Unstructured Mesh Generation and Repairing in the Wild

by

Yixin Hu

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January, 2022

[illegible]

Professor Daniele Panozzo

Acknowledgements

This thesis would not have been possible without the support of many people.

First of all, I would like to express my sincere gratitude to my dissertation adviser, Prof. Daniele Panozzo, who guided me into the world of meshing. He provided me with valuable suggestions, continuous encouragement, and motivating guidance. I am fortunate to have been a part of the Geometric Computing Lab at NYU.

I extend my sincere thanks to Prof. Denis Zorin who co-advised me on most of the research works in the dissertation. He provides insights from different aspects and high-level suggestions. I have benefited greatly from his wealth of knowledge.

I would like to say a special thank you to Prof. Wenping Wang who supervised me during my internship at HKU in my undergraduate, sparked my interest in geometry processing, and wrote me a recommendation for my graduate school application.

I also want to give heartfelt thanks to my collaborators, Dr. Qingnan (James) Zhou, Prof. Teseo Schneider, Prof. Alec Jacobson, Dr. Xifeng Gao, Dr. Bolun Wang, and Dr. Sebastian Koch, for their contribution to the research work in the dissertation, for the stimulating discussions, and for the sleepless nights we were working together before the deadlines. Thanks to Qingnan Zhou for mentoring me during my internships at Adobe Research.

I was luckily involved in research projects related to my dissertation and had many talented people as my coauthors: Dr. Jeremie Dumas, Dr. Marco Attene, Zhongshi Jiang, Ziyi Zhang. Thanks to Jeremie Dumas who also helped during the refactoring of the implementation of

TetWild (Chapter 2). I learned from them in both research and engineering.

I would also like to acknowledge Prof. Chee Yap and Prof. Doug James who so generously took time out of their schedules to participate in my dissertation committee and provided the valuable suggestion.

I have been fortunate to have the opportunity to discuss my research with Prof. Tamy Boubekeur and Prof. Marco Tarini as they visited our lab, and with Dr. Danny Kaufman and Prof. David Bommes during the conferences. Thanks for the enlightening discussions

I am also very grateful to the NYU HPC team, especially Shenglong Wang, for the support and help in using the clusters and the convenience offered before deadlines.

Thanks to Bradley Rothenberg for hosting my internship at nTopology Inc. and Trevor Laughlin for providing valuable advice and contributing to the TetWild GitHub repository. Also, thanks to Dr. Yujia Chen and Dr. Michael Goesele for mentoring me during my internship at Facebook Inc., giving me a chance to learn more about 3D reconstruction.

Thanks to New York University for awarding me MacCracken Fellowship in 2016, Jacob T. Schwartz Ph.D. fellowship in 2019, and Sandra Bleistein Prize in 2021, and Adobe System Inc. for awarding me Adobe Research Fellowship in 2019, providing me with the financial support during my Ph.D. study.

Lastly and most importantly, I would like to express my profound gratitude to my parents for their continuous support and love during my many years of study. I am forever indebted to my parents for giving me the opportunities and experiences that have made me who I am.

Abstract

A mesh is a representation used to digitally represent the boundary or volume of an object for manipulation and analysis. Meshes can be used in many fields, including physical simulation in manufacturing, architecture design, medical scan analysis. In this thesis, we propose a series of meshing algorithms, named WildMeshing, that tackles one of the long-standing, yet fundamental, problems in geometry modeling: robustly and automatically generating high-quality triangle and tetrahedral meshes and repairing imperfect geometries in the wild. Different from existing methods that have assumptions about the input and thus often fail on real-world input geometries, WildMeshing provides strict guarantees of termination and is a black box that can be easily integrated into any geometry processing pipelines in research or industry.

This thesis first investigates the problem of tetrahedralizing 3D geometries represented by piecewise linear surfaces. We propose an algorithm, TetWild, that is unconditionally robust, requires no user interaction, and can directly convert a triangle soup into an analysis-ready volumetric tetrahedral mesh. It relies on three core principles: hybrid geometric kernel, tolerance of the mesh relative to the surface input, and iterative mesh optimization with guarantees on the output validity. We then consider improving the algorithm efficiency for tetrahedralizing large-scale geometries. We design a new algorithm, fTetWild, that is based on the principles of TetWild but replaces the hybrid kernel with a floating-point kernel, which largely reduces runtime while keeping the same robustness. Next, this thesis explores meshing curved geometries. We start from the problem of triangulating 2D planar shapes whose boundaries are represented by curves. We introduce TriWild, an algorithm to robustly generate curved triangle meshes reproducing smooth feature curves, which leads to coarse meshes designed to match the simulation requirements necessary by applications and avoids the geometrical errors introduced by linear meshes.

We test our algorithms on over ten thousand real-world input geometries and they achieve 100% success rate. Our methods generate meshes without any assumptions about the input while repairing the imperfect geometries, opening the door to automatic, large-scale processing of real-world geometric data.

Contents

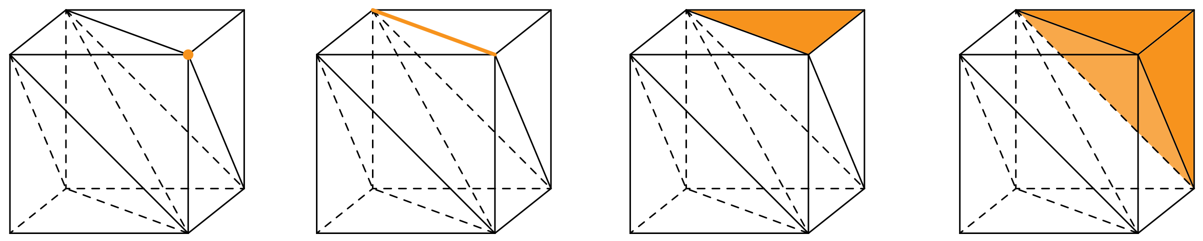
List of Figures

List of Tables

1 Introduction

Meshing is the practice to generate the discrete representation, *mesh*, of a shape and is an indispensable part of computer-aided design (CAD) systems. Meshes can be used to define the shape or volume of objects (Figure 1.1). A surface mesh is a collection of vertices, edges and faces (polygons) that defines the shape of a polyhedral object. A volumetric mesh also contains cells (polyhedra) that define the volume of a solid object. Meshes are used almost everywhere in computer graphics, like rendering, modeling, or Finite Element Method based simulations. Triangulation and tetrahedralization are the most popular and fundamental meshing techniques for generating triangle meshes for 2D shapes and tetrahedral meshes for 3D shapes.

Ideally, a meshing algorithm is a black box: given the boundary of a shape, the algorithm always successfully produces a mesh with high geometric quality conforming to the input boundary. A black box meshing algorithm enables the possibility of making the most of the countless shapes *in the wild* (whose boundary representation may be imperfect) and convert them into a



(a) Vertex (b) Edge (c) Face (Triangle) (d) Cell (Tetrahedron)

Figure 1.1: The tetrahedral mesh of a cube. (a) A vertex highlighted in orange. (b) An edge highlighted in orange. (c) A triangular face of the mesh in orange. (d) A tetrahedral cell in orange.

large collection of clean analysis-ready meshes. These large datasets can be directly integrated into fabrication pipelines, used in large-scale verification for downstream algorithms, or used for benchmarking machine learning algorithms. One of the current major challenges in geometric deep learning is the lack of large scale datasets.

Surprisingly, despite of great research effort and considerable academic and industrial interest, there is no robust and automatic meshing algorithm able to handle input in the wild with artifacts like self-intersections, small gaps, and sharp features. Taking tetrahedral meshing as an example, we tested a collection of popular tetrahedral meshing methods on a large dataset containing 10,000 real-world input, and obtained low success rates ranging from 37.1% to 87.2% (Table 3.3). Their robustness is insufficient for supporting black-box processing. Note that introducing artifacts on the surface is unfortunately common during the real-world design procedure and to the best of my knowledge, there is no existing algorithm that can automatically clean up a defective geometry. Users have to clean up the surface manually, which is a tedious and labor-intensive work. For example, the challenging model in Figure 3.26 has a multitude of issues introduced during the modeling phase and requires about two weeks to fix it manually. In many cases, even if meshing may succeed, the output mesh could be immense due to over-refinement or contains low-quality elements, which makes the later application prohibitively expensive or not sufficiently accurate. Even if the balance between size and quality is achieved, hard-to-detect features of the input may be lost as the examples in Figure 2.14.

In this scenario, I propose WildMeshing, a family of meshing algorithms: TetWild [Hu et al. 2018] and fTetWild [Hu et al. 2020] for linear tetrahedralization, TriWild [Hu et al. 2019] for 2D curved triangulation, and cTetWild for 3D curved tetrahedralization, which I am currently working on. *My goal is to shift the paradigm and design black-box triangulation and tetrahedralization algorithms verified by large-scale testing and able to produces high-quality output robustly and automatically.*

Linear tetrahedral meshing takes as input the linear surface representation of an 3D object, uses linear tetrahedra to fill in the interior space of the object, and outputs a linear tetrahedral mesh that conforms to the input surface exactly or approximately depending on the downstream application. Different from previous methods, We reformulate the meshing problem accordingly: Our new algorithms, TetWild and fTetWild, mesh volumes represented by arbitrary surface meshes, with no assumptions on surface manifoldness, watertightness, or self-intersections etc. Instead of viewing surface repair as a separate preprocessing problem, we recognize the fact, that "clean" surfaces are more of an exception than a rule in many settings.

Thus, we design TetWild algorithm (Chapter 2) that is unconditionally robust, requires no user interaction, and can directly convert a triangle soup into an analysis-ready volumetric mesh. The approach is based on several core principles: (1) initial mesh construction based on a fully robust, yet efficient, filtered exact computation (2) explicit (automatic or user-defined) tolerancing of the mesh relative to the surface input (3) iterative mesh improvement with guarantees, at every step, of the output validity. The quality of the resulting mesh is a direct function of the target mesh size and allowed tolerance: increasing allowed deviation from the initial mesh and decreasing the target edge length both lead to higher mesh quality.

We then propose fTetWild (Chapter 3) that 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.

TetWild and fTetWild generate high-quality tetrahedral meshes with clean surface and thus can also be used for automatic surface repairing. Our approaches enable "black-box" analysis and allow to automatically solve partial differential equations on geometrical models available in the wild, offering a robustness and reliability comparable to, e.g., image processing algorithms, opening the door to automatic, large scale processing of real-world geometric data.

Curved triangular Meshing whose input and output are curved and can make precise reproduction of curved shapes, such as Bézier curves, independently of the resolution used. Curved meshes, i.e. meshes with curved edges and faces, provides significantly superior geometric approximation of a shape in a small size like the example in Figure 4.14. In downstream simulation application, the smaller mesh size leads to higher efficiency for a given desired accuracy.

We propose a robust 2D meshing algorithm, TriWild (Chapter 4), to generate curved triangles reproducing smooth feature curves, leading to coarse meshes designed to match the simulation requirements necessary by applications and avoiding the geometrical errors introduced by linear meshes. The robustness and effectiveness of our technique are demonstrated by batch processing an SVG collection of 20k images, and by comparing our results against state-of-the-art linear and curvilinear meshing algorithms. We demonstrate for our algorithm the practical utility of computing diffusion curves, fluid simulations, elastic deformations, and shape inflation on complex 2D geometries.

Chapter 5 concludes the dissertation by summarizing the contribution and limitation of WildMeshing algorithms. We also discuss as feature work the potential method to tetrahedralize 3D CAD models with the input features preserved.

2 | Tetrahedral Meshing in the Wild

This chapter introduces our work, TetWild, for robustly and automatically generating high-quality tetrahedral meshes for possibly imperfect geometries in the wild.

2.1 Introduction

Triangulating the interior of a shape is a fundamental subroutine in 2D and 3D geometric computation.

For two-dimensional problems requiring meshing a domain, robust and efficient software for constrained Delaunay triangulation problem has been a tremendous boon to the development of robust and efficient automatic computational pipelines, in particular ones requiring solving PDEs. Robust 2D triangulations inside a given polygon boundary are also an essential spatial

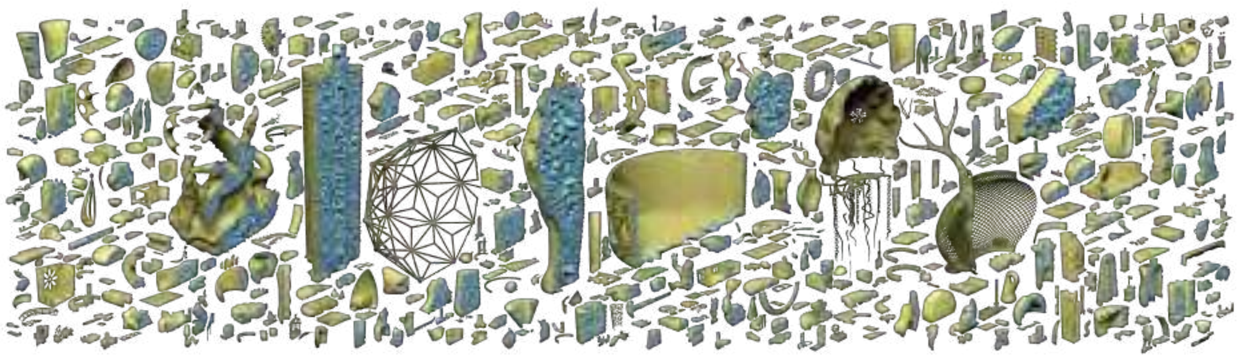


Figure 2.1: A selection of the ten thousand meshes *in the wild* tetrahedralized by our novel tetrahedral meshing technique.

partitioning useful for fast point location, path traversal, and distance queries.

In 3D, the problem of robustly triangulating the interior of a given triangle surface mesh is just as well, if not more, motivated. While tremendous progress was made on various instances of the problem, it is far from solved by existing methods. While pipelines involving 3D *tetrahedralization* of smooth implicit surfaces are quite mature, pipelines using meshes as input either are limited to simple shapes or routinely fallback on manual intervention. The user may have to "fix" input surface meshes to cajole meshers to succeed due to unspoken pre-conditions, or output tetrahedral meshes must be repaired due to failure to meet basic post-conditions (such as manifoldness). Existing methods typically fail too often to support automatic pipelines, such as massive data processing for machine learning applications, or shape optimization. In many cases, while meshing may succeed, the size of the output mesh may be prohibitively expensive for many applications, because a method lacks control between the quality of approximation of the input surface and the size of the output mesh. Even when such controls are present, hard-to-detect features of the input mesh may not be preserved.

In this paper, we propose a new approach to mesh domains that are represented (often ambiguously) by arbitrary meshes, with no assumptions on mesh manifoldness, watertightness, absence of self-intersections etc. Rather than viewing mesh repair as a separate preprocessing problem, we recognize the fact, that "clean" meshes are more of an exception than a rule in many settings.

The key features of our approach, based on careful analysis of practical meshing problems, and shortcomings of existing state of the art solutions are:

- We consider the input as fundamentally imprecise, allowing deviations from the input within user-defined envelope of size ϵ;

- We make no assumptions about the input mesh structure, and reformulate the meshing problem accordingly;

- We follow the principle that robustness comes first (i.e., the algorithm should produce a valid and, to the extent possible, useful output for a maximally broad range of inputs), with quality improvement done to the extent robustness constraints allow.
- While allowing deviations from the input, which is critical both for quality and performance, we aim to make our algorithm conservative, using the input surface mesh as a starting point for 3D mesh construction, rather than discarding its connectivity and using surface sampling only.

Our method is explicitly designed to output floating point coordinates, but at the same time is strictly *closed* under rationals allowing it to fit neatly into robust, exact rational computational geometry pipelines.

We empirically compare both the performance and robustness of state-of-the-art methods and our novel method on a large database of 10 thousand models from the web [Zhou and Jacobson 2016b]. To foster replicability of results, we release a complete reference implementation of our algorithm, all the data shown in the paper, and scripts to reproduce our results.

Our method –while slower– demonstrates a significant improvement in robustness and quality of the results on a number of quality measures, when applied to meshes found *in the wild*.

2.2 Related work

Tetrahedral mesh generation has remained a perennial problem, both for computational geometers and practitioners in graphics, physics and engineering ([Cheng et al. 2012a; Carey 1997; Owen 1998]). We are specifically interested in methods that are *constrained* to output a 3D tetrahedral mesh whose 2D surface closely matches an input surface. We categorize related work with respect to the high-level methodology employed. We place special emphasis on methods with reproducible results thanks to their openly accessible implementations. One confusion during comparisons is that most existing software implements *multiple* algorithms, triggered discretely (and somewhat discreetly) by input flags or parameters (e.g., TetGen or CGAL). Our comparisons are done in best faith and using default parameters where applicable; when controls similar to the ones used in our method are available, we tried to choose them in a similar way.

Background Grids In 3D, a regular lattice of points is trivial to tetrahedralize (e.g., using either five, six, or 12 tetrahedra per cube). To tetrahedralize the interior of a solid given its surface, *grid-based* methods fill the ambient space with either a uniform grid or an adaptive octree. Grid cells far from the surface can be tetrahedralized immediately and efficiently using a predefined, combinatorial stencil, with excellent quality. Trouble arises for boundary cells.

Molino et al. [Molino et al. 2003] propose the red-green tetrahedron refinement strategy, while cells intersecting the domain boundary are pushed into the domain via physics-inspired simulation. Alternatively, boundary cells can be cut into smaller pieces [Bronson et al. 2012]. Labelle & Shewchuck [Labelle and Shewchuk 2007a] snap vertices to the input surface and cut crossing elements. This method provides bounds on dihedral angles and a proof of convergence for sufficiently *smooth* (bounded curvature) isosurface input. Doran et al. [Doran et al. 2013] improves this method to detect and handle feature curves, providing an open source implementation, Quartet [Bridson and Doran 2014a] with which we thoroughly compare. *Average* element quality tends to be good: for volumes with high volume-to-surface ratio, most of the mesh will be filled by the high-quality stencil. Near the boundary, grid-based methods struggle to simultaneously provide parsimony and element quality: either the surface is far denser than the interior making volume gradation difficult to control or the surface is riddled with low-quality elements.

