Fast Tetrahedral Meshing in the Wild

YIXIN HU, New York University, USA
TESEO SCHNEIDER, New York University, USA
BOLUN WANG, Beihang University, China and New York University, USA
DENIS ZORIN, New York University, USA
DANIELE PANOZZO, New York University, USA

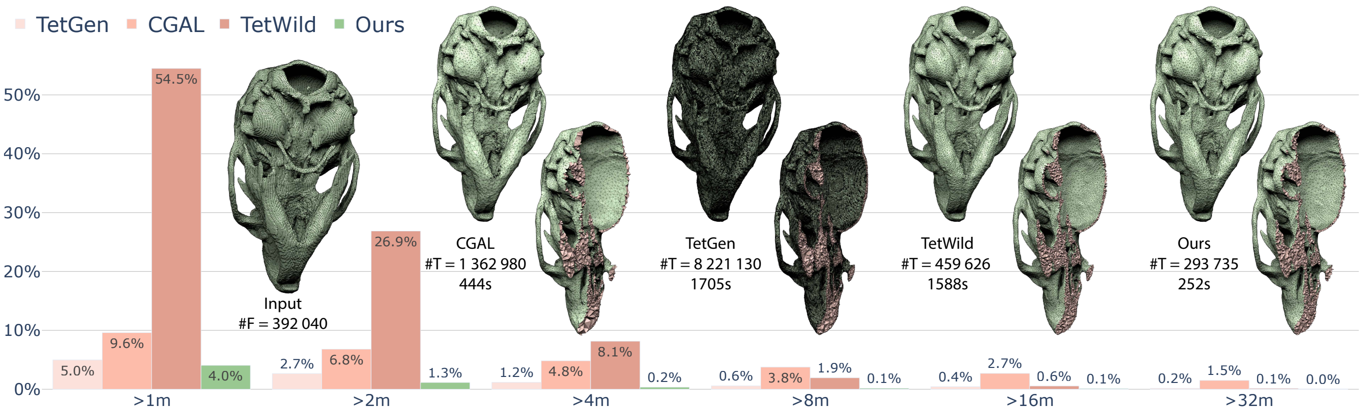


Fig. 1. The bar charts show the percentage of models requiring more than the indicated time for the different approaches over 4 540 inputs (the subset of Thingi10k where all 4 compared algorithms succeed). Our algorithm successfully meshes 98.7% of the input models in less than 2 minutes, and processes all models within 32 minutes. The comparison has been done using the experimental setup of TetWild [Hu et al. 2018] and selecting a similar target resolution for all methods. The CGAL surface approximation parameter has been selected to be comparable to the envelope size used for TetWild and for our method. The images above the plot show a mouse skull model (from micro-CT) tetrahedralized with fTetWild (right) compared with other popular tetrahedral meshing algorithms.

We propose a new tetrahedral meshing method, fTetWild, to convert triangle soups into high-quality tetrahedral meshes. Our method builds on the TetWild algorithm, replacing the rational triangle insertion with a new incremental approach to construct and optimize the output mesh, interleaving triangle insertion and mesh optimization. Our approach makes it possible to maintain a valid floating-point tetrahedral mesh at all algorithmic stages, eliminating the need for costly constructions with rational numbers used by TetWild, while maintaining full robustness and similar output quality. This allows us to improve on TetWild in two ways. First, our algorithm is significantly faster, with running time comparable to less robust Delaunay-based tetrahedralization algorithms. Second, our algorithm is guaranteed to produce a valid tetrahedral mesh with floating-point vertex coordinates, while TetWild produces a valid mesh with rational coordinates which is not guaranteed to be valid after floating-point conversion. As a trade-off, our algorithm no longer guarantees that all input triangles are present in the output mesh, but in practice, as confirmed by our tests on the Thingi10k dataset, the algorithm always succeeds in inserting all input triangles.

CCS Concepts: • **Mathematics of computing → Mesh generation**.

Additional Key Words and Phrases: Mesh Generation, Tetrahedral Meshing, Robust Geometry Processing

ACM Reference Format:
Yixin Hu, Teseo Schneider, Bolun Wang, Denis Zorin, and Daniele Panozzo. 2020. Fast Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 39, 4, Article 117 (July 2020), 18 pages. https://doi.org/10.1145/3386569.3392385

Authors' addresses: Yixin Hu, New York University, USA, yixin.hu@nyu.edu; Teseo Schneider, New York University, USA, teseo.schneider@nyu.edu; Bolun Wang, Beihang University, China, New York University, USA, wangbolun@buaa.edu.cn; Denis Zorin, New York University, USA, dzorin@cs.nyu.edu; Daniele Panozzo, New York University, USA, panozzo@nyu.edu.

1 INTRODUCTION

Tetrahedral meshes are commonly used in graphics and engineering applications. Tetrahedral meshing algorithms usually take a 3D surface triangle mesh as input and output a volumetric tetrahedral mesh filling the volume bounded by the input mesh. Traditional tetrahedral meshing algorithms have strong assumptions on the input, requiring it to be a closed manifold, free of self-intersections and numerical unstably close elements, and so on. However, those assumptions often do not hold on imperfect 3D geometric data in the wild.

The recently proposed Tetrahedral Meshing in the Wild (TetWild) [Hu et al. 2018] algorithm makes it possible to reliably tetrahedralize triangle soups by combining exact rational computations with a geometric tolerance to automatically address self-intersections, gaps and other imperfections in the input. The algorithm imposes no formal assumptions on the input mesh and is extremely robust, opening the door to automatic processing and repair of large collections of 3D models.

However, TetWild has two downsides, one theoretical and one practical. The theoretical downside is that it does not guarantee the generation of a floating point tetrahedral mesh: the algorithm internally uses rational numbers, which are then converted to floating point in the process of mesh optimization. While quite unlikely, it is possible that the mesh optimization stage will be unable to round all coordinates of the output mesh to floating point. The practical downside is the long running time compared with Delaunay-based tetrahedralization algorithms.

We introduce fTetWild, a variant of the TetWild algorithm addressing both these limitations while keeping the important properties of TetWild: robustness to imperfect input and ability to batch process large collections of models without parameter tuning, while producing high-quality tetrahedral meshes. Differently from TetWild, which generates a polyhedral rational mesh inserting all triangles at once, we start from a floating point tetrahedral mesh, insert one input triangle at a time and re-tetrahedralize locally, rejecting the operations producing inverted or degenerate elements. We then improve the quality of the mesh iteratively, and attempt to insert the rejected triangles into a higher quality mesh, which is less likely to fail.

Our algorithm always guarantees to generate a valid tetrahedral mesh with floating point vertex positions, independently from the stopping criteria or quality of the mesh. It might fail to insert few input triangles leading to a “less accurate” boundary preservation, however we never observe this behavior in our experiments. The new algorithm can be implemented using floating point constructions, avoiding the overhead associated with rational numbers. The use of floating point numbers also simplifies parallelization, which we use during mesh optimization to further improve the running time on large models. Consequently, our new algorithm is significantly faster than TetWild, with running times comparable to Delaunay-based algorithms (Figure 1), while providing the stronger guarantee of always producing a valid floating point output at the same time.

These improvements make fTetWild more practical than TetWild not only for volumetric meshing problems, but also for mesh repair and approximate mesh arrangements. By combining fTetWild and some elements of [Zhou et al. 2016], we obtain an approximate mesh arrangement algorithm for input triangle soups guaranteed to produce a valid floating point output. In comparison, the original algorithm presented in [Zhou et al. 2016] may fail to produce a floating-point output due to impossibility of rounding after the rational-arithmetic arrangement computation.

We demonstrate the robustness and practical utility of our algorithm by computing tetrahedral meshes on the Thingi10k dataset (10 000 models) and computing approximate Booleans. We use the generated tetrahedral meshes to solve elasticity, fluid flow, and heat diffusion equations on complex geometric domains. The complete implementation of fTetWild is provided in the additional material, together with scripts to reproduce all results in the paper.

2 RELATED WORK

We briefly review the literature on tetrahedral meshing (Section 2.1), with an emphasis on envelope-based techniques, and we refer to [Cheng et al. 2012; Shewchuk 2012] for a more detailed overview of the topic. We also review mesh repair and mesh arrangement algorithms (Section 2.2), since our technique can be also used in these settings to enable processing of imperfect geometry.

2.1 Tetrahedral Meshing

Delaunay Meshing. The most studied and most widely used algorithms to generate tetrahedral meshes are based on the Delaunay condition [Alliez et al. 2005a; Aurenhammer 1991; Aurenhammer et al. 2013; Bishop 2016; Boissonnat et al. 2002; Boissonnat and Oudot 2005; Busaryev et al. 2009; Chen and Xu 2004; Cheng et al. 2008, 2012; Chew 1989, 1993; Cohen-Steiner et al. 2002; Dey and Levine 2008; Du and Wang 2003; George et al. 2003; Jamin et al. 2015; Murphy et al. 2001; Remacle 2017; Ruppert 1995; Sheehy 2012; Shewchuk 1996, 1998, 1999, 2002a; Si 2015; Si and Gartner 2005; Si and Shewchuk 2014; Tournois et al. 2009; Weatherill and Hassan 1994]. These methods are efficient and are widely used in commercial software. They can be applied to either point clouds inputs, or to tessellate the interior of manifold non self-intersecting meshes with no degenerate faces, but are not designed to deal with imperfect input, and, as a consequence, these techniques fail on a significant fraction of data *in the wild* [Hu et al. 2018]. We provide a direct comparison with TetGen [Si 2015], the most commonly used code in this category, which builds on the techniques developed in [George et al. 2003; Weatherill and Hassan 1994], and CGAL [Wein et al. 2018] in Section 4.

Grid Methods. An alternative approach is the use of a background grid as a starting point [Baker et al. 1988; Bern et al. 1994; Bridson and Doran 2014; Bronson et al. 2013; Doran et al. 2013; Labelle and Shewchuk 2007; Molino et al. 2003; Yerry and Shephard 1983]. These algorithms fill the entire bounding box of the input with a regular lattice or with a hierarchical space partitioning, optionally intersect the background mesh with the input surface, and then discard the elements outside of the input. These methods are simpler and more robust than Delaunay methods, but still struggle with imperfect input geometry, and create high-quality elements only in the interior of the mesh, where the background mesh is preserved exactly. However, placing badly shaped triangles on the boundary is problematic for many applications. Our algorithm borrows the idea of a background mesh from these methods, but inserts the elements incrementally, interleaving mesh optimization stages to ensure that the final quality of the mesh is uniformly high.

Front-Advancing Methods. Another family of methods starts from the boundary, and inserts one element at a time, growing the volumetric mesh (i.e. marching in space), until the entire volume is filled [Alauzet and Marcum 2014; Cuilliere et al. 2013; George 1971; Haimes 2014; Peraire et al. 1987; Sadek 1980]. These methods create high quality elements close to the boundary, but introduce many corner cases in the interior regions where the fronts meet, lowering the quality of the elements and making a robust implementation challenging.

Envelope Meshing. All methods discussed above assume a valid, manifold, non self-intersecting boundary input mesh, and are not designed to handle the imperfections which are common in real-world CAD and scanned data. This issue has been tackled for surface meshes in Mandad et al. [2015], by creating a surface approximation within a tolerance volume using a modified Delaunay refinement process, and for implicit surfaces [Shen et al. 2004], using different filter radius. A similar idea has been exploited for volumetric meshing in TetWild [Hu et al. 2018], and its 2D counterpart TriWild [Hu et al. 2019]. The main idea of this work is to combine exact computation, using a hybrid kernel similar to [Attene 2017], and a surface envelope [Hu et al. 2017], which allows the resulting mesh to approximate the input instead of reproducing it exactly. Our method closely follows [Hu et al. 2018], but we design our algorithm to avoid the use of exact computation. We compare the two techniques in Section 4.

Mesh Improvement. Many algorithms have been proposed to improve the quality of an existing tetrahedral mesh by displacing vertices or changing the local connectivity [A. Freitag and Ollivier-Gooch 1998; Alexa 2019; Alliez et al. 2005b; Canann et al. 1996, 1993; Chen and Xu 2004; Faraj et al. 2016; Feng et al. 2018; Hu et al. 2018; Klingner and Shewchuk 2007; Lipman 2012]. Our method relies on the algorithm proposed in Hu et al. [2018], which uses a set of local operations to optimize the conformal AMIPS energy [Fu et al. 2015; Rabinovich et al. 2017]. We parallelized some of the steps of that algorithm (Section 3), which is easier in our case since we only have floating point coordinates.

2.2 Applications

Mesh Repair. Since our algorithm can be used for mesh repair, we review the most recent works on this topic, and we refer to [Attene et al. 2013] for a complete overview.

MeshFix [Attene 2010, 2014] detects problematic regions in triangle meshes, and uses a set of local operations to heal them. The tool is very effective, but due to its use of a greedy algorithm it might delete large parts on a mesh. The most recent mesh repair technique has been introduced in [Hu et al. 2018]: the algorithm generates a tetrahedral mesh and discards the generated tetrahedra, only keeping the boundary surface. While simple and effective, this techniques is computationally expensive, and thus only usable in batch processing applications. Our algorithm can be used in the same way, but its higher efficiency makes it more practical. We also propose a simple modification to the surface mesh extraction procedure to guarantee a manifold output.

Booleans and Mesh Arrangements. Many approaches to performing Boolean operations on meshes were proposed, with some methods emphasizing robustness, other methods aiming to produce exact results, and another set prioritizing performance. In most cases, non-trivial assumptions are made on the input meshes: most commonly, these are required to be closed; in other cases, no self-intersections are allowed, or most restrictively vertices may be assumed in general position.

CGAL, one of the most robust implementations of Boolean operations available [Granados et al. 2003], relies on exact arithmetic, and uses a very general structure of Nef polyhedra [Bieri and Nef 1988] to represent shapes. This allows one to obtain exact Boolean results in degenerate cases (e.g., when the result is a line or a point). At the same time, the assumptions on the input are quite restrictive: the surfaces need to be closed and manifold (although the latter constraint could be eliminated).

