Convex polyhedral meshing for robust solid modeling

LORENZO DIAZZI, IMATI - CNR, Italy
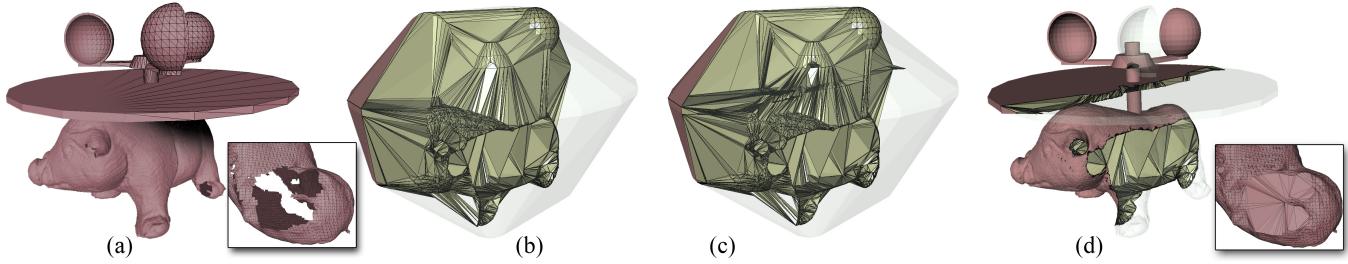MARCO ATTENE, IMATI - CNR, Italy

Fig. 1. The model on the left (a) is made of four intersecting components, with a total of 433K triangles that include both non-manifold configurations and several surface holes. In a few seconds, our algorithm transforms an initial tetrahedrization of the vertices (b) into a polyhedral mesh which is exactly *conformal* to the input surface (c). After having removed external cells, the boundary of the so-filtered polyhedral mesh is a closed oriented surface with no defects which replicates the input geometry while gently filling in missing or contradictory information (d). Note that our notion of *conformity* is purely geometrical; the input surface geometry is guaranteed to coincide with the geometry of a subset of the facets in the polyhedral mesh, though they may be triangulated differently.

We introduce a new technique to create a mesh of convex polyhedra representing the interior volume of a triangulated input surface. Our approach is particularly tolerant to defects in the input, which is allowed to self-intersect, to be non-manifold, disconnected, and to contain surface holes and gaps. We guarantee that the input surface is exactly represented as the union of polygonal facets of the output volume mesh. Thanks to our algorithm, traditionally *difficult* solid modeling operations such as mesh booleans and Minkowski sums become surprisingly robust and easy to implement, even if the input has defects. Our technique leverages on the recent concept of indirect geometric predicate to provide an unprecedented combination of guaranteed robustness and speed, thus enabling the practical implementation of robust though flexible solid modeling systems. We have extensively tested our method on all the 10000 models of the Thingi10k dataset, and concluded that no existing method provides comparable robustness, precision and performances.

Authors' addresses: Lorenzo Diazzi, IMATI - CNR, Italy, lorenzo.diazzi@ge.imati.cnr.it; Marco Attene, IMATI - CNR, Italy, marco.attene@ge.imati.cnr.it.

1 INTRODUCTION

Interactive 3D modeling using polygon meshes is an extremely powerful paradigm, especially as it allows an enormous freedom in the class of representable shapes. Polygon mesh processing is the natural enabling technology to disclose this freedom, though many limitations still exist that prevent a large-scale adoption of this paradigm for general 3D modeling. Polygon meshes may easily have defects [Attene et al. 2013], and advanced modeling operations such as booleans or Minkowski sums cannot be exploited unless their input unambiguously defines a solid. Even worse, existing algorithms that perform these operations on meshes may be guaranteed to produce the correct topology, but the exact coordinates of the resulting vertices need to be approximated using floating point numbers. The problem of approximating coordinates without introducing invalid configurations is known as *3D snap rounding* [Devillers et al. 2018] and, to the best of our knowledge, no algorithm is still known to resolve it in an acceptable amount of time. Stated differently, if input models have no defects, we are currently able to compute boolean operations efficiently and robustly, but we cannot guarantee that the result can be used as a new input due to possible defects. Though some recent methods sidestep the rounding problem thanks to a generic approximation [Hu et al. 2020], this approach cannot be used in interactive modeling systems where such approximations would accumulate.

In this paper, we propose a completely different approach: instead of trying to resolve the snap rounding problem or control the approximations, we simply relax the input requirements and allow our polygon meshes to have defects. To do this, we exploit the geometry of input polygons to define an explicit solid model represented as a mesh of convex polyhedra. If the input surface has no defects, the outer surface of our volume mesh will coincide with the input (they may be meshed differently, but their overall shape is exactly the same). In any other case, we guarantee that the input surface is

exactly represented as the union of polygonal facets of the volume mesh, and show that defects are compensated by gently interpolating missing or contradictory input information (Fig. 1). The crux of our contribution is in the technique we use to make this process both exact and efficient, though the underlying algorithm is very similar to the first phase of TetWild [Hu et al. 2018]. As in TetWild, we create our volume mesh by iteratively splitting Delaunay tetrahedra using the input constraint planes but, instead of exploiting rational numbers as in [Hu et al. 2018], we adopt the recent concept of indirect geometric predicate [Attene 2020] to robustly deal with intersection points. Indirect predicates are unconditionally robust, but they require all their parameters to be a direct expression of input values. No intermediate construction is allowed. In contrast, in a naively-designed BSP subdivision, intermediate vertices might be necessary to compute new vertices. In this paper, we show that any vertex at any time of our space partitioning can be represented as a direct combination of input coordinates, and describe an algorithm that cleverly forwards a minimal amount of information to construct these vertices. Once the space is completely partitioned, we show how to classify the polyhedral cells in internal and external by exploiting the input polygons.

Being particularly fast, our algorithm can be used to implement interactive mesh-based solid modeling systems featuring a wide spectrum of operations, including booleans and standard geometry processing algorithms. Also, thanks to our unconditional robustness, the result of each operation can be naively rounded and reused within the system for further modeling.

We quantitatively evaluate our algorithm on the entire Thingi10k dataset and compare it with alternatives (Section 4). The Thingi10k dataset is available for download at https://ten-thousand-models. appspot.com/. The reference implementation and data (other than Thingi10k) needed to reproduce the results in the paper are provided in the additional material and are released as an open-source project (https://github.com/MarcoAttene/VolumeMesher).

2 RELATED WORK

Our state of the art review describes existing methods that create volume meshes and classifies them based on both the approach and the features they exhibit. We also briefly discuss algorithms that directly tackle specific applications and clarify why all these techniques are not appropriate to implement interactive modeling systems.

2.1 Delaunay-based methods

When the input is well formed, the concept of constrained Delaunay tetrahedrization (CDT) provides a quite natural solution to the problem. Unfortunately, the CDT is not guaranteed to exist for arbitrary input triangles, even if they have no defects. In these cases, the input must be enriched with additional Steiner points that make the CDT exist. The widely-used software tool `tetgen` keeps the number of these points reasonably small [Si 2015]. `tetgen` is extremely efficient as it is purely based on floating point arithmetic and on fast geometric predicates pioneered by Shewchuk [Shewchuk 1997]. [Alexa 2020] shows that, in some cases, a set of weights exists so that the weighted Delaunay tetrahedrization is a superset of

the input triangles, and shows a heuristic approach to determine these weights. Hence, when the input admits a tetrahedrization, this approach can realize it without the need of Steiner points. Unfortunately, these methods cannot deal with input defects and cannot process degenerate or even *nearly* degenerate input.

2.2 Exact arithmetic

To make algorithms unconditionally robust, arbitrarily precise numbers (e.g., rational numbers) can be used at the cost of a general slowdown. NEF polyhedra are defined in CGAL [Hemmer et al. 2019] to represent the space enclosed by a set of polygons as an explicit volume. When rooted on exact arithmetic, this approach can be used to robustly compute booleans [Hachenberger et al. 2007] and Minkowski sums [Hachenberger 2009]. Thanks to a generalized Delaunay refinement technique [Shewchuk 1998], CGAL also offers a functionality to create high quality tetrahedral meshes out of input triangles that define a domain. Though being robust and exact, these approaches are unacceptably slow for use in interactive modeling systems.

2.3 Binary Space Partitioning

In an attempt to sidestep the speed issue, input triangles can be converted to their implicit plane equations, and fast and robust orientation predicates can be used on these equations to determine the space partition using floating point numbers [Bernstein and Fussell 2009]. Unfortunately, the conversion itself is not error free, and creating a mesh out of the resulting clipped planes involves a nontrivial repairing step (see Sect 2.4) with no guarantees. This was improved in [Campen and Kobbelt 2010b], where it is shown that the conversion is error free if input edges are short enough. A pre-clipping procedure is described to shorten edges appropriately, but no robustness guarantees are given for such a preprocessing itself. Instead of using them as temporary structures, BSPs can be maintained throughout the entire pipeline to implement iterated CSG operations with no intermediate approximations [Nehring-Wirxel et al. 2021]. Though guaranteeing robustness and supporting cascading, this method is not suitable for non-CSG operations, and no input defects are tolerated.

2.4 Mesh repairing

One possible approach to take advantage of the aforementioned fast Delaunay-based algorithms is to pre-process the input and turn it to a well-formed polyhedron. This process is known as *mesh repairing* and comes in two forms [Attene et al. 2013]: (1) global approaches, where the input is completely rebuilt out of an intermediate volumetric representation; (2) local approaches, where the input is kept unmodified in regions that do not exhibit defects. Global approaches are extremely robust but produce approximated results (e.g. [Ju 2004]), whereas local approaches may be precise (e.g. [Cherchi et al. 2020]) but their results may become invalid when coordinates are rounded to floating point values (see Sect 2.7). Local approaches may need to be complemented with hole filling [Zhao et al. 2007] or mesh completion [Podolak and Rusinkiewicz 2005] algorithms to turn the valid simplicial complex to a closed polyhedron.

2.5 Approximated methods

A new class of algorithms has been recently introduced to produce quality meshes for finite element analysis while guaranteeing an unconditional robustness for any kind of input. In [Hu et al. 2018] a hybrid coordinate representation is used: exact numbers are used to deal with intersections, whereas floating point numbers are sufficient for all the other parts of the algorithm. This is the first algorithm able to tetrahedrize with quality all the 10000 models within the thingi10k dataset [Zhou and Jacobson 2016]. Though the hybrid approach is an improvement wrt pure exact arithmetic, the algorithm is still far too slow for use in interactive applications. For that reason, a faster version was later invented to exploit floating point arithmetic [Hu et al. 2020] but, differently from the original version, no formal guarantees of success can be given. Both the versions produce a mesh which is close to the original only up to a user-defined approximation error ϵ, and lowering ϵ causes an increase in execution time. It is worth mentioning that the original `tetwild` [Hu et al. 2018] can produce an exact volume mesh if quality requirements are not necessary. In that case ϵ can be set to zero, but the running time remains unacceptable. In contrast, the fast version [Hu et al. 2020] is inherently approximated and cannot be used with $\epsilon = 0$.