Delaunay The problem of tetrahedralizing a *set of points* is very well studied [Cheng et al. 2012a; Sheehy 2012]. Efficient, scalable [Remacle 2017] algorithms exist to create Delaunay meshes.

When the input includes surface mesh constraints, the challenge is to extend the notion of a Delaunay mesh in a meaningful way. In two dimensions, constrained Delaunay methods provide a satisfactory solution. In contrast to 2D, the situation in 3D is immediately complicated by the fact that there exist polyhedra that cannot be tetrahedralized without adding extra interior Steiner vertices [Schönhardt 1928].

The simple and elegant idea of *Delaunay refinement* [Chew 1993; Shewchuk 1998; Ruppert 1995] is to insert new vertices at the center of the circumscribed sphere of the worst tetrahedron measured by radius-to-edge ratio. This approach guarantees termination and provides bounds on radius-edge ratio. This approach has been robustly implemented by many [Si 2015a; Jamin et al. 2015], and, in our experiments, proved to be consistently successful. However, robustness problems immediately appear if the boundary facets have to be preserved.

More importantly, even in situations when the method is guaranteed to produce a mesh with bounded radius-to-edge ratio, it does not –unlike the 2D case– guarantee that quality measures relevant for applications are sufficiently good. The notorious "sliver" tetrahedra satisfy the radius-to-edge ratio criteria. Thus, unavoidably, Delaunay refinement needs to be followed by various mesh improvement heuristics: exudation [Cheng et al. 2000], Lloyd relaxation [Du and Wang 2003], ODT relaxation [Alliez et al. 2005a], or vertex perturbation [Tournois et al. 2009]. Our approach also relies on a variational-type mesh improvement (Section 2.3.2). *Conforming Delaunay tetrahedralization* [Murphy et al. 2001; Cohen-Steiner et al. 2002] splits input boundary by inserting additional Steiner points, until all input faces appear as supersets of element faces. Even with additional assumptions on the input, this process may require impractically many additional points and tetrahedra. In contrast, *constrained Delaunay tetrahedralization* [Chew 1989; Si and Gärtner 2005; Shewchuk 2002a; Si and Shewchuk 2014] proposes to relax the Delaunay requirement for boundary faces so fewer Steiner points are needed. The popular open source software TetGen [Si 2015a] is based on constrained Delaunay tetrahedralization, enforcing inclusion of input faces in the mesh.

Restricted Delaunay tetrahedralization [Cheng et al. 2008; Boissonnat and Oudot 2005] completely resamples the input surface to obtain better tet quality while generating a good approximation of the domain boundary at the same time. The software DelPSC and CGAL 3D tetrahedral meshing module [Jamin et al. 2015; Dey and Levine 2008] is based on this approach. Engwirda [Engwirda 2016] uses an advancing front method as a refinement and point placement strategy for constructing a restricted Delaunay mesh.

Variations of these methods are difficult to implement robustly, as in their original form they require exact predicates that go beyond the typically available set, so a careful reduction to the robustly implementable operations is needed. This may account for a percentage of failures that we observe.

A conceptual feature of many restricted Delaunay meshers (using meshes as input) is that they do not allow any slack on the boundary geometry, thus requiring heavy refinement in certain cases to achieve acceptable quality, for any target tetrahedron size. However, tetrahedra incident at features are invariably excluded from quality improvement.

In contrast, our algorithm by design, admits practical robust implementation, and, also by design, allows the surface to change within user-specified bounds, which greatly reduces unnecessary over-refinement due to surface irregularities.

The state-of-the-art method based on restricted Delaunay refinement, [Jamin et al. 2015], is highly robust for important classes of inputs (smooth implicit surfaces) and yields high-quality meshes. However, as we demonstrate in the results section, if the input is polygonal, it cannot be easily reduced to the problem of meshing an implicit surface, due to nonsmoothness, and the need for feature preservation. Currently, [Jamin et al. 2015] and related methods preserve features using the protection ball method: spheres are placed on feature points and weighted Delaunay meshing and refinement are performed, treating ball radii as point weights. This approach requires explicit detection and representation of feature lines; in its current form, it results in reduction of robustness and in some cases over refinement.

Variational meshing The duality between Delaunay meshes and Voronoi diagrams, leads to a *variational* or energy-minimizing view of the meshing problem. Centroidal Voronoi Tessellation energy minimizers can leverage Lloyd's algorithm of BFGS optimization to produce regular or adaptive meshes with well spaced vertices [Du and Wang 2003], though this does not guarantee good element quality [Eppstein 2001]. An alternative is to minimize the "Optimal Delaunay Triangulation" energy [Chen and Xu 2004; Alliez et al. 2005a], for better element quality. These algorithms require an initial starting point (which cannot be generated starting from noisy input geometry), in order to stay near any input surface constraints. Our method is designed to generate this valid starting point, and it then uses a variant of these methods, which is designed to work with a hybrid kernel, to improve quality.

Other variational mesh improvement methods exist [Klingner and Shewchuk 2007; Gargallo-Peiró et al. 2013; Misztal and Bærentzen 2012], but all require and depend heavily on the initial base mesh. In contrast, we propose a complete meshing algorithm. Our first step generates a base mesh that complements our choice of mesh improvement strategy later on. The result is unprecedented robustness and element quality.

Tetrahedral meshing is a hard problem. The strategies found in the literature span a wide range of ideas, from the use of machine learning to predict hard cases [Chen et al. 2012] to the various advancing front methods to generate initial meshes [Cuillière et al. 2012; Alauzet and Marcum 2013; Haimes 2015]. The quality of advancing front outputs can be deceptive: problems are pushed into the interior. Even if the exterior looks perfect, quality in the interior may be arbitrarily poor. We found no reliable advanced front methods suitable for our full-scale comparison.

Surface Envelope. Explicit envelopes have been used to guarantee a bounded approximation error in surface reconstruction. Shen et al. [2004] convert a polygon soup into an implicit representation using a novel interpolation scheme, where a watertight ϵ-isosurface can be extracted for surface approximation purposes. Mandad et al. [2015a] create an isotopic surface approximation within a tolerance volume using a modified Delaunay refinement process followed by an envelope-aware and topology-preserving simplification procedure. Our approach uses a similar, implicit, ϵ-envelope to ensure that the tracked surface does not move too far from the input triangle soup.

2.3 Method

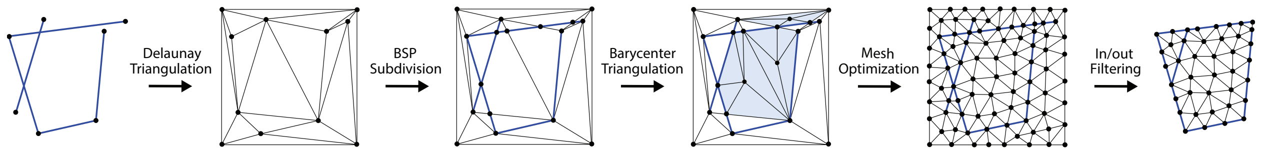


Figure 2.2: A diagram illustrating the pipeline of our algorithm in 2D. The points of the original input segments (left) are triangulated using Delaunay triangulation (second left). Each line segment is then split by all triangles that intersect it, constructing a BSP-tree (third left). Each of the resulting convex polygons (colored blue) is divided into triangles by adding a point at its barycenter and connecting it to the vertices of the polygon (third from the right). Local operations are used to improve the quality (second from the right), and finally winding number is used to filter out the elements outside of the domain (right).

We start by defining our problem more precisely. As input we assume a triangle soup, a user-specified tolerance ϵ, and a desired target edge length ℓ. *The goal is to construct an approximately constrained tetrahedralization, that is, a tetrahedral mesh that (1) contains an approximation of the input set of triangles, within user-defined ϵ of the input, (2) has no inverted elements, and (3) edge lengths below user-defined bound ℓ. Mesh quality is optimized while satisfying these constraints.* We call a mesh *valid* if it satisfies the first two properties.

The resulting tetrahedralization can be used for a variety of purposes; most importantly, we can use any definition of the interior of a set of triangles to extract a tetrahedralized volume contained "inside" the input triangle soup.

Throughout this paper, we use the term *surface* to refer to collections of faces, not necessarily manifold, connected, or self-intersection free. Our algorithm tackles this problem in two distinct phases: (1) the generation of a valid mesh, disregarding its geometric quality, representing its coordinates with arbitrary-precision rational numbers and (2) improvement of the geometric quality of its elements and rounding the coordinates of the vertices to floating point numbers, while preserving the validity of the mesh. Decoupling these two sub-problems is the key to the robustness of our algorithm and it is in contrast with the majority of competing methods, which attempt to directly generate a high-quality mesh.

The first phase relies only on operations closed under rational numbers, i.e., the entire computation can be performed exactly if the vertex coordinates are rational, sidestepping all robustness issues (but increasing the computational cost). The second phase uses a hybrid geometric kernel (inspired by [Attene 2017]), allowing us to switch to floating point operations whenever possible to keep the running time sensible (Section 2.3.4). Our algorithm is thus guaranteed to produce a valid mesh (Phase 1), but we cannot provide any formal bound on its quality (Phase 2): in practice, the quality obtained with our prototype on a dataset of ten thousand *in the wild* models is high (Section 2.4).

Overview. The algorithm creates a volumetric Binary Space Partitioning (BSP) tree, containing one plane per input triangle and storing its coordinates as exact rational numbers. By construction, the resulting convex (but not necessarily strictly convex) cell decomposition is conforming to the input triangle soup, and a tetrahedral mesh can be trivially created by independently tetrahedralizing each cell (Section 2.3.1). The volumetric mesh is not only created inside the model, but also around the model, filling a bounding box slightly larger than the input. This allows us to robustly deal with imperfect geometry that contains gaps or self-intersections, postponing the inside/outside segmentation of the space to a later stage in the pipeline. The quality of the mesh is then optimized with a set of local operations to refine, coarsen, swap, or smooth the mesh elements (Section 2.3.2). These operations are performed only if they do not break a set of invariants that ensure the validity of the mesh at each step. The final mesh is then extracted using winding-number filtering [Jacobson et al. 2013], which is robust to imperfect, real-world input

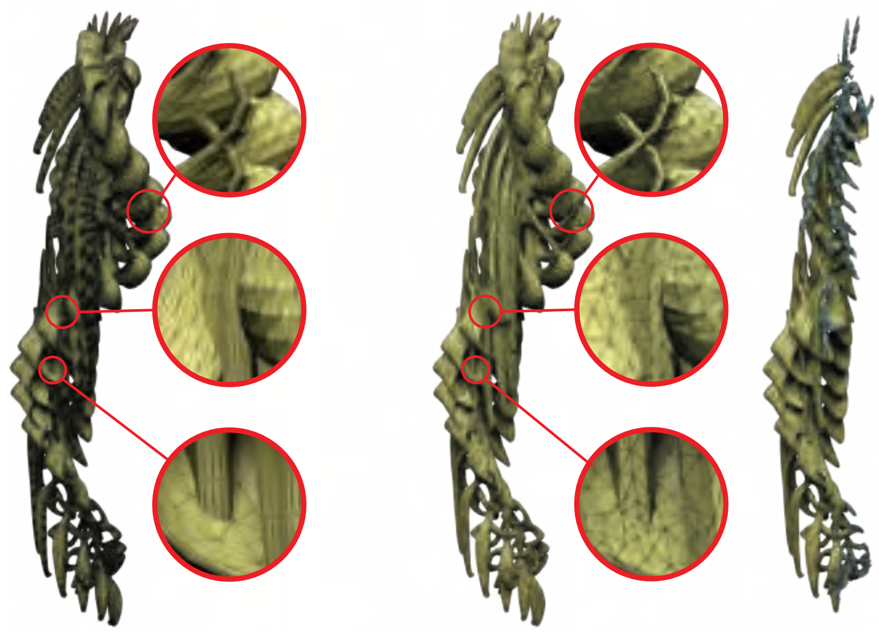


Figure 2.3: Self-intersections in the input (left) are automatically handled by our meshing algorithm (right).

(Section 2.3.3).

2.3.1 Generation of a Valid Tetrahedral Mesh

The robust generation of a valid tetrahedral mesh that preserves the faces of an original triangle soup is challenging, even ignoring any quality consideration. Real-world meshes are often plagued by a zoo of defects, including degenerate elements, holes, self-intersection, and topological noise [Zhou and Jacobson 2016b; Attene et al. 2013]. Even manually modeled CAD geometry cannot be exported to a clean boundary format, since the most common modeling operations are not closed under spline representation [Farin 2002; Sederberg et al. 2003], unavoidably leading to small "cracks" and self-intersections. Cleaning polygonal meshes or CAD models is a long-standing problem, for which bullet-proof solutions are still elusive [Attene et al. 2013]. We thus propose to use the input geometry as is, and rely on a robust geometrical construction to fill the entire volume with tetrahedra, without committing to the exact topology or geometry of the boundary at this stage, and postponing this challenge to a later stage in the pipeline, after all degeneracies have been removed.

BSP-Tree Approach. We build an exact BSP subdivision, using infinite-precision rational coordinates, and only relying on operations closed under this representation. An illustration of the pipeline in 2D is shown in Figure 2.2: we use a 2D illustration since it is difficult to visualize the effect of operations on tetrahedral meshes in a static figure. In contrast to the surface-conforming Delaunay tetrahedralization [Si 2015a], for which designing a robust implementation is challenging (Section 2.2), the unconstrained version can be robustly implemented with exact rational numbers [Jamin et al. 2015]. We thus create an initial, non-conforming tetrahedral mesh $\mathcal{M}$, whose vertices are the same as the input triangle soup, using the exact rational kernel in CGAL [Jamin et al. 2015].

The generated tetrahedral mesh does not preserve the input surface, making it unusable for most downstream applications. To enforce conformity, we use an approach inspired by [Joshi and Ourselin 2003], but designed to guarantee a valid output. We consider each triangle of the input triangle soup as a plane, and intersect it with all the tetrahedra in $\mathcal{M}$ that contain it. In other words, we consider each tetrahedron as the root of a BSP cell, and we cut the cell using all the triangles of the input geometry intersecting it. This computation can be performed entirely using rational coordinates, since intersections between planes are closed under rationals, ensuring robustness and correctness even for degenerate input. This polyhedral mesh is converted into a tetrahedral mesh taking advantage of convexity of the cells: we triangulate its faces, add a vertex at the barycenter, and connect it to all the triangular faces on the boundary. Since the only operation necessary is an average of vertex positions, the barycenter can be computed exactly with rationals. As long as at least four input vertices are linearly independent, then *all* convex cells will be non-degenerate, i.e., the resulting tetrahedra connected to the barycenter will also be non-degenerate (though perhaps poor quality). The output mesh is *valid* and exactly conforming to the input triangle soup. Self-intersections in the input are naturally handled by this formulation: they are explicitly meshed, splitting the corresponding triangles accordingly (Figure 2.3).

2.3.2 Mesh Improvement

Given a valid tetrahedral mesh represented using rational numbers, we propose an algorithm to improve its quality, and round its vertices to floating point positions, while preserving its validity. We follow the common greedy optimization pipeline based on local mesh improvement operations [Dunyach et al. 2013; Faraj et al. 2016; Freitag and Ollivier-Gooch 1997], but with four important differences:

1. We explicitly prevent inversions using exact predicates (*Validity Invariant 1*).
2. We track the surface mesh during the operations, and we only allow operations that keep them within an ϵ distance from the input triangle mesh (inspired by a similar criteria used for surface meshing by [Hu et al. 2017]) (*Validity Invariant 2*).
3. We directly penalize bad elements in all shapes using a conformal energy which has been recently introduced for mesh parametrization [Rabinovich et al. 2017].
4. We use a hybrid geometric kernel to reduce the computation time while ensuring correctness and termination, using floating point whenever possible and relying on exact coordinates only where it is strictly necessary.

Invariant 1: Inversions. We disallow every operation introducing inverted tetrahedra whose orientation is negative, using the exact predicates in [Brönnimann et al. 2017] for both rational and floating point coordinates. This ensures an output without inversions, since the algorithm starts from an inversion-free tetrahedral mesh produced by our BSP-tree construction (Section 2.3.1).

Invariant 2: Input Surface Tracking and Envelope. By construction, the tetrahedral mesh produced in Section 2.3.1 contains an exact representation of all input triangles, in the form of a collection of faces of the tetrahedra. That is, the tetrahedral mesh contains one (or more) tetrahedra whose faces exactly match any given input triangle. We call this collection of faces the *embedded surface*, and all operations performed on the tetrahedral mesh keeps track of it.

To bound the geometric approximation error introduced during the mesh improvement procedure, we only accept operations that keep the faces of the embedded surface at a distance smaller than a user-defined ϵ. Intuitively, this can be depicted as an *envelope* of thickness ϵ built around the input triangle soup. We ensure that the embedded surface is always contained in the envelope at all times by disallowing any operation breaking this invariant (Figure 2.4).

Quality Measure. As a measure of quality to optimize, we use the 3D conformal energy recently explored in [Rabinovich et al. 2017], which is well-correlated with many common measures of quality (we evaluate the results on a number of measures). It is expressed as:

$$\mathcal{E} = \sum_{t \in T} \frac{\mathrm{tr}(\mathbf{J}_t^T \mathbf{J}_t)}{\det(\mathbf{J}_t)^{\frac{2}{3}}} \tag{2.1}$$

where $\mathbf{J}_t$ is the Jacobian of the unique 3D deformation that transforms the tetrahedron t into a regular tetrahedron. This energy is oblivious to isotropic scaling, but naturally penalizes needle-like elements, flat and fat elements, slivers, and prevents inversions since it diverges to infinity as an element approaches zero volume. It is also differentiable [Rabinovich et al. 2017], and can be efficiently minimized using Newton or Quasi-Newton iterations [Kovalsky et al. 2016; Rabinovich et al. 2017].