Another approach to achieve robustness at the expense of accuracy, is to convert input meshes to level sets e.g. by sampling a signed distance function for each object [Museth et al. 2002] and perform all operations on the level set functions. The obvious disadvantage of these methods is that their accuracy is limited by the resolution of the grid; the original mesh geometry is lost, and it is non-trivial to maintain even explicitly tagged features. These downsides are partially addressed by adaptive [Varadhan et al. 2004] and hybrid [Pavic et al. 2010; Wang 2011; Zhao et al. 2011], the latter preserving mesh geometry away from intersections. All these methods rely on well-defined signed distance function, i.e., assume that input meshes are closed, and may still significantly alter the input geometry near intersections. [Schmidt and Singh 2010] does not use a signed distance function, but resembles these methods, in that it removes existing geometry near intersections and replaces it by new mesh connecting the two objects and approximating the result of the Boolean. Binary Space Partitioning (BSP) based methods, starting from [Naylor et al. 1990; Thibault and Naylor 1987] are closest in their approach to ours. Using BSP trees preserves the input more accurately, and, along the way, creates a volume partition. However, it is prone to errors due to numerical instability of intersection calculations, and, due to global intersections of triangle planes, performs excessive refinement. [Bernstein and Fussell 2009] addresses the issue of non-robustness by using exact predicates, and [Campen and Kobbelt 2010] reduces refinement by creating localized BSP trees in an octree. Examples of highly efficient but non-robust software for computing Booleans are [Douze et al. 2015], [Barki et al. 2015], and [Bernstein 2013]. A general position assumption is often required explicitly or implicitly. In [Zhou et al. 2016] a robust way to compute *mesh arrangements* is introduced, with Boolean operations as an application. Robustness is achieved by using rational numbers for critical computations. To perform Booleans the mesh is required to be Positive Winding Number (PWN), which does not always hold in meshes in the wild [Zhou and Jacobson 2016].

Sheng et al. [2018a,b] use a combination of plane-based and vertex-based representations of mesh faces to improve robustness of basic operations needed for Boolean operations performed in floats. Their method achieves very high efficiency, at the expense of somewhat lower robustness compared to the state of the art [Granados et al. 2003; Zhou et al. 2016]. Their method assumes that the input meshes enclose solids and are free of self-intersections. [Magalhães et al. 2017] is an efficient technique using simulation-of-simplicity techniques to handle general intersections between objects, self-intersections or holes are not handled. [Paoluzzi et al. 2017] considers a general problem of arrangements of complexes in 2D and

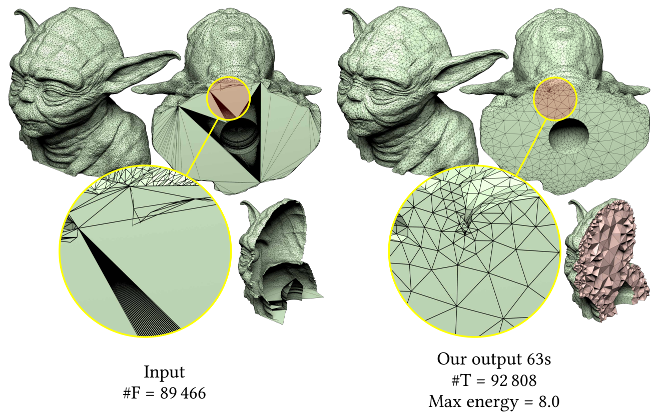


Fig. 2. Example of an input surface mesh with self-intersections and a bad triangulation on the base. fTetWild converts this model into a high-quality tetrahedral mesh.

3D, presenting a theoretical general merge algorithm, but do not consider the questions of robustness and handling imperfect inputs.

Compared to existing methods, the application of fTetWild to Boolean operations is more conservative, in terms of mesh geometry changes and refinement, compared to level set and BSP-based methods, while maintaining their level of robustness. At the same time, thanks to the geometric tolerance, fTetWild is capable of eliminating near-degenerate or overly refined triangles in the input model, which [Zhou et al. 2016] cannot do. We also make fewer assumptions on the inputs, allowing gaps, self-intersections, and degeneracies.

3 METHOD

fTetWild takes as input a 3D triangle soup (i.e., a set of arbitrarily connected, potentially intersecting triangles with vertices potentially duplicated) whose vertices are represented in floating-point coordinates, representing the surface of an object. The algorithm has two user-defined parameters: target edge length ℓ, and envelope size ϵ. The ϵ-envelope represents the maximal deviation from the input surface the user is willing to accept. For instance, in additive manufacturing applications ϵ can be the machining precision. It outputs a volumetric tetrahedral mesh of the axis-aligned bounding box containing the input, with floating-point vertex coordinates, whose elements are (1) non-inverted (i.e., positive volume) and (2) with some faces approximating the input soup within a user-defined ϵ-envelope. fTetWild makes *no assumptions* on the input triangle soup and it is robust when handling imperfect input with self-intersections or small gaps. This robustness is achieved by allowing the faces of the tetrahedral mesh corresponding to the input surface to move inside an ϵ-envelope (up to ϵ far from the input): self-intersections, degenerate and near-degenerate faces and gaps contained in the envelope are automatically removed when combined with proper mesh improvement operations (Figure 2).

The output tetrahedral mesh can be optionally post-processed to remove the tetrahedra outside the input surface (Section 3.6). We note that this optional stage relies on the input triangles (geometry and orientation) to represent a valid volume. This heuristic filtering might fail, for instance if the input is far from a closed surface (e.g., a half-sphere) fTetWild will generate a valid tetrahedral mesh with faces conforming to the input, but the filtering stages might discard all tetrahedra since the “outside” region is not well defined.

Similarities and Differences to Existing Face Insertion Algorithms. The main challenge tackled in many existing tetrahedral meshing algorithm is the preservation of the input faces, which can be exact or approximate. One of the best known algorithms exactly preserving the input faces is [George et al. 2003], which proposed to subdivide a background mesh by intersecting it with input faces. This procedure can, however, introduce inverted elements due to floating-point rounding, which then need to be untangled, a difficult task for which no robust algorithm currently exists. A robust solution is proposed in TetWild [Hu et al. 2018], that initially inserts the faces exactly using rational numbers to avoid numerical problems, but is then forced to allow them to move to round the rational coordinates back to floating point. Although robust and conservative, this solution relies on expensive rational constructions, and it is not guaranteed to succeed in the rounding phase.

Our method follows an approach similar to TetWild (see Appendix A for a brief description of the algorithm), enabling small and controlled deviations from the input surface, but sidesteps the need for constructing a rational mesh, always using floating-point coordinates, while inheriting the robustness of TetWild. Algorithmically, there are three major differences:

(1) fTetWild preserves the input faces by inserting one input triangle at a time into an existing background tetrahedral mesh. To facilitate the insertion it relaxes the insertion with a snapping tolerance (relatively larger than floating point machine precision) which is only possible thanks to the ϵ-envelope.
(2) fTetWild always tetrahedralizes the region affected by the newly inserted face by looking up a pre-computed table and always maintains a valid inversion-free tetrahedral mesh (using exact predicates).
(3) fTetWild represents the vertices using only floating point coordinates, reducing the running time and memory consumption.

We note that inserting a triangle might fail due to limitations of the floating-point representation. For instance, the inserted face can be arbitrarily close to one of the existing vertices and the insertion will introduce a tetrahedron with a volume numerically equal to zero. In this scenario, we rollback the problematic operation, mark the problematic face as un-inserted, iteratively perform mesh improvement operations on the whole mesh, and try to insert the face again when the mesh quality has increased. This procedure shows the only possible failure of fTetWild: the impossibility of adding some input faces. While this is indeed possible, it never manifested in our experiments. Note that even if some faces could not be inserted, fTetWild still outputs a valid mesh conforming to all other faces.

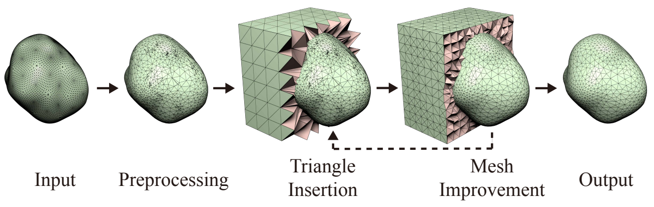


Fig. 3. Overview of our algorithm. From left to right, the input mesh is simplified, a background mesh is created and the input faces are inserted, the mesh quality is optimized, and the final result is obtained by filtering the elements lying outside the input surface.

3.1 Algorithm Overview

Our algorithm consists of four phases (Figure 3): (1) the input triangle soup is simplified while ensuring it stays in the ϵ-envelope of the input (Section 3.3), (2) a background mesh is generated and the triangles are iteratively inserted into it, if the insertion does not introduce inverted elements (Section 3.4), (3) the mesh is improved using local operations (Section 3.5) and at the end of every three improvement iteration we re-attempt the insertion of input triangles that could not be inserted at phase 2, (4) the mesh elements are optionally filtered to remove the elements outside the surface or to perform Boolean operations (Section 3.6).

During the whole procedure we ensure that the tetrahedral mesh remains *valid*, that is, we ensure that (1) each element has positive volume (checked using exact predicates [Lévy 2019; Shewchuk 1997]) and (2) all successfully inserted triangles, from now on called the *tracked surface*, stay inside the ϵ-envelope of the input.

Throughout the algorithm, we consider a distance between two points zero if it is below a numerical tolerance ϵ_{zero}. Similarly, we use ϵ^2_{zero}, ϵ^3_{zero} for areas and volumes respectively. We found that the performance of the algorithm are not heavily affected by this tolerance, as long as $\epsilon_{\text{zero}} > 10^{-20}$: in our experiments we used $\epsilon_{\text{zero}} = 10^{-8}$.

3.2 Envelope

We use the envelope definition and the algorithms introduced in [Hu et al. 2018] to build the envelope and check if a triangle is contained in it. In particular, testing if a triangle is contained within the envelope is done by sampling the input triangle and checking if the samples are all within a slightly smaller envelope with the sampling error conservatively compensated [Hu et al. 2018].

3.3 Preprocessing

We use the same preprocessing procedure proposed in [Hu et al. 2018] for simplifying the input: we merge vertices closer than ϵ_{zero} and collapse an edge (by merging one endpoint to the other) if: (1) it is a manifold edge (has no more than two incident triangles) and vertex-adjacent edges are also manifold, and (2) the collapsing operation does not move triangles outside a *smaller* envelope of size $\epsilon_{\text{prep}} < \epsilon$. At this stage, we use $\epsilon_{\text{prep}} = 0.8\epsilon$ since this value gives space for snapping in triangle insertion (Section 3.4), and prevents vertices to be too close to the boundary of the envelope, thus leaving more freedom for surface vertices to move in the mesh improvement stage (Section 3.5). On our dataset, we observed that changing this parameter has a minor impact on the running time and negligible effect on the output when in the range 0.7 to 0.999. Note that it cannot be set to 1 because it will prevent snapping (Section 3.4). We use the value 0.8 since it is far from the bounds of this range.

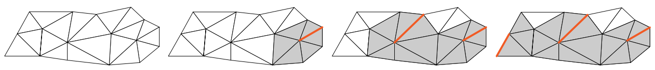


Fig. 4. A two-dimensional example of edge coloring. From left to right: one parallel-independent edge is selected (in red), its vertex-adjacent triangles are colored in black. The algorithm proceeds until no edges can be marked (last image). In the end, all red edges can be safely collapsed in parallel without affecting other red edges.

Since the preprocessing step is computationally expensive, due to the envelope containment checks, we propose a basic parallelization strategy which leads, on average, to a 4x speedup of the preprocessing step when using 8 cores. Our parallel edge collapsing procedure uses a serial 2-coloring pass over all input faces. We mark all input triangles white in the initial stage. Then, iteratively, we mark all edges as *parallel-independent* if all its vertex-adjacent triangles are white, and then mark these triangles black (Figure 4). At this point, we can safely collapse all marked parallel-independent edges in parallel. We iterate this procedure until we are unable to remove more than 0.01% of the original input vertices.

3.4 Incremental Triangle Insertion

3.4.1 Background Mesh Generation. The triangle insertion stage requires a background mesh (which does not necessarily conform to the input triangles) which we create using Delaunay tetrahedralization [Lévy 2019] on the vertices from the preprocessing stage. Since we allow the surface to move within an ϵ-envelope, we generate the background mesh for a bounding box 2ϵ larger than the bounding box of the input vertices. Similarly to TetWild, additional points are added uniformly inside the box and at least ϵ away from the input faces before Delaunay tetrahedralization to obtain more uniformly-shaped initial elements. More precisely, the additional points are added in a regular grid with spacing of $d/20$ (where d is the diagonal of bounding box of the input mesh), skipping inserting the additional points with distance to the input faces smaller than ϵ.

3.4.2 Single Triangle Insertion. The key component of our algorithm is three-stage procedure for inserting one triangle T into a valid tetrahedral mesh M, adding new vertices and tetrahedra, and adjusting mesh connectivity, to minimize the number of insertion failures and number of badly shaped tetrahedra created by insertion. Note that we do not insert degenerate triangles. Our algorithm uses ideas from marching tetrahedra [Doi and Koide 1991] and other tetrahedralization methods [George et al. 2003; Weatherill and Hassan 1994], as well as marching cubes [Lorensen and Cline 1987]. It consists of the following steps: (1) Find the set $\mathcal{T}_I$ consisting of the tetrahedra of M that triangle T cuts, as defined below; (2) Compute the intersection points of the plane spanned by T (denoted as P) and the edges of the tetrahedra in $\mathcal{T}_I$; (3) Subdivide all cut tetrahedra using a connectivity pattern from a pre-computed *tet-subdivision table*.

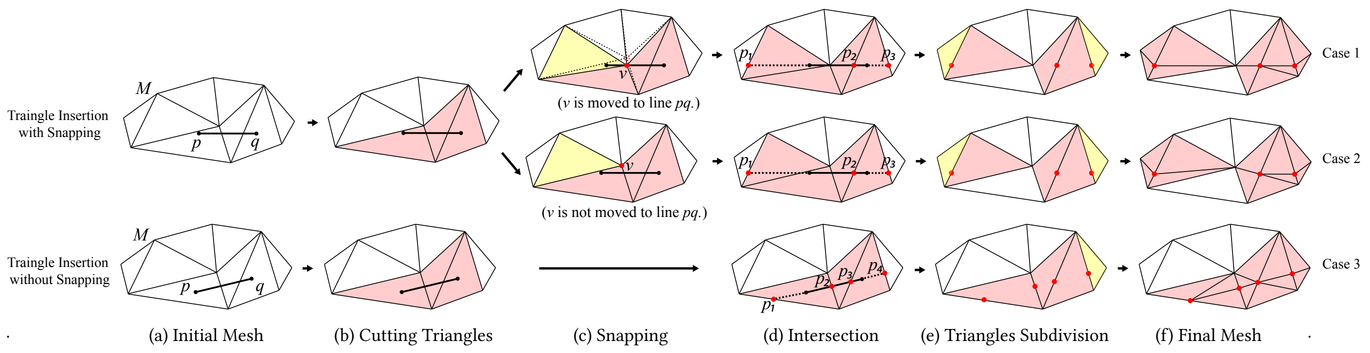


Fig. 5. Segment insertion into a triangle mesh (a 2D analog of triangle insertion) with and without snapping. (a) Insertion of segment pq into mesh M. (b) Identification of cut triangles $\mathcal{T}_I$ (in red). (c) Snapping vertex v to line pq and updating $\mathcal{T}_I$, where v is δ-close to pq. v is moved to its closest points on line pq if this does not invert any elements of M (case 1). Vertex v is added in $\mathcal{V}_\delta$ both if v is moved (case 1) or not (case 2). The yellow triangle is added to $\mathcal{T}_I$. (d) Computing the intersection of line pq with the edges of $\mathcal{T}_I$ (points p_1, p_2, p_3). (e) Triangles requiring subdivision (shown in red). (f) The final mesh after subdivision.

