0.36 seconds to resolve intersections. The same step with libigl took 2.13 seconds.

Intersection detection. In certain situations it may be necessary just to check that a geometry is free of intersection, before even proceeding further with any computation. To this end, the first step

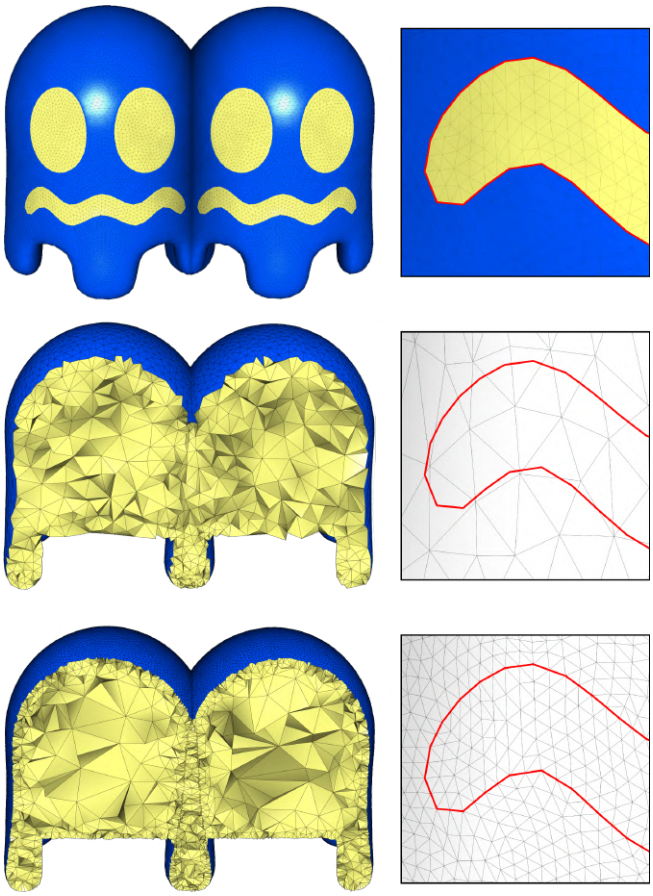


Fig. 13. Top: two copies of the ghost compenetrate to one another, generating a non valid PLC that cannot be turned into a tetrahedral mesh with tools like Tetgen [Si 2015]. Middle: recent methods for robust tetrahedralization allow to sidestep the resolution of intersections [Hu et al. 2018], but the result is non conforming and color features are lost (closeup). Bottom: resolving intersections with our method produces a valid PLC, which can then be successfully turned into a tetrahedral mesh, also preserving the boundaries of color features. Meshes of this kind are useful e.g. in fabrication, to design single colored assemblable components [Araújo et al. 2019].

of our pipeline offers an efficient yet exact tool to verify that a mesh does not self-intersect. We tested the remaining 5335 models in Thingi10k [Zhou and Jacobson 2016] not involved in the previous experiments, and we verified that they are intersection free. Processing all these models with the serial code required 12 minutes of computation. The parallel code took 9 minutes to complete. Since all these tests involve only the explicit coordinates of the input points, standard orientation predicates could be used. Therefore, using our tool or any previous tool for the computation of a mesh arrangement is equivalent.

7 CONCLUSIONS

We presented a novel algorithm for the robust and efficient computation of mesh arrangements. Previous methods had to sacrifice either robustness or performances, choosing to represent intersection points exactly (facing a major slowdown) or approximately (facing failures). Our method does not compromise, and is able to fully address both requirements thanks to a smart exploitation of the floating point hardware. Our major contribution is an implicit representation of intersection points, coupled with a set of exact predicates which allow to query them on a set of fundamental geometric tests. We demonstrated the capabilities of our method on a rich yet challenging dataset, showing that even a single threaded implementation of our algorithm outperforms a parallel implementation of the most efficient prior methods. To this end, we believe this work constitutes an important step forward in the field. To grant maximum adoption of our techniques and full reproducibility, we release both our predicates and mesh arrangement code to the public domain at `github.com/gcherchi/FastAndRobustMeshArrangements.git`.