Local Operations. We use four local operations for mesh improvement [Freitag and Ollivier-Gooch 1997; Faraj et al. 2016]: edge splitting, edge collapsing, face swapping, and vertex smoothing (Figure 2.5). These operations only affect a local region of the mesh, and can thus be performed

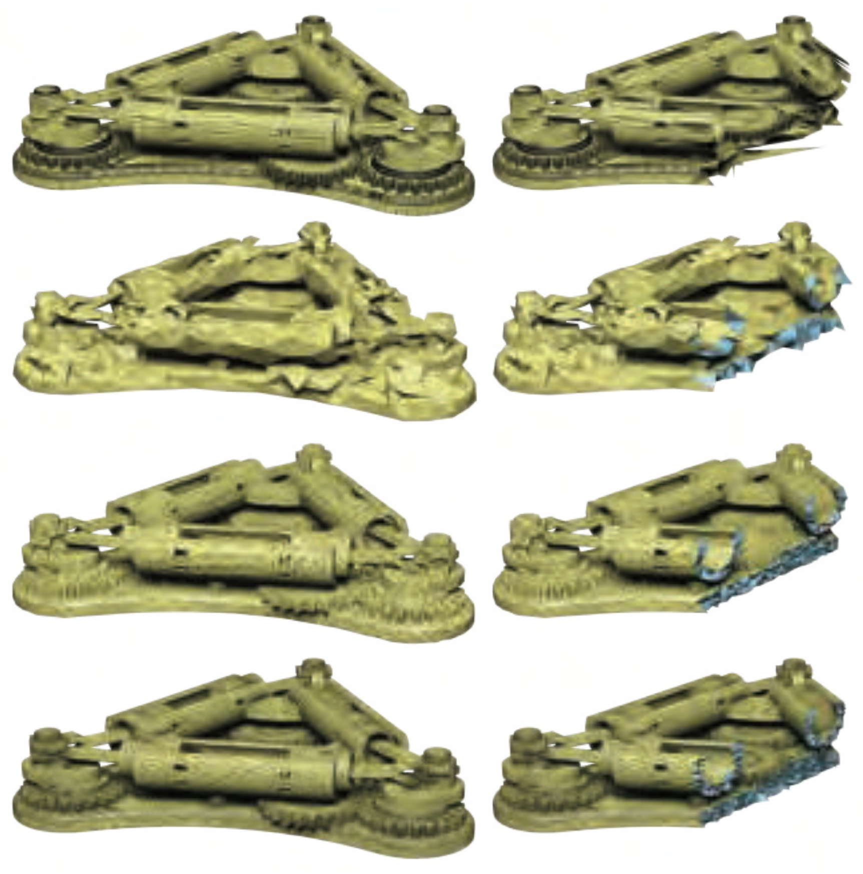


Figure 2.4: An oversized ϵ ($\frac{b}{100}$, with b being the bounding box diagonal) creates a tetrahedral mesh (2nd row) that fails to capture the features of the input triangle mesh (1st row). Reducing ϵ to $\frac{b}{300}$ and $\frac{b}{3000}$ increases the geometric fidelity (3rd and 4th row).

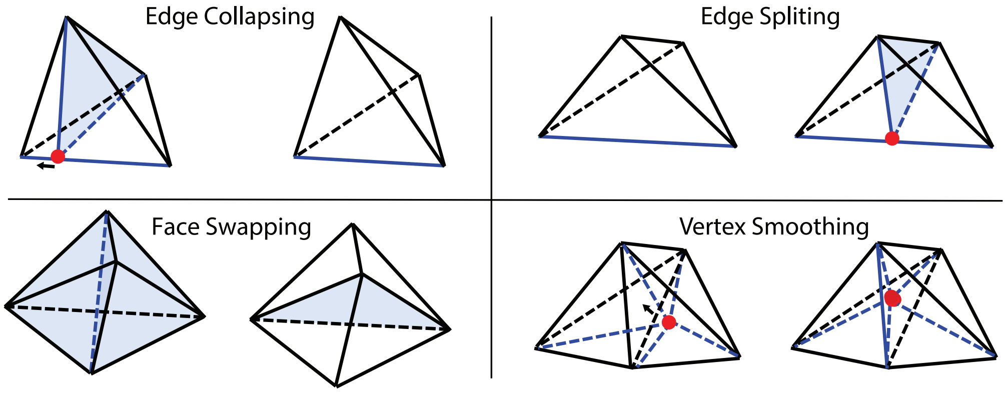


Figure 2.5: Overview of the local mesh improvement operations. For face swapping, our algorithm uses 3-2, 4-4, 5-6 bistellar flips [Freitag and Ollivier-Gooch 1997], where 3-2 flip is illustrated here.

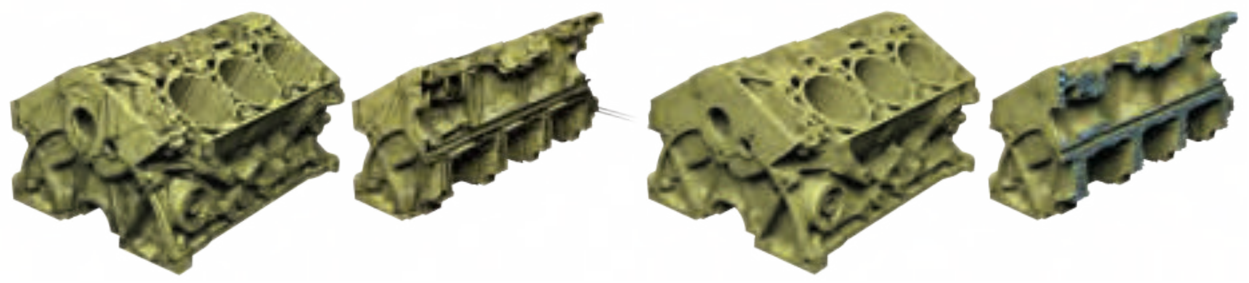


Figure 2.6: A low quality triangle mesh exported from a CAD model with OpenCascade (left) is automatically converted into a high-quality tetrahedral mesh by our algorithm (right), without requiring any manual cleanup.

efficiently. We propose an asymmetric optimization scheme: coarsening and optimization operators are applied only if they improve the mesh quality, while the refinement operator is applied until a predefined edge length (user-controlled) is reached, or whenever a region is locked due to the lack of enough degrees of freedom. The rationale behind this strategy is that we want to avoid over-refinement in regions where it is not necessary to improve quality, and we thus add additional vertices only to match the user-provided density or locally if they are necessary to improve the quality. This strategy allows us to produce high-quality meshes even if the input surface has low quality (Figure 2.6).

We optimize the mesh using 4 passes: (1) splitting (refining), (2) collapsing (coarsening), (3) swapping, and (4) smoothing. We store a target edge length value at the vertices of the tetrahedral mesh, initialized with the user-specified desired edge length ℓ. In (1) each edge whose length is larger than $\frac{4}{3}$ [Botsch and Kobbelt 2004; Dunyach et al. 2013] times the average of the target edge lengths assigned to its endpoints is split once, and the average is assigned to the new vertex. After (1), the target edge length assigned to a vertex v is divided by 2 if there is a low-quality tetrahedron ($\mathcal{E} > 8$, Equation 2.1) within its ℓ_v ball, and multiplied by 1.5 otherwise. To ensure that the user-specified density is always reached, we limit the length by the user-specified parameter ℓ. To prevent unnecessary over-refinement in problematic regions, we cap below the length by ϵ. In (2), we collapse an edge if its length is smaller than $\frac{4\ell}{5}$. In (3), we swap faces if they improve the quality. In (4), we smooth all vertices individually minimizing the average of Equation 2.1 over their one-ring, using Newton's iteration. Only vertices roundable to floats are smoothed, the others are skipped. All these operations are performed only if they do not break any of the invariants described above, and if they increase the mesh quality (with the exception of (1)). In each pass, we use a priority queue to decide the orders of the operations (longest edge first for (1) and (3) and shortest edge first for (2)), except for (4) where the vertices are processed in random order. For (4), we use analytic gradient and Hessian. In Figure 2.7, we show the effects of the mesh improvement step.

The mesh improvement process stops when either the maximum energy is sufficiently small (default: less than 10) or the maximum number of iteration is reached (default: 80 iterations).

2.3.3 Interior volume extraction

Note that until this point, our algorithm has not attempted to define a closed surface bounding a volume: the result of the previous stage is a construction of the approximately constrained tetrahedralization, with a possibly nonmanifold, disconnected and open embedded surface.

We use the method proposed in [Jacobson et al. 2013] to address possible imperfections in the embedded surface, by defining an inside-outside function that can be used to extract an interior volume associated with the mesh.

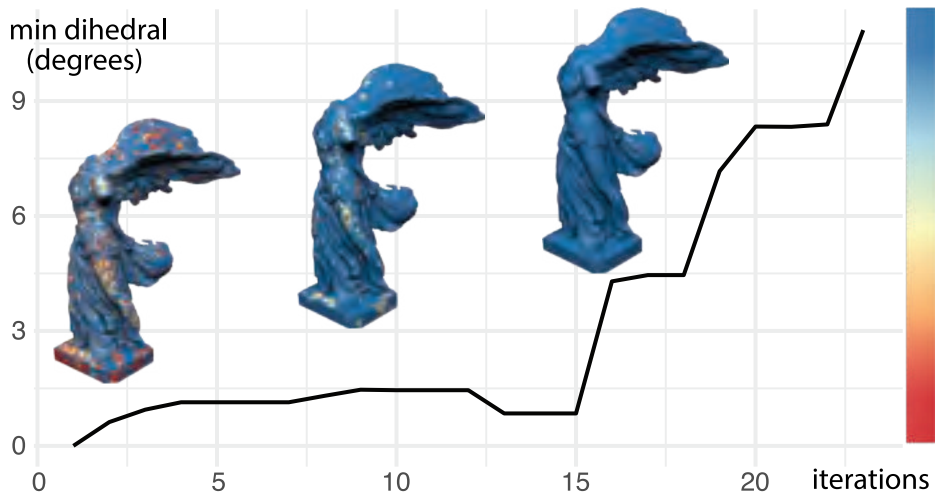


Figure 2.7: A mesh generated with the BSP-tree approach is processed by our iterative mesh optimization algorithm. The quality might decrease during the iterations due to the local refinement ignoring quality, but it quickly improves after additional passes of collapsing, swapping, and smoothing.

We calculate the winding number of the centroid of each tetrahedron with respect to the embedded surface. If the winding number of the centroid of an element is smaller than 0.5, we consider it outside of the surface and drop it before exporting the mesh. Note that this technique must be applied only after mesh optimization due to numerical reasons: the computation of the winding number cannot be performed in rational numbers and it is numerically unstable close to the surface (where we care the most), due to the use of trigonometric functions.

As a result of this step, both small gaps and large surface holes will be filled according to the induced winding number field (Figures 2.8 and 2.11). Consequently, if the input mesh has holes, our algorithms produces a tetrahedral mesh whose surface is not completely inside the ϵ envelope, since the triangles used for hole filling may be outside.

2.3.4 Technical Detail

Hybrid Kernel. Implementing the mesh optimization with only exact rational numbers to store the position of the vertices is not practical for two reasons: (1) the size of the rational representation grows every time a vertex is modified (dramatically increasing the computation time as

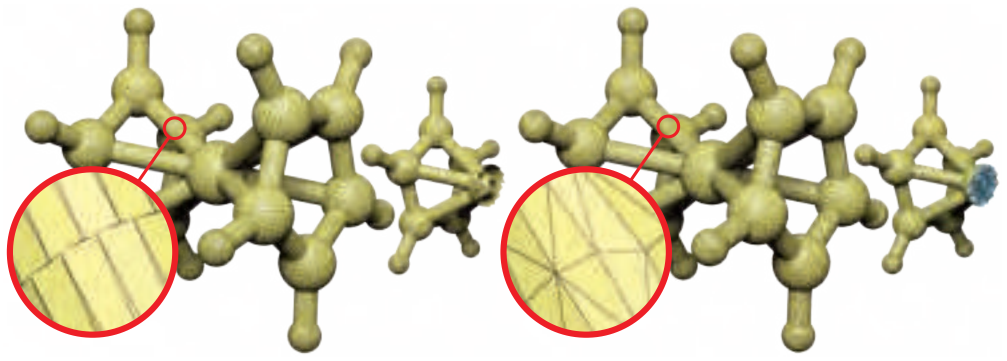


Figure 2.8: Any gap or hole in the input geometry (top) is automatically filled by our algorithm (bottom), generating an analysis-ready tetrahedral mesh.

the algorithm proceeds, especially in the smoothing step), and (2) rational operations are not supported directly in hardware, and are much slower than floating point operations. We implemented our algorithm using an hybrid geometric kernel, similar in spirit and design to [Attene 2017]. For each vertex, we store its coordinates in exact rational numbers only if any of the incident tetrahedra invert after rounding its vertices to floating point representation. This has two major benefits: it avoids the growth of the rational representations, since it trims their length as soon as it is possible to round a vertex, and reduces the memory consumption. Note that this does not affect the correctness of the algorithm since problematic regions containing almost degenerate elements will continue to use an exact rational representation.

Voxel Stuffing. While guaranteed to produce a valid mesh for any input, the algorithm described in Section 2.3.1 can (and will) generate poorly-shaped initial cells whose size is different from what the user prescribed, requiring extensive cleanup in the mesh improvement step. To reduce running times, we found it beneficial to preemptively add some proxy points in a regular lattice inside the bounding box of the input triangle soup. To avoid creating degenerate cells, we remove proxy points that are within δ ($\delta > \epsilon$, default: $\delta = \frac{b}{40}$) from the surface. These points are passed to the Delaunay tetrahedralization algorithm (Figure 2.9), producing a superior starting point that requires fewer local operations to reach a usable quality. In addition to reducing the

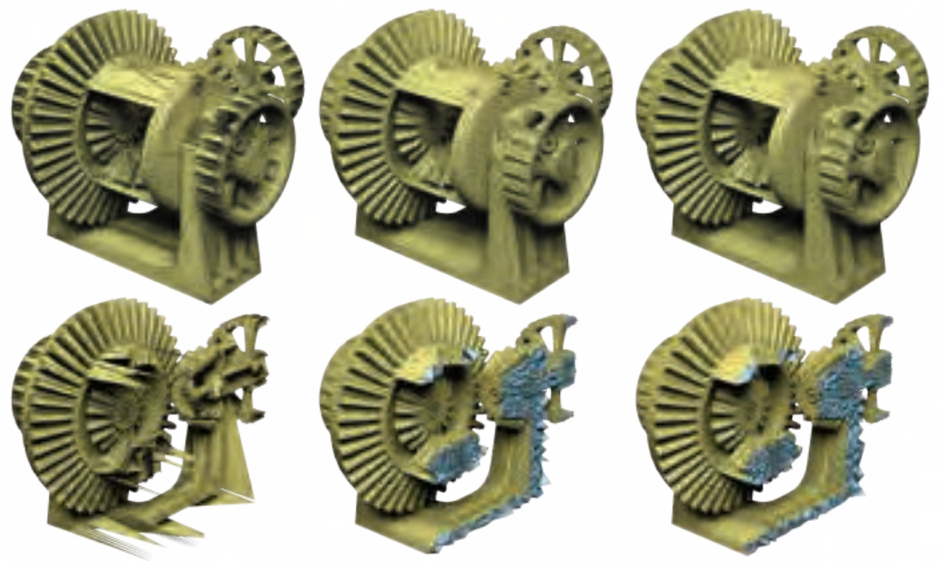


Figure 2.9: Voxel stuffing produces a tetrahedral mesh (middle) of quality comparable to a direct BSP-tree construction (right), but reduces the running times from 3292.3 seconds to 2476.6 seconds.

timing in the optimization stage, this step also localizes the BSP construction around the input surface. We experimentally found that setting the grid edge length to $\frac{b}{20}$ provides the highest benefit, with b being the length of the diagonal of the bounding box.

Input Simplification. The BSP-tree construction potentially introduces a quadratic number of intersections with respect to the number of faces. This only happens in rare pathological cases and it is not an issue for the majority of real-world models, but we did find two problematic ones over ten thousand in Thingi10k [Zhou and Jacobson 2016b] (one of which is shown in Figure 2.10). In these two models, this issue is sufficiently severe to make the BSP tree mesh larger than 64GB, making our implementation crawl due to memory swapping. We propose a preprocessing step that, while not changing the upper bound complexity of our algorithm, resolves this issue on all meshes we tested it with. The preprocessing tries to: (1) collapse all manifold edges of the input triangle soup, accepting the operations that do not move the surface outside of the envelope and (2) improve the quality of the mesh (in terms of angles) by flipping edges, still keeping the surface in the envelope. This procedure simplifies regions with low curvature, and effectively reduces

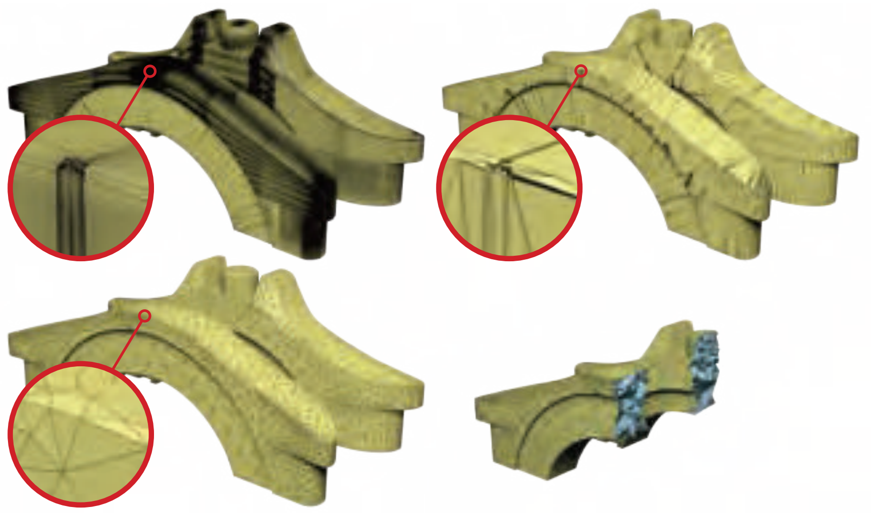


Figure 2.10: A heavily tessellated bridge model from Thingi10k (top, left), is simplified by our algorithm, while keeping the surface in the envelope (top, right), and then converted into a tetrahedral mesh (bottom).

the number of vertices introduced by the BSP tree. We were not able to construct a synthetic case that breaks this procedure when a realistic ϵ is provided. We used this procedure for all our results, since it improves performance also on non-pathological meshes.