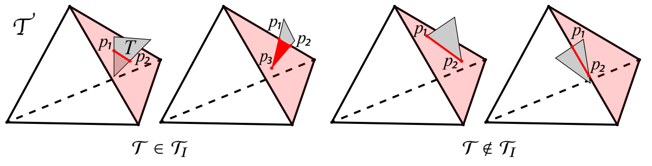

$\mathcal{T}$
p_1
p_2
p_3
T
$\mathcal{T} \in \mathcal{T}_I$
$\mathcal{T} \notin \mathcal{T}_I$

Fig. 6. Examples of tetrahedra $\mathcal{T}$ included into, or excluded from, $\mathcal{T}_I$. The intersections are shown in red. Left two: T intersects a face of $\mathcal{T}$ at a segment ($[p_1, p_2]$) or a polygon ($[p_1, p_2, p_3]$) that contains interior points of both T and the intersected face of $\mathcal{T}$. In this case, we put $\mathcal{T}$ into $\mathcal{T}_I$. Right two: T intersects a face of $\mathcal{T}$ (in light red) at a segment ($[p_1, p_2]$) that does not contain any interior points of either T or the intersected face of $\mathcal{T}$. In this case, we do not put $\mathcal{T}$ into $\mathcal{T}_I$.

These patterns guarantee that a valid tetrahedral mesh connectivity is maintained.

Finding Cut Tetrahedra. We first define that object A *cuts though* object B if their intersection contains interior points of both A and B. We say that triangle T *cuts* tetrahedron $\mathcal{T}$ if (1) it is completely contained inside $\mathcal{T}$, or (2) it cuts through at least one face of $\mathcal{T}$ (Figure 6). We initialize $\mathcal{T}_I$ to be the set of the tetrahedra of M that T cuts. Note that this set will be iteratively expanded by the algorithm.

We use exact predicates [Lévy 2019; Shewchuk 1997] for checking if a triangle is contained inside a tetrahedron. To detect if one triangle cuts through another, we combine the exact predicates with the algorithm [Guigue and Devillers 2003]. The use of predicates ensures topological correctness when using floating-point coordinates.

Plane-Tetrahedra Intersection. To insert a triangle T defining a plane P into $\mathcal{T}_I$ (Figure 5(c)(d)), we need to ensure that after the insertion: (1) for every point $p \in T$ there is a face F of the refined tets in the set $\mathcal{T}_I$ such that $\min_{q \in F} \|p-q\| < \delta$; (2) the projection of faces F in $\mathcal{T}_I$, that are within the distance δ from T to the plane P covers T. We call sets with these properties *covers* of T. We allow the cover of triangles to deviate from P. This is crucial for robustly inserting triangles using floating point computations: without it, we observe a significantly higher running time due to more insertion failures,

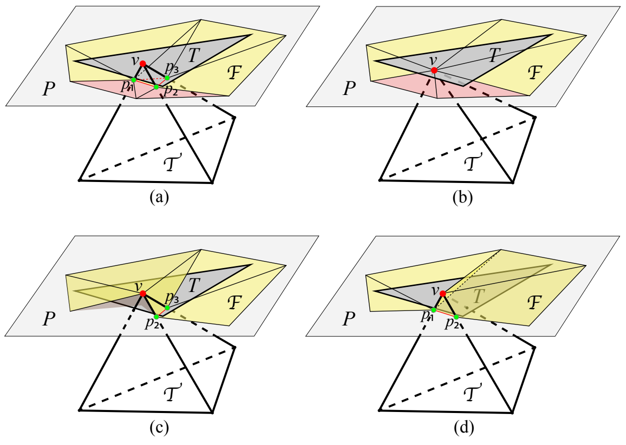


Fig. 7. Plane P intersects $\mathcal{T}_I$ ($\mathcal{T} \in \mathcal{T}_I$). (a) The faces of $\mathcal{F}$ (marked in yellow) are the covering of triangle T. (b) Snapping v to its closest point on P and expanding $\mathcal{F}$ to include red triangles makes $\mathcal{F}$ safely covering T. (c) Snapping a boundary vertex p_1 of $\mathcal{F}$ to v changes the area of $\mathcal{F}$ and make it not covering T. (d) Snapping an interior vertex p_3 of $\mathcal{F}$ to v does not change the boundary of $\mathcal{F}$: $\mathcal{F}$ is still covering T.

which leads to additional iterations of mesh optimization. Also, more faces remain uninserted in the final output. For the first pass of triangle insertion (i.e., before any mesh improvement is done), we use a larger $\delta = \max(\epsilon_{\text{zero}}, 10^{-3}\epsilon)$, while for all subsequent passes we reduce it to $\delta = \epsilon_{\text{zero}}$.

We start with the idealized case of infinite-precision arithmetics. In this case, we can easily realize the covering of T by intersecting all the faces of the tetrahedra in $\mathcal{T}_I$ with plane P. This generates a planar polygonal mesh $\mathcal{F}$ on P covering T and the maximal distance from T to $\mathcal{F}$ is zero (Figure 7 (a)). The vertices of $\mathcal{F}$ are intersection points of P and edges of $\mathcal{T}_I$.

When the floating point representation is used to represent the coordinates of vertices, round-off error may result in degenerate or inverted tetrahedra. Our approach is to reject insertion in these

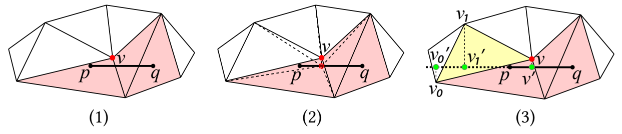


Fig. 8. 2D illustration of step (1) to (3) of snapping when inserting segment $[p, q]$. The cut triangles in $\mathcal{T}_I$ are marked in red. (1) Vertex v is within δ distance to segment $[p, q]$. (2) Check the effect of moving v to line $[p, q]$. (Vertex v cannot be moved to $[p, q]$ in this case, because a triangle reverts its orientation.) (3) One-ring triangle $[v, v_0, v_1]$ (marked in yellow) of v is intersecting with line $[p, q]$, and the projection of its edges $[v, v_0]$, $[v, v_1]$ on line $[p, q]$, segment $[v', v_0']$, $[v', v_1']$, intersect with segment $[p, q]$.

cases. However, to minimize the number of triangles that have to be rejected, we either snap vertices of the tetrahedral mesh M to P (Figure 7 (b)) or snap intersection points to vertices of M (Figure 7 (c)(d)). Moving a vertex v of $\mathcal{T}_I$ (thus changing M) changes the cover $\mathcal{F}$, because P intersects the edges of $\mathcal{T}_I$ in different locations. As no intersection of P with an edge of $\mathcal{T}_I$ disappears (at most, it may move to an endpoint), and no new intersections appear (other than at the endpoints shared with already intersected edges), the connectivity of $\mathcal{T}_I$ can be viewed as unchanged, possibly with some zero-length edges. We can view snapping vertices of $\mathcal{T}_I$ to P as a deformation of $\mathcal{F}$, keeping it on plane P. If the affected vertices are in the interior of $\mathcal{F}$, $\mathcal{F}$ still covers T since the boundary of $\mathcal{F}$ does not change. However, if moving v changes the boundary of $\mathcal{F}$, the covering might be invalidated (Figure 7 (b)). In this case, *before* moving a boundary vertex, we extend $\mathcal{T}_I$ by adding its 1-ring neighbourhood, intersect it with P, and extend $\mathcal{F}$ accordingly. We repeat this process until all affected vertices are in the interior.

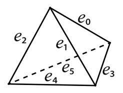
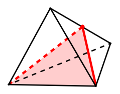
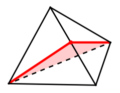
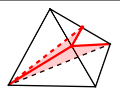
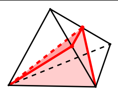
Moving the point v to the plane P might not always be possible, since it could invert some tetrahedra in M. In these cases, instead of moving v to P, we deform $\mathcal{F}$ by moving some vertices of $\mathcal{F}$ to v, which is at δ distance from P by definition (Figure 7 (c)(d)). Similarly to the previous case, this operation can only be applied on interior vertices of $\mathcal{F}$. We thus extend $\mathcal{T}_I$ if this operation affects vertices on the boundary of $\mathcal{F}$.

In practice, we never explicitly compute $\mathcal{F}$ on the plane P since it is uniquely defined by the intersection points (Appendix E), but instead use the following 4 steps, that directly determine the vertices of $\mathcal{F}$ (the faces of $\mathcal{F}$ are obtained by table-based refinement of $\mathcal{T}_I$).

(1) Find all vertices of the tetrahedra in $\mathcal{T}_I$, with distance to P smaller than δ and put them in $\mathcal{V}_\delta$ (e.g., vertex v in Figure 8(1)).
(2) Move vertices in $\mathcal{V}_\delta$ to their closest points on P if it does not invert any elements of M (Figure 8(2)).
(3) For each vertex in $\mathcal{V}_\delta$, add all of its vertex-adjacent tetrahedra to $\mathcal{T}_I$ if these are cut by P and have faces covering T (i.e., the projection of the face to P intersects with T). For example, $[v, v_0, v_1]$ in Figure 8(3) is added.
(4) Repeat steps (1) to (3) until no more new tetrahedra are added to $\mathcal{T}_I$.

Table-based Tetrahedron Subdivision. All tetrahedra sharing the edges of tetrahedra in $\mathcal{T}_I$ cut by P are subdivided into sub-tetrahedra according to the *tet-subdivision* table. Note that this set of tetrahedra usually contains some tetrahedra from $\mathcal{T}_I$ (red elements in Figure 5(e)) and some neighboring tetrahedra of $\mathcal{T}_I$ (yellow elements in Figure 5(e)).

Table 1. A subset of the tet-subdivision table, the complete table is provided in the additional material. The first row corresponds to the case of a tetrahedron without cut edges, the second and third to the case of one cut edge, e_0 and e_1 respectively, and the last row to two cut edges e_0 and e_1. All tetrahedra shown in the table have same edge label.

I \ **II**	0	1	…
$0_{(10)} = 000000_{(2)}$			…
$1_{(10)} = 000001_{(2)}$			…
$2_{(10)} = 000010_{(2)}$			…
$3_{(10)} = 000011_{(2)}$			…
⋮	⋮	⋮	⋱

Since an edge can have at most one intersection point with P, the decomposition of the subdivided tetrahedron is largely (but not entirely) determined by which edges are cut. (An edge will be cut if it intersects with P and neither endpoints are snapped.) We record all possible decompositions of a tetrahedron in a subdivision table, indexed by *primary cut indices* and *secondary cut indices*. The primary index (I) (Table 1), is a binary string, indicating which edges are cut. If two edges on a face are cut (3 edges of a face cannot be cut at the same time), there are two possible triangulations of this face and also multiple decompositions of the tetrahedron; the secondary index (II) is the number of a specific decomposition (Table 1). A primary index paired with a secondary index retrieves a unique decomposition of a tetrahedron.

For an oriented tetrahedron $\mathcal{T}$, there are $2^6 = 64$ combinations of possible intersection points on its edges, but not all 64 combinations can be practically realized. A direct enumeration shows that the following edge-cut configurations are impossible: (1) $\mathcal{T}$ has six cut edges, (2) $\mathcal{T}$ has five cut edges, (3) $\mathcal{T}$ has 4 cut edges, and 3 of them are on the same face, and (4) $\mathcal{T}$ has 3 cut edges on the same face. In total, there are $\binom{6}{6} + \binom{6}{5} + 3 \cdot 4 + 4 = 23$ impossible edge-cut configurations.

The remaining 41 realizable edge-cut configurations cover all subdivision cases and we can categorize them into 7 symmetry classes (Figure 9). Five of them were discussed in [Schweiger and Arridge 2016] and used for a tetrahedron cut by a plane. We need 2 extra configurations (Figure 9, (4) and (6)) for subdividing the neighboring tetrahedra of $\mathcal{T}_I$ with only certain edges cut by P (yellow elements in Figure 5(e)).

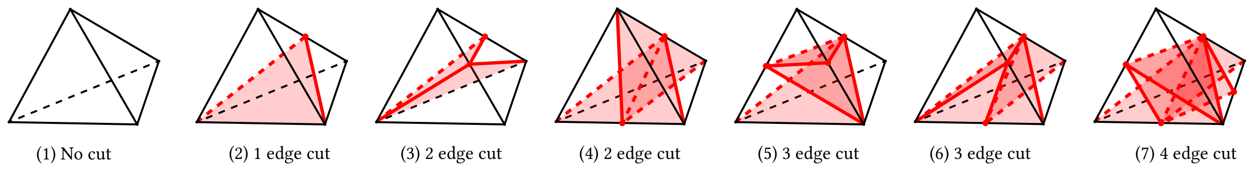
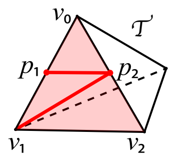


Fig. 9. 7 symmetry classes of edge-cut configurations. (4) and (6) can only happen on neighboring tetrahedra of $\mathcal{T}_I$ with only certain edges cut by P.

We retrieve a list of decompositions of $\mathcal{T}$ corresponding to a primary index; we now need to select a secondary index corresponding to a decomposition that preserves the validity of mesh M after the subdivision, that is, we want M to have a valid topology and no inverted tetrahedra. To ensure a valid topology, two adjacent subdivided tetrahedra must have the same triangulation on the shared face. We set a rule for choosing such triangulation using only the global ordering of the vertices of M. For a face $[v_0, v_1, v_2]$ of tetrahedra $\mathcal{T}$ with two intersection points p_1, p_2 on it (see inset), we select the triangulation containing the edge $[p_2, v_1]$ if the unique integer label of vertex v_1 is larger than the one of v_2. Otherwise, we select the triangulation containing the edge $[p_1, v_2]$. This simple rule completely identifies a secondary index and preserves the topology of the mesh. For completeness, we remark that some configurations might require additional vertices (Appendix C). However, our rule automatically excludes them. We attach the visualization of the tet-subdivision table in the supplementary material. We then check if all sub-tetrahedra have volume larger than ϵ^3_{zero} (since we observed that elements with positive but extremely small volume could delay later insertions in this local region) and reject the insertion if this is not the case.