7.1 Future works

We foresee a number of extensions for this work. First and foremost, we identify in the last step of the pipeline the weak ring of the chain. Indeed, when an implicit point is converted to a float, a possible numerical error may still produce locally flipped triangles. We observe that this limitation is common to all methods that do not represent intersection points directly with floats (i.e. all robust methods). Despite recent literature has proposed solutions to this problem [Devillers et al. 2018; Milenkovic and Sacks 2019], the complexity of the current algorithms is such that these methods are not usable in practice. More efficient algorithms need to be developed to secure end to end black box geometry processing with guarantees. Minor improvements to our technique regard: (i) the extension of our input data to a generic PLC containing also points, segments, and planar polygons; (ii) a constrained Delaunay triangulation of refined triangles, which would require an additional incircle predicate. Triangles are currently tessellated using trivial earcut, and may therefore be arbitrarily badly shaped. To this end, we conjecture that besides improving elements' quality, a CDT may also reduce the number of triangles intersecting a segment constraint, possibly further reducing our running times. Finally, we plan to further improve the parallel version of our code. We currently use simple OMP directives to parallelize intersection queries and triangle refinement. Despite less interesting from an academic point of view, we believe there is plenty of room for improvement, and a careful re-engineering of our code could greatly impact the performances of the algorithm.

ACKNOWLEDGMENTS

Gianmarco Cherchi gratefully acknowledges the support to his research by PON R&I 2014-2020 AIM1895943-1 (http://www.ponricerca.gov.it).

REFERENCES

Chrystiano Araújo, Daniela Cabiddu, Marco Attene, Marco Livesu, Nicholas Vining, and Alla Sheffer. 2019. Surface2Volume: Surface Segmentation Conforming Assemblable Volumetric Partition. *ACM Transaction on Graphics* 38, 4 (2019). https://doi.org/10.1145/3306346.3323004

Marco Attene. 2010. A lightweight approach to repair polygon meshes. *The Visual Computer* (2010), 1393–1406.

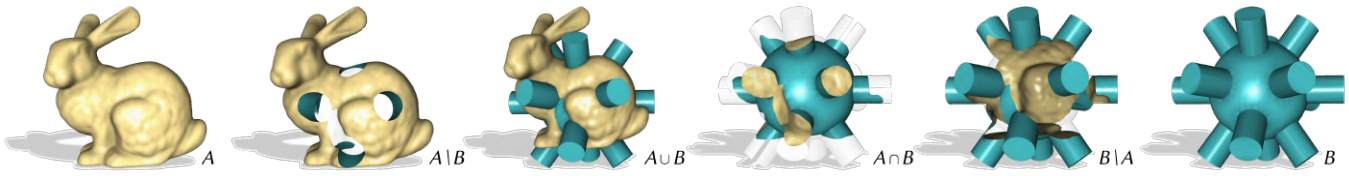


Fig. 14. Boolean operations between two watertight 3D meshes, obtained by first computing a mesh arrangement with our algorithm and then filtering triangles based on the winding number of their barycenter.

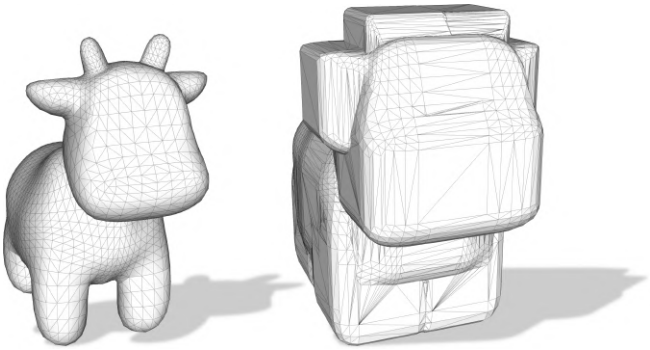


Fig. 15. Minkowski sum of the cow, obtained with a cubic structural element.

Marco Attene. 2014. Direct repair of self-intersecting meshes. *Graphical Models* 76, 6 (2014), 658–668.

Marco Attene. 2017. ImatiSTL - Fast and Reliable Mesh Processing with a Hybrid Kernel. *LNCS Transactions on Computational Science XXIX* (2017), 86–96.

Marco Attene. 2018. As-exact-as-possible repair of unprintable STL files. *Rapid Prototyping Journal* (2018).