2.6 Mesh booleans

Boolean composition is an extremely intuitive modeling paradigm, and big efforts have been spent to adapt it to polygon meshes. If the input is well-formed, exact arithmetic [Hachenberger et al. 2007] can be avoided by using coordinate-plane conversions [Campen and Kobbelt 2010a], and efficient algorithms exist that exploit parallel architectures [de Magalhães et al. 2020]. Thanks to a lightweight representation and exact rational arithmetic, [Barki et al. 2015] can handle a variety of near degeneracies but, still, it is not guaranteed to produce a valid output. In all these methods the result may have defects due to rounding, and hence cannot be used as a new input for further modeling. If the sequence of all the boolean operations is known in advance, and if no other *non boolean* operations are used inbetween, the variadic method proposed in [Zhou et al. 2016] can be used instead. This method can be used in combination with the notion of generalized winding number [Jacobson et al. 2013] to also deal with defects in the input but, in any case, it necessarily relies on slow exact arithmetic.

2.7 Snap rounding

Many of the algorithms described in this section are *exact*, in the sense that their results have exactly the expected topology. Some methods delay the calculation of coordinates to the very last phase [Cherchi et al. 2020], but this requires a rounding that may invalidate the embedding. Conversely, when the method uses exact arithmetic [Hachenberger et al. 2007], the geometry is also exact, but reusing this geometry as a new input causes an exponential growth in the size of the coordinate representation [Zhou et al. 2016] and a consequent slowdown. Thus, even in these cases, rounding to floating point values is still necessary in practice. The first snap rounding algorithm for 3D geometry was proposed by Fortune [Fortune 1999],
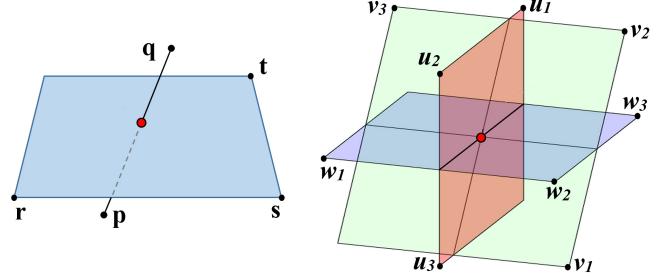

Fig. 2. Construction of an LPI point (left) and a TPI point (right).

but its requirements on the input make it impractical. A more practical solution was proposed later in [Milenkovic and Sacks 2019], though no guarantees are given. Recently, a 3D snap rounding algorithm for general input has been proposed in [Devillers et al. 2018]. To the best of our knowledge, this is the first and only working algorithm developed so far with guarantees, but its algorithmic complexity of $O(n^{19})$ makes it hardly useful in practice.

2.8 Indirect Predicates (background)

A geometric algorithm can be made robust either by using slow exact arithmetic or by simply guaranteeing that the program flow is *correct* while accepting a rounded output [Li et al. 2005]. The flow is correct if it is the same as if infinite precision was used. It is typically determined by *predicates* that, by analyzing the relative configuration of points, take one or the other branch.

Geometric algorithms can be classified based on the number of *construction layers* which are necessary to produce the input to predicates [Attene 2020]. In the easiest class, where predicates operate directly on input coordinates, filtering techniques and adaptive precision can be used to combine speed and robustness [Hemmer et al. 2019; Shewchuk 1997]. Delaunay triangulation of point sets belongs to this first class. Slightly more difficult problems involve one intermediate layer, that is, input to predicates includes points represented as unevaluated combinations of input coordinates. Predicates that accept these *implicit* point representations are called *indirect predicates* [Attene 2020] and can be efficiently evaluated using filtering techniques. When more than one layer is necessary, indirect predicates cannot be used.

In this paper, we show that our meshing problem can be formulated using one layer only, so that indirect predicates can be employed. Besides traditional explicit points represented as triplets of coordinates, we make use of two implicit point types called LPI (Line-Plane Intersection) and TPI (Three-Planes Intersection). An LPI point $l = \{p, q, r, s, t\}$ represents the intersection of a straight line by p, q and a plane by r, s, t, where p, q, r, s, t are all explicit points. A TPI point is made of three triplets of explicit points, each triplet representing a plane. Formal definitions of LPIs, TPIs, and indirect predicates operating on them are given in [Cherchi et al. 2020; Wang et al. 2020]. Examples are depicted in Fig. 2.

3 METHOD

Our algorithm considers an unstructured set of input triangles and computes a subdivision of the space in convex cells which do not
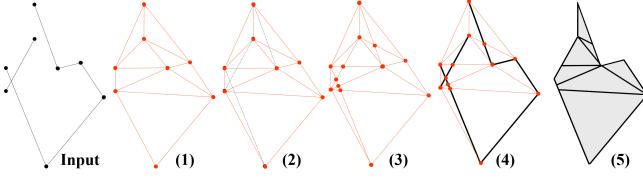
Fig. 3. Algorithm overview: (1) Delaunay mesh; (2) Constraint mapping; (3) Cell split; (4) Constrained facets; (5) Internal/external cell classification.
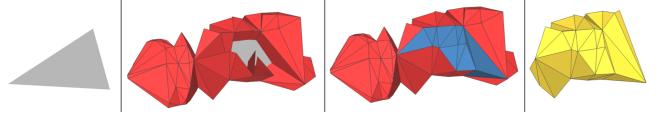

Fig. 4. From left to right: a constraint c, the set of tetrahedra that intersect its boundary ∂c (red), the set S of all tetrahedra intersecting c (blue tetrahedra intersect only the interior of c) and the subset of tetrahedra in S that improperly intersect c (yellow).

intersect any input triangle. The surface represented by input triangles coincides with a collection of polygonal facets in the output mesh, and our algorithm tags such facets so as to establish a correspondence with the input. Finally, such a tagging is exploited to classify cells in internal and external. No assumption is made on the input triangles. In the remainder of the paper, input triangles will be called *constraints* just to distinguish them from the possibly triangular facets of cells used to subdivide the space.

To determine the space subdivision, our algorithm is organized in five consecutive phases (see Fig. 3): (1) computing a tetrahedrization of the input constraint vertices (Sect. 3.2), (2) mapping each tetrahedron to the constraints that intersect its interior (Sect. 3.3), (3) iteratively splitting each tetrahedron into convex polyhedral cells by using the constraints in its map (Sect. 3.4), (4) identifying facets of the resulting polyhedral cells which belong to the input surface (Sect. 3.6) and (5) classify cells in internal and external (Sect. 3.7).

Notice that, similarly to TetWild [Hu et al. 2018], our algorithm first calculates the Delaunay tetrahedrization of the input vertices and successively subdivides tetrahedra so as to ensure conformity with input triangles. However, thanks to our approach based on implicit representations of point coordinates, we do not need to rely on slow exact arithmetic. Another relevant difference wrt TetWild is the type of output which, in our case, is a polyhedral mesh (not necessarily tetrahedral).

3.1 Terminology

From now on, the input is considered to be an *abstract simplicial complex* Σ over a set of vertices V, endowed with a function $\phi : V \rightarrow R^3$ that maps each abstract vertex to a unique position in space [Ferrario and Piccinini 2011]. The *geometric realization* $|\sigma|$ of a simplex $\sigma \in \Sigma$ is the convex hull of the image of its vertices under ϕ. The union of the geometric realizations of all the simplexes of Σ is the geometric realization of Σ and is denoted with $|\Sigma|$.

Let $\sigma = \{v_1, ..., v_n\}$ be a simplex in Σ, the *boundary* of its geometric realization $|\sigma|$ is the union of the geometric realizations of its proper subsimplexes (i.e. its *faces*) and is denoted with $\partial|\sigma|$. The set of points in $|\sigma|$ which are not in $\partial|\sigma|$ is the *interior* of $|\sigma|$ and is denoted with $I(|\sigma|)$. Hence, $I(|\sigma|) = |\sigma| \setminus \partial|\sigma|$.

From now on, when no ambiguity arises, we shall omit the vertical bars $|.|$ when referring to geometric realizations. We say that two simplexes σ and τ intersect if the intersection of their geometric realizations is not empty, and say that σ and τ *cross* each other if the intersection is made of a single point.

3.2 Tetrahedral Mesh

In this first stage, we only consider the position of the input constraint vertices, and tetrahedrize their convex hull using a classical incremental Delaunay insertion algorithm [Bowyer 1981]. Some of the input constraints may coincide with facets of the resulting tetrahedral mesh, whereas some others may not. The procedure to distinguish between these two cases is described in the following Sect. 3.3.

3.3 Mapping tetrahedra to intersecting constraints

If a constraint is not a triangle in the tetrahedral mesh, it may intersect the interior of some tetrahedra, and/or be coplanar with some facets of the mesh. In the former case, we need to split the intersected tetrahedron using the constraint plane. If the input surface has no boundaries, splitting all the tetrahedra in this way is sufficient to guarantee that the input surface can be represented as the union of mesh facets (see appendix A). For the sake of simplicity, in this section we assume that the input has no boundary. We will see how to deal with the other cases in Sect. 3.5.

We create a map that, for each tetrahedron, stores the set of all the constraints that intersect its interior. This step may appear simple, but implementing it efficiently and with guarantees requires a particular care: every branch in the program flow must be uniquely determined by *certified* predicate values.

The tetrahedral mesh and the constraints share their vertices. Hence, for each constraint, we consider one of its vertices and verify that its incident tetrahedra do not have the constraint itself as one of their facets. If this occurs, the constraint is already part of the mesh and does not need to be mapped for later subdivision. Conversely, if the constraint is not found, our strategy is organized in two consecutive phases: first, we collect all the tetrahedra that intersect its three edges (Sect. 3.3.1); second, we grow this initial set of tetrahedra inward to cover the whole constraint area (Sect. 3.3.2). During these two phases, we consider two simplexes to intersect if their closures share at least a point, so that the set S of all tetrahedra intersecting a given constraint forms a compact hull containing the triangle. Stated differently, S is a combinatorial 3-manifold with boundary. This characteristic is key for efficiency.

Once S has been determined, we classify the constraint-tetrahedron intersections in *proper* and *improper* intersections. We say that an intersection is proper if it is completely contained in the boundary of the tetrahedron, otherwise it is improper (i.e if it involves the interior of the tetrahedron). Only improper intersections lead to splits during the BSP phase, and hence are included in the eventual map (Sect. 3.3.3). An example of this procedure is depicted in Fig. 4.

3.3.1 Tetrahedra intersecting constraint edges. Given a constraint, we start building S by collecting tetrahedra that intersect its three bounding edges. Let $s = <s_0, s_1>$ be one such edge: we initialize S with the set of all the tetrahedra incident at s_0. Then, we search the smallest-dimensional simplex σ on the boundary of S that intersects with s, and add all its incident tetrahedra to S. We repeat the process as long as $\sigma \neq s_1$.

Note that at each iteration σ might be a triangle, an edge or a vertex of the tetrahedral mesh. If σ is a triangle, only one tetrahedron will be added to S. In the other cases, the number of new tetrahedra in S is variable. We also observe that the intersection check can be performed by evaluating standard 3D orientation predicates.