3.4.3 Open-boundary Edge Preservation. After triangle insertion, the input edges shared by two non-coplanar triangles are preserved through the insertion of adjacent triangles, as the plane of the next inserted triangle will intersect the cover $\mathcal{F}$ of a previously inserted triangle. This does not hold for boundary edges. An edge is an *open-boundary edge* if it has only one incident triangle or has multiple coplanar incident triangles on the same side of the edge in their common plane.

To preserve an open-boundary edge e of a triangle T, we project e and the cover $\mathcal{F}$ of T to the plane P' that best fits the faces of $\mathcal{F}$. Then, we compute the intersection of the projection of e to P' with the projections of the faces of $\mathcal{F}$. The intersection points of the projection of e and $\mathcal{T}$ are then computed on P' and are lifted back to the corresponding faces of $\mathcal{F}$. Since we have a set of edges cut into two, we can subdivide the affected tetrahedra using the previous table-based tetrahedron subdivision. An example can be found in Appendix D.

Note that the open-boundary edge preservation might fail due to numerical reasons, in this case we rollback the operation and postpone the insertion of the open-boundary triangle to later stages.

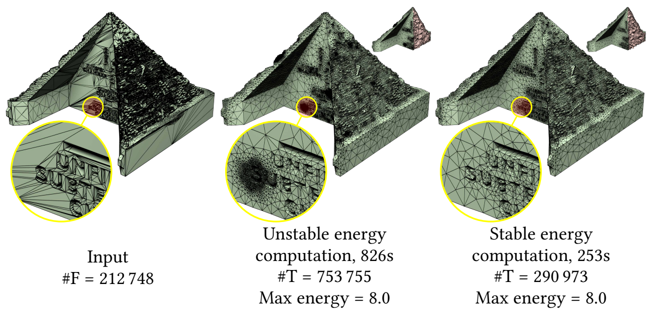


Fig. 10. Example of model where the numerical instability of the AMIPS energy causes over-refinement (middle). By evaluating the energy using rational numbers (when it is above 10^8) the issue disappears (right).

3.5 Mesh Improvement

We adapt the mesh improvement framework proposed in TetWild [Hu et al. 2018] that optimizes the conformal AMIPS 3D energy [Rabinovich et al. 2017] to increase the mesh quality, but avoid the overhead introduced by the hybrid kernel by specializing the framework for floating point computation. Note that, as mentioned in [Hu et al. 2018], we use the AMIPS energy since it is differentiable and scale-invariant. We made three changes to the original optimization:

(1) We try to insert the uninserted input faces every three mesh improvement iterations until all input faces are inserted or the mesh improvement terminates.
(2) We parallelize the vertex smoothing step using a simple graph coloring strategy.
(3) We discovered an instability in the evaluation of the AMIPS energy computation in floating points, which sometimes leads to overrefinement in TetWild. We propose a fix using a hybrid evaluation that uses rational numbers to compute intermediate results.

(1) is a change required by the new algorithm, since not all faces can be inserted when computations are done in floating point. (2) is a minor, yet effective, modification that slightly improves performance (Figure 16). (3) is a subtle problem, which we now explain in more detail. The conformal AMIPS 3D energy is a Jacobian-based energy defined as:

$$\text{AMIPS} = \frac{\operatorname{tr}(\mathbf{J}^T\mathbf{J})}{\det(\mathbf{J})^{2/3}}, \tag{1}$$

where $\mathbf{J}$ is the Jacobian of the transformation from a regular tetrahedron to the tetrahedron $\mathcal{T}$. The larger the energy is, the worse the quality of $\mathcal{T}$ is. The minimal value is 3, the energy of a regular tetrahedron. The AMIPS energy is invariant under permutation of the vertices of $\mathcal{T}$, however its numerical evaluation in floating-point arithmetic is not, due to floating-point rounding. Usually the difference is negligible, but when the energy of $\mathcal{T}$ is large (on the order of 10^8), the floating point computation becomes unstable and the resulting energy could differ by two orders of magnitude, which means that the descent direction that appears to be decreasing the energy may be determined incorrectly (see Appendix B for a concrete example). This numerical instability might prevent mesh improvement and thus lead to over-refinement, since the algorithm is trying to add degrees of freedom unnecessary to improve the quality (Figure 10).

To address this issue, we raise the energy to the third degree making it completely rational, and evaluate it using rational computation for elements with energy larger than 10^8. We round the computed value of its third degree to the 64-bit floating point representation, and then compute the cubic root. The rational computation is more accurate (and permutation invariant) but significantly slower. However, the cases of precision loss in the energy are rare, and the overall computational overhead is negligible. Note that we only use the rational evaluation of the energy to ensure validity of the line search step: the search direction is always computed using floating-point. This change has a major effect on the speed and effectiveness of our mesh optimization (Figure 10), avoiding unnecessary refinement and decreasing the overall runtime.

The mesh improvement terminates when either a user-specified mesh quality or a user-controlled number of iterations is reached. To ensure a fair comparison, for the large dataset testing and all examples in the paper, we use the same stopping criteria (max AMIPS energy is smaller than 10 or the number of optimization iterations reaches 80) and input parameters (envelope size $\epsilon = 10^{-3}d$, targeted edge length $\ell = d/20$, where d is the diagonal's length of the bounding box of the input mesh) as in [Hu et al. 2018].

We note that our method provides no theoretical guarantees on the quality of the final mesh. In our experiments, it achieves a quality higher or comparable to other methods (Section 4). Quartet [Bridson and Doran 2014], a grid based method, produces uniformly sized tetrahedral mesh whose dihedral angles are bounded between $10.7°$ and $164.8°$, or between $8.9°$ and $158.8°$ [Labelle and Shewchuk 2007], but it does not preserve sharp edges or corners. The Constrained Delaunay refinement method used in TetGen [Si 2015] guarantees a radius-edge ratio larger than 2 if the input does not have sharp features or angles smaller than $70.53°$.

We use the same strategy as in TetWild for handling inputs with open boundaries. We track the vertices on the open boundary and project them back to the open boundary during mesh improvement (Section 3.5). Elements are classified as inside or outside the surface using the generalized winding number (Section 3.6). An example of input with open boundary is shown in Figure 11.

3.6 Filtering

The output of the mesh improvement step is a volumetric tetrahedral mesh of the expanded bounding box of the input triangle soup, with the preprocessed input triangles inserted. We provide three ways of optionally filter the result, targeting two different applications.

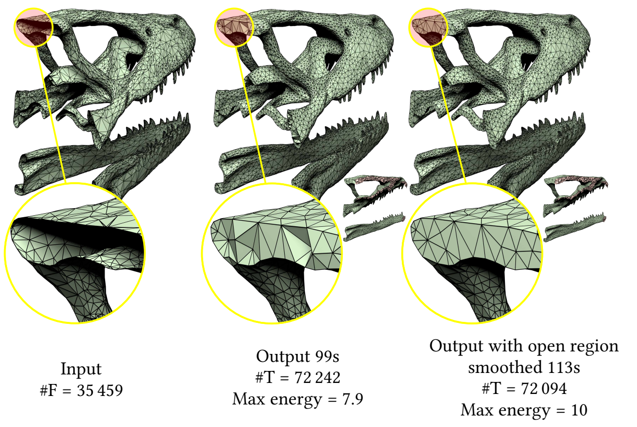


Fig. 11. Input with open boundary on the bottom (left). The output tetrahedral mesh preserves the input geometry and closes the open side (middle). Users can choose to enable an additional smoothing process for smoothing the open region (right).

The first strategy is to use a simple flood-fill algorithm starting from the boundary. This method is well-suited for watertight input models with incorrect normal orientations and several component nested. This simple strategy only removes elements outside the outer boundary of the object but can not remove unwanted elements inside nested components as shown in the middle of the sliced output in Figure 13.

The second strategy uses the fast winding number [Barill et al. 2018] to filter the tetrahedra outside of the preserved/tracked input [Hu et al. 2018]. This strategy is particularly suited to inputs with gaps, since it is able to extend the notion of in-out to these regions. In this case, the volume of output extracted by the winding number filter depends on the orientation of the input triangles.

The third strategy is a volumetric extension of the mesh arrangement algorithm [Zhou et al. 2016]. In this case, the input becomes a set of triangle soups, coupled with a set of Boolean operations to perform on them. During the triangle insertion stage, we keep track of the provenance of each triangle, and use it at the end to compute a set of generalized winding numbers (one for each tracked input surface) for the centroids of all tetrahedra in the volumetric mesh. We use the set of winding numbers to decide which tetrahedron to keep by checking if it is supposed to be contained in the result of the Boolean operation. For instance, when intersecting two triangle soups, we keep all tetrahedra that are inside both input triangle soups, according to the winding number definition.

There are three major advantages of this method over [Zhou et al. 2016]: (1) Boolean operations can be performed on non-PWN surfaces, (2) the output is equipped with a tetrahedral mesh, which could be useful in downstream applications, and (3) the surface quality is high since the algorithm is allowed to remesh within the ϵ envelope.

4 RESULTS

Our algorithm is implemented in C++ and uses Eigen [Guennebaud et al. 2010] for the linear algebra routines. We perform a large-scale

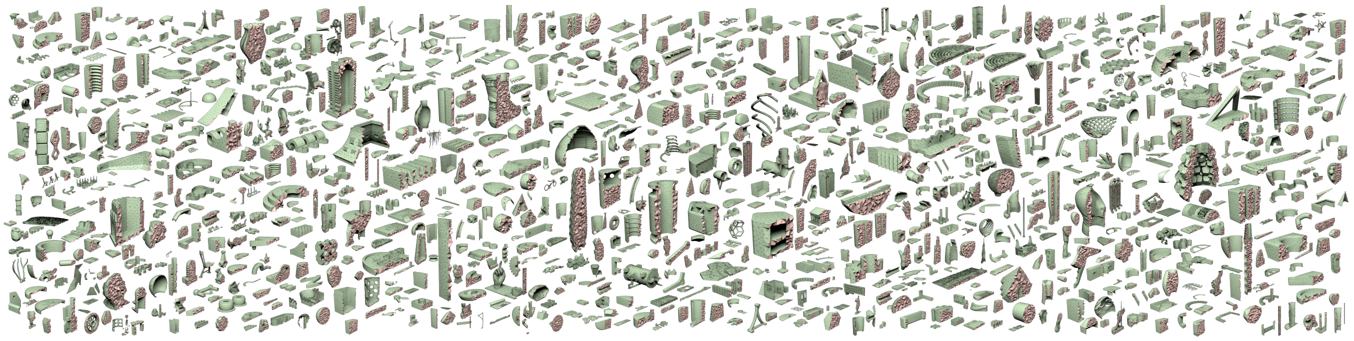


Fig. 12. 1000 random samples of fTetWild output on Thingi10k dataset.

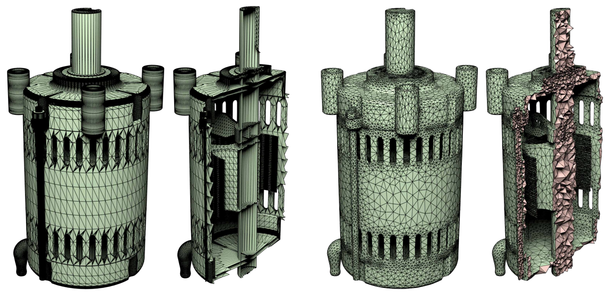


Fig. 13. An input model (left) where the heuristic winding number filtering fails to extract the volume (it drops most of the tetrahedra in the output) due to inconsistent triangle orientations in the input. By changing the heuristic to the flood-fill algorithm, we can obtain the expected output (right).

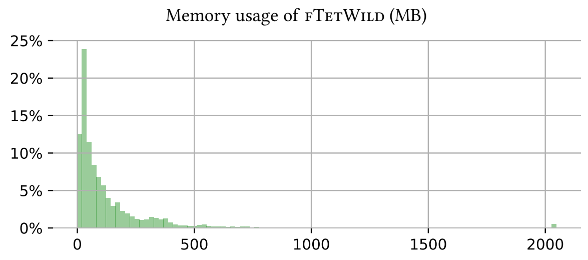


Fig. 14. Histogram of memory usage of fTetWild over the Thingi10k dataset (data truncated at 2GB).

comparison of our method with other meshing methods on the Thingi10k dataset [Zhou and Jacobson 2016], which contains 10 000 real-world surface triangle meshes. We run our experiments on cluster nodes with a Xeon E5-2690v4 2.6GHz, allowing every model to use up to 8 threads, 128GB memory, and 24 hours running time. The reference implementation and testing data are open-source and available on GitHub: https://github.com/wildmeshing/fTetWild.

Table 2. Comparison of code robustness and performance on the Thingi10k dataset.

Method	Success rate	Out of memory (>32GB)	Time exceeded (>3h)	Algorithm limitation	Average time(s)
CGAL	79.00%	0.00%	0.00%	21.00%	11.7
TetGen	49.50%	0.10%	1.70%	48.70%	32.3
TetWild	*99.89%	0.05%	0.11%	0.00%	360.0
Ours	****99.97%**	**0.02%**	**0.03%**	**0.00%**	**49.8**

Note: The maximum resources allowed for each model are 3 hours and 32GB of memory. The first 3 lines of data are from [Hu et al. 2018], Table 2. Note that the average time (last column) is computed over all the models for which each method succeeded, and it is thus not directly comparable between different methods. *: TetWild exceeds the 3h time on 11 models. If 27 hours of maximal running time are allowed, TetWild achieves 100% success rate. **: Our method exceeds the 3h time limit on 3 models. If 11 hours of maximal running time are allowed, fTetWild achieves 100% success rate.

4.1 Success Rate

With the above memory and time constraints, fTetWild successfully tetrahedralizes 100% of the 10 000 input meshes (Figure 12). Most of the input models can be tetrahedralized with less than 1GB of RAM as detailed in Figure 14. Note that very complex models might require more memory, for instance the one in Figure 23 uses around 17GB of memory.

As observed in [Hu et al. 2018], most of the state-of-the-art tetmeshers have low success rate on *in-the-wild* data. We summarize the results on the whole Thingi10k dataset in Table 2. Note that only our method and TetWild have high success rates: our average timing is however seven times faster than TetWild.

4.2 Running Time

Thingi10k Dataset (10000 Models). We compare the running time of our method with TetWild. For a fair comparison, we disable our code optimizations that could be easily ported to TetWild, such as parallelization of the preprocessing and smoothing step, and using the recent fast winding number algorithm for the final filtering. Without these optimizations, our algorithm is 4 times faster than TetWild on average (80.4s vs 360s). With code optimizations, we further improve our running time to 49.8s on average on a machine with 8 cores, which is 7 times faster than the serial implementation of TetWild (Figure 16). On more complex examples, like the model in Figure 17, our method is up to 17 times faster than TetWild.