Open Boundaries. If the surface contains an open boundary, using only the surface envelope is not always sufficient to ensure a good approximation of the input triangle soup: while unlikely to happen, the boundary is free to move anywhere inside it, potentially moving away from the open boundary, while staying inside the envelope. We address this problem, tracking the open boundaries and reprojecting its vertices back to it in the smoothing step (Figure 2.11). We consider an edge an open boundary if only one triangle is incident to it.

Envelope Test. Our algorithm heavily relies on testing whether a triangle is contained inside the mesh envelope or not to ensure that the embedded surface stays within the envelope during optimization (Section 2.3.2). An exact solution would be prohibitively expensive for our purpose [Bartoň et al. 2010; Tang et al. 2009], and we thus use a conservative floating point approximation. Since the approximation error is bounded, our method guarantees that none of the output surface

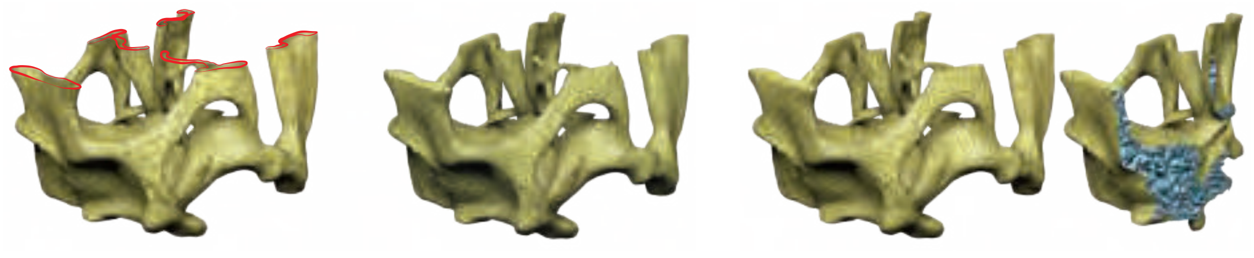


Figure 2.11: For an input model with open boundaries (left, red lines), we add a reprojection in the smoothing step to preserve them (middle). To improve the surface quality, we apply Laplacian smoothing to the output faces used to fill the open regions (right).

points is outside the envelope.

We implicitly construct the envelope by measuring point-to-mesh distance to the unprocessed input mesh, accelerated by an AABB tree [Samet 2005; Lévy 2019]. To check if an embedded surface triangle face is inside of the envelope, we sample this face using a regular triangular lattice with d as the length of the lattice edge. We also add additional samples on the edges of the face, ensuring a maximal sampling error of $d/\sqrt{3}$ (Figure 2.12, left). The triangle is considered inside if all the samples are closer than $\hat{\epsilon} = \epsilon - d_{err}$ ($d_{err} = d/\sqrt{3}$), which is a *conservative* envelope. Since the maximal sampling error is bounded by d_{err}, this ensures a correct result, up to floating point rounding. This construction allows us to control the computational cost: a small d means denser sampling and more computational cost but leads to a wider envelope, allowing our algorithm more flexibility in relocating the vertices. Our experiments showed that $d = \epsilon$ ($\hat{\epsilon} = (1 - 1/\sqrt{3})\epsilon$) is a good compromise.

However, the discrete nature of the sampling introduces a subtle problem: our envelope check is conservative, but not consistent, i.e. reallocating samples on a face of embedded surface by editing its vertices could make it erroneously classified as outside, since some samples might land outside the conservative envelope $\hat{\epsilon}$ (but not outside the user-specified envelope ϵ) (Figure 2.12, right). This could prevent the optimization algorithm for improving the quality of some regions, since operations might be rejected due to the excessively conservative envelope check. This is a rare occurrence, we observed it on only 3 models over 10k (0.03%).

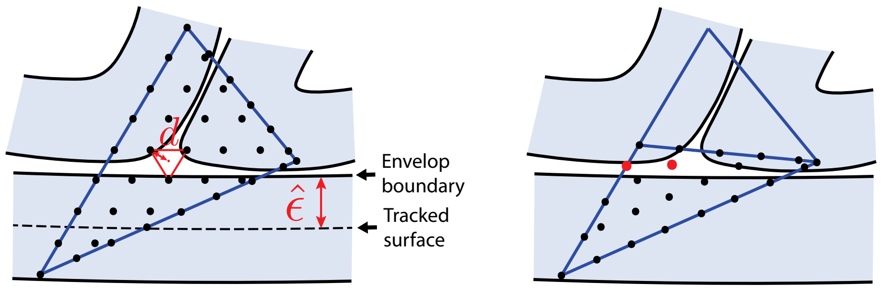


Figure 2.12: A triangle face sampled using a triangular lattice has all samples inside the conservative $\hat{\epsilon}$-envelope can have points outside the envelope by at most $d/\sqrt{3}$ (left). Splitting the triangle into two changes the sampling pattern (right), and some samples on one of its sub-faces are now outside the conservative envelope (marked in red).

We propose a robust, yet expensive, solution for these problematic cases: observing that if there are locked elements, enlarging $\hat{\epsilon}$ by d_{err} guarantees that all elements will be free to move again, we increase the sampling density to make enough space for enlarging the envelope, so that locked regions are freed without violating the user-specified envelope ϵ. Let k an integer representing the current stage (the initial stage is denoted by $k = 1$). In stage k: we (1) set the sampling distance to $d_k = d/k$, (2) run the algorithm, and then (3) enlarge the envelope for $k - 1$ times by d_{err}/k each time during the geometric optimization (see Figure 2.13). If a model is still invalid (i.e. the output contains no unroundable vertex) after the geometric optimization in stage k, we then enter into stage $k + 1$, rerun the algorithm with a denser sampling, and repeat this procedure until it succeeds.

Across the Thingi10K dataset, 9997 models produced *valid* outputs after stage 1, and the remaining 3 models succeed after stage 2. Since enlarging envelope gives more freedom for moving vertices and cleaning surface, this method can also help to improve quality to some degree: we got 99.98% output tetrahedral meshes have minimal dihedral angle larger than 1 degree with $k = 2$, while this percentage is only 99.52% with $k = 1$.

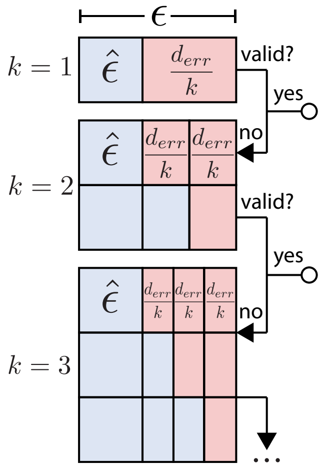


Figure 2.13: Different stages of envelope.

Table 2.1: Comparison of code robustness and performance on the Thingi10k dataset

Software	Success rate	Out of memory (>32GB)	Time exceeded (>3h)	Algorithm limitation	Average time(s)
CGAL (explicit, w features)	57.2%	5.4%	15.7%	21.7%	160.2
CGAL (explicit, wo features)	79.0%	0.0%	0.0%	21.0%	11.7
CGAL (implicit, wo features)	55.7%	0.0%	32.6%	11.7%	997.3
TetGen	49.5%	0.1%	1.7%	48.7%	32.3
DelPSC	37.1%	0.0%	31.1%	31.7%	174.8
Quartet	87.2%	0.0%	0.0%	12.8%	15.3
MMG3D	56.2%	1.2%	10.8%	31.8%	2182.3
Ours	99.9*%	0.0%	0.1%	0.0%	360.0

Note: The maximum resource allowed for each model are 3 hours and 32GB of memory. *Our method exceeds the 3h time on 11 models. If 27 hours of maximal running time are allowed, our algorithm achieves 100% success rate.

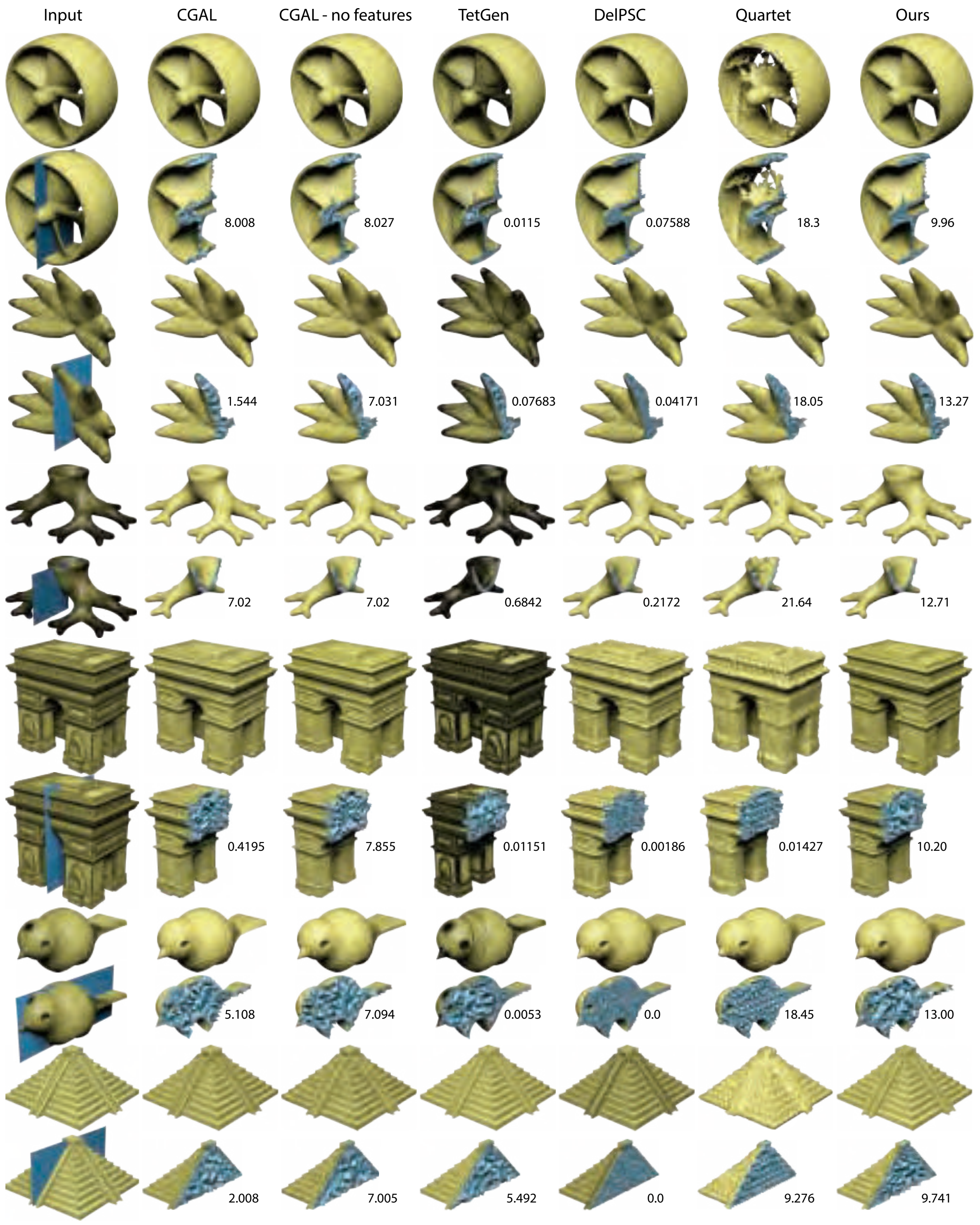


Figure 2.14: Comparison with state-of-art tetrahedralization algorithms. The number close to each model is the minimal dihedral angle.

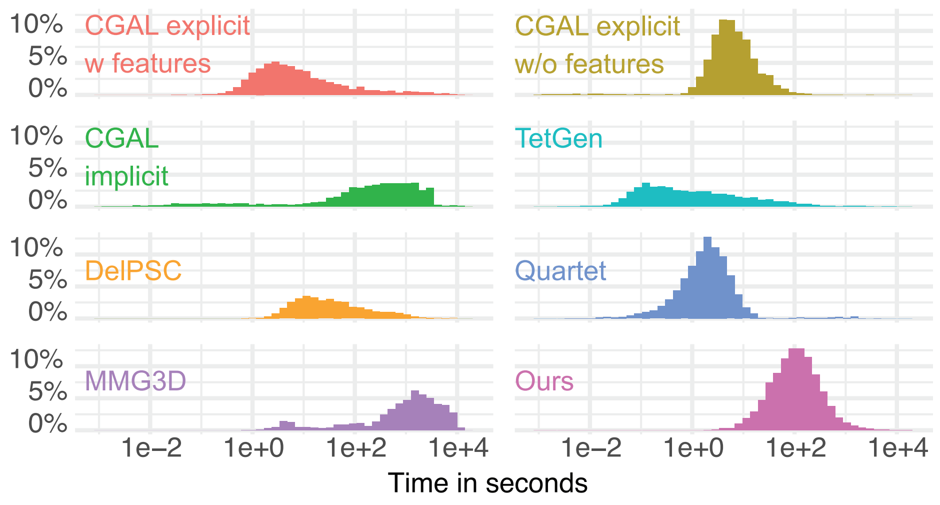


Figure 2.15: Comparison of running time.

2.4 Results

We implemented our algorithm in C++, using Eigen for linear algebra routines, CGAL and GMP for rational computations. The source code of our reference implementation is available at `https://github.com/Yixin-Hu/TetWild`.

Robustness and Performance. We tested our algorithm and a selection of competing methods over the entire Thingi10k dataset [Zhou and Jacobson 2016b]: we show a few examples in Figure 2.14, report aggregate statistics in Table 3.3, running times in Figure 2.15, and output mesh quality in Figure 2.16. We also report detailed statistics for all models shown in the paper (with the exception of Figure 2.1) in Table 2.2. We selected their parameters to make the comparison as fair as possible.

CGAL. We compared our method with [Jamin et al. 2015] in 3 scenarios: (1) CGAL with polyhedral oracle with feature protection, (2) CGAL with polyhedral oracle without feature protection, and (3) CGAL with implicit surface oracle. (1) and (2) are run using the standard implementation inside CGAL, enabling and disabling feature protection (Section 2.2), respectively.

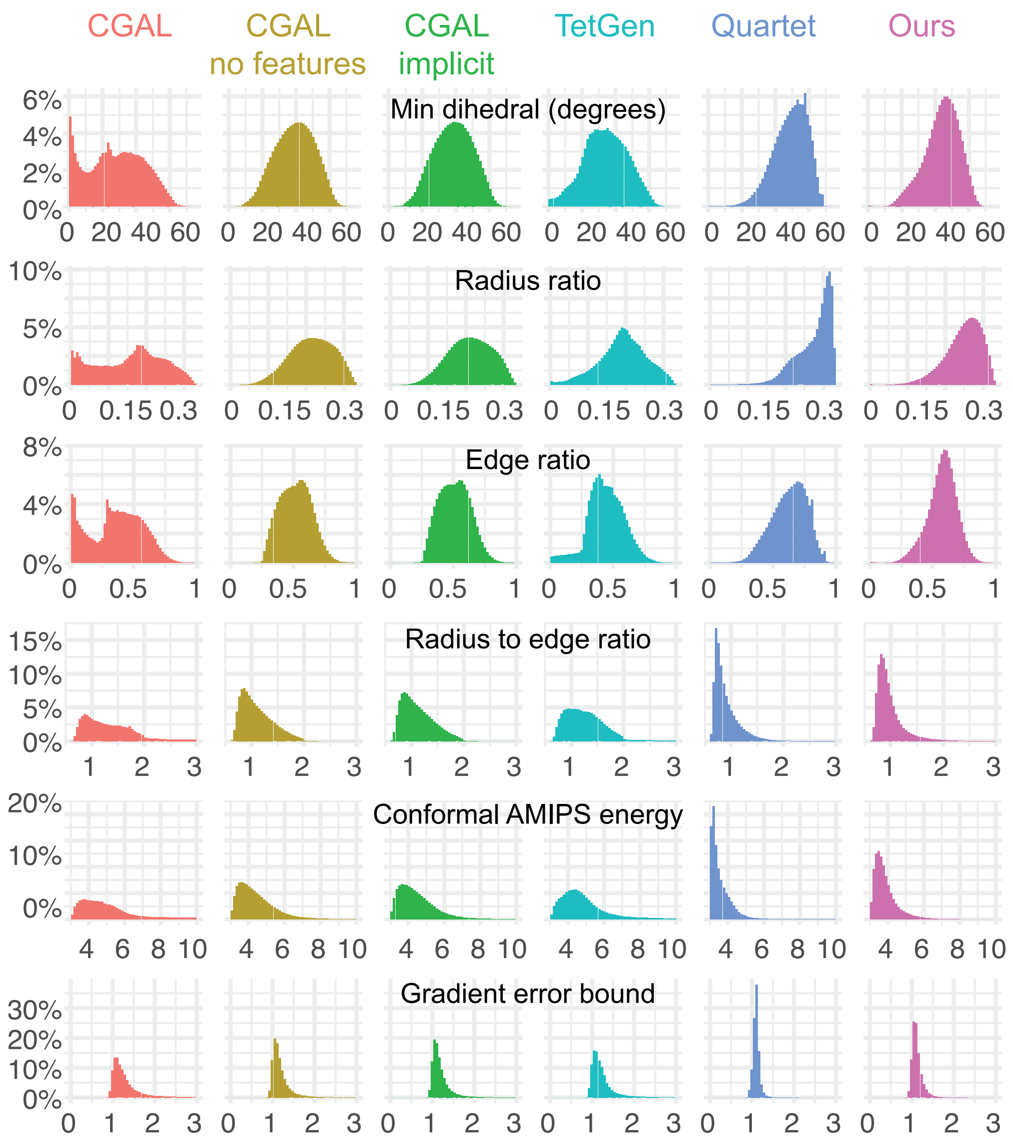
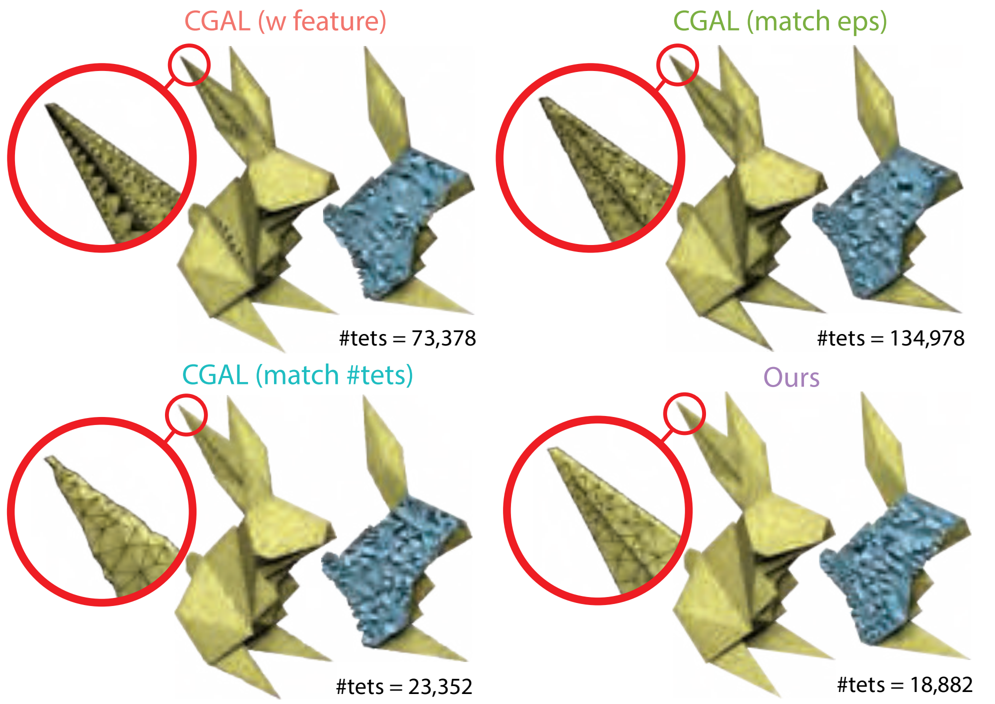


Figure 2.16: Comparison of generated mesh quality on Thingi10k dataset. For each software, we show the distribution of 6 common quality measures of all tetrahedra in 1000 randomly sampled meshes generated from Thingi10k dataset. Quality measures: dihedral angle, inscribed/circumscribed sphere radius ratio, conformal AMIPS energy, and normalized Shewchuk's gradient error estimate factor ([Shewchuk 2002d]).