Marco Attene. 2020. Indirect predicates for Geometric Constructions. *Computer-Aided Design* (2020). https://doi.org/10.1016/j.cad.2020.102856

Marco Attene, Marcel Campen, and Leif Kobbelt. 2013. Polygon mesh repairing: An application perspective. *ACM Computing Surveys (CSUR)* 45, 2 (2013), 15.

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.

Gilbert Bernstein and Don Fussell. 2009. Fast, Exact, Linear Booleans. In *Proceedings of the Symposium on Geometry Processing (SGP '09)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 1269–1278. http://dl.acm.org/citation.cfm?id=1735603.1735606

Stephan Bischoff and Leif Kobbelt. 2005. Structure Preserving CAD Model Repair. *Computer Graphics Forum* 24, 3 (2005), 527–536. https://doi.org/10.1111/j.1467-8659.2005.00878.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2005.00878.x

Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. 1998. Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry (SCG '98)*. ACM, New York, NY, USA, 165–174. https://doi.org/10.1145/276884.276903

Marcel Campen and Leif Kobbelt. 2010a. Exact and Robust (Self-)Intersections for Polygonal Meshes. *Computer Graphics Forum* 29, 2 (2010), 397–406. https://doi.org/10.1111/j.1467-8659.2009.01609.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01609.x

Marcel Campen and Leif Kobbelt. 2010b. Polygonal Boundary Evaluation of Minkowski Sums and Swept Volumes. *Computer Graphics Forum* 29, 5 (2010), 1613–1622. https://doi.org/10.1111/j.1467-8659.2010.01770.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01770.x

Salles Viana Gomes de Magalhães, W Randolph Franklin, and Marcus Vinícius Alvim Andrade. 2020. An Efficient and Exact Parallel Algorithm for Intersecting Large 3-D Triangular Meshes Using Arithmetic Filters. *Computer-Aided Design* 120 (2020), 102801.

Olivier Devillers, Sylvain Lazard, and William J. Lenhart. 2018. 3D Snap Rounding. In *34th International Symposium on Computational Geometry (SoCG 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Bettina Speckmann and Csaba D. Tóth (Eds.), Vol. 99. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 30:1–30:14. https://doi.org/10.4230/LIPIcs.SoCG.2018.30

Olivier Devillers and Sylvain Pion. 2003. Efficient exact geometric predicates for Delaunay triangulations. In *Procs. of 5th Workshop Algorithm Eng. Exper.* 37–44.

Steven Fortune and Christopher J. Van Wyk. 1993. Efficient Exact Arithmetic for Computational Geometry. In *Proceedings of the Ninth Annual Symposium on Computational Geometry (SCG '93)*. ACM, New York, NY, USA, 163–172. https://doi.org/10.1145/160985.161015

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

Akash Garg, Alec Jacobson, and Eitan Grinspun. 2016. Computational design of reconfigurables. *ACM Trans. Graph.* 35, 4 (2016), 90–1.

Philippe Guigue and Olivier Devillers. 2003. Fast and robust triangle-triangle overlap test using orientation predicates. *Journal of graphics tools* 8, 1 (2003), 25–32.

Peter Hachenberger. 2009. Exact Minkowksi Sums of Polyhedra and Exact and Efficient Decomposition of Polyhedra into Convex Pieces. *Algorithmica* 55, 2 (01 Oct 2009), 329–345. https://doi.org/10.1007/s00453-008-9219-6

Yixin Hu, Teseo Schneider, Bolun Wang, Denis Zorin, and Daniele Panozzo. 2019. Fast Tetrahedral Meshing in the Wild. *arXiv preprint arXiv:1908.03581* (2019).

Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral meshing in the wild. *ACM Trans. Graph.* 37, 4 (2018), 60–1.

Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–12.

Wonhyung Jung, Hayong Shin, and Byoung Kyu Choi. 2003. Self-intersection Removal in Triangular Mesh Offsetting.

Bruno Lévy. 2016. Robustness and efficiency of geometric programs: The Predicate Construction Kit (PCK). *Computer-Aided Design* 72 (2016), 3–12.

C. Li, S. Pion, and C.K. Yap. 2005. Recent progress in exact geometric computation. *The Journal of Logic and Algebraic Programming* 64, 1 (2005), 85 – 111. https://doi.org/10.1016/j.jlap.2004.07.006 Practical development of exact real number computation.