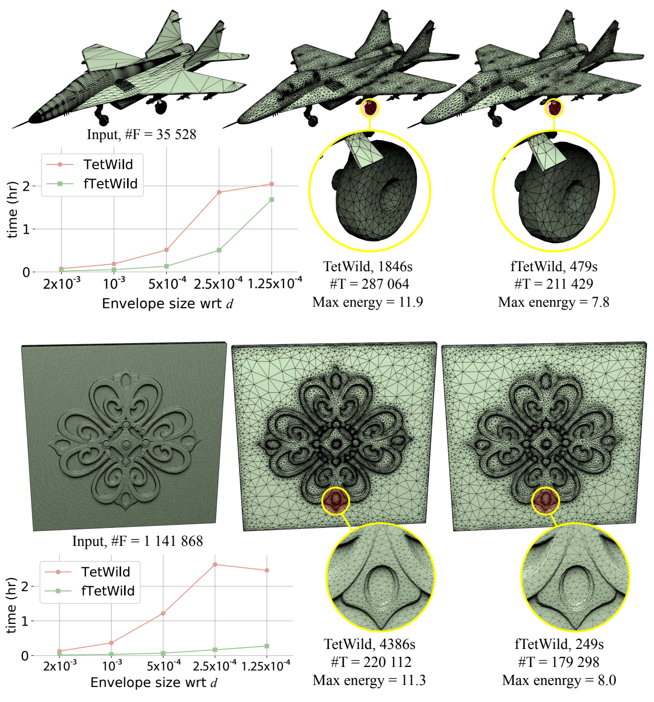


Fig. 15. Input models and running time plots of fTetWild and Tetwild with ϵ reduced from $2\,10^{-3}d$ to $1.25\,10^{-4}d$ (left). Output tetrahedral meshes of the two methods at $\epsilon = 5\,10^{-4}d$ (middle and right). Note that we flipped all the normals of the input triangles of the airplane model for visualization purposes (see Figure 30 for a detailed discussion).

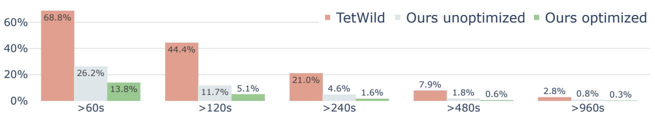


Fig. 16. Percentage of models requiring more than a certain time for our parallel and serial algorithm compared with TetWild on Thingi10k dataset.

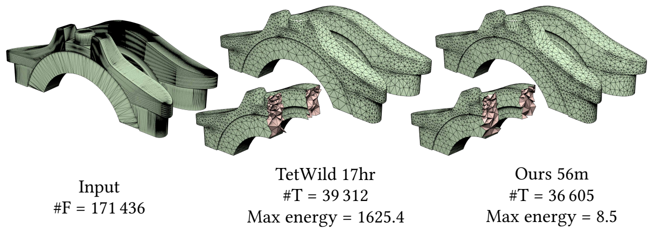


Fig. 17. Example of a challenging model where fTetWild is 17 times faster than TetWild.

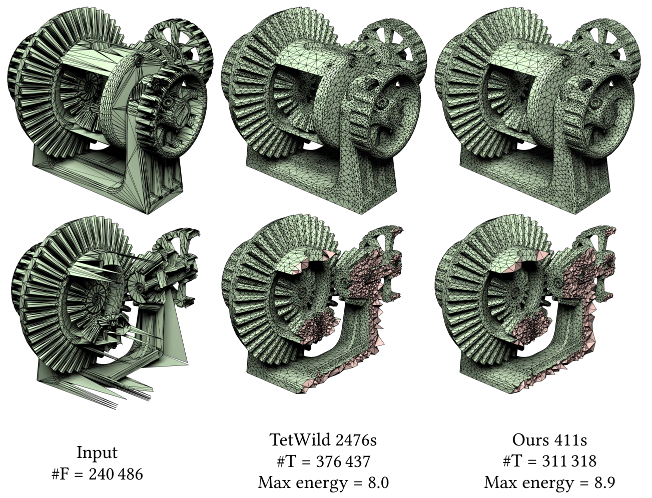


Fig. 18. Our method (right) produces high-quality tet-meshes that are similar to TetWild (middle).

The running time of our algorithm (and of TetWild too) depends, among other factors, on the envelope size. Checking envelope containment using sampling has a cost that grows quadratically as the envelope shrinks. This results in a trade-off between running time and detail preservation. Figure 15 shows how the performance of fTetWild and TetWild are affected by the envelope size: while both methods are fast with large envelope size, the running times dramatically increase when the envelope shrinks. Alternative strategies could be used to check the envelope to mitigate this issue [Wang et al. 2020]. If a small envelope is required, the runtime could be reduced by sacrificing element quality by stopping the algorithm prematurely during the mesh optimization.

Reduced Thingi10k Dataset (4540 Models). We use a reduced dataset containing the intersection of the Thingi10k models that TetGen, CGAL, TetWild, and our method all succeed on. The dataset contains 4540 models, and allows us to fairly compare the performance of the different methods. On average, our method is comparable (18.5s) to the widely used, Delaunay-based tetrahedral mesher TetGen (22s), and is faster than CGAL (95s) and TetWild (107s), while robustly handling imperfect inputs. Figure 1 shows the number of models requiring more than a given time. For example, within less than 2 minutes, our method successfully tetrahedralizes 98.7% of the inputs. It is interesting to note that the tail of the distribution of our method is shorter than both TetGen and CGAL. For instance, there are only 4 models where our method requires more than 16 minutes, differently from TetGen, CGAL, and TetWild which have 20, 122, and 25 models, respectively.

4.3 Mesh Quality

The geometric quality of meshes produced by our algorithm is similar to the meshes produced by TetWild (Figure 18), since our method implements a similar mesh optimization strategy. We quantitatively evaluate and compare the element quality of TetWild and our output using five different measures:

(1) AMIPS energy (Equation (1)), range $[3, +\infty)$, optimal 3,
(2) Minimal dihedral angle, range $(0, 1.23]$, optimal 1.23,
(3) Volume-to-edge ratio $6\sqrt{2}\,V/\ell_{\max}^3$, range $(0, 1]$, optimal 1,
(4) Aspect ratio $\sqrt{3/2}\,h_{\min}/\ell_{\max}$, range $(0, 1]$, optimal 1,
(5) Radius-to-edge ratio $2\sqrt{6}\,r_{\text{in}}/\ell_{\max}$, range $(0, 1]$, optimal 1,

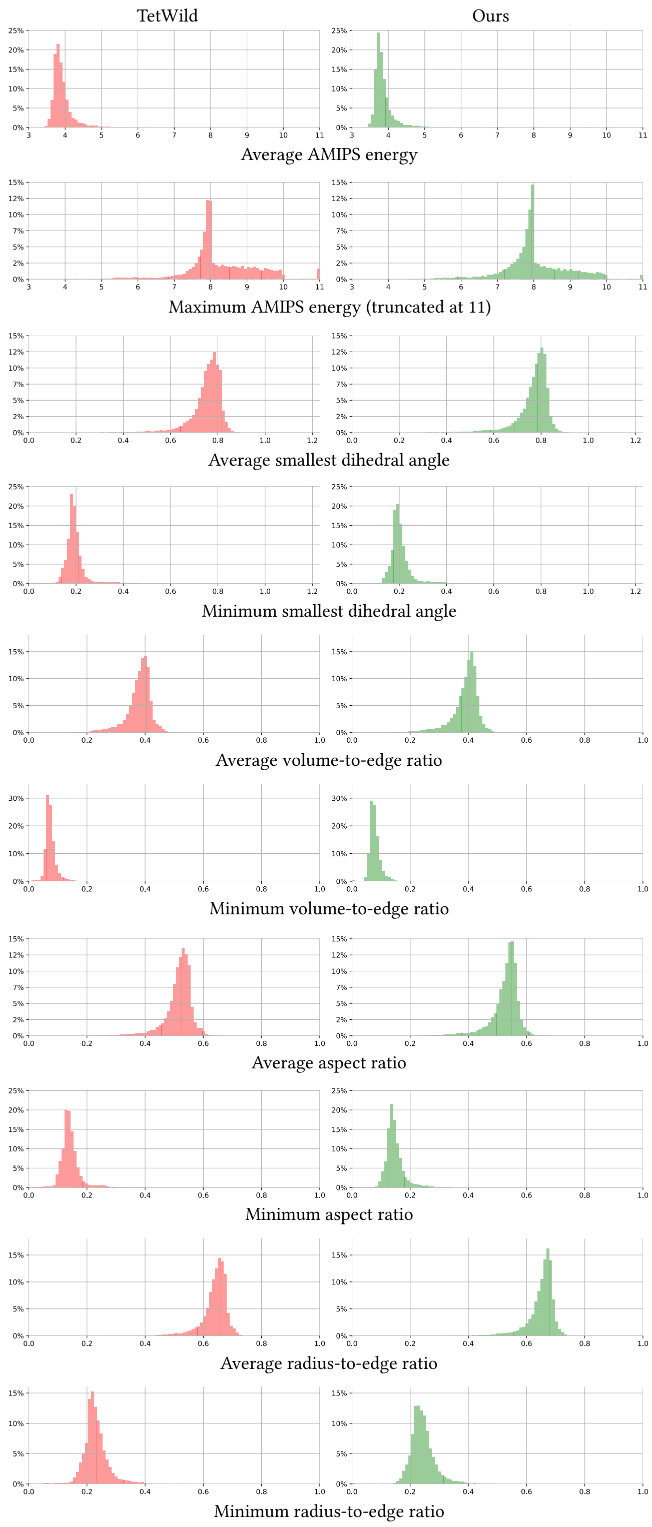


Fig. 19. Histogram for mesh quality comparison of TetWild (red) and our method (green) in five different quality measures. The statistic is based on the output of the whole Thingi10k dataset.

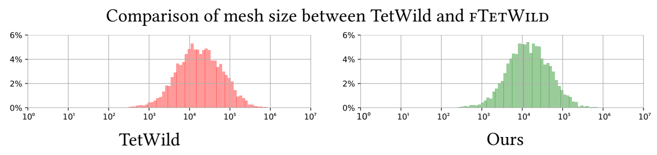


Fig. 20. Histograms of number of in log scale for the output meshes of Thingi10k dataset.

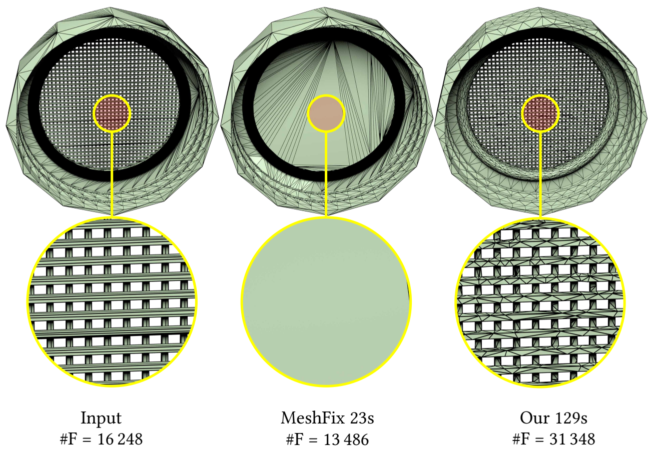


Fig. 21. Example of repairing an invalid triangular mesh (left) with MeshFix (middle) and our algorithm (right). MeshFix is fast but loses details during processing, while our method preserves them. The max AMIPS energy of our intermediate tetrahedral mesh is 1975. Here we stop mesh improvement when maximum energy reaches 2000.

where V is the volume, $\ell_{\max}$ is the longest edge, $h_{\min}$ the minimum height, and r_{in} the radius of the inscribed circle of a tetrahedron $\mathcal{T}$. We use (3), (4) and (5) since these are standard measures for tetrahedral quality [Shewchuk 2002b].

Figure 19 shows the histograms of worst and average element quality of 10 000 output meshes of TetWild and our method. The quality of our outputs are quite similar to TetWild's output. We refer to the study in [Hu et al. 2018, Figure 14] for the full quality comparison of TetWild and other tetrahedral meshing algorithms.

4.4 Mesh Density

Compared with TetWild, our method generates meshes of similar density (Figure 20). Both TetWild and our method aim to generate as-coarse-as possible meshes while preserving the input surface. This choice is useful in downstream applications to reduce their computational cost. Optionally, the algorithm supports a user-specified sizing field to increase the density if desired.

In contrast to our method, TetGen preserves the input surface geometry *exactly* and thus generates a dense tetrahedral mesh around the surface if the input surface mesh is dense, as visible in the model shown in Figure 1. CGAL approximates the surface by means of an implicit function, but sometimes over-refines sharp features and tiny artifacts as illustrated in Figure 1, where the dark spots are over-refined regions.

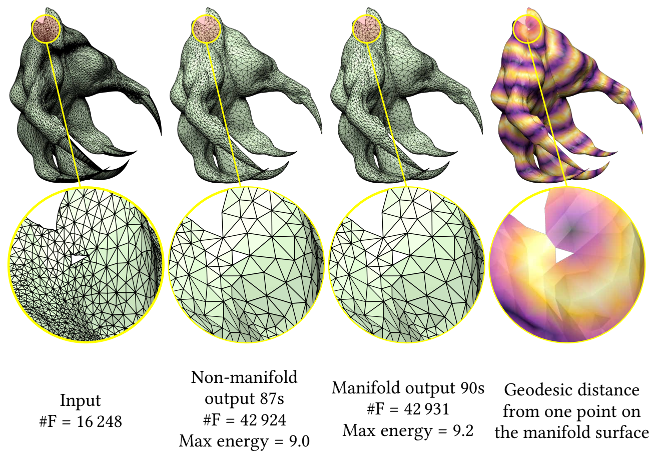


Fig. 22. Example of a non-manifold surface mesh (left) which is automatically repaired by our algorithm (right second).

5 APPLICATIONS

5.1 Mesh Repair

Similarly to TetWild, our algorithm can be used to repair imperfect triangle meshes by tetrahedralizing the volume and extracting the surface of the generated tetrahedral mesh. However, the mesh improvement step of our method (Section 3.5) can be stopped at any time since we maintain an inversion-free floating point tetrahedral mesh at all stages of our algorithm. High tetrahedral mesh quality is not required for this application, and we can stop mesh optimization as soon as all input faces are inserted, further reducing the running time. We compared our result with the state-of-the-art mesh repairing tool MeshFix [Attene 2010] in Figure 21. Our method, while slower, provides a higher-quality result with controllable geometric error. A minor, yet important, observation is that keeping only the boundary of a valid tetrahedral mesh might generate a non-manifold surface mesh (Figure 22). To avoid this problem, we identify the non-manifold edges and split them. Then we duplicate every non-manifold vertex to ensure a global manifold output, using the algorithm proposed in [Attene et al. 2009]. Note that this procedure ensures manifoldness, but introduces vertices in the same geometric position. With this minor change, our algorithm can be used to repair triangle meshes, guaranteeing the extraction of an high-quality, manifold boundary surface mesh within the prescribed distance from the input triangle soup.