Table 2.2: Statistics for the datasets in the paper.

Model Id	Fig.	Input #V	Output #V	Angle	AMIPS	Time(m)
255648	2.3	91550	61506	4.8/41.3	16.1/4.1	48.8
134705	2.4	66045	2208	5.6/41.4	11.5/4.1	3.0
134705	2.4	66045	11341	11.7/46.4	7.8/3.7	10.3
134705	2.4	66045	470742	10.3/47.3	11.4/3.7	168.5
114029	2.6	123565	118347	10.3/45.4	9.2/3.7	47.2
376252	2.7	980051	31734	11.1/45.8	8.0/3.7	10.9
62526	2.8	7818	25773	8.9/43.7	9.6/3.9	17.7
38416	2.9	120172	87648	10.2/46.3	8.0/3.7	44.6
996816	2.10	76111	12663	0.02/45.0	1625.4/4.0	747.7
48354	2.11	10945	21211	10.5/45.8	8.0/3.7	3.4
486859	2.14	14629	15011	10.0/45.3	9.3/3.7	5.3
42155	2.14	24646	7248	13.3/45.4	7.3/3.7	2.1
78481	2.14	298370	11385	12.7/46.4	7.9/3.7	3.9
551021	2.14	174066	51011	10.2/46.1	9.4/3.7	16.5
488049	2.14	23036	3574	13.0/43.2	7.8/4.0	1.3
47076	2.14	768	5491	9.7/44.7	9.6/3.8	1.0
964933	2.17	148	4991	10.0/44.5	8.3/3.8	1.2
1036403	2.18	87046	46220	10.5/45.1	8.1/3.8	20.3
1036403	2.18	87046	202846	12.4/50.1	7.7/3.5	162.7
252683	2.19	906835	34721	10.0/44.5	8.2/3.8	14.1
252683	2.19	906835	119087	10.1/46.4	8.0/3.7	113.4
78211	2.20	320	2042	11.3/34.2	9.9/4.6	0.5
78211	2.20	320	8661	9.3/43.5	10.1/3.9	14.2
63465	2.21	592	6238	14.1/44.9	8.2/3.8	0.9
76538	2.22	14169	10098	12.0/44.9	7.9/3.8	3.9
1065032	2.23	48506	27362	8.5/45.4	9.4/3.8	9.2
1036658	2.24	4244	3713	12.3/43.7	7.9/3.8	1.4
Bunny	2.25	11247	38326	7.7/43.8	9.3/3.9	7.2
Bunny	2.25	11247	87359	9.9/43.0	8.1/4.0	20.8
1505037	2.26	19218	37782	10.2/44.2	8.0/3.9	16.8

Note: From left to right: Thingi10k model ID, figure where it appears, number of input vertices, number of output vertices, dihedral angle (min/avg), AMIPS energy (Equation 2.1) (max/avg), running time in minutes.



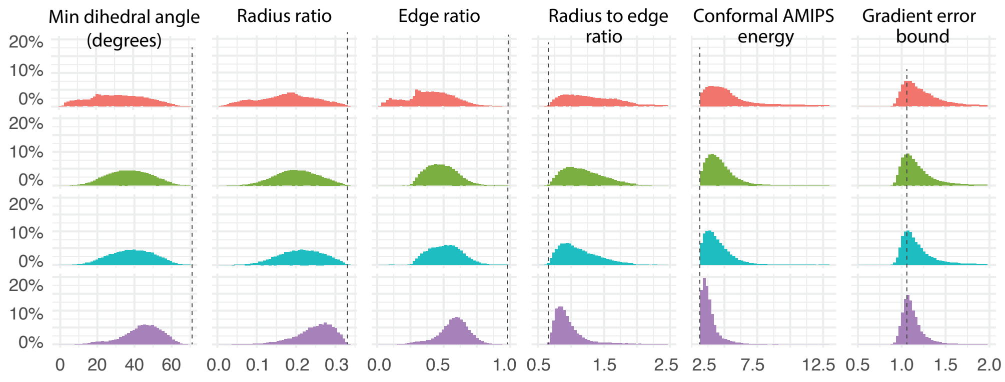


Figure 2.17: (Top): With the same meshing parameters ($\epsilon = b/2000$ and $\ell = b/20$), CGAL's algorithm with and without feature protection (top row) used more than 4 and 7 times the number of tets than ours (second row right) respectively. When using roughly the same number of tets, CGAL's result (second row left) struggles to preserve sharp features. (Bottom): Histograms of various tet quality measures for all tets generated from CGAL and our algorithm. The dotted lines indicate the ideal quality values computed on a regular tetrahedron. Note that our results (bottom row) have better quality in all measures.

For (3), we passed an implicit function based on the winding number calculation, used in our filtering. We provide a signed distance field as oracle (computed with the AABB tree in [Jacobson et al. 2016]), and use the winding number [Jacobson et al. 2013] to decide its sign. In all cases, we have observed lower robustness compared to our algorithm. The quality is slightly better for our algorithm. CGAL with the polyhedral oracle is on average 3 to 4 times faster than our algorithm, while CGAL with implicit oracle is much slower: nearly a third of the inputs timed out after 3 hours (Table 3.3). We show a more detailed comparison of the quality (measured using 6 different criteria) in Figure 2.17. **Tetgen** [Si 2015a] is an order of magnitude faster than our method, but cannot process around half of Thingi10k. It produces meshes with a quality consistently lower than ours, despite introducing more elements. **DelPSC** [Dey and Levine 2008] suffers from robustness problems, successfully processing only around 38% of Thingi10k. The quality is consistently lower than ours. **Quartet** [Bridson and Doran 2014a] is the most robust competing method, with a success rate of 88%. It unfortunately struggles to preserve thin features, and often uses a much higher element count than our method.

Parameters. Our algorithm requires two parameters: the target edge length ℓ, which controls the density of the output mesh, and the maximal Hausdorff distance bound ϵ, which controls the geometric faithfulness of the result. For all our experiments (except where noted otherwise) we used $\ell = b/20$ and $\epsilon = b/1000$, where b is the length of the diagonal of the bounding box of the input. The parameter ℓ controls the mesh density directly (Figure 2.18), while ϵ does it indirectly. Prescribing a small ϵ forces the algorithm to refine more to enforce the tighter bound. Providing a larger ϵ allows our algorithm to get close to the user-prescribed lenghts (Figure 2.19).

Spatially Varying Sizing Field. By replacing the uniform target edge length ℓ with a spatially varying function $\ell(p)$, our algorithm can be extended to create graded meshes. Figure 2.20 illustrates a result with target edge length smoothly varying from coarse to fine in a single model. Note that the output mesh quality remains high despite the large change in the sizing field.

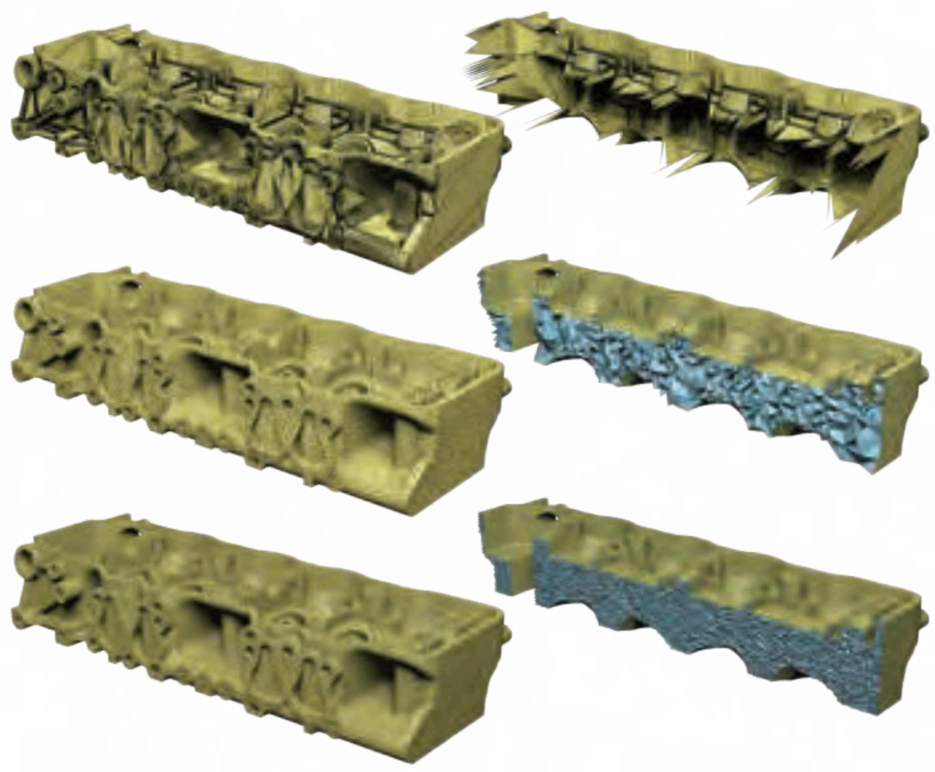


Figure 2.18: ℓ controls the density of the output mesh. Input (top), $\ell = b/20$ (middle) and $\ell = b/150$ (bottom).

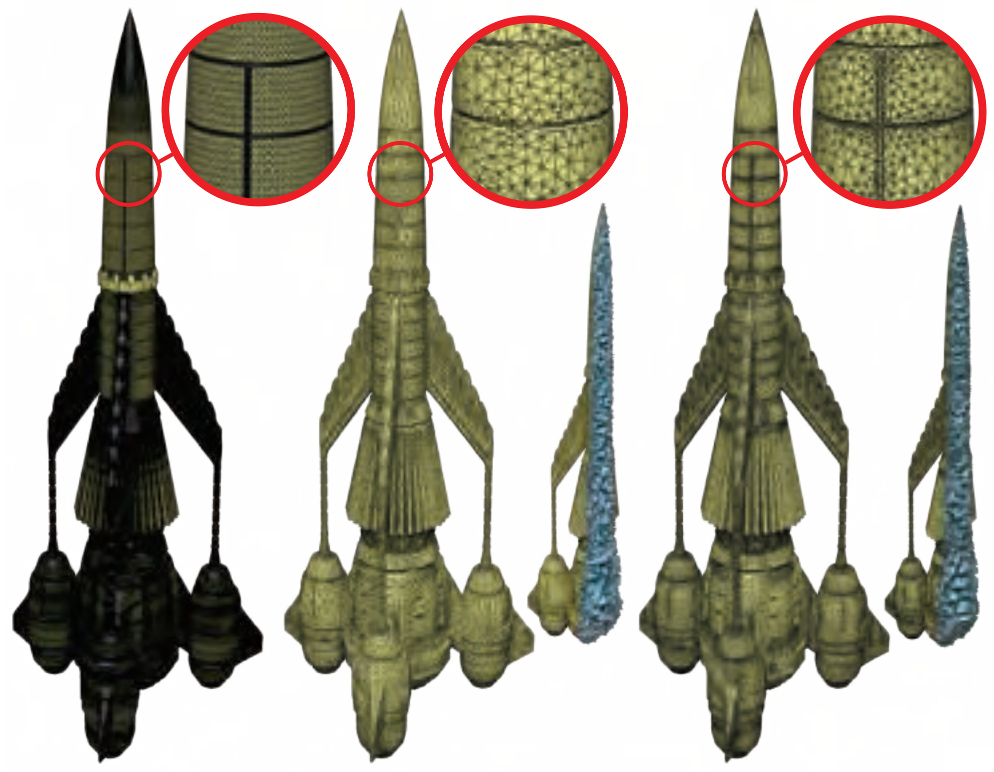


Figure 2.19: ϵ bounds the maximal distance between the input and output mesh. Input (left), $\epsilon = b/1000$ (middle) and $\epsilon = b/3000$ (right).

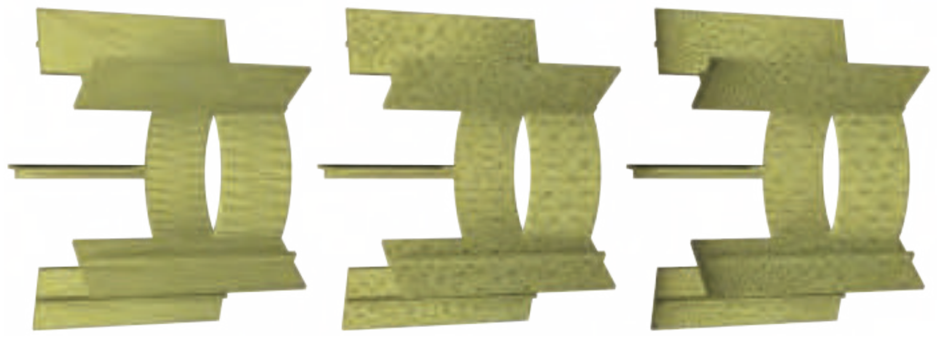


Figure 2.20: Example for spatially varying sizing field using background mesh. Input (left), output tetrahedral mesh without sizing control (middle), and output tetrahedral mesh with sizing field applied (right).

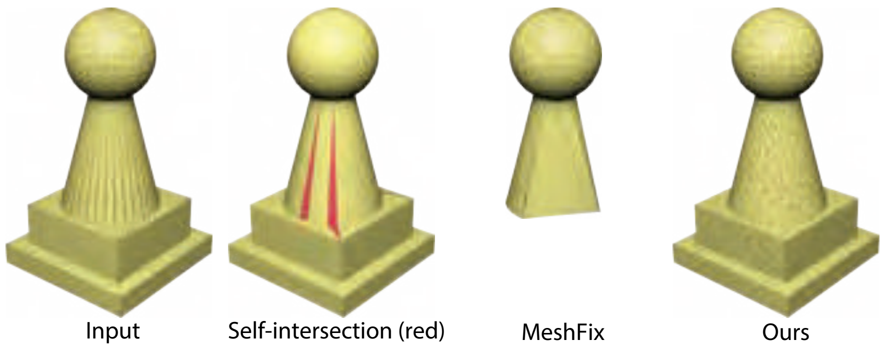


Figure 2.21: A self-intersecting triangle soup, is cleaned using meshfix by removing the base. Our algorithm instead creates a tetrahedral mesh of its interior, whose boundary corresponds to a clean triangle mesh of the pawn.

Surface Repair. Our algorithm can be used as an effective mesh repair tool for closed surfaces by creating a tetrahedral mesh of their interior, and then extracting its boundary. Self-intersections are robustly resolved when constructing the BSP-tree, degeneracies are removed by the mesh improvement step, surface gaps/holes are filled based on generalized winding number, and the output surface is trivially the boundary of a valid volume. While computationally more expensive than alternative methods that only work on the surface, our technique can robustly handle extremely challenging cases. In Figure 2.21, we compare our method to MeshFix [Attene 2010a] on a self-intersecting chess pawn.

Finite Element Method Validation. We demonstrate that our algorithm can be used as a black box to solve PDEs on the entire Thingi10k dataset. We normalize all our output meshes to fit in the unit cube and create an analytic volumetric harmonic function by summing 12 radial kernels ($1/r$), placed randomly on a sphere centered at the origin of radius $1.5b$. This function is sampled on the boundary of the mesh and used as a boundary condition for a Poisson problem, solved using [Jacobson et al. 2016]. We successfully solve this PDE over all models, and we report a sample solution and the histograms of L^2 and L^∞ errors with respect to the analytic solution evaluated on the internal nodes in Figure 2.22.