Marco Livesu. 2019. cinolib: a generic programming header only C++ library for processing polygonal and polyhedral meshes. *Transactions on Computational Science XXXIV* (2019). https://doi.org/10.1007/978-3-662-59958-7_4 https://github.com/mlivesu/cinolib/.

Andreas Meyer and Sylvain Pion. 2008. FPG: A code generator for fast and certified geometric predicates. In *Real Numbers and Computers*. 47–60.

Victor Milenkovic and Elisha Sacks. 2019. Geometric rounding and feature separation in meshes. *Computer-Aided Design* 108 (2019), 12–18.

Alessandro Muntoni, Marco Livesu, Riccardo Scateni, Alla Sheffer, and Daniele Panozzo. 2018. Axis-aligned height-field block decomposition of 3d shapes. *ACM Transactions on Graphics (TOG)* 37, 5 (2018), 1–15.

Daniele Panozzo and Alec Jacobson. 2014. *LIBIGL: A C++ library for geometry processing without a mesh data structure.* SGP 2014 Graduate School.

Sylvain Pion and Andreas Fabri. 2011. A generic lazy evaluation scheme for exact geometric computations. *Science of Computer Programming* 76, 4 (2011), 307 – 323. https://doi.org/10.1016/j.scico.2010.09.003 Special issue on library-centric software design (LCSD 2006).

Rohan Sawhney and Keenan Crane. 2020. Monte Carlo Geometry Processing: A Grid-Free Approach to PDE-Based Methods on Volumetric Domains. *ACM Trans. Graph.* 39, 4, Article 123 (July 2020), 18 pages. https://doi.org/10.1145/3386569.3392374

Silvia Sellán, Herng Yi Cheng, Yuming Ma, Mitchell Dembowski, and Alec Jacobson. 2019. Solid Geometry Processing on Deconstructed Domains. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 564–579.

Jonathan Richard Shewchuk. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry* 18, 3 (1997), 305–363.

Jonathan Richard Shewchuk and Brielin C Brown. 2015. Fast segment insertion and incremental construction of constrained Delaunay triangulations. *Computational Geometry* 48, 8 (2015), 554–574.

Hang Si. 2015. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)* 41, 2 (2015), 1–36.
K. Sugihara and M. Iri. 1990. A Solid Modelling System Free from Topological Inconsistency. *J. Inf. Process.* 12, 4 (April 1990), 380–393. https://doi.org/10.5555/81617.81622
The CGAL Project. 2019. *CGAL User and Reference Manual* (4.14.1 ed.). CGAL Editorial Board. https://doc.cgal.org/4.14.1/Manual/packages.html
Bolun Wang, Teseo Schneider, Yixin Hu, Marco Attene, and Daniele Panozzo. 2020. Exact and Efficient Polyhedral Envelope Containment Check. *ACM Trans. Graph.* 39, 4, Article 114 (July 2020), 14 pages. https://doi.org/10.1145/3386569.3392426
Jiaxian Yao, Danny M Kaufman, Yotam Gingold, and Maneesh Agrawala. 2017. Interactive design and stability analysis of decorative joinery for furniture. *ACM Transactions on Graphics (TOG)* 36, 2 (2017), 1–16.
Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 39.
Qingnan Zhou and Alec Jacobson. 2016. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797* (2016).

A ORIENT2D_XY

The expression for all the versions of the `orient2d_XY` predicate are reported in this appendix. Note that the `YZ` and `ZX` versions can be obtained by simply replacing all the subscripts accordingly. All the semi-static filters were calculated using [Attene 2020].

`orient2d_XY_EEE`$(p_1, p_2, p_3) = sign(\Delta)$

$\Delta = (p_{2x} - p_{1x})(p_{3y} - p_{1y}) - (p_{2y} - p_{1y})(p_{3x} - p_{1x})$

$\varepsilon_\Delta = 8.881784197001252\ 10^{-16}\delta_\Delta^2$

$\delta_\Delta = \max\{|p_{2x} - p_{1x}|, |p_{3y} - p_{1y}|, |p_{2y} - p_{1y}|, |p_{3x} - p_{1x}|\}$

`orient2d_XY_LEE`$(p_L, p_2, p_3) = sign(\Delta)sign(d_L)$

$\Delta = d_L(p_{2x}p_{3y} - p_{2y}p_{3x}) + \lambda_{Lx}(p_{2y} - p_{3y}) + \lambda_{Ly}(p_{3x} - p_{2x})$