We also tested an extremely challenging model coming from an industrial application in additive manufacturing (the part is copyrighted by Velo3D): the design of an exhaust pipe using a volume filled with a structure based on the gyroid triply periodic minimal surface. The model has a multitude of issues introduced during the modeling phase, but it can be cleaned up by our algorithm within 55 minutes (or 122 minutes with the envelope size decreased by a factor of two), compared to around two weeks of manual labor required by Velo3D's current processing pipeline. Our output mesh (Figure 23) is directly usable for FEA, further editing, or fabrication. As a reference point, the original implementation of TetWild takes 215

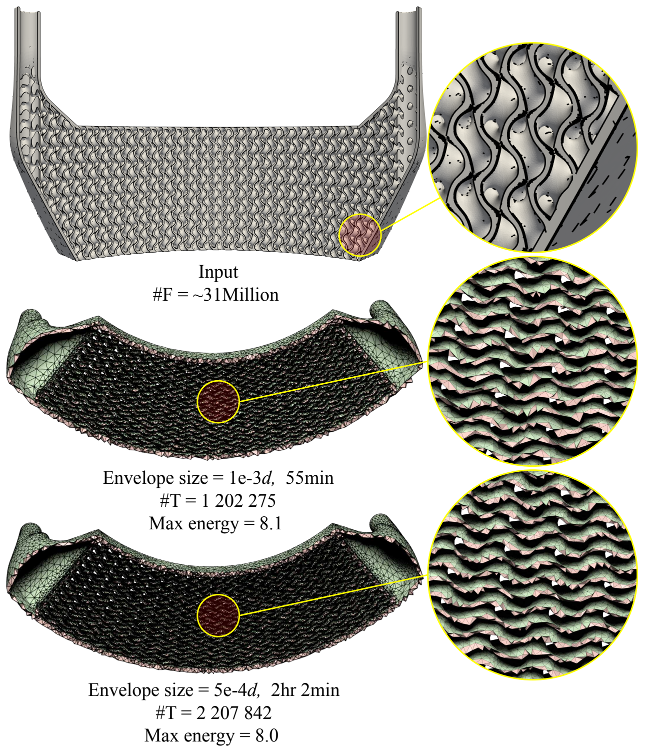


Fig. 23. Meshing a complex model with 93 million vertices and 31 million faces with different envelope sizes (top). The input mesh contains degenerate triangles and severe self-intersections. Our output tetrahedral meshes are in geometric high quality with either default envelope size (middle) or half envelope size (bottom).

minutes with a default envelope size. Another challenging model we tested contains complex thin structures coming from architecture (Figure 24). The method in [Ghomi et al. 2018; Masoud 2016] optimizes for the layout of a graph, then replaces the graph edges with cylinders of varying radii. To ensure solidity of the final structure, all cylinders are intersecting as shown in the close up. Although the mesh contains many irregularities, fTetWild successfully meshes the domain into an analysis-ready mesh.

5.2 Mesh Arrangements

Zhou et al. [2016] proposes to compute the arrangement between multiple surfaces using an algorithm to map Boolean operations into simple algebraic expressions involving the winding number of the input surfaces. Their method is robust, but only supports clean PWN surfaces as input. We propose a simple extension of this algorithm (as explained in Section 3.6) to arbitrary triangle soups. The advantages of our method is evident when the input surfaces come from CAD models containing small gaps or self-intersections: both Mesh Arrangements [Zhou et al. 2016] and CGAL [Hachenberger and Kettner 2019] are unable to perform the operation (since it is not well-defined for non-PWN surface), while fTetWild can compute an approximate (since it allows for an ϵ-deviation from the input surfaces) union, difference, and intersection between them

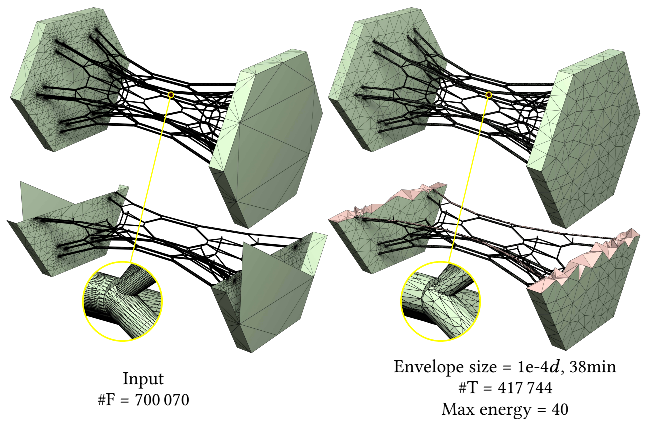


Fig. 24. Example of an architectural application with 80 999 self-intersecting faces. The cylinders in the input are intersecting with each other as shown in the closeup. fTetWild successfully cleaned and tetrahedralized this input. Here we stop mesh optimization when maximum energy reaches 50.

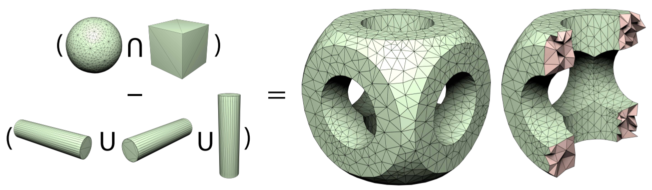


Fig. 25. Four Boolean operations among 5 objects. fTetWild takes 34s and products output with #T = 8 060 and max energy = 7.2.

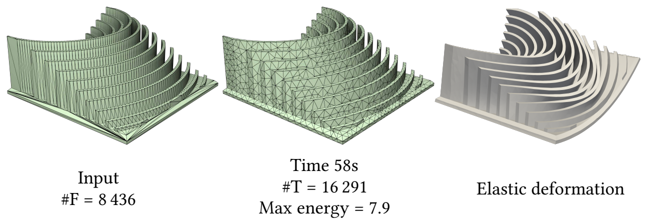


Fig. 26. Example of non-linear elastic deformation of a body (right).

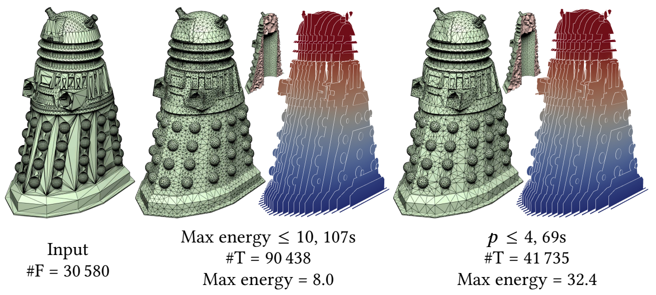


Fig. 27. Two different stopping criteria of our algorithm. The full optimization (middle) improves the mesh to high quality, while using the criterion in [Schneider et al. 2018] (right) results in lower mesh quality but faster meshing and smaller mesh size. The color shows the solution of the volumetric Laplace equation.

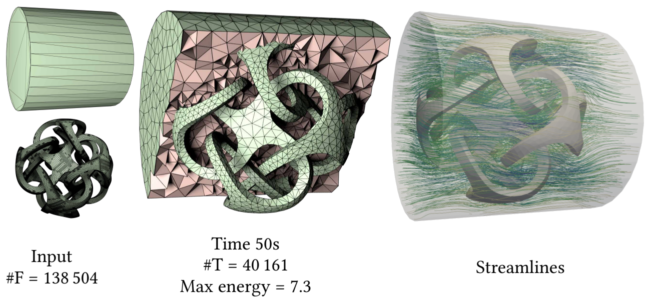


Fig. 28. Streamlines of a fluid (right) moving in a cylindrical pipe (left top) with a complicated obstacle (left bottom) in the center. The background mesh (middle) is obtained by subtracting the obstacle from a cylinder using our method.

(Figures 29, 25), providing robust (but slower) Boolean operations on imperfect geometries. The output is a tetrahedral mesh, which can be useful in downstream applications, and its boundary is a high quality surface triangular mesh.

5.3 Simulation

The main application of tetrahedral meshing is physical simulations, and the high-quality of our results makes them ideal to be directly used in downstream finite element software (Figure 26).

Additionally, the recently proposed *a priori* p-refinement [Schneider et al. 2018] is an ideal fit for our approach when targeting FEM applications, since fTetWild *always* produces a valid floating-point mesh. Schneider et al. [2018] provides a simple formula to determine the order of each element to compensate for its, possibly bad, shape. We can use this criterion to terminate the mesh optimization early in our algorithm (thus reducing the meshing time) without affecting the quality of the simulation, Figure 27.

We use the Boolean difference (Section 5.2) to generate the background mesh required for simulating the fluid flow on a cylindrical tube containing an obstacle (Figure 28).

6 CONCLUDING REMARKS

We introduced fTetWild, a novel robust tetrahedral meshing algorithm for triangle soups which combines the robustness of TetWild with a running time comparable to Delaunay-based methods. The improved performance makes this algorithm suitable not only for applications requiring a volumetric discretization, but also for surface mesh repair and Boolean operations.

Our current naive parallelization approach shows that our algorithm benefits from shared-memory parallelization; exploring more advanced parallelization techniques and extending it to distributed computation on HPC clusters are important directions for future work. Our iterative triangle insertion algorithm could be used in dynamic remeshing tasks, potentially allowing to reuse an existing mesh and insert new faces only in regions with high deformation.

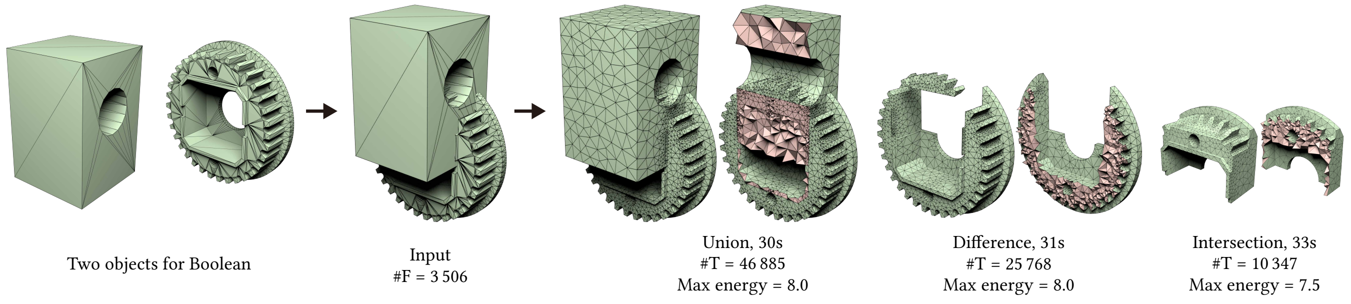


Fig. 29. Three Boolean operations computed on non-manifold, self-intersecting, and non-PWN input surface meshes. The left are two objects for Boolean operation. The middle is the input surface mesh of fTetWild. The right are our output meshes after computing the union, difference, and intersection between the two objects. The average max AMIPS energy of outputs and average time of different operations are with small variance.

While conceptually trivial, extending our algorithm to 2D triangle meshing could improve the performance of [Hu et al. 2019].

Our algorithm optionally uses the winding number or flood fill filters to extract the volume of the interior of the object bounded by the input surface. While these heuristics are very effective for imperfect inputs representing closed input models with consistent normal orientation, they might fail if the input surface contains open shells not bounding a volumes or nested components with wrongly oriented normals (Figure 30). In these cases, the volume is not well defined and our filtering will arbitrarily discard or keep tetrahedra around these regions. We recommend to not rely on these heuristics if the input contains open shells, and do the filtering using an ad-hoc algorithm. In case of nested components we recommend to correct the orientation to ensure a proper definition of in-out [Takayama et al. 2014].

fTetWild uses the conformal AMIPS energy [Rabinovich et al. 2017] to measure and optimize the quality of the tetrahedra. An interesting alternative has been introduced concurrently to our work by [Alexa 2019]: they propose to optimize directly for the Dirichlet energy of the tetrahedralization and show that this measure is effective at removing slivers, while being computationally efficient to evaluate. A comparative study of the two measures would be interesting, and using the Dirichlet energy could lead to further reductions in the running time of our method.

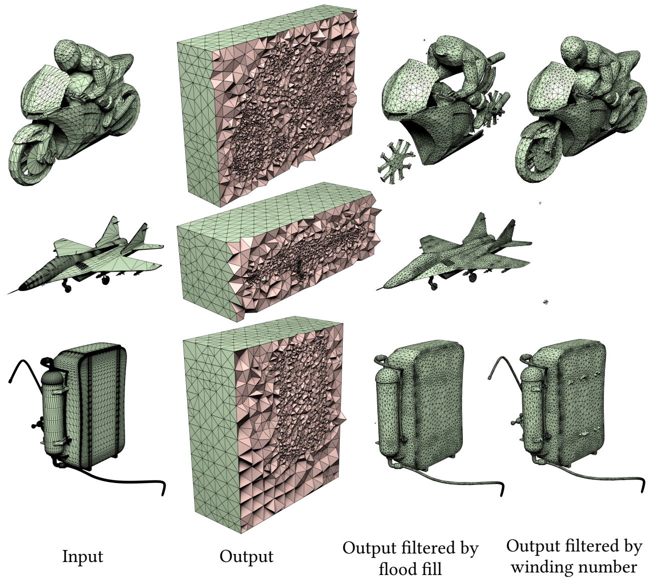

Input
Output
Output filtered by flood fill
Output filtered by winding number

Fig. 30. The output of fTetWild is a tetrahedral mesh of the bounding box containing the input (second column). The output can be optionally filtered to delete the tetrahedra in the exterior using the flood fill or the winding number heuristic (last two columns), which may fail on inputs (first column) with open shells or nested components with inconsistent normal orientation.

ACKNOWLEDGMENTS

This work was supported in part through the NYU IT High Performance Computing resources, services, and staff expertise. This work was partially supported by the NSF CAREER award 1652515, the NSF grants IIS-1320635, DMS-1436591, DMS-1821334, OAC-1835712, OIA-1937043, CHS-1908767, CHS-1901091, the National Key Research and Development Program of China grant 2018YFB1107402, the SNSF grant P2TIP2_175859, a gift from Adobe Research, a gift from nTopology, and a gift from Advanced Micro Devices, Inc.

REFERENCES

L. A. Freitag and C. Ollivier-Gooch. 1998. Tetrahedral Mesh Improvement Using Swapping and Smoothing. *Internat. J. Numer. Methods Engrg.* 40 (05 1998).

F. Alauzet and D. Marcum. 2014. A Closed Advancing-Layer Method With Changing Topology Mesh Movement for Viscous Mesh Generation. In *Proceedings of the 22nd International Meshing Roundtable.* Springer International Publishing, Cham, 241–261.