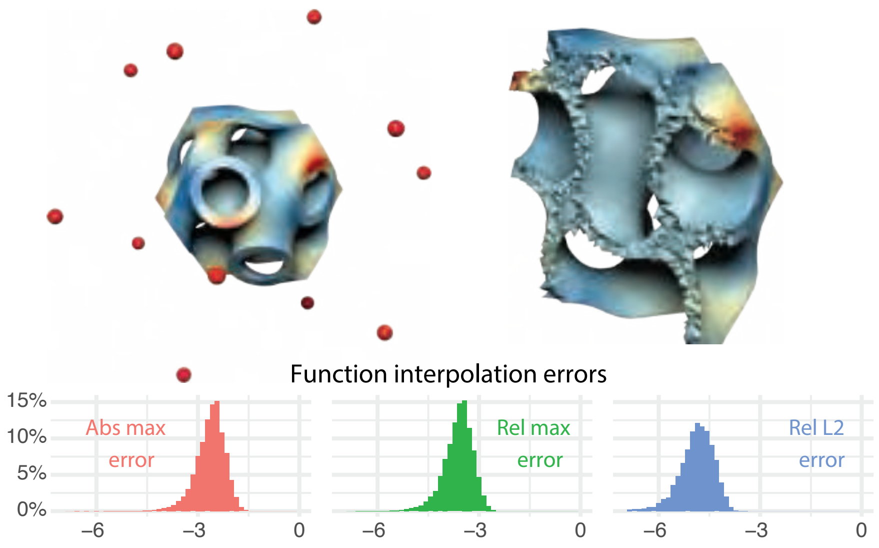


Figure 2.22: We test our generated tet meshes by solving a harmonic PDE using finite element method with linear elements. For each model in Thingi10K, we compare the computed solution with the ground truth (radial basis functions with kernel $1/r$ centered at the red spheres). We show the absolute max error, relative max error, and relative L_2 error histograms (log scale) in the bottom row.

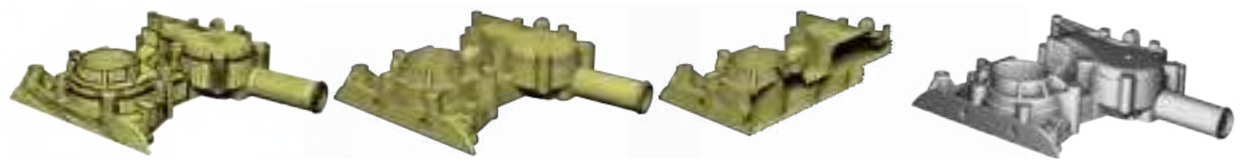


Figure 2.23: Our algorithm can be used to bootstrap quadrilateral remeshing.

Structured Meshing. Structured meshing algorithms [Bommes et al. 2012] usually rely on an existing clean boundary representation of the geometry (triangle meshes in 2D and tetrahedral meshes in 3D) to generate a structured mesh. Our algorithm can be used to convert triangle soups into meshes suitable for remeshing. We show the examples of quadrilateral meshing using [Jakob et al. 2015] in Figure 2.23 and hexahedral-dominant meshing [Gao et al. 2017] in Figure 2.24.

Noise Stress-Test. We stress test our method under geometrical noise (Figure 2.25), by randomly displacing its vertices using Gaussian noise. Even in this extreme case our algorithm produces meshes close to the noisy input and have a large minimal dihedral angle.

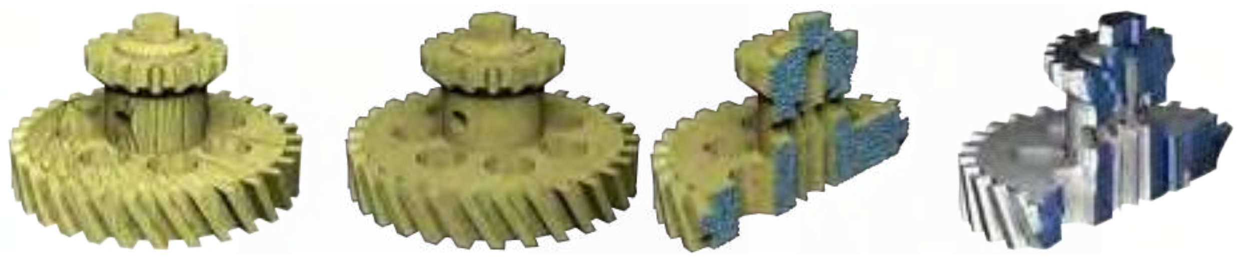


Figure 2.24: Our algorithm can be used to bootstrap hex-dominant remeshing.

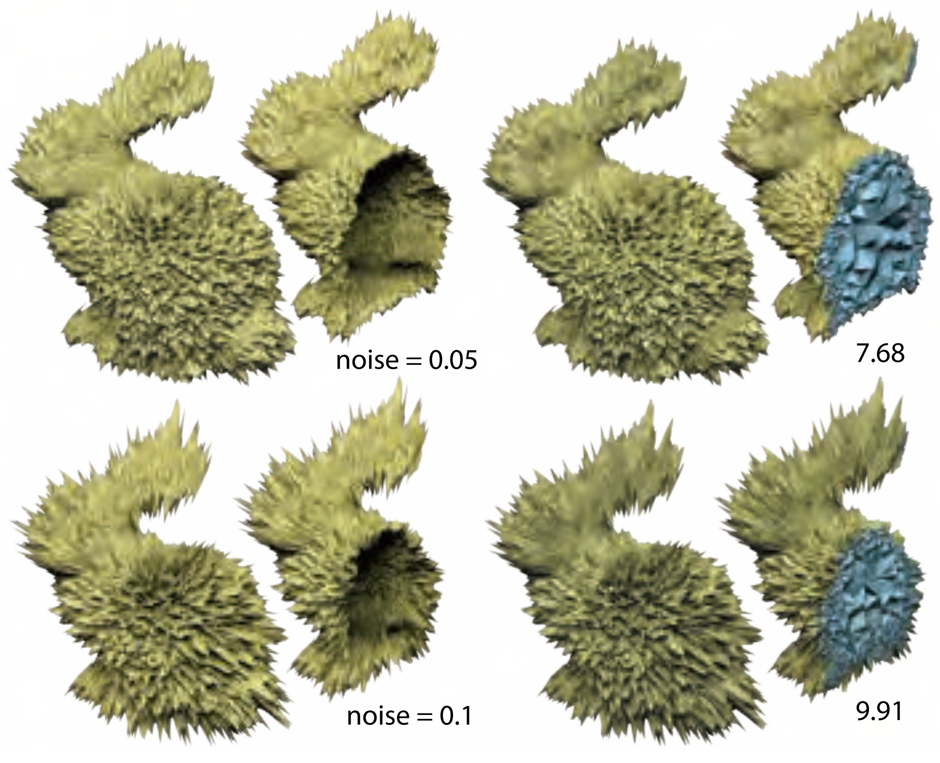


Figure 2.25: Our algorithm is robust to geometrical noise. The numbers denote the minimal dihedral angle of output meshes.

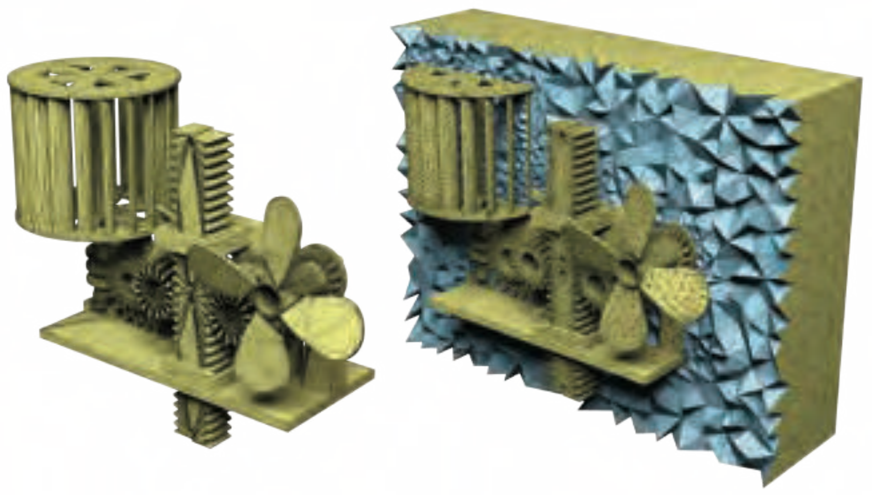


Figure 2.26: The volume around a complex mechanical piece is automatically meshed by our algorithm, preserving the surface of the embedded object.

Meshing for Multimaterial Solids Our algorithm naturally supports the generation of tetrahedral meshes starting from multiple enclosed surfaces by simply skipping the filtering step (Section 2.3.3), as shown in Figure 2.26.

2.5 Limitations and Concluding Remarks

Our algorithm handles sharp features in a soft way: they are present in the output, but their vertices could be displaced, causing a straight line to zigzag within the envelope. While this is acceptable for most graphics applications, extending our algorithm to support exact preservation of sharp features is an interesting research direction that we plan to pursue. We demonstrated that our algorithm can be used as a mesh repair tool, but it is, however, limited to closed surfaces: extending it to support mesh repair over shells is an interesting and challenging problem. Our single threaded implementation is slower than most competing methods: since most steps of our algorithm are local, we believe that a performance boost could be achieved by developing a parallel (and possibly distributed) version of our approach.

To conclude, we presented an algorithm to compute *approximately constrained tetrahedral-*

izations from triangle soups. Our algorithm can robustly process thousands of models without parameter tuning or manual interaction, opening the door to black-box processing of geometric data.

3 | Fast Tetrahedral Meshing in the Wild

This chapter describes our work fTetWild that extends TetWild in Chapter 2 for improving the meshing efficiency.

3.1 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.

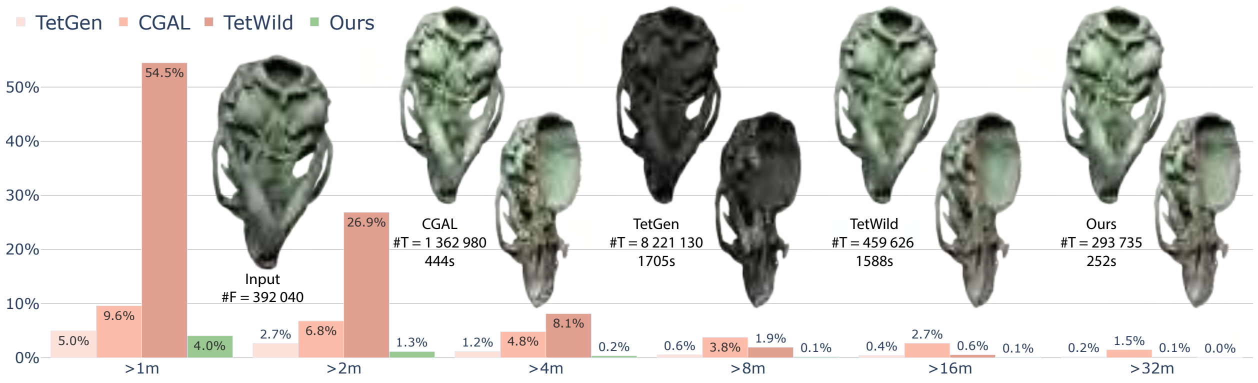


Figure 3.1: The bar charts show the percentage of models requiring more than the indicated time for the different approaches over 4 540 inputs (the subset of Thingi10k where all 4 compared algorithms succeed). Our algorithm successfully meshes 98.7% of the input models in less than 2 minutes, and processes 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.

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 3.1), 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.

3.2 Related Work

We have reviewed the literature on tetrahedral meshing in Chapter 2 Section 2.2 and we refer to [Cheng et al. 2012b; Shewchuk 2012] for a more detailed overview of the topic. Here we give an emphasis on envelope-based techniques, We also review mesh repair and mesh arrangement algorithms, since our technique can be also used in these settings to enable processing of imperfect geometry.

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 2010b, 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 [Thibault and Naylor 1987; Naylor et al. 1990] 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 2016a].

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 [Zhou et al. 2016; Granados et al. 2003]. 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 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.

3.3 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

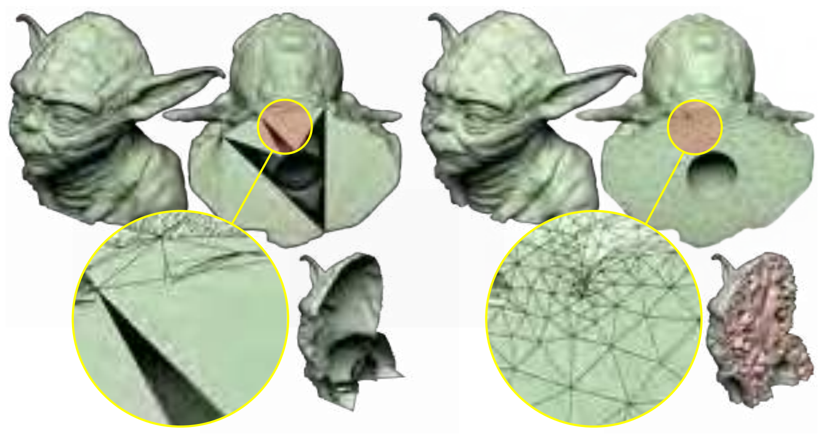


Figure 3.2: Example of an input surface mesh with self-intersections and a bad triangulation on the base. fTetWild converts this model into a high-quality tetrahedral mesh.

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 3.2).

The output tetrahedral mesh can be optionally post-processed to remove the tetrahedra outside the input surface (Section 3.3.6). 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 Chapter 2 for a 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 run-

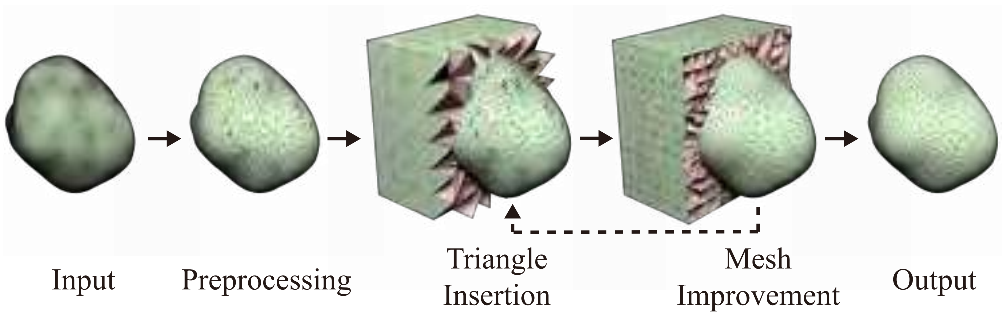


Figure 3.3: Overview of our algorithm. From left to right, the input mesh is simplified, a background mesh is created and the input faces are inserted, the mesh quality is optimized, and the final result is obtained by filtering the elements lying outside the input surface.

ning 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.

3.3.1 Algorithm Overview

Our algorithm consists of four phases (Figure 3.3): (1) the input triangle soup is simplified while ensuring it stays in the ϵ-envelope of the input (Section 3.3.3), (2) a background mesh is generated and the triangles are iteratively inserted into it, if the insertion does not introduce inverted elements (Section 3.3.4), (3) the mesh is improved using local operations (Section 3.3.5) 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 3.3.6).

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 [Shewchuk 1997; Lévy 2019]) 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_{zero}^2, ϵ_{zero}^3 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 $\epsilon_{\text{zero}} = 10^{-8}$.

3.3.2 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].

3.3.3 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 3.3.4), and prevents vertices to be too close to the boundary of the

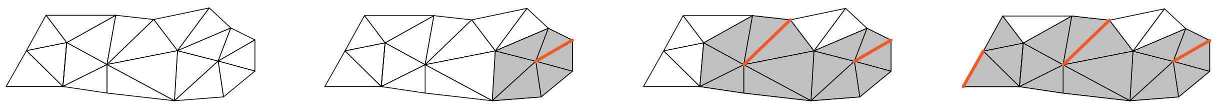


Figure 3.4: A two-dimensional example of edge coloring. From left to right: one parallel-independent edge is selected (in red), its vertex-adjacent triangles are colored in black. The algorithm proceeds until no edges can be marked (last image). In the end, all red edges can be safely collapsed in parallel without affecting other red edges.

envelope, thus leaving more freedom for surface vertices to move in the mesh improvement stage (Section 3.3.5). 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 3.3.4). We use the value 0.8 since it is far from the bounds of this range.

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 3.4). 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.

3.3.4 Incremental Triangle Insertion

3.3.4.1 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

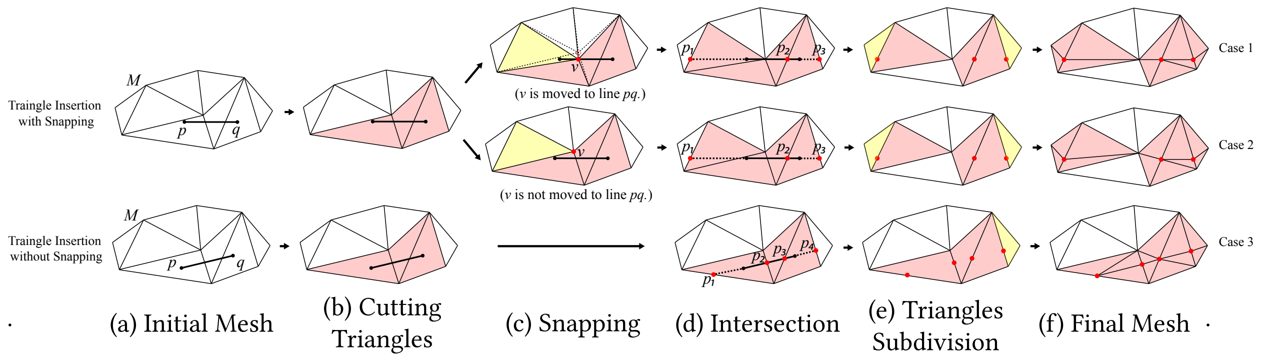


Figure 3.5: Segment insertion into a triangle mesh (a 2D analog of triangle insertion) with and without snapping. (a) Insertion of segment pq into mesh M. (b) Identification of cut triangles $\mathcal{T}_I$ (in red). (c) Snapping vertex v to line pq and updating $\mathcal{T}_I$, where v is δ-close to 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.

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.