$\varepsilon_\Delta = 4.75277369543781\ 10^{-14}\delta_\Delta^5$

$\delta_\Delta = \max\{|\delta_L|, |p_{2x}|, |p_{2y}|, |p_{3x}|, |p_{3y}|, |p_{2y} - p_{3y}|, |p_{3x} - p_{2x}|\}$

`orient2d_XY_LLE`$(p_{L1}, p_{L2}, p_3) = sign(\Delta)sign(d_{L2})$

$\Delta = (d_{L1}\lambda_{L2x} - d_{L2}\lambda_{L1x})(d_{L1}p_{3y} - \lambda_{L1y}) -$
$\quad (d_{L1}\lambda_{L2y} - d_{L2}\lambda_{L1y})(d_{L1}p_{3x} - \lambda_{L1x})$

$\varepsilon_\Delta = 1.699690735379461\ 10^{-11}\delta_\Delta^{11}$

$\delta_\Delta = \max\{|\delta_{L1}|, |\delta_{L2}|, |p_{3x}|, |p_{3y}|\}$

`orient2d_XY_LLL`$(p_{L1}, p_{L2}, p_{L3}) = sign(\Delta)sign(d_{L2})sign(d_{L3})$

$\Delta = (d_{L1}\lambda_{L2x} - d_{L2}\lambda_{L1x})(d_{L1}\lambda_{L3y} - d_{L3}\lambda_{L1y}) -$
$\quad (d_{L1}\lambda_{L2y} - d_{L2}\lambda_{L1y})(d_{L1}\lambda_{L3x} - d_{L3}\lambda_{1x})$

$\varepsilon_\Delta = 1.75634284893534\ 10^{-10}\delta_\Delta^{14}$

$\delta_\Delta = \max\{|\delta_{L1}|, |\delta_{L2}|, |\delta_{L3}|\}$

`orient2d_XY_LLT`$(p_{L1}, p_{L2}, p_T) = sign(\Delta)sign(d_{L2})sign(d_T)$

$\Delta = (d_{L1}\lambda_{L2x} - d_{L2}\lambda_{L1x})(d_{L1}\lambda_{Ty} - d_T\lambda_{L1y}) -$
$\quad (d_{L1}\lambda_{L2y} - d_{L2}\lambda_{L1y})(d_{L1}\lambda_{Tx} - d_T\lambda_{L1x})$

$\varepsilon_\Delta = 2.144556754402072\ 10^{-9}\delta_\Delta^{17}$

$\delta_\Delta = \max\{|\delta_{L1}|, |\delta_{L2}|, |\delta_T|, ||\}$

`orient2d_XY_LTE`$(p_L, p_T, p_3) = sign(\Delta)sign(d_T)$

$\Delta = (d_L\lambda_{Tx} - d_T\lambda_{Lx})(d_Lp_{3y} - \lambda_{Ly}) -$
$\quad (d_L\lambda_{Ty} - d_T\lambda_{Ly})(d_Lp_{3x} - \lambda_{Lx})$

$\varepsilon_\Delta = 2.184958117212875\ 10^{-10}\delta_\Delta^{14}$

$\delta_\Delta = \max\{|\delta_L|, |\delta_T|, |p_{3x}|, |p_{3y}|\}$

`orient2d_XY_LTT`$(p_L, p_{T1}, p_{T2}) = sign(\Delta)sign(d_{T1})sign(d_{T2})$

$\Delta = (d_L\lambda_{T1x} - d_{T1}\lambda_{Lx})(d_L\lambda_{T2y} - d_{T2}\lambda_{Ly}) -$
$\quad (d_L\lambda_{T1y} - d_{T1}\lambda_{Ly})(d_L\lambda_{T2x} - d_{T2}\lambda_{Lx})$

$\varepsilon_\Delta = 2.535681042914479\ 10^{-8}\delta_\Delta^{20}$

$\delta_\Delta = \max\{|\delta_L|, |\delta_{T1}|, |\delta_{T2}|\}$

`orient2d_XY_TEE`$(p_T, p_2, p_3) = sign(\Delta)sign(d_T)$

$\Delta = d_T(p_{2x}p_{3y} - p_{2y}p_{3x}) + \lambda_{Tx}(p_{2y} - p_{3y}) + \lambda_{Ty}(p_{3x} - p_{2x})$