M. Alexa. 2019. Harmonic Triangulations. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 38, 4 (2019), 54.

P. Alliez, D. Cohen-Steiner, M. Yvinec, and M. Desbrun. 2005a. Variational Tetrahedral Meshing. *ACM Transactions on Graphics* 24, 3 (07 2005), 617. https://doi.org/10.1145/1073204.1073238

P. Alliez, D. Cohen-Steiner, M. Yvinec, and M. Desbrun. 2005b. Variational Tetrahedral Meshing. *ACM Trans. Graph.* 24, 3 (July 2005), 617–625. https://doi.org/10.1145/1073204.1073238

M. Attene. 2010. A lightweight approach to repairing digitized polygon meshes. *The Visual Computer* 26, 11 (01 Nov 2010), 1393–1406. https://doi.org/10.1007/s00371-010-0416-3

M. Attene. 2014. Direct Repair of Self-intersecting Meshes. *Graph. Models* 76, 6 (Nov. 2014), 658–668. https://doi.org/10.1016/j.gmod.2014.09.002

M. Attene. 2017. *ImatiSTL - Fast and Reliable Mesh Processing with a Hybrid Kernel.* Springer Berlin Heidelberg, Berlin, Heidelberg, 86–96.

M. Attene, M. Campen, and L. Kobbelt. 2013. Polygon Mesh Repairing: An Application Perspective. *ACM Comput. Surv.* 45, 2, Article 15 (March 2013), 33 pages.

M. Attene, D. Giorgi, M. Ferri, and B. Falcidieno. 2009. On converting sets of tetrahedra to combinatorial and PL manifolds. *Computer Aided Geometric Design* 26, 8 (2009), 850 – 864. https://doi.org/10.1016/j.cagd.2009.06.002

F. Aurenhammer. 1991. Voronoi Diagrams&Mdash;a Survey of a Fundamental Geometric Data Structure. *ACM Comput. Surv.* 23, 3 (Sept. 1991), 345–405. https://doi.org/10.1145/116873.116880

F. Aurenhammer, R. Klein, and D.-T. Lee. 2013. *Voronoi Diagrams and Delaunay Triangulations.* WORLD SCIENTIFIC, River Edge, NJ, USA. https://doi.org/10.1142/8685 arXiv:https://www.worldscientific.com/doi/pdf/10.1142/8685

B. S. Baker, E. Grosse, and C. S. Rafferty. 1988. Nonobtuse triangulation of polygons. *Discrete & Computational Geometry* 3, 2 (01 Jun 1988), 147–168.

G. Barill, N. Dickson, R. Schmidt, D. I. Levin, and A. Jacobson. 2018. Fast Winding Numbers for Soups and Clouds. *ACM Transactions on Graphics* 37, 4 (2018), 43:1–43:12.

H. Barki, G. Guennebaud, and S. Foufou. 2015. Exact, robust, and efficient regularized Booleans on general 3D meshes. *Computers and Mathematics with Applications* 70, 6 (2015), 1235–1254.

M. Bern, D. Eppstein, and J. Gilbert. 1994. Provably good mesh generation. *J. Comput. System Sci.* 48, 3 (1994), 384 – 409.

G. Bernstein. 2013. Cork Boolean Library . https://github.com/gilbo/cork.

G. Bernstein and D. Fussell. 2009. Fast, Exact, Linear Booleans. In *SGP*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 1269–1278.

H. Bieri and W. Nef. 1988. Elementary Set Operations with D-dimensional Polyhedra. In *Proc. IWCGA*. Springer-Verlag, Berlin, Heidelberg, 97–112.

C. J. Bishop. 2016. Nonobtuse Triangulations of PSLGs. *Discrete & Computational Geometry* 56, 1 (2016), 43–92.

J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. 2002. Triangulations in CGAL. *Computational Geometry* 22 (2002), 5–19.

J.-D. Boissonnat and S. Oudot. 2005. Provably Good Sampling and Meshing of Surfaces. *Graphical Models* 67, 5 (09 2005), 405–451. https://doi.org/10.1016/j.gmod.2005.01.004

R. Bridson and C. Doran. 2014. Quartet: A tetrahedral mesh generator that does isosurface stuffing with an acute tetrahedral tile. https://github.com/crawforddoran/quartet.

J. R. Bronson, J. A. Levine, and R. T. Whitaker. 2013. Lattice Cleaving: Conforming Tetrahedral Meshes of Multimaterial Domains With Bounded Quality. In *Proceedings of the 21st International Meshing Roundtable*. Springer Berlin Heidelberg, Berlin, Heidelberg, 191–209. https://doi.org/10.1007/978-3-642-33573-0_12

O. Busaryev, T. K. Dey, and J. A. Levine. 2009. Repairing and Meshing Imperfect Shapes with Delaunay Refinement. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling (SPM '09)*. ACM, 25–33.

M. Campen and L. Kobbelt. 2010. Exact and Robust (self-)intersections for Polygonal Meshes. *Comput. Graph. Forum* 29, 2 (2010), 397–406.

S. A. Canann, S. N. Muthukrishnan, and R. K. Phillips. 1996. Topological refinement procedures for triangular finite element meshes. *Engineering with Computers* 12, 3 (01 Sep 1996), 243–255. https://doi.org/10.1007/BF01198738

S. A. Canann, M. B. Stephenson, and T. Blacker. 1993. Optismoothing: An optimization-driven approach to mesh smoothing. *Finite Elements in Analysis and Design* 13, 2 (1993), 185 – 190. https://doi.org/10.1016/0168-874X(93)90056-V

L. Chen and J.-c. Xu. 2004. Optimal Delaunay Triangulations. *Journal of Computational Mathematics* 22, 2 (2004), 299–308.

S.-W. Cheng, T. K. Dey, and J. A. Levine. 2008. A Practical Delaunay Meshing Algorithm for a Large Class of Domains. In *Proceedings of the 16th International Meshing Roundtable*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 477–494.

S.-W. Cheng, T. K. Dey, and J. Shewchuk. 2012. *Delaunay Mesh Generation.* Chapman and Hall/CRC, Boca Raton, Florida.

L. P. Chew. 1989. Constrained delaunay triangulations. *Algorithmica* 4, 1 (01 Jun 1989), 97–108. https://doi.org/10.1007/BF01553881

L. P. Chew. 1993. Guaranteed-Quality Mesh Generation for Curved Surfaces. In *Proceedings of the ninth annual symposium on Computational geometry - SCG '93*. ACM Press, New York, NY, USA, 274–280. https://doi.org/10.1145/160985.161150

D. Cohen-Steiner, E. C. de Verdière, and M. Yvinec. 2002. Conforming Delaunay Triangulations in 3D. In *Proceedings of the eighteenth annual symposium on Computational geometry - SCG '02*. ACM Press, 217 – 233.

J.-C. Cuilliere, V. Francois, and J.-M. Drouet. 2013. Automatic 3D Mesh Generation of Multiple Domains for Topology Optimization Methods. In *Proceedings of the 21st International Meshing Roundtable*. Springer Berlin Heidelberg, Berlin, Heidelberg, 243–259. https://doi.org/10.1007/978-3-642-33573-0_15

T. K. Dey and J. A. Levine. 2008. Delpsc: A Delaunay Mesher for Piecewise Smooth Complexes. In *Proceedings of the twenty-fourth annual symposium on Computational geometry - SCG '08*. ACM Press, New York, NY, USA, 220–221. https://doi.org/10.1145/1377676.1377712

A. Doi and A. Koide. 1991. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE TRANSACTIONS on Information and Systems* 74, 1 (1991), 214–224.

C. Doran, A. Chang, and R. Bridson. 2013. Isosurface Stuffing Improved: Acute Lattices and Feature Matching. In *ACM SIGGRAPH 2013 Talks on - SIGGRAPH '13*. ACM Press, New York, NY, USA, 38:1–38:1. https://doi.org/10.1145/2504459.2504507

M. Douze, J.-S. Franco, and B. Raffin. 2015. *QuickCSG: Arbitrary and Faster Boolean Combinations of N Solids.* Technical Report 01121419. Inria Research Centre Grenoble, Rhone-Alpes.

Q. Du and D. Wang. 2003. Tetrahedral Mesh Generation and Optimization Based on Centroidal Voronoi Tessellations. *International journal for numerical methods in engineering* 56, 9 (2003), 1355–1373.

N. Faraj, J.-M. Thiery, and T. Boubekeur. 2016. Multi-Material Adaptive Volume Remesher. *Compurer and Graphics Journal (proc. Shape Modeling International 2016)* 58 (2016), 150 – 160.

L. Feng, P. Alliez, L. Busé, H. Delingette, and M. Desbrun. 2018. Curved Optimal Delaunay Triangulation. *ACM Trans. Graph.* 37, 4 (2018), 61:1–61:16.

X. M. Fu, Y. Liu, and B. Guo. 2015. Computing Locally Injective Mappings by Advanced MIPS. *ACM Trans. Graph.* 34, 4, Article 71 (July 2015), 12 pages.

J. A. George. 1971. *Computer Implementation of the Finite Element Method.* Ph.D. Dissertation. Stanford University, Stanford, CA, USA. AAI7205916.

P. L. George, H. Borouchaki, and E. Saltel. 2003. 'Ultimate' robustness in meshing an arbitrary polyhedron. *Internat. J. Numer. Methods Engrg.* 58, 7 (2003), 1061–1089.

A. T. Ghomi, M. Bolhassan, A. Nejur, and M. Akbarzadeh. 2018. Effect of Subdivision of Force Diagrams on the Local Buckling, Load-Path and Material Use of Founded Forms. In *Proceedings of the IASS Symposium 2018, Creativity in Structural Design.* MIT, Boston, USA.

M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, and M. Seel. 2003. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, and implementation. In *Proc. ESA*. Springer Berlin Heidelberg, Berlin, Heidelberg, 654–666.

G. Guennebaud, B. Jacob, et al. 2010. Eigen v3.

P. Guigue and O. Devillers. 2003. Fast and Robust Triangle-Triangle Overlap Test Using Orientation Predicates. *Journal of graphics tools* 8, 1 (2003), 39–52. https://doi.org/10.1080/10867651.2003.10487580

P. Hachenberger and L. Kettner. 2019. 3D Boolean Operations on Nef Polyhedra. In *CGAL User and Reference Manual* (4.14 ed.). CGAL Editorial Board.

R. Haimes. 2014. MOSS: Multiple Orthogonal Strand System. In *Proceedings of the 22nd International Meshing Roundtable*. Springer International Publishing, Cham, 75–91. https://doi.org/10.1007/978-3-319-02335-9_5

K. Hu, D. Yan, D. Bommes, P. Alliez, and B. Benes. 2017. Error-Bounded and Feature Preserving Surface Remeshing with Minimal Angle Improvement. *IEEE Transactions on Visualization and Computer Graphics* 23, 12 (Dec 2017), 2560–2573.

Y. Hu, T. Schneider, X. Gao, Q. Zhou, A. Jacobson, D. Zorin, and D. Panozzo. 2019. TriWild: Robust Triangulation with Curve Constraints. *ACM Trans. Graph.* (2019).

Y. Hu, Q. Zhou, X. Gao, A. Jacobson, D. Zorin, and D. Panozzo. 2018. Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 37, 4, Article 60 (July 2018), 14 pages. https://doi.org/10.1145/3197517.3201353

C. Jamin, P. Alliez, M. Yvinec, and J.-D. Boissonnat. 2015. CGALmesh: A Generic Framework for Delaunay Mesh Generation. *ACM Trans. Math. Software* 41, 4 (10 2015), 1–24. https://doi.org/10.1145/2699463

B. Klingner and J. Shewchuk. 2007. Aggressive Tetrahedral Mesh Improvement. *Proceedings of the 16th International Meshing Roundtable, IMR 2007*, 3–23.

F. Labelle and J. R. Shewchuk. 2007. Isosurface Stuffing: Fast Tetrahedral Meshes With Good Dihedral Angles. In *ACM SIGGRAPH 2007 papers on - SIGGRAPH '07*. ACM Press, New York, NY, USA, 57. https://doi.org/10.1145/1275808.1276448

B. Lévy. 2019. Geogram. http://alice.loria.fr/index.php/software/4-library/75-geogram.html.

Y. Lipman. 2012. Bounded Distortion Mapping Spaces for Triangular Meshes. *ACM Trans. Graph.* 31, 4 (2012), 108.

W. E. Lorensen and H. E. Cline. 1987. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.* 21, 4 (Aug. 1987), 163–169. https://doi.org/10.1145/37402.37422

S. V. Magalhães, W. R. Franklin, and M. V. Andrade. 2017. Fast exact parallel 3D mesh intersection algorithm using only orientation predicates. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, ACM, New York, NY, USA, 44.

M. Mandad, D. Cohen-Steiner, and P. Alliez. 2015. Isotopic Approximation Within a Tolerance Volume. *ACM Trans. Graph.* 34, 4, Article 64 (July 2015), 12 pages. https://doi.org/10.1145/2766950

A. Masoud. 2016. *3D Graphical Statics Using Reciprocal Polyhedral Diagrams.* Ph.D. Dissertation. ETH Zruich, Stefano Franscini Platz 5, Zurich, CH, 8093.

N. Molino, R. Bridson, and R. Fedkiw. 2003. Tetrahedral Mesh Generation for Deformable Bodies. In *Proc. Symposium on Computer Animation*.

M. Murphy, D. M. Mount, and C. W. Gable. 2001. A Point-Placement Strategy for Conforming Delaunay Tetrahedralization. *International Journal of Computational Geometry & Applications* 11, 06 (12 2001), 669–682.

K. Museth, D. E. Breen, R. T. Whitaker, and A. H. Barr. 2002. Level set surface editing operators. *ACM Trans. Graph.* 21, 3 (2002), 330–338.

B. Naylor, J. Amanatides, and W. Thibault. 1990. Merging BSP trees yields polyhedral set operations. In *Proc. SIGGRAPH*. ACM, New York, NY, USA, 115–124.

A. Paoluzzi, V. Shapiro, and A. DiCarlo. 2017. Arrangements of cellular complexes. *CoRR* abs/1704.00142 (2017). arXiv:1704.00142 http://arxiv.org/abs/1704.00142

D. Pavic, M. Campen, and L. Kobbelt. 2010. Hybrid Booleans. *Comput. Graph. Forum* 29 (2010), 75–87.

J. Peraire, M. Vahdati, K. Morgan, and O. C. Zienkiewicz. 1987. Adaptive Remeshing for Compressible Flow Computations. *J. Comput. Phys.* 72, 2 (Oct. 1987), 449–466.

M. Rabinovich, R. Poranne, D. Panozzo, and O. Sorkine-Hornung. 2017. Scalable Locally Injective Mappings. *ACM Trans. Graph.* 36, 2 (April 2017), 16.