3.3.4.2 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)

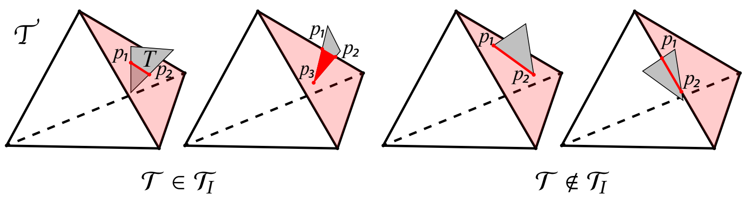

$\mathcal{T}$
p_1
T
p_2
p_1
p_2
p_3
p_1
p_2
p_1
p_2
$\mathcal{T} \in \mathcal{T}_I$
$\mathcal{T} \notin \mathcal{T}_I$

Figure 3.6: 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 $\mathcal{T}_I$.

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*. 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 3.6). 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 [Shewchuk 1997; Lévy 2019] 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 3.5(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

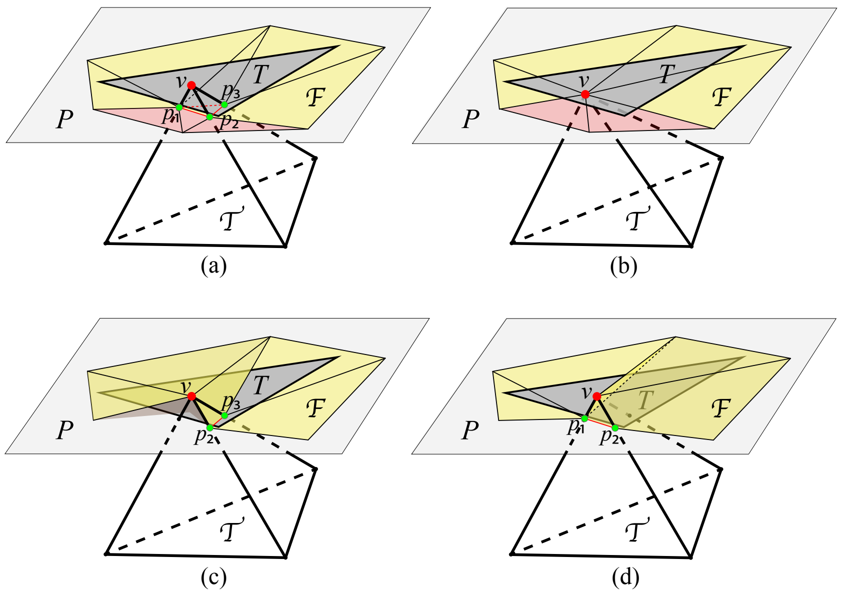


Figure 3.7: Plane P intersects $\mathcal{T}_I$ ($\mathcal{T} \in \mathcal{T}_I$). (a) The faces of $\mathcal{F}$ (marked in yellow) are the covering of triangle T. (b) Snapping v to its closest point on P and expanding $\mathcal{F}$ to include red triangles makes $\mathcal{F}$ safely covering T. (c) 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.

more insertion failures, 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 3.7 (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 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 3.7 (b)) or snap intersection points to vertices of M (Figure 3.7 (c)(d)). Moving a vertex v of $\mathcal{T}_I$ (thus changing M) changes the cover $\mathcal{F}$, because

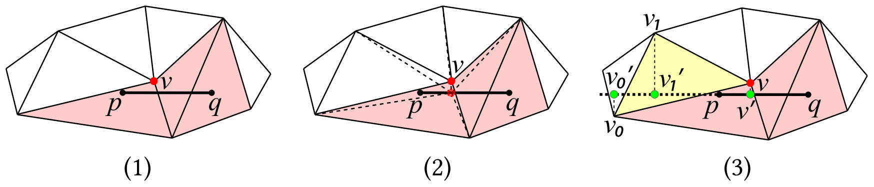


Figure 3.8: 2D illustration of step (1) to (3) of snapping when inserting segment $[p, q]$. The cut triangles in $\mathcal{T}_I$ are marked in red. (1) Vertex v is within δ distance to segment $[p, q]$. (2) Check the effect of moving v to line $[p, q]$. (Vertex v cannot be moved to $[p, q]$ in this case, 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 $[p, q]$.

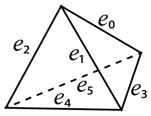
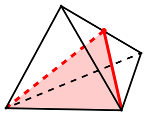
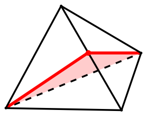
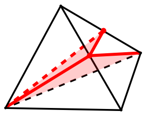
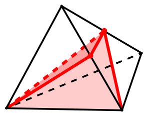
P intersects the edges of $\mathcal{T}_I$ in different locations. As no intersection of P with an edge of $\mathcal{T}_I$ disappears (at most, it may move to an endpoint), and no new intersections appear (other than at the endpoints shared with already intersected edges), the connectivity of $\mathcal{T}_I$ 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 3.7 (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 3.7 (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 (Section 3.3.7), 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 3.8(1)).

Table 3.1: 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	⋯
$0_{(10)} = 000000_{(2)}$			⋯
$1_{(10)} = 000001_{(2)}$			⋯
$2_{(10)} = 000010_{(2)}$			⋯
$3_{(10)} = 000011_{(2)}$			⋯
⋮	⋮	⋮	⋱

2. Move vertices in $\mathcal{V}_\delta$ to their closest points on P if it does not invert any elements of M (Figure 3.8(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 3.8(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 3.5(e)) and some neighboring tetrahedra of $\mathcal{T}_I$ (yellow elements in Figure 3.5(e)).

Since an edge can have at most one intersection point with P, the decomposition of the subdi-

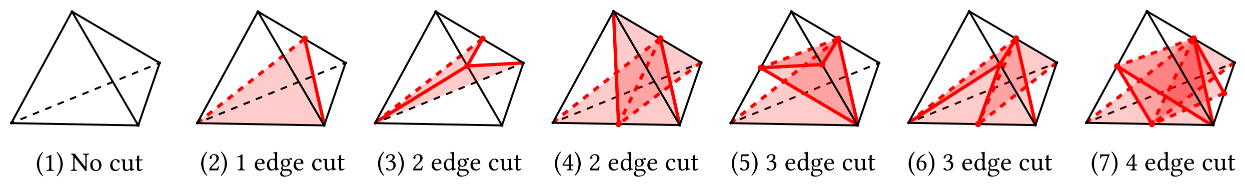


Figure 3.9: 7 symmetry classes of edge-cut configurations. (4) and (6) can only happen on neighboring tetrahedra of $\mathcal{T}_I$ with only certain edges cut by P.

vided 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 3.1), 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 3.1). 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 3.9). 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 3.9, (4) and (6)) for subdividing the neighboring tetrahedra of $\mathcal{T}_I$ with only certain edges cut by P (yellow elements in Figure 3.5(e)).

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.

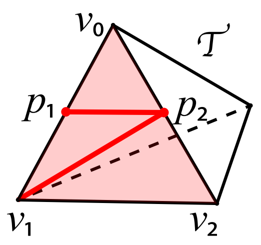


Figure 3.10: Triangulation on the shared face.

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 Figure 3.10), 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 (Section 3.3.7). 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_{zero}^3 (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.

3.3.4.3 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

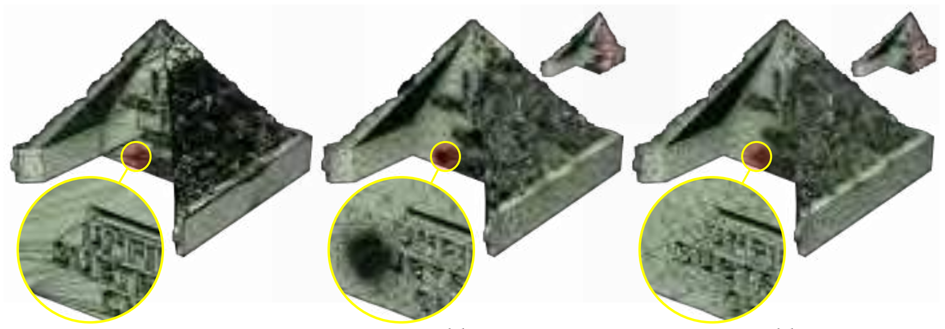


Figure 3.11: Example of model where the numerical instability of the AMIPS energy causes over-refinement (middle). By evaluating the energy using rational numbers (when it is above 10^8) the issue disappears (right).

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 Section 3.3.7.

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.

3.3.5 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 3.18). (3) is a subtle problem, which we now explain in more detail. The conformal AMIPS 3D energy is a Jacobian-based energy defined as:

$$\text{AMIPS} = \frac{\operatorname{tr}(\mathbf{J}^T\mathbf{J})}{\det(\mathbf{J})^{2/3}}, \tag{3.1}$$

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. Here is a concrete example: 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

$$\text{AMIPS}_{1234} = 5.027711906288343\text{e}10, \text{AMIPS}_{2341} = 2.171615254548946\text{e}11$$

$$\text{AMIPS}_{3412} = 8.865129658843354\text{e}10, \text{AMIPS}_{4123} = 7.103076229685612\text{e}10,$$

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

$$\text{AMIPS}^3_{1234} = 9.401446861483944\text{e}25, \text{AMIPS}^3_{2341} = 1.834560196543814\text{e}25,$$

$$\text{AMIPS}^3_{3412} = 1.006679363250288\text{e}26, \text{AMIPS}^3_{4123} = 3.462536408842030\text{e}26.$$

As reference the correct value computed with rational number is 1.127562687503913e11.

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 3.11).

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 3.11), avoiding unnecessary refinement and decreasing the overall runtime.

The mesh improvement terminates when either a user-specified mesh quality or a user-

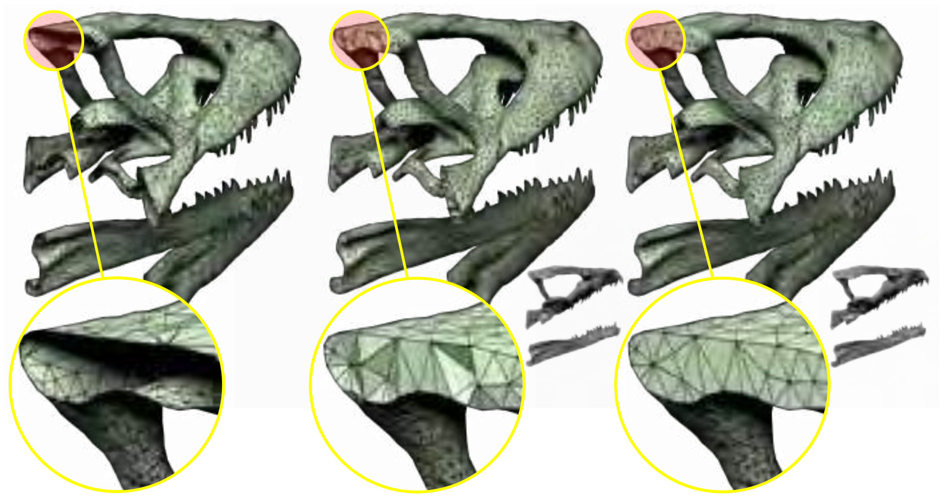


Figure 3.12: Input with open boundary on the bottom (left). The output tetrahedral mesh preserves the input geometry and closes the open side (middle). Users can choose to enable an additional smoothing process for smoothing the open region (right).

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 3.4). Quartet [Bridson and Doran 2014b], 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 2007b], but it does not preserve sharp edges or corners. The Constrained Delaunay refinement method used in TetGen [Si 2015b] 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

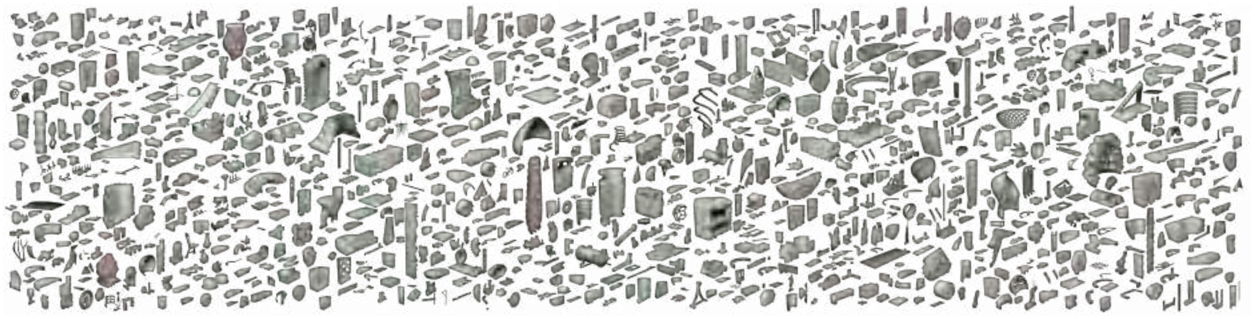


Figure 3.13: 1000 random samples of fTetWild output on Thingi10k dataset.

the vertices on the open boundary and project them back to the open boundary during mesh improvement (Section 3.3.5). Elements are classified as inside or outside the surface using the generalized winding number (Section 3.3.6). An example of input with open boundary is shown in Figure 3.12.

3.3.6 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.

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 3.14.

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.

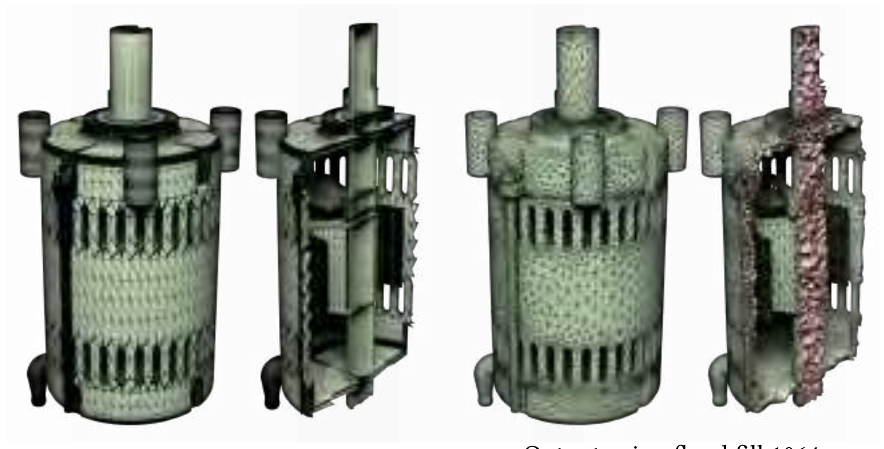


Figure 3.14: An input model (left) where the heuristic winding number filtering fails to extract the volume (it drops most of the tetrahedra in the output) due to inconsistent triangle orientations in the input. By changing the heuristic to the flood-fill algorithm, we can obtain the expected output (right).

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.

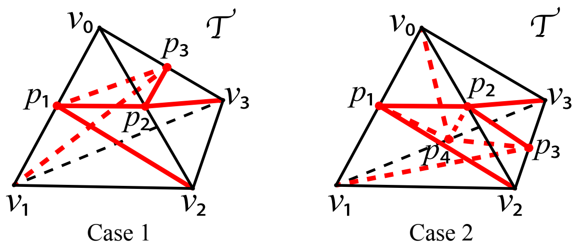


Figure 3.15: Two unused configurations requiring an additional vertex.

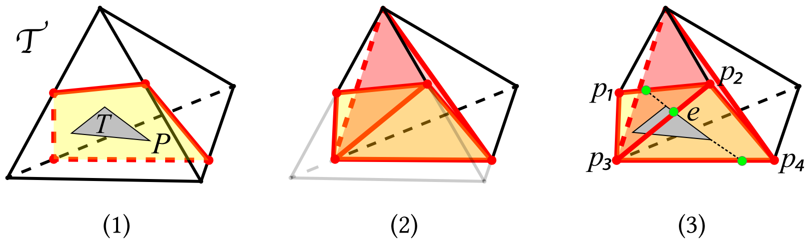


Figure 3.16: Example for preserving an open-boundary edge e of triangle T. (1) Insert T and $\mathcal{T}_I = \{\mathcal{T}\}$ in this case. (2) The sub-tetrahedra of $\mathcal{T}$ after subdivision. (Only sub-tetrahedra behind T are shown for better visualization.) (3) Inserting edge e and get the intersection points (in green).

3.3.7 Technical Detail

Unused Decompositions of a Tetrahedron We enumerated all the possible decompositions of a tetrahedron and discovered two symmetry classes (Figure 3.15) 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 3.3.4.2) never selects these two cases.

We show that our rule does not select case 1 (Figure 3.15 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 3.15 right) is also not selected following a similar argument.

Table 3.2: Edge-cut configurations of a cutting tetrahedron before and after snapping. Numbers corresponds to the configurations in Figure 3.9.

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)

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 3.16(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 3.16(1), $\mathcal{T}$ first get decomposed into sub-tetrahedra (Figure 3.16(2)). Then the faces covering T are $\mathcal{F} = \{[p_1, p_2, p_3], [p_4, p_2, p_3]\}$ Figure 3.16(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 3.16(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.

Changes of Edge-cut Configuration After Snapping Table 3.2 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 3.9 (2)(3)(5)(7)), but excludes the remaining 3 classes where $\mathcal{T}$ is not cut or is just affected by their neighbors (Figure 3.9 (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.

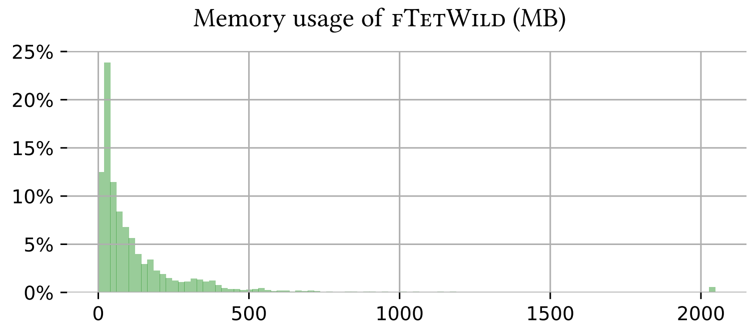


Figure 3.17: Histogram of memory usage of fTetWild over the Thingi10k dataset (data truncated at 2GB).

Table 3.3: 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.