$\varepsilon_\Delta = 9.061883188277186\ 10^{-13}\delta_\Delta^8$

$\delta_\Delta = \max\{|\delta_T|, |p_{2x}|, |p_{2y}|, |p_{3x}|, |p_{3y}|, |p_{2y} - p_{3y}|, |p_{3x} - p_{2x}|\}$

`orient2d_XY_TTE`$(p_{T1}, p_{T2}, p_3) = sign(\Delta)sign(d_{T2})$

$\Delta = (d_{T1}\lambda_{T2x} - d_{T2}\lambda_{T1x})(d_{T1}p_{3y} - \lambda_{T1y}) -$
$\quad (d_{T1}\lambda_{T2y} - d_{T2}\lambda_{T1y})(d_{T1}p_{3x} - \lambda_{T1x})$

$\varepsilon_\Delta = 3.307187945722513\ 10^{-8}\delta_\Delta^{20}$

$\delta_\Delta = \max\{|\delta_{T1}|, |\delta_{T2}|, |p_{3x}|, |p_{3y}|\}$

`orient2d_XY_TTT`$(p_{T1}, p_{T2}, p_{T3}) = sign(\Delta)sign(d_{T2})sign(d_{T3})$

$\Delta = (d_{T1}\lambda_{T2x} - d_{T2}\lambda_{T1x})(d_{T1}\lambda_{T3y} - d_{T3}\lambda_{T1y}) -$
$\quad (d_{T1}\lambda_{T2y} - d_{T2}\lambda_{T1y})(d_{T1}\lambda_{T3x} - d_{T3}\lambda_{T1x})$

$\varepsilon_\Delta = 3.103174776697444\ 10^{-6}\delta_\Delta^{26}$

$\delta_\Delta = \max\{|\delta_{T1}|, |\delta_{T2}|, |\delta_{T3}|\}$

B POINTCOMPARE_ON_X

The expression for all the versions of the `pointCompare_on_X` predicate are reported in this appendix. `pointCompare_on_X_EE` can be implemented without any explicit subtraction, and hence without the need for a filter. The `Y` and `Z` versions can be obtained by replacing the subscripts accordingly.

`pointCompare_on_X_LE`$(p_L, p_2) = sign(\Delta)sign(d_L)$

$\Delta = \lambda_{Lx} - p_{2x}d_L$

$\varepsilon_\Delta = 1.932297637868842\ 10^{-14}\delta_\Delta^4$

$\delta_\Delta = \max\{|\delta_L|, |p_{2x}|\}$

`pointCompare_on_X_LL`$(p_{L1}, p_{L2}) = sign(\Delta)sign(d_{L1})sign(d_{L2})$

$\Delta = d_{L2} * \lambda_{L1x} - d_{L1} * \lambda_{L2x}$

$\varepsilon_\Delta = 2.92288762637760\ 10^{-13}\delta_\Delta^7$

$\delta_\Delta = \max\{|\delta_{L1}|, |\delta_{L2}|\}$

`pointCompare_on_X_LT`$(p_L, p_T) = sign(\Delta)sign(d_L)sign(d_T)$

$\Delta = d_T * \lambda_{Lx} - d_L * \lambda_{Tx}$

$\varepsilon_\Delta = 4.321380059346694\ 10^{-12}\delta_\Delta^{10}$

$\delta_\Delta = \max\{|\delta_L|, |\delta_T|\}$

`pointCompare_on_X_TE`$(p_T, p_2) = sign(\Delta)sign(d_T)$

$\Delta = \lambda_{Tx} - p_{2x}d_T$

$\varepsilon_\Delta = 3.980270973924514\ 10^{-13}\delta_\Delta^7$

$\delta_\Delta = \max\{|\delta_T|, |p_{2x}|\}$

`pointCompare_on_X_LL`$(p_{T1}, p_{T2}) = sign(\Delta)sign(d_{T1})sign(d_{T2})$

$\Delta = d_{T2} * \lambda_{T1x} - d_{T1} * \lambda_{T2x}$

$\varepsilon_\Delta = 5.504141586953918\ 10^{-11}\delta_\Delta^{13}$

$\delta_\Delta = \max\{|\delta_{T1}|, |\delta_{T2}|\}$