At each iteration, we first check the intersection of s against the interior of triangles on the boundary of S. The interior of a triangle $t = <t_0, t_1, t_2>$ intersects s if both the following conditions hold:

$$\text{orient3d}(s_0, s_1, t_0, t_1) =$$
$$\text{orient3d}(s_0, s_1, t_1, t_2) = \qquad (1)$$
$$\text{orient3d}(s_0, s_1, t_2, t_0)$$

$$\text{orient3d}(s_0, t_0, t_1, t_2) \neq \text{orient3d}(s_1, t_0, t_1, t_2) \qquad (2)$$

where $\text{orient3d}(a, b, c, d)$ represents the sign of the volume determinant for the tetrahedron $<a, b, c, d>$:

$$\text{orient3d}(a, b, c, d) = \text{sign}\left(\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}\right)$$

If none of the triangles satisfies these conditions, we may check the intersection of s against the interior of edges on the boundary of S and, finally, against vertices. However, this is not necessary. Indeed, since we assume that t is not degenerate (possible degenerate constraints are ignored), the three orientations in Eqn. 1 cannot be all zeroes. This means that, if two of them are zero while Eqn. 2 is satisfied, we can skip any further search and conclude that σ is the common vertex of t in the two null orientations. Otherwise, if only one such orientation is zero while the other two are equal and Eqn. 2 is satisfied, σ is the edge corresponding to the null orientation.

3.3.2 Tetrahedra intersecting the constraint interior. The set S constructed so far is then grown to include tetrahedra that intersect the interior of the constraint. Similarly to the previous phase, we consider the smallest-dimensional simplexes on the boundary ∂S of S that intersect the constraint interior. These may be edges or vertices. At each step, we pick one such simplex σ and enrich S by adding all the tetrahedra incident at σ. The procedure terminates when no simplexes on ∂S intersect the interior of the constraint.

To check for intersection, we consider an edge $s = <s_0, s_1> \in \partial S$ and evaluate Eqns. 1 and 2. If they both hold and one of the two orientations in Eqn. 2 is zero (say for vertex s_0), then the corresponding vertex (s_0) belongs to the constraint and we add all its incident tetrahedra to S. If both Eqns. 1 and 2 hold but none of the orientations in Eqn. 2 is zero, then the edge crosses the constraint and all its incident tetrahedra are added to S.

3.3.3 Filling The Map. Once the set S is completely constructed, we create the actual map by associating the constraint to all the tetrahedra in S whose intersection with the constraint is improper.

We recall that, in this paper, we say that a tetrahedron t improperly intersects a constraint c if the interior of t intersects c. Establishing whether the intersection of a tetrahedron $t \in S$ with the constraint is improper amounts to check the value of a few cleverly selected geometric predicates. An exhaustive list of the possible configurations is shown in Fig. 5. An exact definition of all the predicates involved is given in Appendix B. One may argue that calculating all the intersections and then throwing away those that are proper is a waste of time. Actually, tetrahedra that improperly intersect a constraint do not form a compact hull, and their set may even be disconnected. This would prevent us from using advancing front techniques, thus making an efficient calculation much more difficult.

3.3.4 Implementation. The creation of the map does not rely on intermediate constructions, therefore efficiency can be achieved on the basis of standard geometric predicates [Shewchuk 1997] coupled with an appropriate data structure to store the terahedral mesh. For this latter aspect, we have extended the data structure proposed in [Marot et al. 2019] by associating one of the incident tetrahedra to each of the vertices. Since the mesh is manifold, this is sufficient to reconstruct all the topological relations in optimal time [De Floriani and Hui 2005].
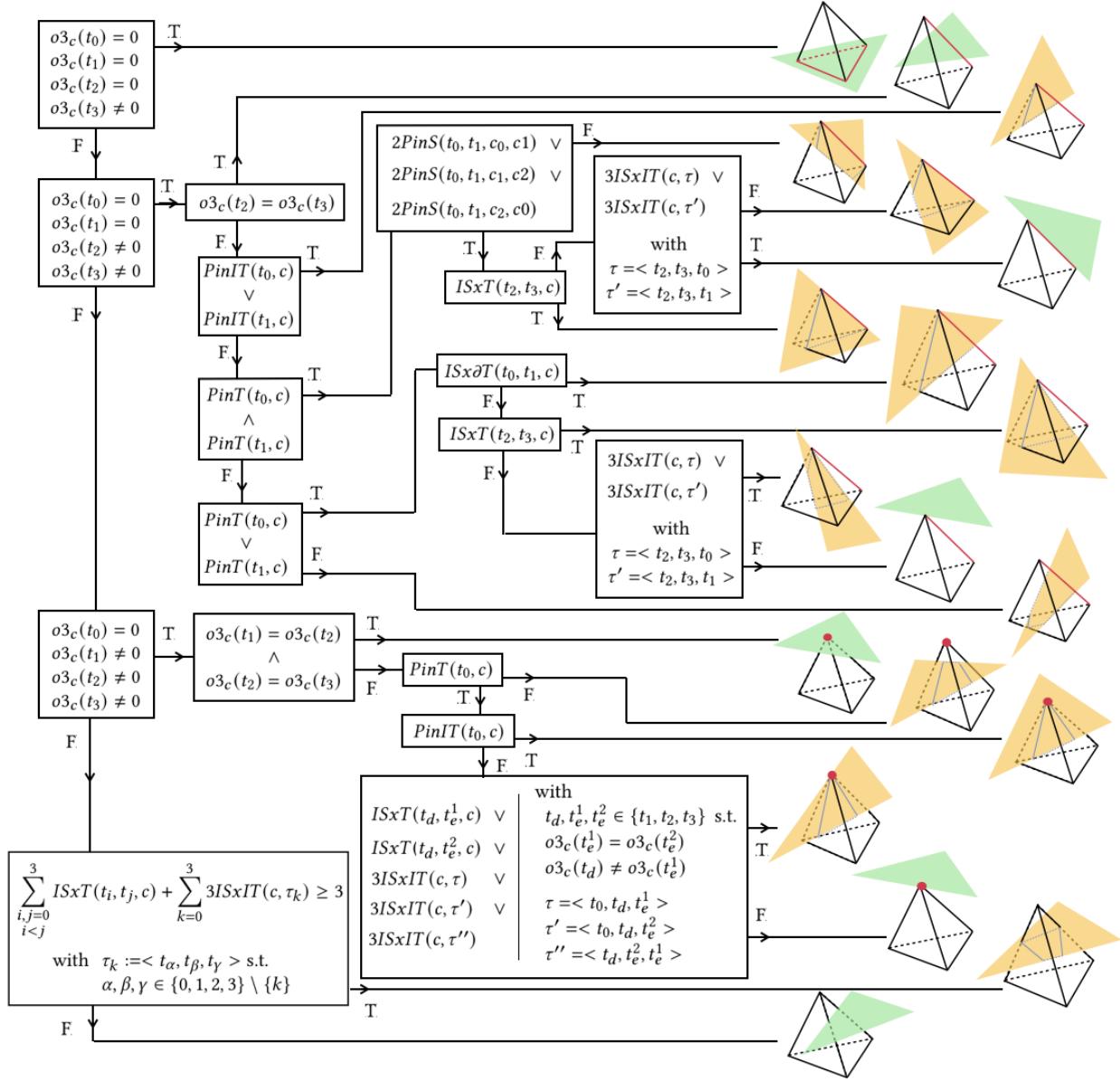
3.4 Binary Space Partition

Thanks to the maps constructed so far, each tetrahedron is associated to all the constraints that intersect its interior. To make the volume mesh *conformal* to the input constraints, we iteratively cut each tetrahedron using the planes of its associated constraints. Specifically, we use the constraint planes to construct a Binary Space Partition (BSP) out of each tetrahedron of the initial mesh.

We say that the volume mesh is *conformal* to the input constraints if there exists a collection of facets of the volume mesh whose geometric realization corresponds to the geometric realization of the input constraints. Note that the geometric realizations may coincide even if the underlying abstract simplicial complexes are different (see inset).

When designing this phase, combining efficiency and guaranteed correctness is particularly challenging, and the use of a proper data structure becomes crucial. Besides storing all the necessary connectivity information, our structure keeps track of the original points that generate any newly created element. This will become fundamental to use indirect predicates instead of exact arithmetic (Sect. 3.4.2)

3.4.1 Subdivision. Our data structure is designed to efficiently represent a manifold cellular mesh. We employ a simplified version of the classical Incidence Graph [Edelsbrunner 1987], where we store a partial co-boundary relation for edges. Each polyhedral cell in the mesh is required to be convex, but not necessarily *strictly* convex (i.e. a cell may have coplanar facets).

The complex initially coincides with the tetrahedra of the Delaunay mesh, and each cell inherits the map of its intersecting constraints (Sect. 3.3). We then consider one cell at a time, and split it in two sub-cells by using the plane of one of the constraints in the map.

Fig. 5. Possible configurations in which a constraint $c = <c_0, c_1, c_2>$ (yellow or green) intersects a tetrahedron $t = <t_0, t_1, t_2, t_3>$. We assume that an intersection exists. The flow chart shows how we determine whether an intersection is proper or improper. Pictures on the right exemplify corresponding configurations: the constraint color indicates the intersection type (green=proper, yellow=improper). Tetrahedron edges or vertices that belong to the constraint plane are marked in red. Small light-blue lines bound the area of the constraint included in the tetrahedron interior.

LIST OF CONTRACTIONS:

$o3_c(P) := \texttt{orient3d}(P, c_0, c_1, c_2)$

$PinIT(P, c) := \texttt{point\_in\_inner\_triangle}(P, c_0, c_1, c_2)$

$PinT(P, c) := \texttt{point\_in\_triangle}(P, c_0, c_1, c_2)$

$2PinS(P_0, P_1, s_0, s_1) := \texttt{point\_in\_segment}(P_0, s_0, s_1) \land \texttt{point\_in\_segment}(P_1, s_0, s_1)$

$ISxT(s_0, s_1, c) := \texttt{inner\_segment\_crosses\_triangle}(s_0, s_1, c_0, c_1, c_2)$

$ISxIT(s_0, s_1, c) := \texttt{inner\_segment\_crosses\_inner\_triangle}(s_0, s_1, c_0, c_1, c_2)$

$3ISxIT(c, \tau) := ISxIT(c_0, c_1, \tau) \lor ISxIT(c_1, c_2, \tau) \lor ISxIT(c_2, c_0, \tau)$

$ISx\partial T(s_0, s_1, c) := \texttt{inner\_segments\_cross}(s_0, s_1, c_0, c_1) \lor \texttt{inner\_segments\_cross}(s_0, s_1, c_1, c_2) \lor \texttt{inner\_segments\_cross}(s_0, s_1, c_2, c_0)$

After the split, the remaining constraints in the map are re-mapped to the two resulting sub-cells, depending on whether they intersect one, the other, or both the sub-cells. Each of the two sub-cells is then split again by using the plane of another constraint, and so on. This procedure guarantees that conformity will be eventually reached.