J.-F. Remacle. 2017. A Two-Level Multithreaded Delaunay Kernel. *Computer-Aided Design* 85 (04 2017), 2–9. https://doi.org/10.1016/j.cad.2016.07.018

J. Ruppert. 1995. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. *Journal of Algorithms* 18, 3 (05 1995), 548–585. https://doi.org/10.1006/jagm.1995.1021

E. A. Sadek. 1980. A scheme for the automatic generation of triangular finite elements. *Internat. J. Numer. Methods Engrg.* 15, 12 (1980), 1813–1822.

R. Schmidt and K. Singh. 2010. Meshmixer: an interface for rapid mesh composition. In *ACM SIGGRAPH 2010 Talks*. ACM, ACM, New York, NY, USA, 6.

T. Schneider, Y. Hu, J. Dumas, X. Gao, D. Panozzo, and D. Zorin. 2018. Decoupling simulation accuracy from mesh quality. *ACM Transactions on Graphics* 37, 6 (dec 2018), 1–14. https://doi.org/10.1145/3272127.3275067

M. Schweiger and S. Arridge. 2016. Basis mapping methods for forward and inverse problems: BASIS MAPPING METHODS. *Internat. J. Numer. Methods Engrg.* 109 (05 2016). https://doi.org/10.1002/nme.5271

D. R. Sheehy. 2012. New Bounds on the Size of Optimal Meshes. *Computer Graphics Forum* 31, 5 (08 2012), 1627–1635. https://doi.org/10.1111/j.1467-8659.2012.03168.x

C. Shen, J. F. O'Brien, and J. R. Shewchuk. 2004. Interpolating and Approximating Implicit Surfaces from Polygon Soup. In *Proceedings of ACM SIGGRAPH 2004*. ACM Press, 896–904.

B. Sheng, P. Li, H. Fu, L. Ma, and E. Wu. 2018a. Efficient non-incremental constructive solid geometry evaluation for triangular meshes. *Graphical Models* 97 (2018), 1–16.

B. Sheng, B. Liu, P. Li, H. Fu, L. Ma, and E. Wu. 2018b. Accelerated robust Boolean operations based on hybrid representations. *Computer Aided Geometric Design* 62 (2018), 133–153.

J. Shewchuk. 2012. *Unstructured Mesh Generation.* Chapman and Hall/CRC, Boca Raton, Florida, Chapter 10, 257 – 297.

J. R. Shewchuk. 1996. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry Towards Geometric Engineering*, Ming C. Lin and Dinesh Manocha (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 203–222.

J. R. Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry* 18, 3 (Oct. 1997), 305–363.

J. R. Shewchuk. 1998. Tetrahedral Mesh Generation by Delaunay Refinement. In *Proceedings of the fourteenth annual symposium on Computational geometry - SCG '98*. ACM Press, New York, NY, USA, 86–95. https://doi.org/10.1145/276884.276894

J. R. Shewchuk. 1999. Lecture Notes on Delaunay Mesh Generation. (1999).

J. R. Shewchuk. 2002a. Constrained Delaunay Tetrahedralizations and Provably Good Boundary Recovery. In *Eleventh International Meshing Roundtable*. Sandia National Laboratories, 193–204.

J. R. Shewchuk. 2002b. What is a good linear element? interpolation, conditioning, and quality measures. In *In 11th International Meshing Roundtable*. 115–126.

H. Si. 2015. TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. *ACM Trans. Math. Softw.* 41, 2, Article 11 (Feb. 2015), 36 pages. https://doi.org/10.1145/2629697

H. Si and K. Gartner. 2005. Meshing Piecewise Linear Complexes by Constrained Delaunay Tetrahedralizations. In *Proceedings of the 14th international meshing roundtable*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 147–163.

H. Si and J. R. Shewchuk. 2014. Incrementally Constructing and Updating Constrained Delaunay Tetrahedralizations With Finite-Precision Coordinates. *Engineering with Computers* 30, 2 (04 2014), 253–269. https://doi.org/10.1007/s00366-013-0331-0

K. Takayama, A. Jacobson, L. Kavan, and O. Sorkine-Hornung. 2014. A Simple Method for Correcting Facet Orientations in Polygon Meshes Based on Ray Casting. *Journal of Computer Graphics Techniques* 3, 4 (2014), 53–63.

W. C. Thibault and B. F. Naylor. 1987. Set operations on polyhedra using binary space partitioning trees. In *Proc. SIGGRAPH*. ACM, New York, NY, USA, 153–162.

J. Tournois, C. Wormser, P. Alliez, and M. Desbrun. 2009. Interleaving Delaunay Refinement and Optimization for Practical Isotropic Tetrahedron Mesh Generation. *ACM Transactions on Graphics* 28, 3 (07 2009), 1.

G. Varadhan, S. Krishnan, T. Sriram, and D. Manocha. 2004. Topology preserving surface extraction using adaptive subdivision. In *SGP*. ACM, New York, NY, USA, 235–244.

B. Wang, T. Schneider, Y. Hu, M. Attene, and D. Panozzo. 2020. Exact and Efficient Polyhedral Envelope Containment Check. *ACM Trans. Graph.* 39, 4 (July 2020).

C. C. L. Wang. 2011. Approximate Boolean Operations on Large Polyhedral Solids with Partial Mesh Reconstruction. *IEEE Trans. Vis. Comput. Graph.* 17, 6 (2011), 836–849.

N. P. Weatherill and O. Hassan. 1994. Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints. *Internat. J. Numer. Methods Engrg.* 37, 12 (1994), 2005–2039.

R. Wein, E. Berberich, E. Fogel, D. Halperin, M. Hemmer, O. Salzman, and B. Zukerman. 2018. 2D Arrangements. In *CGAL User and Reference Manual* (4.13 ed.). CGAL Editorial Board.

M. A. Yerry and M. S. Shephard. 1983. A Modified Quadtree Approach To Finite Element Mesh Generation. *IEEE Computer Graphics and Applications* 3, 1 (Jan 1983), 39–46.

H. Zhao, C. C. Wang, Y. Chen, and X. Jin. 2011. Parallel and efficient Boolean on polygonal solids. *The Visual Computer* 27, 6-8 (2011), 507–517.

Q. Zhou, E. Grinspun, D. Zorin, and A. Jacobson. 2016. Mesh Arrangements for Solid Geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 39.

Q. Zhou and A. Jacobson. 2016. Thingi10K: A Dataset of 10, 000 3D-Printing Models. *CoRR* abs/1605.04797 (2016). arXiv:1605.04797

A A BRIEF DESCRIPTION OF THE TETWILD ALGORITHM

The TetWild algorithm [Hu et al. 2018] takes a 3D triangle soup as input and generates a tetrahedral mesh that (1) has no inverted or degenerate tetrahedra and (2) contains an approximation of the input surface within a user-defined ϵ*-envelope*.

The method starts with an initial background mesh generated using an unconstrained Delaunay tetrahedralization on the input points plus an additional set of evenly-spaced points sampled from a regular grid. These additional points are added to improve the shape of the tetrahedra in the background mesh.

This step generates tetrahedra that might not represent the input faces of the triangle soup: to ensure that they are preserved TetWild uses a Binary Space Partitioning (BSP) subdivision step. Each input triangle is converted into a plane that cuts the tetrahedra of the background mesh. The output of this stage is a polyhedral mesh. To avoid numerical issues and to guarantee that the sub-elements in the polyhedral mesh are convex and non-inverted, TetWild converts the coordinates of the vertices of the initial background mesh into rational numbers and performs all computations using rational numbers.

Since any convex polyhedron can be trivially subdivided into tetrahedra by adding an additional point in its barycenter, a tetrahedral mesh that exactly preserves the input triangles can be naturally obtained after BSP subdivision. However, the vertices of this tetrahedral mesh are represented in rational coordinates. Rounding them to floating point is not simple, since the BSP subdivision introduces badly-shaped tetrahedra which could invert after rounding.

TetWild thus increases the quality of the elements using a hybrid optimization procedures that mixes floating point and rational representation. During this procedure, the preserved input triangle's faces are tracked and are allowed to move inside the ϵ-envelope. The ϵ-envelope limits the tracked surface from deviating from the input further than ϵ.

TetWild uses four local operations for mesh improvement: (1) edge splitting, (2) edge collapsing, (3) edge swapping, and (4) vertex smoothing. Every operation is rolled back if the tracked surface leaves the envelope after the operation or if any tetrahedra are inverted, ensuring a valid output. Differently from other mesh improvement methods, TetWild uses the 3D conformal AMIPS energy for measuring the geometric quality of the tetrahedra. The AMIPS energy is scaling-invariant and easily differentiable, which boosts these traditional local operations.

As the quality of the mesh is improved, the rational coordinates can be gradually rounded into floating points. Theoretically, there might be some unroundable vertices, but it does not occur on the ten thousand models that TetWild has been tested on.

The final step is the removal of the tetrahedra outside of the tracked surface. To handle potentially noisy inputs, TetWild computes the winding number of the centroids of all tetrahedra with respect to the tracked surface, and filters out all tetrahedra with centroid's winding-number larger than 0.5.

B EXAMPLE OF UNSTABLE AMIPS ENERGY

If we compute the 3D AMIPS energy for a tetrahedron with these 4 vertices

$$v_1 = (22.8289586180569, 31.46598870690956, 2.000000016196326)$$
$$v_2 = (22.83955896584259, 31.46598870610162, 2.000000016081439)$$
$$v_3 = (22.85206254968259, 31.46598870514861, 2.000000015945925)$$
$$v_4 = (22.83955896584259, 30.48801551784109, 2.616041190648805)$$

we obtain

$$\mathrm{AMIPS}_{1234} = 5.027711906288343\text{e}10$$
$$\mathrm{AMIPS}_{2341} = 2.171615254548946\text{e}11$$
$$\mathrm{AMIPS}_{3412} = 8.865129658843354\text{e}10$$
$$\mathrm{AMIPS}_{4123} = 7.103076229685612\text{e}10,$$

where the subscript indicates the vertex permutations. There are 24 permutations in total and here we pick 4 of them as an example. Even if we use the cube of the energy without rational we obtain fluctuations

$$\mathrm{AMIPS}^3_{1234} = 9.401446861483944\text{e}25$$
$$\mathrm{AMIPS}^3_{2341} = 1.834560196543814\text{e}25$$
$$\mathrm{AMIPS}^3_{3412} = 1.006679363250288\text{e}26$$
$$\mathrm{AMIPS}^3_{4123} = 3.462536408842030\text{e}26.$$

As reference the correct value computed with rational number is

$$\mathrm{AMIPS} = 1.127562687503913\text{e}11.$$

C UNUSED DECOMPOSITIONS OF A TETRAHEDRON

We enumerated all the possible decompositions of a tetrahedron and discovered two symmetry classes (Figure 31) of triangulation of faces whose decomposition requires an additional internal vertex. Note that these two cases are never selected by our algorithm (we include them here for completeness), as our rule (Section 3.4.2) never selects these two cases.

We show that our rule does not select case 1 (Figure 31 left). By contradiction: since the configuration is selected then the edges $[p_1, v_2]$, $[p_2, v_3]$ and $[p_3, v_1]$ are present, thus $v_2 > v_1$, $v_3 > v_2$, and $v_1 > v_3$, according to our rule. Combining these inequalities, the indices of the vertices must satisfy $v_3 > v_2 > v_1 > v_3$, which is impossible. Case 2 (Figure 31 right) is also not selected following a similar argument.

D AN EXAMPLE FOR OPEN-BOUNDARY EDGE PRESERVATION

If triangle T is the only inserted triangle and is entirely contained inside a tetrahedron $\mathcal{T}$ (Figure 32(1)), the intersection of the plane P and $\mathcal{T}$ will be a larger polygon (marked in yellow) containing T. In this case, the edges of T, which are open-boundary edges, are not preserved. To preserve them, we subdivide the tetrahedra once more.

In Figure 32(1), $\mathcal{T}$ first get decomposed into sub-tetrahedra (Figure 32(2)). Then the faces covering T are $\mathcal{F} = \{[p_1, p_2, p_3], [p_4, p_2, p_3]\}$ Figure 32(3). The open-boundary edge e and the faces in $\mathcal{F}$ are projected to the best-fitting plane of p_1, p_2, p_3, and p_4. The intersection points of the projection of e and $\mathcal{T}$ are then computed in 2D and are lifted to 3D (3 green points in Figure 32(3)). Now there are 3 edges $[p_1, p_2]$, $[p_2, p_3]$, $[p_3, p_4]$ cut into two. We thus subdivide all the neighbouring tetrahedra with the table-based subdivision.

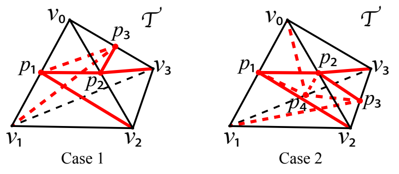


Fig. 31. Two unused configurations requiring an additional vertex.

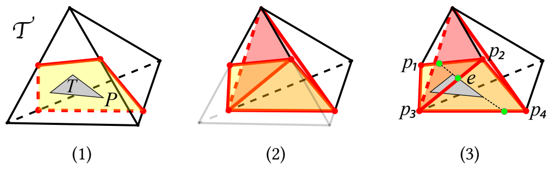


Fig. 32. Example for preserving an open-boundary edge e of triangle T. (1) Insert T and $\mathcal{T}_I = \{\mathcal{T}\}$ in this case. (2) The sub-tetrahedra of $\mathcal{T}$ after subdivision. (Only sub-tetrahedra behind T are shown for better visualization.) (3) Inserting edge e and get the intersection points (in green).

Table 3. Edge-cut configurations of a cutting tetrahedron before and after snapping. Numbers corresponds to the configurations in Figure 9.

Before	1 vertex snapped	2 vertices snapped	3 vertices snapped
(2)	(1)	(2)	(1)
(3)	(1)(2)	(1)(2)	(1)
(5)	(1)(3)	(1)(2)	(1)(2)
(7)	(3)	(1)(3)	(1)

E CHANGES OF EDGE-CUT CONFIGURATION AFTER SNAPPING

Table 3 shows all possible edge-cut configurations of a cutting tetrahedron $\mathcal{T}$ after snapping. The final configurations have no more than two vertices which makes the triangulation of $\mathcal{F}$ uniquely defined by the points. The table includes only the 4 symmetry classes where $\mathcal{T}$ is cut by plane P and contains a face in $\mathcal{F}$ (Figure 9 (2)(3)(5)(7)), but excludes the remaining 3 classes where $\mathcal{T}$ is not cut or is just affected by their neighbors (Figure 9 (1)(4)(6)).

A tetrahedron $\mathcal{T}$ can have at most 3 vertices snapped. If $\mathcal{T}$ has all its 4 vertices within a δ distance to the P, we only snap the 3 vertices closer to P.