Upon a cell split, each edge of the cell that crosses the constraint plane at a certain point p is split into two sub-edges by inserting p as a new vertex. Then, each cell facet having at least two vertices on opposite sides of the constraint plane, is split in two sub-facets by adding a new edge: this new edge connects two face vertices that lie on the constraint plane and may be either original face vertices or new vertices produced by an edge split. Finally, the cell itself is split in two new cells, one consisting of faces and edges which are *above* the constraint plane, and the other consisting on the remaining faces and edges (which are *below* the plane). Both the sub-cells share a new common face built by collecting the edges that lie on the constraint plane. At this point, the maps must be updated: each of the remaining constraints in the map is associated to one or both the two sub-cells.

This procedure is summarized in Algorithm 1, and is iteratively repeated until all polyhedral cells have no more associated constraints. All the geometrical decisions in this phase can be reduced to a series of orient3d evaluations but, differently from the previous map-creation phase, here we must deal with intermediately constructed implicit points.

3.4.2 Implicit Points. When a tetrahedron of the Delaunay mesh is split in two convex cells by using a constraint plane, at least one of its edges is split too. Let this edge be e, and let p be the intersection point at which e is split. p becomes an endpoint for both the new sub-edges e' and e'' generated by the split. We use an LPI point (See Sect. 2.8) to implicitly represent p as a combination of five explicit input vertices (three for the constraint and two for the tetrahedron edge). This approach works as long as the edge to be split has two explicit endpoints, which is the case for the initial tetrahedral cells. Unfortunately, when the time comes to split e' or e'', this is no longer true.

To solve this problem, we enrich our data structure by keeping track, for each edge, of its *original* two endpoints. Specifically, any edge e has two pairs of endpoints $< s_0, s_1 >, < o_0, o_1 >$, which are initially coincident. Upon a split at a point p, the two sub-edges become $e' =< s_0, p >, < o_0, o_1 >$ and $e'' =< p, s_1 >, < o_0, o_1 >$. Essentially, while the first pair is used to *bound* the actual edge, the second pair represents the straight line by the edge and does not change after a split. While the first pair may include implicit points, the second pair is always made of two explicit points, and therefore can be used to construct other implicit points.

Unfortunately, this is still not sufficient. Indeed let us consider the common facet between the two sub-cells created by a split. This facet is bounded by edges connecting points on the constraint plane (e_{new} in Algorithm 1). These edges are not created after an edge split. Nevertheless, this is the only exception and we make a fundamental observation: in these cases the straight line by the edge may be represented by the two intersecting planes that generated the edge itself, that is, by the constraint and the intersected facet.

ALGORITHM 1: splitCell(c_{par})

Input: a convex polyhedral cell c_{par}. We can figure it as the "parent" cell. c_{par} has associated a set of constraints $K \neq \emptyset$.
Output: when the split occurs two convex polyhedral cells c_{par} and c'; otherwise c_{par} itself.

---

k = last element of the K;
remove k from K;
π = plane for k;
E = { edges of c_{par} } ;
for $e =< e_0, e_1 > \in E$ **do**
    if π *intersects the interior of* e **then**
        create new point $p_{new} = \pi \cap e$;
        split e at p_{new} in e' and e'';
        // $e' =< e_0, p_{new} >$ and $e'' =< p_{new}, e_1 >$
    end
end
F = { faces of c_{par} } ;
for $f \in F$ **do**
    if π *intersects the interior of* f **then**
        create new edge $e_{new} = \pi \cap f$;
        split f at e_{new} in f' and f'';
        // Edges of f are partitioned between f' and f''
           depending on their position (above/below) wrt
           π
    end
end
create new face $f_{new} = \pi \cap c_{par}$;
split c_{par} at f_{new} in c' and c'';
// Faces of c_{par} are partitioned between c' and c''
   depending on their position (above/below) wrt π
if $K \neq \emptyset$ **then**
    for $k \in K$ **do**
        V^+ = { vertices of k above π};
        V^- = { vertices of k below π};
        if $V^+ \neq \emptyset$ **then**
           assign k to c';
        end
        if $V^- \neq \emptyset$ **then**
           assign k to c'';
        end
    end
end
return;

Therefore, we further enrich our data structure to keep track of this information. First, for each facet we store the three explicit points that define its plane: if a facet is (or derives from a split of) a triangle in the Delaunay mesh, we associate the three triangle vertices; if a facet is part of a constraint, we associate the three vertices of the constraint. Second, when we build an edge e_{new} by intersecting a constraint c_e and a facet f_e, we associate the two corresponding triplets of points to e_{new}. When the time comes to split e_{new} with another constraint c, we represent the split point p as an implicit TPI point whose three constituting triplets are c's vertices and the two triplets associated to e_{new}.

By analyzing the subdivision procedure (Algorithm 1) it is clear that no other cases can occur, and hence our approach enables the use of indirect predicates to resolve the problem.

To summarize, a facet in our data structure stores:

- The list of its bounding edges;
- Three explicit points representing its plane;
- Its two incident cells.

whereas an edge stores:

- Its two endpoints (possibly implicit);
- Six explicit points representing its straight line (four of them may be unused if the edge is (a sub-edge of) an element of the initial tetrahedral mesh);
- One of its incident facets.

When an edge is split, we count the number of explicit points that define its straight line in the data structure. If they are two we represent the split point as an LPI, if they are six we use a TPI.

3.5 Virtual Constraints

Up to now we have assumed that the surface defined by the input constraints has no boundaries. However, our algorithm can accept in input any kind of triangle soup, and boundaries must be accounted for. To understand why boundaries are an issue, let us imagine a square pyramid, where the base is split in two triangles. Now, remove one of the triangles from the base. When the initial Delaunay tetrahedrization is built, there is no guarantee that the correct diagonal is chosen for the base. As a consequence, the unique constraint belonging to the base may be not represented by the union of facets as we require, each of the two triangular facets corresponding to the base overlaps with the input constraints only partially, and this makes it impossible to identify the constrained facets (see Sect. 3.6).

Our solution to this problem is based on the construction of *virtual constraints*. The idea is that, if an edge $e = < e_1, e_2 >$ is on the boundary, we may force the polyhedral mesh to contain e (or a subdivision of e) by adding an additional constraint $k = < e_1, e_2, v >$ such that:

- $t = < t_1, t_2, t_3 >$ is a constraint incident at e;
- v is one of the vertices in the input s.t. $orient3d(v, t_1, t_2, t_3) \neq 0$.

In the pyramid example, the only possible position for v is the pyramid apex. If the tetrahedral mesh uses the *wrong* diagonal, the two tetrahedra are split by the virtual constraint, and the new base is made of four right triangles. Each of these triangles may be a subset of the input surface or it may be disjoint from that surface, but partial overlaps are no longer possible.

When it exists, the boundary of the input set of constraints consists of a set of edges. Identifying these edges is important to create as many virtual constraints as necessary. It is tempting to say that an edge is on the boundary if it has only one incident triangle, but this is true only in a combinatorial sense. For the sake of our space subdivision, we want that the geometric realization of the constraints has no boundaries. Hence, for each edge we verify whether the geometric realization of the edge is in the interior (wrt the surface topology) of the geometric realization of its incident constraints. If it is not, the edge is considered a boundary edge and leads to the creation of

a virtual constraint k. The vertex v to be used when creating k is selected from one of the tetrahedra incident at the boundary edge in the Delaunay mesh. This guarantees a fast selection of a nearby non-coplanar vertex.

Checking whether an edge e is in the interior of the geometric realization of its incident constraints amounts to the evaluation of $orient3d$ and $orient2d$ predicates. We start from the first incident constraint $t = < t_1, t_2, t_3 >$ and search another incident constraint whose opposite vertex v wrt e is such that $orient3d(v, t_1, t_2, t_3) \neq 0$. If we can find such other constraint, then e is not on the boundary. Otherwise, all the incident constraints are coplanar, and we must verify whether all of them are on the same side of e. To do this, we project t on one of the coordinate planes by dropping one of the coordinates (we pick one for which the projected t is not degenerate), and perform the side checks in 2D by adopting the same projection for all the other constraints.

3.6 Identification of constrained facets

Thanks to the iterative cell subdivision, the initial tetrahedral mesh is replaced by a polyhedral mesh that conforms to the input set of constraints. The next step consists in identifying which of the polygonal facets actually constitute the input constraints. From now on, we refer to these facets as to *constrained* facets.

We use a colouring metaphor, which is essentially based on three colours (white, grey and black) associated to the mesh facets throughout the algorithm execution. If $|\Sigma|$ is the geometric realization of the input, we say that a facet f is white if the intersection of its interior $I(f)$ with $|\Sigma|$ is empty, whereas it is black if the intersection is the whole facet. In all the other cases (i.e. the intersection consists of only a portion of the facet, or we cannot say based on current information), the facet is grey. It is important to notice that a black facet may be: (1) coincident with a constraint, (2) fully contained into a constraint, (3) contained in the union of a number of co-planar constraints.
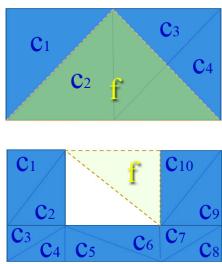
A constraint may be associated with a tetrahedron if their intersection is improper, and a map collecting all these associations is built as described in Sect. 3.3. An additional map is created to associate each triangular facet of the initial tetrahedral mesh with all the constraints that (1) are co-planar with the facet itself, and (2) realize a positive area intersection with the facet. Upon any facet split, both the resulting subfacets inherit the set of coplanar constraints from their parent.

Before subdivision, each triangular facet of the initial tetrahedral mesh is coloured black if it coincides with a constraint, whereas it is coloured white if it has no mapped coplanar constraints. In all the other cases the facet is coloured grey.

During subdivision, each sub-face created by a facet split inherits the colour from its parent facet. Conversely, a grey colour is assigned to the new common face shared by the two sub-cells produced by a cell split. Consequently, when the BSP procedure is complete, each facet of the polyhedral mesh may be white, grey or black. At this stage white facets are necessarily unconstrained because they are sub-facets of a formerly white facet. For the same reason, black facets are necessarily constrained. Each grey facet is either entirely contained in $|\Sigma|$ or its interior is disjoint from $|\Sigma|$. No partial

intersections are possible. Hence, we can finalize the colour of any grey facet to white or black as described in the following sub-section.

3.6.1 Grey Facet finalization. To decide if a grey facet f must be eventually coloured in white or black, we check the geometric realization $|f|$ and the geometric realization $|c| = \bigcup_i |c_i|$ of its associated co-planar constraints c_i. If $|f| \subset |c|$, then the facet is black, otherwise it is white. Note that this procedure makes sense only if the intersection between a facet and its coplanar constraints is either the whole facet or empty. Implementing such a check while using exact predicates only is not as easy as it may appear. We proceed as follows: first, we check whether at least one vertex of f is in the interior of one of the c_i. If this occurs, the facet is black. Otherwise, we check whether one vertex of f is not part of any c_i (boundary included). If this occurs, the facet is white. If neither of these cases occur, all the vertices of f are on the boundary of some constraint c_i. Unfortunately, the realization $|c|$ of the coplanar constraints is not necessarily convex, therefore we cannot give a final response without further analysis (see inset).

In this case each of the vertices of a grey facet f is on the boundary of one of its coplanar constraints c_i, and we may verify whether $|f|$ is a subset of $|c| = \bigcup_i |c_i|$ as follows. We assume that either $|f| \subseteq |c|$ or $I(|f|) \cap I(|c|) = \emptyset$, therefore if there exists one c_i such that $I(|f|) \cap I(|c_i|) \neq \emptyset$, we can conclude that $|f| \subseteq |c|$.

For each c_i, we verify whether $I(|f|) \cap I(|c_i|) \neq \emptyset$ by searching three misaligned points on the boundary of c_i that are also on the boundary of f. If three such points exist, then an intersection occurs. Unfortunately we cannot construct these points because f may be bounded by implicit points, and this would prevent us from using indirect predicates. However, we just need to know if these points are misaligned, and this occurs if they do not belong to the same edge of c_i (endpoints included). Hence, for each edge of f, we check which edges of c_i it intersects and verify whether, among all the possible intersections, there are three that do not belong to a common edge of c_i. This procedure is guaranteed to produce exact results, but it is too slow even if implemented with fast indirect predicates as we do.

Therefore, we first try with a faster approach and revert to this slow version only in case of failure. Our faster method is fairly simple: we calculate the barycenter of f and check whether it is part of one of the c_i. If it does, the facet is black, otherwise it is white. The problem with this approach is that the barycenter is necessarily approximated and hence may lead to the wrong result, but all that we need is a point in the interior of f. Thus, we calculate the approximated barycenter and check whether it is actually contained in the interior of the facet. If it does, it can be safely used for our purpose, otherwise we must necessarily rely on the slow method. Our experiments show that this approach accelerates the slow method by approximately one order of magnitude.

3.7 Interior/Exterior Cells Partition

We consider black faces as the suggested *skin* of our solid. If they form a closed surface, classifying the cells in internal and external

is trivial. Conversely, if they form an incomplete surface with holes and/or disconnected patches (e.g. the bust in Fig. 14), our approach is based on selecting as many white facets as necessary to close all the holes and gaps between patches. Specifically, we wish to select white faces so that their total area is minimum.

We achieve this goal indirectly, and use the available black faces to drive the classification of cells in internal and external. We create a dual graph G of our polyhedral mesh, where a node exists for each cell, and two nodes are connected by an arc if the two corresponding cells share a white facet. Stated differently, a node in G is the dual of a cell, whereas an arc is the dual of a white facet. Furthermore, G has an additional node corresponding to the *outer* infinite cell, and such a node is connected to any other node being the dual of a cell having a white facet on the boundary. As an example, if the black faces constitute a single closed surface, G has exactly two components, one corresponding to external cells, and the other to internal cells. Conversely, if the surface has holes, G may be connected and determining which nodes correspond to internal cells can be cast as a minimum cut problem. In general, we may need to split G in at least two components, but not necessarily just two.

In our formulation, the cut minimizes the following energy:

$$E(G, l) = \sum_{v \in nodes} d(v, l(v)) + \sum_{a \in arcs} s(a, l(a.v_1), l(a.v_2)) \quad (3)$$

where l is a labeling that assigns either *in* or *out* to each node of G, and $a.v_1, a.v_2$ are the two nodes connected by the arc a. This energy is made of *data costs* d plus *smooth costs* s. The data cost $d(v, l(v))$ is the cost of labeling v according to l, whereas the smooth cost $s(a, l(a.v_1), l(a.v_2))$ is the cost of labeling the two connected nodes $a.v_1$ and $a.v_2$ according to l.

The data cost is defined as follows:

$$d(v, in) = \sum_{f \in c_{out}(v)} A(f) \quad (4)$$

$$d(v, out) = \sum_{f \in c_{in}(v)} A(f) \quad (5)$$

where $c_{out}(v)$ (resp. $c_{in}(v)$) is the subset of facets f_i of the dual cell of v such that (1) f_i is black and (2) the normal of its coplanar constraints points towards the interior (resp. the exterior) of the cell. $A(f)$ is the area of f. Intuitively, if the normal of a constraint t points *up*, a black face f is part of t, and the cells c_1 and c_2 share f, if c_1 is *below* t, then t suggests that c_1 is internal. This means that we must penalize its labeling as external, and we do that by an amount proportional to the total *offending facet* areas.

The smooth cost is defined as follows:

$$s(a, in, in) = s(a, out, out) = 0 \quad (6)$$

$$a(a, in, out) = s(a, out, in) = A(f(a)) \quad (7)$$

where $f(a)$ is the dual (white) face of a. Essentially, this cost represents a penalization for two connected nodes from having different labels. Note that for an input with no defects (i.e. a closed oriented manifold) there exists a labeling that makes both the data cost and the smooth cost vanish.
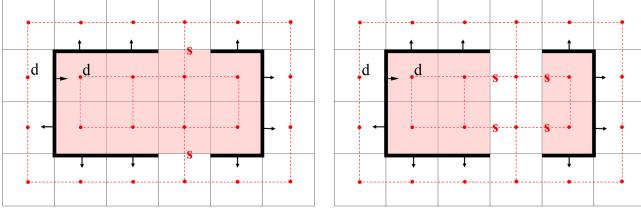
Fig. 6. In this 2D example all the cells are squares and black faces are depicted by thick lines along with the normal of their coplanar constraints. Red dotted lines represent the dual graph, whereas pink and white filling denote internal and external labeling respectively. A black "d" near a dual node means that the node contributes a nonzero data cost. Similarly, a red "s" near a dual arc means that the arc contributes a nonzero smooth cost. The left and right labelings correspond to a total energy $E(G, l_{left}) = 2A + 2A$ and $E(G, l_{right}) = 2A + 4A$, therefore the minimum is achieved by the left labeling (here A is the area of any of the faces). Note that the data cost is nonzero because one of the input normals has a "wrong" orientation.

To calculate the optimal labeling according to our energy definition we adopt the widely-used `graphcut` algorithm [Boykov and Kolmogorov 2004; Boykov et al. 2001]. Note that the use of graphcut to classify internal and external elements was already adopted in [Hornung and Kobbelt 2006; Jacobson et al. 2013; Wan et al. 2013; Zhou et al. 2008], though all these previous formulations are different from ours.

4 RESULTS

To test our algorithm we have implemented a standalone tool with options to execute various applications (Sect. 5). Source code is written in C++ and depends on the Indirect Predicates library (https://github.com/MarcoAttene/Indirect_Predicates). Our tool was compiled on a Windows 10 home desktop PC with Intel i9 and 32Gb RAM. Experiments were run on the same machine. The reference implementation is open-source and available on GitHub: https://github.com/MarcoAttene/VolumeMesher.

We have tested our implementation on the whole Thingi10k dataset [Zhou and Jacobson 2016], which consists of 10000 triangulated surfaces modeling both real-world and synthetic shapes. Our tests demonstrate that our algorithm is exceptionally fast (Fig. 7). Over than 65% of the models are meshed in less than 1 second and the percentage of those that require at most 10 seconds surpasses the 90%. Memory usage is appropriate for use on standard home desktop PCs such as ours (see fig. 8). Only one model (ID: 996816) over the 10000 in Thingi10k could not be processed on our reference machine because it requires more than the memory available. All the other models were succesfully processed at an average speed of 4.34 seconds per model, and with an average memory usage of 225 Mb per model.

To better understand how demanding each phase is along our pipeline, and hence to suggest directions for re-engineering and optimizing our prototype code, we report phase-by-phase time requirements in Fig. 9. We observe that there is no specific bottleneck that requires optimization, but in all the phases there is plenty of space for improvement (see Sect. 6).
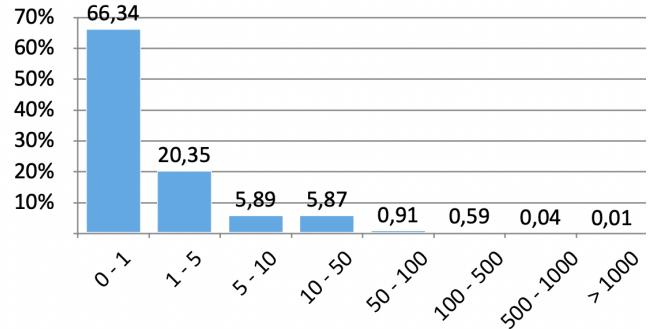

Fig. 7. Histogram of execution timing over the Thingi10k dataset. Each column represents the percentage of models that could be processed in a time belonging to the abscissa interval.
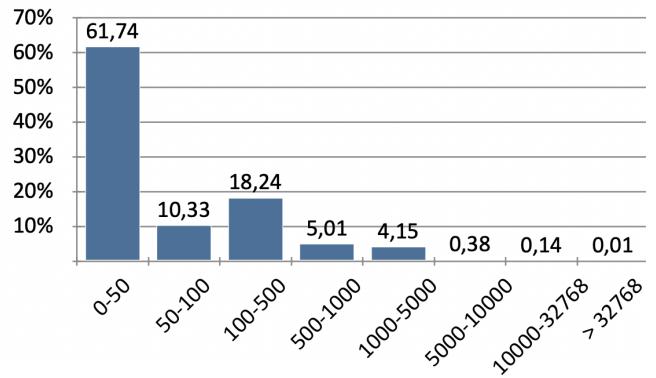

Fig. 8. Histogram of memory usage over the Thingi10k dataset. Each column represents the percentage of models whose processing requires an amount of memory (Mb) belonging to the abscissa interval.
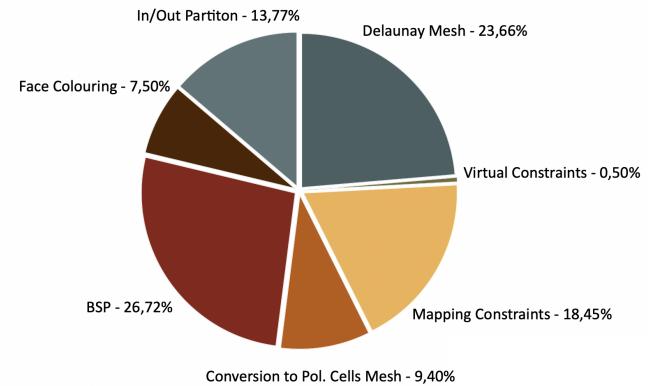

Fig. 9. Pie chart reporting how the total running time of our method distributes across the various steps of the pipeline. These data refer to the time spent processing the entire Thingi10k dataset.
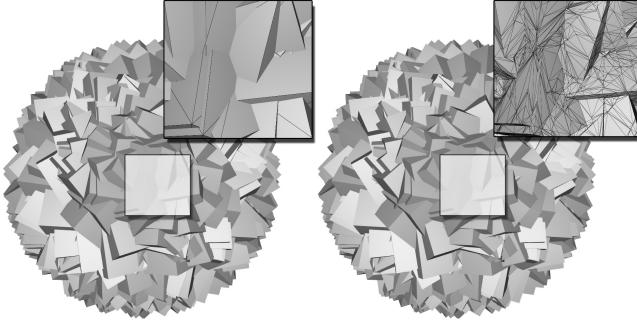
Fig. 10. A cube was replicated 100 times with random rotations. The resulting self-intersecting model (left) was correctly meshed by our method (right).

Table 1. Running time in seconds on ten models randomly selected from the Thingi10k dataset. Specific options were used for each software tool. TetGEN: -zp; fTetWild: −skip-simplify −max-its 0 −max-threads 1; TetWild: original code modified to skip input simplification and mesh optimization, $\epsilon = 0$; CGAL: used PyMesh command tet.py −config cgal.

Model	TetGEN	fTetWild	TetWild	CGAL	Ours
35269	1,104	7,171	58,275	112,815	1,422
42194	7,990	276,797	931,399	26,544	5,528
54161	0,151	1,309	28,808	4,510	0,493
66484	0,043	0,465	1,659	0,457	0,053
73161	0,022	0,457	0,606	0,703	0,032
81309	0,173	1,235	4,120	1,568	0,121
97493	0,021	0,095	0,904	0,857	0,025
119238	5,778	53,910	272,242	41,019	3,944
133079	6,363	25,799	58,735	4,727	6,733
209086	0,200	18,284	15,663	0,651	0,154

Input files with numerous intersections are particularly challenging as they require the construction of many implicit points. To stress our method we have generated a synthetic model made of 100 intersecting cubes with random rotations. Even in this case, the algorithm produces the expected single component mesh in about four minutes (see Fig. 10).

An input with many coplanarities is also a typical challenging experiment that stresses the underlying numerical engine. To test our algorithm in this sense, we have computed two complementary solids having a rather complex common surface, and then used our method to successfully calculate their boolean union (Fig. 13).

4.1 Comparison with existing methods

The only existing method that can exactly resolve our conformal meshing problem with the same guarantees is TetWild [Hu et al. 2018]. If run with $\epsilon = 0$ and neither initial simplification nor mesh optimization, TetWild produces an exact explicit mesh of the input model as we do and is tolerant to defects. Hence we slightly modified the author-provided code (https://github.com/Yixin-Hu/TetWild) to switch these unnecessary steps off and compared with this custom version. The fast version fTetWild [Hu et al. 2020] uses approximations from the very beginning, and cannot be made exact. Nonetheless, by preventing the input simplification and using zero optimization iterations, fTetWild can produce a rather precise result and is worth a comparison. We also compare against TetGEN [Si 2015] and CGAL meshing with polyhedral oracle and feature protection. For the latter we used PyMesh wrapper (https://github.com/PyMesh) with default options. The behaviour of all these existing algorithms on the Thingi10k dataset is well described in [Hu et al. 2020], while the performances of our method on that same dataset are reported in the previous section.

In numbers, the Thingi10k dataset contains 4540 models where all the algorithms being compared succeed. Over such a sub-dataset, an average of 2 seconds per model are used by TetGEN, 38 by fTetWild, 137 by TetWild, 19 by CGAL, and 1.85 by our method (difference wrt [Hu et al. 2020] is due to the different configurations used). Therefore, as far as speed is concerned, our method is approximately equivalent to TetGEN while being much more robust and tolerant to defects. We are one order of magnitude faster than CGAL with

feature protection, 20 times faster than fTetWild, and 70 times faster than TetWild. Table 1 reports the specific running time on ten randomly selected models where all the methods succeed. The result of our method on the same set of models is shown in Fig. 11.

5 APPLICATIONS

5.1 Mesh Repairing

Our method creates a valid 3-manifold out of an input with defects. Therefore, if we consider the boundary surface as our result, our technique can be regarded as a mesh repairing algorithm. Our repairing tool is extremely precise and complete as it can handle all the complex aspects related to this procedure (auto-intersections, disconnected components, surface holes and gaps). Regarding precision, when no ambiguity occurs, our method produces the exact expected topology, and the vertex position is as precise as the floating point representation permits. When input is ambiguous, our approach strives to minimize the total area of the *correcting* facets, and hence provides an intuitive repairing (see Fig. 12).

5.2 Boolean composition

The boolean composition of polygon meshes is a notoriously difficult problem, especially due to the inherent robustness issues that arise during the process, even if the input models are clean and well-defined. Besides ensuring an unconditional robustness, our solution to the problem also expands the class of possible input models to virtually any collection of triangles.

Hence, let A and B be two input files, each representing an unstructured set of triangles (see Sect. 3.1). Our method defines the two volumes enclosed by A and B, and then performs a simple set operation. Specifically, we create a volume mesh which is conformal to both A and B, and classify each cell as belonging to A, B, both A and B, or none of the two.

Our basic algorithm requires a small adaptation: input constraints belonging to A and B are merged in a single set and are used together to construct the space subdivision, but their origin is tracked to correctly label the black facets as belonging to A or to B (or to both, in case of overlaps). Then, the interior/exterior classification is done
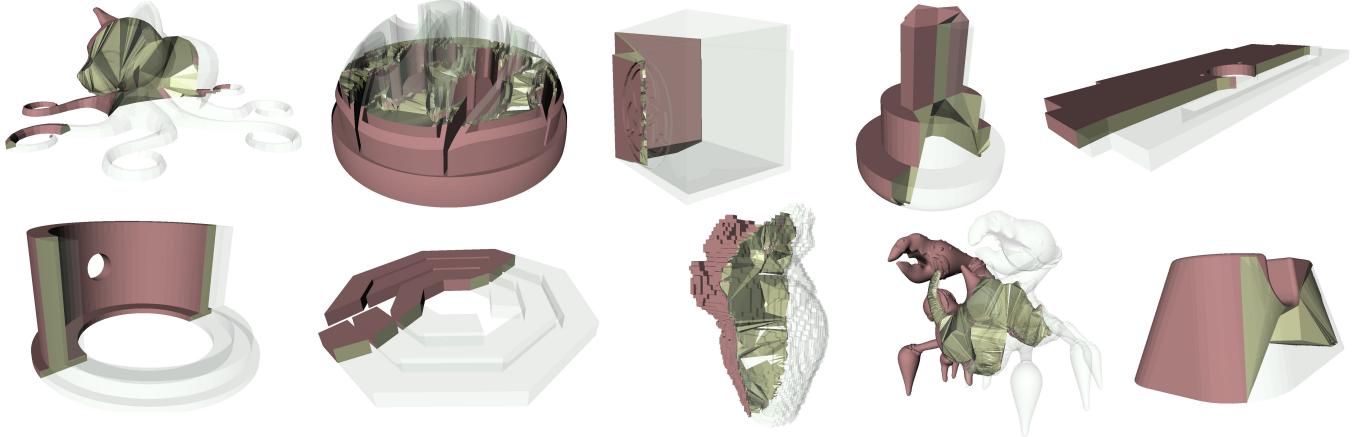
Fig. 11. Ten models randomly selected from the Thingi10k dataset and meshed by our algorithm.
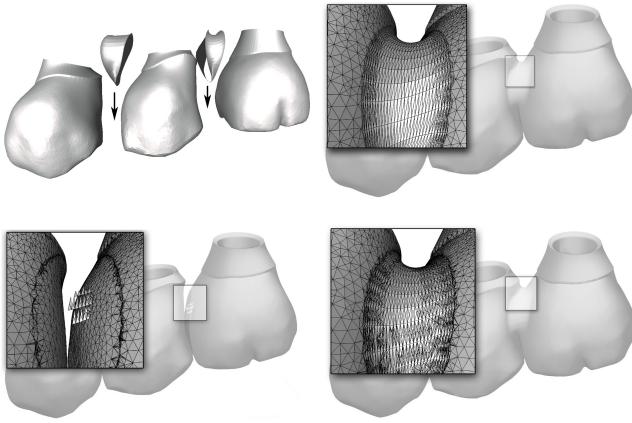

Fig. 12. Top row: a dental bridge modeled by a professional combines digitized parts (the teeth) and modeled surfaces. The resulting model is made of five intersecting components. Bottom-left: Model repaired by tetwild [Hu et al. 2018] using $\epsilon = 0$ and switching the mesh optimization off. Though elegant in their formulation, in this practical case generalized winding numbers do not capture the design intent. Bottom-right: Model repaired by our method, where the approach based on area minimization gives the expected single-component result.

twice, once using the black facets from A and once using those from B. As a result, each cell is labeled as belonging to A, B, both A and B, or none of the two as we need. Using this labeled volume mesh to extract a boolean combination is trivial, and the result is regular by construction [Feito et al. 2013].

Our prototype implements three basic boolean operations: union, intersection and difference (Fig. 13).

Thanks to the inherent method robustness and efficiency, boolean operations can be cascaded and interleaved with other geometry processing operations along the solid modeling pipeline (see Fig. 14).

Table 2. Resolution of mesh self-intersections using three different approaches on a common dataset. Average per-model runtime and memory consumed are reported.

Dataset	Method	Time (s)	Memory (Mb)
4408 models	[Cherchi et al. 2020]	2.43	119.1
in Thingi10k	LibIGL	5.27	148.7
with self-int.	Ours	6.98	366.5

5.3 Minkowski sums

The Minkowski sum of two subsets of the 3D space A and B is the set $A \oplus B = \{a + b | a \in A, b \in B\}$. If the surface of both A and B is a polygon mesh, then their Minkowski sum $A \oplus B$ is also a polygon mesh and can be computed exactly. Our approach to calculate $A \oplus B$ is extremely simple: first, we calculate a tight superset of its faces as described in [Campen and Kobbelt 2010b]; then, we run our volume meshing algorithm on that superset, which might be nonmanifold and/or self-intersecting, to realize a well-defined polyhedron representing the sum. An example is shown in Fig. 15.

5.4 Resolution of self-intersections

The extraction of black faces from the polyhedral mesh (Sect. 3.6) effectively resolves the self-intersections of the input triangle soup. We observe that, though our method can be effectively used to resolve this problem, explicitly reconstructing the whole volume is overkill, and more focused algorithms might perform better than ours. That is why we are not comparable with [Cherchi et al. 2020] where only the surface is processed. Nevertheless, our runtime is still comparable with LibIGL (see discussion in [Cherchi et al. 2020]) which also processes the surface only. The results of our experiments are reported in Table 2, where the serial implementation was used for all the tests.

5.5 Quality tetrahedral meshing

Tetrahedral meshes with well-shaped elements can be obtained in different ways thanks to our algorithm. The most straightforward
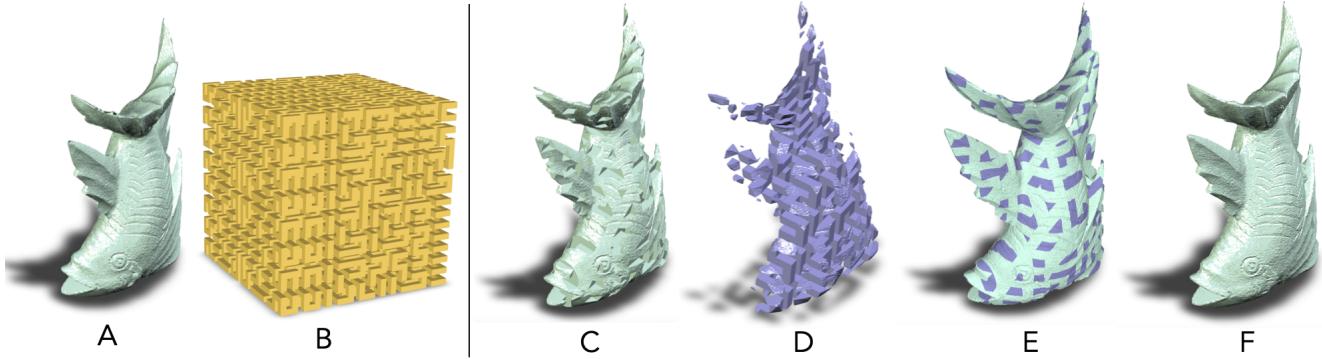
Fig. 13. Boolean composition of a fish model (A) and a Hilbert cube (B). The difference $A \setminus B$ is shown in (C), whereas the intersection $A \cap B$ is shown in (D). The difference and intersection are displayed together in (E) (two different models), whereas their union is calculated and shown in (F) (single model).

Fig. 14. Our algorithm enables extremely intuitive solid modeling paradigms that can combine boolean composition and classical geometry processing. In this example, (1) the "SigAsia2021" model is subtracted from a "box". (2) The resulting mesh is then smoothed out through uniform remeshing followed by a Laplacian smoothing iteration and a mesh simplification to remove redundant vertices with coplanar neighborhood. (3) The smooth mesh is united with the "tile" model and, finally, (4) the resulting union is further united with the "bust" model. Note that possible intersections, degenerate or inverted elements, non-manifold configurations, surface holes, or even disconnected components in the input parts are handled as expected.
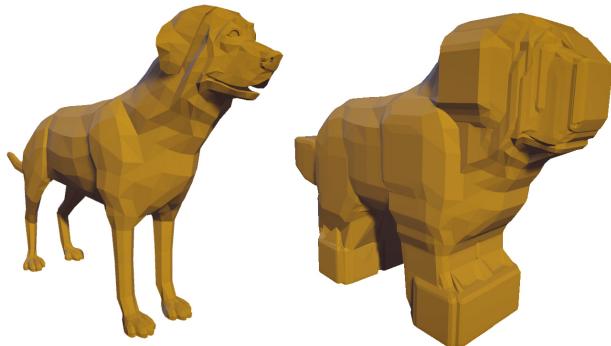

Fig. 15. The Minkowski sum of the dog model with a cube.

method exploits our repairing functionality (Sect. 5.1) to transform an input with defects to a well-formed surface that existing, non robust, quality mesh generators can easily process. This synergy overcomes both types of methods limitations, enlarging the possibility of producing high quality tetrahedral meshes whose natural application is the finite elements analysis (for example see Fig. 16). Although this synergy enlarges the class of input models that can be meshed, for some models the process may still fail. Indeed, our output meshes may have arbitrarily bad-shaped facets that may not be correctly processed by current quality meshers such as, e.g., TetGEN [Si 2015].

Alternatively, it would be possible to first transform our polyhedral mesh to a tetrahedral mesh, and then execute a sequence of primitive modifications to optimize a target quality metric (e.g. the conformal AMIPS energy as in [Hu et al. 2018]). We observe
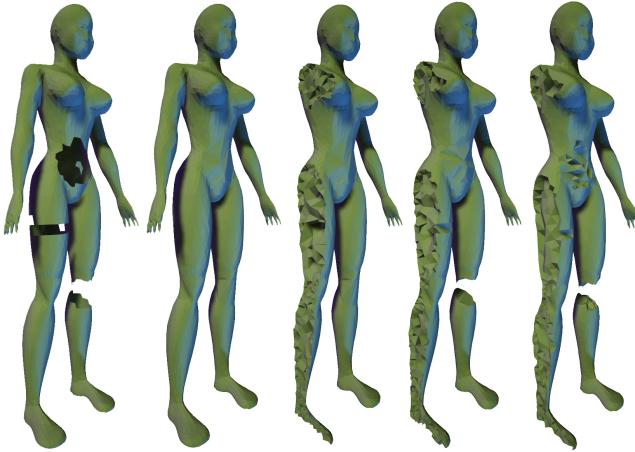
Fig. 16. A mutilated mannequin model (left) was successfully processed by our algorithm (mid-left) in 0.17 seconds. The boundary surface mesh could then be processed by tetgen in 0.6 seconds to produce a quality tetrahedral mesh (center). For the sake of comparison, the same input was processed by our exact version of TetWild in 1.07 seconds, and the resulting surface could be meshed by tetgen in 0.6 seconds (mid-right). Instead of using tetgen, TetWild was also used to directly produce the quality mesh by using the original implementation with default parameters in a total of 38.6 seconds (right). Note that the use of generalized winding numbers prevents TetWild to produce the expected topology.

that the tetrahedrization of our polyhedral cells can be computed by means of indirect predicates because it does not require further intermediate constructions. Indeed, since our cells are convex, they can be tetrahedrized without Steiner points. Since quality meshing is not the main focus of this paper, we have not implemented this latter approach.

6 CONCLUDING REMARKS

Although volume meshing is useful to implement interactive solid modeling systems, existing conformal meshing algorithms are either efficient (e.g. TetGEN) or unconditionally robust (e.g. TetWild), but not both.

In this research, we have shown that the creation of volume meshes can be made both robust and efficient thanks to a clever algorithmic design that enables the exploitation of indirect geometric predicates. Our experiments demonstrate that a typical input can be meshed in a few seconds even if it has defects. We believe that this time is appropriate for operations within an interactive solid modeling system. We observe that our approach is as efficient as the faster state-of-the-art tools, while guaranteeing an unconditional robustness.

Nonetheless, our method still exhibits a few limitations and there is plenty of room for future research and improvements.

First and foremost, our method is designed to deal with volumetric objects. Though open surfaces still represent a valid input, they might not be processed as expected. Furthermore, although creating an explicit volume guarantees that solids are well-defined, when our method is used to produce surface models the result may become

unnecessarily complex (see e.g. Fig. 10-right). Although this is partly intrinsic because we use convex cells, we believe that appropriately sorting the split operations, or replacing cell splits with face swaps when possible [Shewchuk 2003], may sensibly reduce the output complexity. Alternatively, redundant vertices may be removed in a post-processing step, though that would require an additional time.

Another relevant issue is related to the approximation introduced when the implicit points are turned to explicit coordinate triplets at the end of the process. This is not an issue when the result is reused in our own defect-tolerant system, but problems may still arise when external non-robust systems must be used. This problem is well-known, and very recent research is studying strategies to sidestep it [Nehring-Wirxel et al. 2021] and enable a safe cascading of the most *delicate* operations. By not relying on any additional data structure, our algorithm may simply read a mesh and produce another mesh. This makes its integration into existing modeling systems trivial, and enables a robust solution to the most delicate operations (e.g. booleans) that can be cascaded one after the other and interleaved with any other geometry processing algorithm. This flexibility significantly reduces the actual need of *external* non-robust systems.

Finally, we are aware that our algorithm uses an excessive amount of memory. This is partly inherent as the volume mesh may become rather complex, but in our prototype we have admittedly overestimated the maximum number of tetrahedra and cells, and hence used a too large (64bit) data type to index each mesh element. We believe that a better-engineered implementation may significantly less memory with no major changes in the algorithmic design.

Note that our prototype has not undergone any particular optimization to exploit modern hardware parallelism, but the two most demanding phases in our process, i.e. the initial tetrahedrization and the cell subdivision (See Fig. 9), can be both optimized to take advantage of multi-core architectures as described in [Marot et al. 2019] and [Hu et al. 2020]. Also, the initial tetrahedrization can be further optimized by properly pre-sorting the input points [Amenta et al. 2003].

ACKNOWLEDGMENTS

REFERENCES

Marc Alexa. 2020. Conforming Weighted Delaunay Triangulations. *ACM Trans. Graph.* 39, 6, Article 248 (Nov. 2020), 16 pages. https://doi.org/10.1145/3414685.3417776

N. Amenta, S. Choi, and G. Rote. 2003. Incremental Constructions con BRIO. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*. Association for Computing Machinery, 9.

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, Goslar,

DEU, 1269–1278.

Adrian Bowyer. 1981. Computing Dirichlet tessellations. *Comput. J.* 24, 2 (1981), 162–166. https://doi.org/10.1093/comjnl/24.2.162

Yuri Boykov and Vladimir Kolmogorov. 2004. An experimental comparison of mincut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Analysis and Machine Intelligence* 26, 9 (2004), 1124–1137.

Yuri Boykov, Olga Veksler, and Ramin Zabih. 2001. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Analysis and Machine Intelligence* 23, 11 (2001), 1222–1239.

Marcel Campen and Leif Kobbelt. 2010a. Exact and Robust (Self-)Intersections for Polygonal Meshes. *Comput. Graph. Forum* 29 (05 2010), 397–406.

Marcel Campen and Leif Kobbelt. 2010b. Polygonal Boundary Evaluation of Minkowski Sums and Swept Volumes. *Computer Graphics Forum* 29, 5 (2010), 1613–1622.

Gianmarco Cherchi, Marco Livesu, Riccardo Scateni, and Marco Attene. 2020. Fast and Robust Mesh Arrangements Using Floating-Point Arithmetic. *ACM Trans. Graph.* 39, 6, Article 250 (Nov. 2020), 16 pages. https://doi.org/10.1145/3414685.3417818

Leila De Floriani and Annie Hui. 2005. Data Structures for Simplicial Complexes: An Analysis And A Comparison. In *Eurographics Symposium on Geometry Processing 2005*, Mathieu Desbrun and Helmut Pottmann (Eds.). The Eurographics Association. https://doi.org/10.2312/SGP/SGP05/119-128

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

Herbert Edelsbrunner. 1987. *Algorithms in Combinatorial Geometry.* Springer-Verlag.

F. R. Feito, C. J. Ogayar, R. J. Segura, and M. L. Rivero. 2013. Fast and Accurate Evaluation of Regularized Boolean Operations on Triangulated Solids. *Comput. Aided Des.* 45, 3 (March 2013), 705–716. https://doi.org/10.1016/j.cad.2012.11.004

D. L. Ferrario and R. A. Piccinini. 2011. *Simplicial Structures in Topology.* Vol. CMS Books in Mathematics. Springer-Verlag New York.

Steven Fortune. 1999. Vertex-rounding a three-dimensional polyhedral subdivision. *Discrete and Computational Geometry* 22, 4 (1999), 593–618.

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.

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

Michael Hemmer, Susan Hert, Sylvain Pion, and Stefan Schirra. 2019. Number Types. In *CGAL User and Reference Manual* (5.0 ed.). CGAL Editorial Board. https://doc.cgal.org/5.0/Manual/packages.html#PkgNumberTypes

Alexander Hornung and Leif Kobbelt. 2006. Robust Reconstruction of Watertight 3D Models from Non-Uniformly Sampled Point Clouds without Normal Information *(SGP '06)*. Eurographics Association, 41–50.

Yixin Hu, Teseo Schneider, Bolun Wang, Denis Zorin, and Daniele Panozzo. 2020. Fast Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 39, 4 (July 2020).

Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 37, 4, Article 60 (July 2018), 14 pages.

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.

Tao Ju. 2004. Robust Repair of Polygonal Models. *ACM Trans. Graph.* 23, 3 (2004), 888–895.

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.

Célestin Marot, Jeanne Pellerin, and Jean-Francois Remacle. 2019. One machine, one minute, three billion tetrahedra. *Internat. J. Numer. Methods Engrg.* 117, 9 (2019), 967–990. https://doi.org/10.1002/nme.5987

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

Julius Nehring-Wirxel, Philip Trettner, and Leif Kobbelt. 2021. Fast Exact Booleans for Iterated CSG using Octree-Embedded BSPs. *Computer-Aided Design (to appear)* 135 (July 2021).

J. Podolak and S. Rusinkiewicz. 2005. Atomic Volumes for Mesh Completion. In *Symposium on Geometry Processing*.

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. 1998. Tetrahedral mesh generation by Delaunay refinement. (1998), 86–95.

J. R. Shewchuk. 2003. Updating and Constructing Constrained Delaunay and Constrained Regular Triangulations by Flips. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*. Association for Computing Machinery, 181–190.

Hang Si. 2015. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)* 41, 2 (2015), 1–36.

M. Wan, Y. Wang, E. Bae, X. C. Tai, and D. Wang. 2013. Reconstructing Open Surfaces via Graph-Cuts. *IEEE transactions on visualization and computer graphics* 19, 2 (2013), 306–318. https://doi.org/10.1109/TVCG.2012.119

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

W. Zhao, S. Gao, and H. Lin. 2007. A robust hole-filling algorithm for triangular mesh. *Visual Computer* 23 (2007), 987–997. https://doi.org/10.1007/s00371-007-0167-y

K. Zhou, E. Zhang, J. Bittner, and Wonka P. 2008. Visibility-driven mesh analysis and visualization through graph cuts. *IEEE Trans Vis Comput Graph* 14, 6 (2008), 1667–74. https://doi.org/10.1109/TVCG.2008.176.PMID:18989024

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. *CoRR* abs/1605.04797 (2016). arXiv:1605.04797

A PROOF OF STATEMENT IN SECTION 3

For the sake of clarity, we first demonstrate our statement under a slightly stricter hypotesis, and then show how this hypotesis can be relaxed while maintaining the validity of the demonstration.

Hypothesis. For each edge e in the input Σ there exist two triangles t_1, t_2 that share e and $|t_1|$ and $|t_2|$ are not coplanar.

Thesis. If Ω is the polyhedral mesh produced by our algorithm out of $|\Sigma|$ and σ is a triangle in Σ, then there exist n facets of Ω $f_1, \ldots, f_n$ such that $|f_1| \cup \cdots \cup |f_n| \equiv |\sigma|$.

Proof. The interior of a cell in Ω cannot be intersected by $|\sigma|$. This means that $|\sigma|$ is necessarily contained in the union of facets in Ω. We must still prove that such a containment is actually an equivalence. To do this, we show that each of the three bounding edges of $|\sigma|$ coincides with the union of edges in Ω. Formally, let e_σ be an edge of σ, we prove that there exist m edges in Ω such that $|e_1| \cup \cdots \cup |e_m| \equiv |e_\sigma|$. Our hypothesis guarantees that there exists a triangle $\tau \in \Sigma$ such that $e_\sigma \in \partial\tau$ and $|\sigma|$ and $|\tau|$ are not coplanar. $|e_\sigma|$ coincides with the intersection of $|\sigma|$ and $|\tau|$, and it either exists as a 1-simplex in the initial tetrahedrization or its portions $|e_1| \ldots |e_m|$ are all created while splitting facets in Ω. Specifically, each portion $|e_i|$ within a cell is created by intersecting the facets that contain σ and τ respectively in that cell (e_{new} in Algorithm 1). □

Note that our stricter hypothesis prevents $|\Sigma|$ from having non-triangular faces. Indeed, a non-triangular face f would be necessarily realized as the union of realized coplanar triangles, which is not admitted. Nonetheless, we only require that the bounding edges of f exist in Ω, whereas common edges of its composing triangles might be swapped without affecting the realization $|f|$. Therefore, we may reformulate our hypothesis as follows: for each edge e in the input Σ, there exist two triangles t_1, t_2 that share e and either $|t_1|$ and $|t_2|$ are not coplanar or $I(|t_1|) \cap I(|t_2|) = \emptyset$. This new hypothesis admits the existence of coplanar adjacent constraints, but prevents $|\Sigma|$ from having boundaries in the Euclidean topology as we require (i.e. no point in $|\Sigma|$ has a neighborhood homeomorphic to a half-disk).

B DERIVED PREDICATES

All the geometric predicates we use throughout the process are implemented as a composition of the following fundamental predicates:

- `coincident_points_2D`: $\mathbb{F}^2 \times \mathbb{F}^2 \to \{true, false\}$
- `coincident_points_3D`: $\mathbb{F}^3 \times \mathbb{F}^3 \to \{true, false\}$
- `orient2d`: $\mathbb{F}^2 \times \mathbb{F}^2 \times \mathbb{F}^2 \to \{-1, 0, 1\}$
- `orient3d`: $\mathbb{F}^3 \times \mathbb{F}^3 \times \mathbb{F}^3 \times \mathbb{F}^3 \to \{-1, 0, 1\}$

where $\mathbb{F}$ is the set of representable floating point numbers. These four basic predicates are defined hereabove for explicit points, but corresponding definitions exist for any combination of implicit and explicit points. In the remainder, p_{1x}, p_{1y} and p_{1z} denote the first, second and third coordinate of point p_1, whereas p_{1xy} denotes the projection of p_1 on the XY plane, that is, the 2D point obtained by dropping the third coordinate of the 3D point p_1.

B.1 Point misalignment: true if the three points are not collinear.

$$\text{misaligned}(p_1, p_2, p_3) \equiv$$
$$\text{orient2d}(p_{1xy}, p_{2xy}, p_{3xy}) \neq 0 \vee$$
$$\text{orient2d}(p_{1yz}, p_{2yz}, p_{3yz}) \neq 0 \vee$$
$$\text{orient2d}(p_{1zx}, p_{2zx}, p_{3zx}) \neq 0$$

B.2 Point in segment checks (interior only / endpoints included).

$$\text{point\_in\_inner\_segment}(p, v_1, v_2) \equiv$$
$$\neg\ \text{misaligned}(p, v_1, v_2) \wedge$$
$$(v_{1x} < p_x < v_{2x}) \vee (v_{1x} > p_x > v_{2x}) \vee$$
$$(v_{1y} < p_y < v_{2y}) \vee (v_{1y} > p_y > v_{2y}) \vee$$
$$(v_{1z} < p_z < v_{2z}) \vee (v_{1z} > p_z > v_{2z})$$

$$\text{point\_in\_segment}(p, v_1, v_2) \equiv$$
$$\text{point\_in\_inner\_segment}(p, v_1, v_2) \vee$$
$$\text{coincident\_points\_3D}(p, v_1) \vee$$
$$\text{coincident\_points\_3D}(p, v_2)$$

B.3 Intersection check for the interior of coplanar non collinear segments.

$\text{inner\_segments\_cross}(a, b, p, q) \equiv isc_{xy} \vee isc_{yz} \vee isc_{zx}$

where

$isc_{xy} \equiv$
$(\text{orient2d}(p_{xy}, a_{xy}, b_{xy}) \neq 0 \vee \text{orient2d}(q_{xy}, b_{xy}, a_{xy}) \neq 0 \vee$
$\text{orient2d}(a_{xy}, p_{xy}, q_{xy}) \neq 0 \vee \text{orient2d}(b_{xy}, q_{xy}, p_{xy}) \neq 0\ )\wedge$
$\text{orient2d}(p_{xy}, a_{xy}, b_{xy}) = \text{orient2d}(q_{xy}, b_{xy}, a_{xy}) \wedge$
$\text{orient2d}(a_{xy}, p_{xy}, q_{xy}) = \text{orient2d}(b_{xy}, q_{xy}, p_{xy})$

Coplanarity may be verified by checking whether $\text{orient3d}(a, b, p, q) = 0$.

B.4 Point in triangle checks (interior only / boundary included).

`point_in_inner_triangle`$(p, v_1, v_2, v_3) \equiv piit_{xy} \wedge piit_{yz} \wedge piit_{zx}$

where

$piit_{xy} \equiv$
$\text{orient2d}(p_{xy}, v_{2xy}, v_{3xy}) = \text{orient2d}(v_{1xy}, v_{2xy}, v_{3xy}) \wedge$
$\text{orient2d}(p_{xy}, v_{3xy}, v_{1xy}) = \text{orient2d}(v_{1xy}, v_{2xy}, v_{3xy}) \wedge$
$\text{orient2d}(p_{xy}, v_{1xy}, v_{2xy}) = \text{orient2d}(v_{1xy}, v_{2xy}, v_{3xy})$

Coplanarity may be verified by checking whether $\text{orient3d}(p, v_1, v_2, v_3) = 0$.

$$\text{point\_in\_triangle}(p, v_1, v_2, v_3) \equiv$$
$$\text{point\_in\_segment}(p, v_1, v_2) \vee$$
$$\text{point\_in\_segment}(p, v_2, v_3) \vee$$
$$\text{point\_in\_segment}(p, v_3, v_1) \vee$$
$$\text{point\_in\_inner\_triangle}(p, v_1, v_2, v_3)$$

B.5 Intersection check for the interior of a segment and the interior of a triangle which are not coplanar.

$\text{inner\_segment\_crosses\_inner\_triangle}(u_1, u_2, v_1, v_2, v_3) \equiv$
$\text{orient3d}(u_1, v_1, v_2, v_3) \neq 0 \wedge$
$\text{orient3d}(u_2, v_1, v_2, v_3) \neq 0 \wedge$
$\text{orient3d}(u_1, v_1, v_2, v_3) \neq \text{orient3d}(u_2, v_1, v_2, v_3) \wedge$
$\text{orient3d}(u_1, u_2, v_1, v_2) = \text{orient3d}(u_1, u_2, v_1, v_2) \neq 0 \wedge$
$\text{orient3d}(u_1, u_2, v_1, v_2) = \text{orient3d}(u_1, u_2, v_3, v_1) \neq 0$

B.6 Intersection check for the interior of a segment and a triangle which are not coplanar.

$\text{inner\_segment\_crosses\_triangle}(u_1, u_2, v_1, v_2, v_3) \equiv$
$\text{point\_in\_inner\_segment}(v_1, u_1, u_2) \vee$
$\text{point\_in\_inner\_segment}(v_2, u_1, u_2) \vee$
$\text{point\_in\_inner\_segment}(v_3, u_1, u_2) \vee$
$\text{inner\_segments\_cross}(v_2, v_3, u_1, u_2) \vee$
$\text{inner\_segments\_cross}(v_3, v_1, u_1, u_2) \vee$
$\text{inner\_segments\_cross}(v_1, v_2, u_1, u_2) \vee$
$\text{inner\_segment\_crosses\_inner\_triangle}(u_1, u_2, v_1, v_2, v_3)$