3.4 Results

Our algorithm is implemented in C++ and uses Eigen [Guennebaud et al. 2010] for the linear algebra routines. We perform a large-scale comparison of our method with other meshing methods on the Thingi10k dataset [Zhou and Jacobson 2016a], 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`.

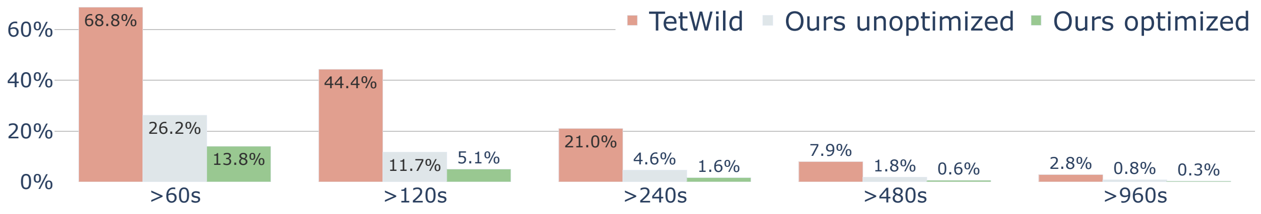


Figure 3.18: Percentage of models requiring more than a certain time for our parallel and serial algorithm compared with TetWild on Thingi10k dataset.

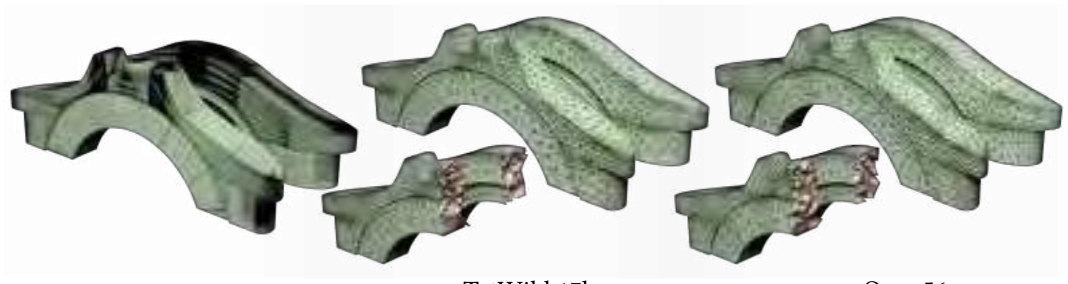


Figure 3.19: Example of a challenging model where fTetWild is 17 times faster than TetWild.

3.4.1 Success Rate

With the above memory and time constraints, fTetWild successfully tetrahedralizes 100% of the 10 000 input meshes (Figure 3.13). Most of the input models can be tetrahedralized with less than 1GB of RAM as detailed in Figure 3.17. Note that very complex models might require more memory, for instance the one in Figure 3.26 uses around 17GB of memory.

As observed in [Hu et al. 2018], most of the state-of-the-art tet-meshers have low success rate on *in-the-wild* data. We summarize the results on the whole Thingi10k dataset in Table 3.3. Note that only our method and TetWild have high success rates: our average timing is however seven times faster than TetWild.

3.4.2 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 3.18). On more complex examples, like the model in Figure 3.19, our method is up to 17 times faster than TetWild.

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 3.20 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 3.1 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

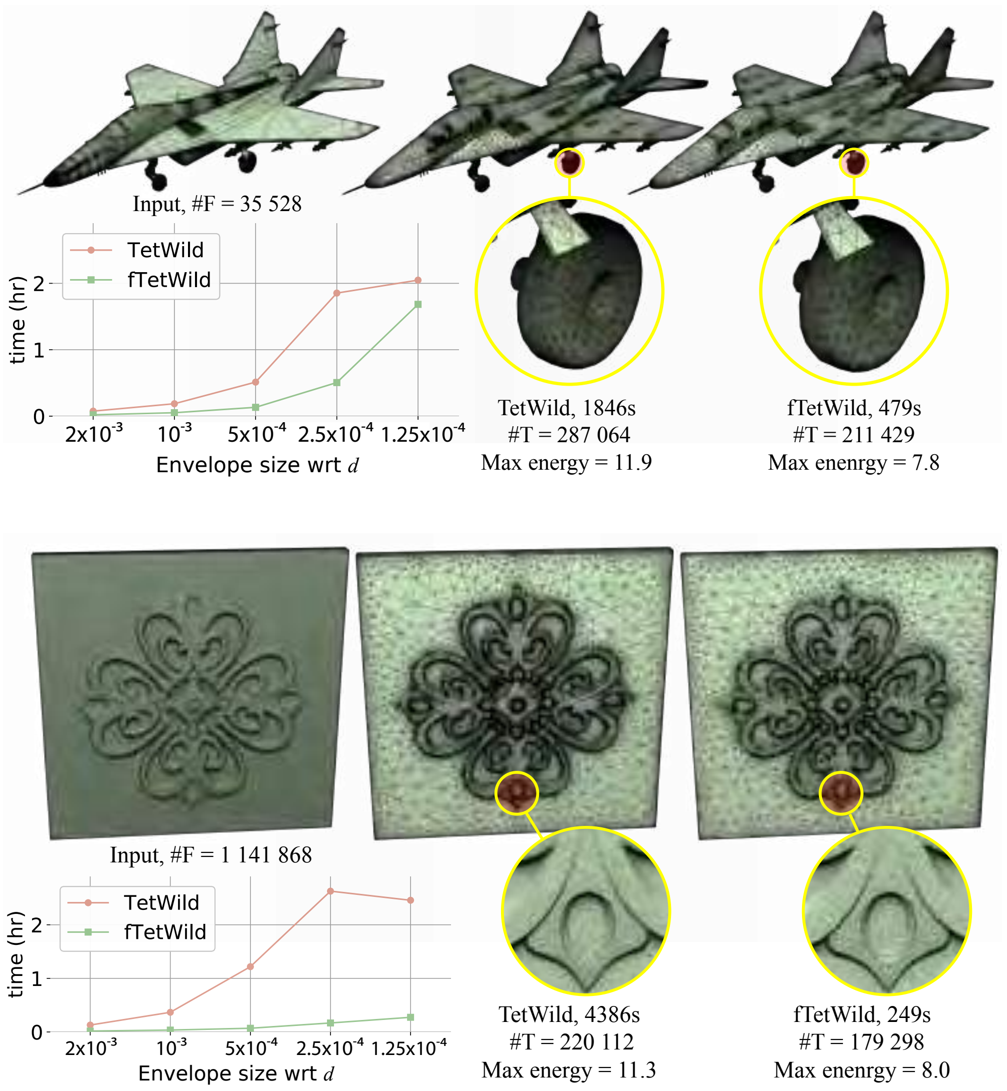


Figure 3.20: Input models and running time plots of fTetWild and Tetwild with ϵ reduced from $2\,10^{-3}d$ to $1.25\,10^{-4}d$ (left). Output tetrahedral meshes of the two methods at $\epsilon = 5\,10^{-4}d$ (middle and right). Note that we flipped all the normals of the input triangles of the airplane model for visualization purposes (see Figure 3.33 for a detailed discussion).

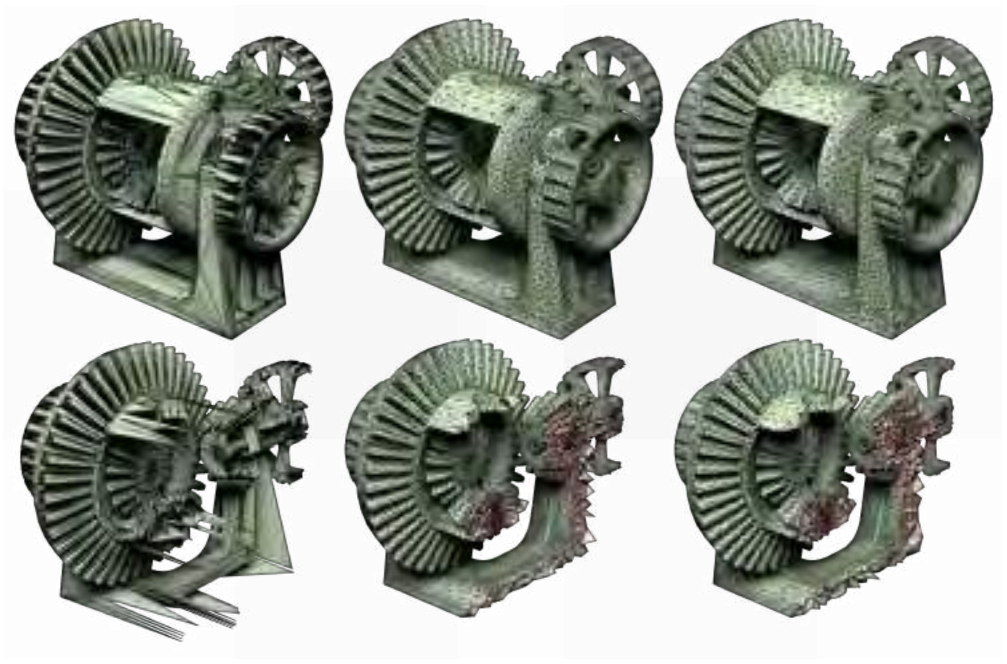


Figure 3.21: Our method (right) produces high-quality tet-meshes that are similar to TetWild (middle).

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.

3.4.3 Mesh Quality

The geometric quality of meshes produced by our algorithm is similar to the meshes produced by TetWild (Figure 3.21), 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 (3.1)), range $[3, +\infty)$, optimal 3,

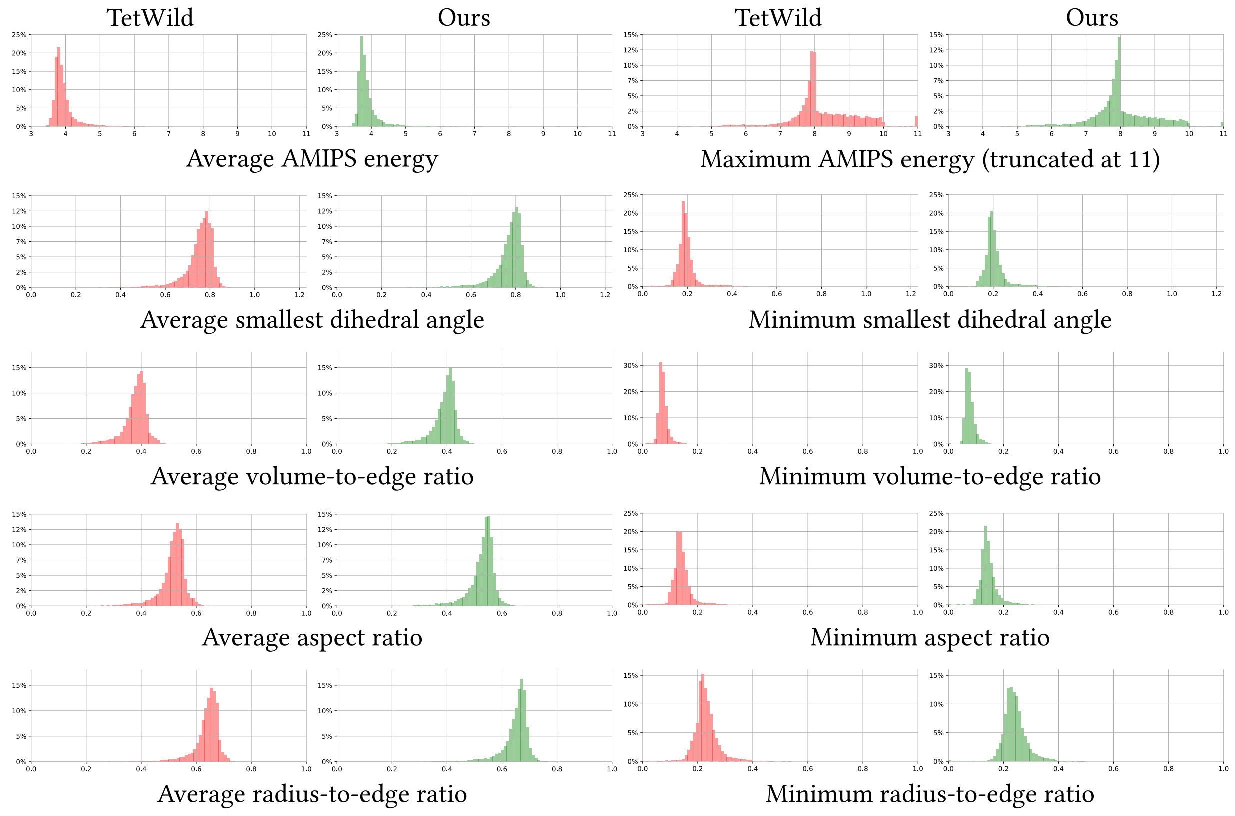


Figure 3.22: Histogram for mesh quality comparison of TetWild (red) and our method (green) in five different quality measures. The statistic is based on the output of the whole Thingi10k dataset.

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,

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 3.22 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

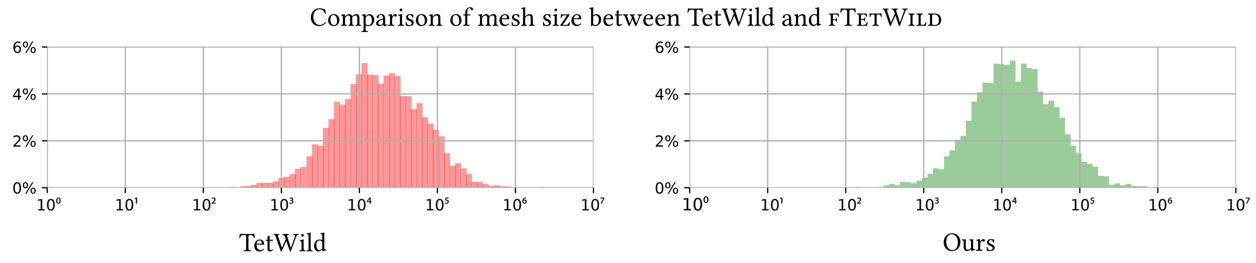


Figure 3.23: Histograms of number of in log scale for the output meshes of Thingi10k dataset.

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.

3.4.4 Mesh Density

Compared with TetWild, our method generates meshes of similar density (Figure 3.23). 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 3.1. CGAL approximates the surface by means of an implicit function, but sometimes over-refines sharp features and tiny artifacts as illustrated in Figure 3.1, where the dark spots are over-refined regions.

3.5 Applications

3.5.1 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 3.3.5) can be stopped at any time since we

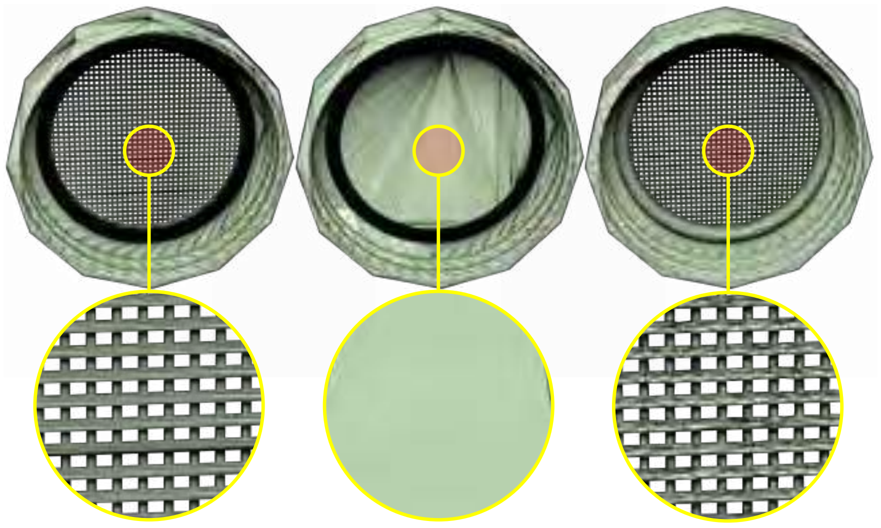


Figure 3.24: Example of repairing an invalid triangular mesh (left) with MeshFix (middle) and our algorithm (right). MeshFix is fast but loses details during processing, while our method preserves them. The max AMIPS energy of our intermediate tetrahedral mesh is 1975. Here we stop mesh improvement when maximum energy reaches 2000.

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 2010b] in Figure 3.24. 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 3.25). 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.

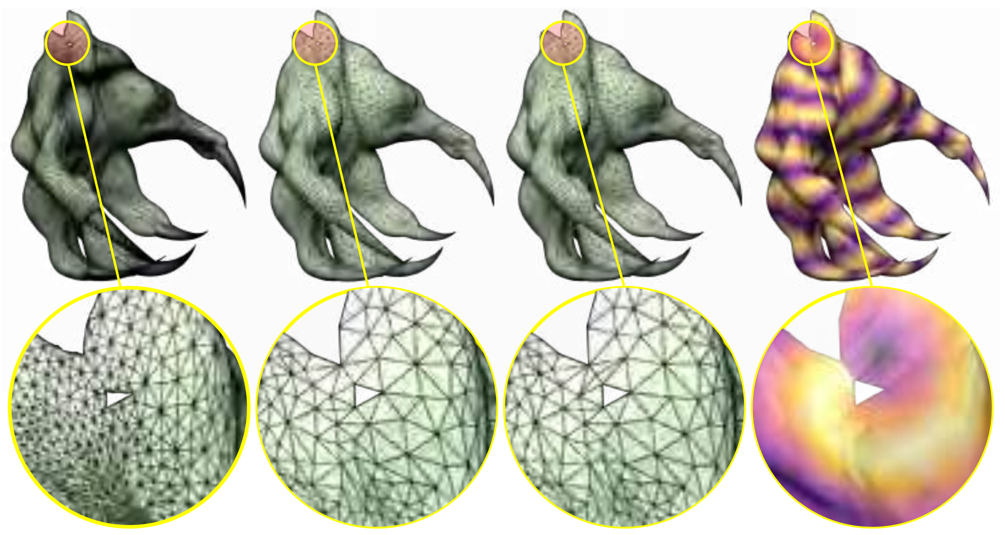


Figure 3.25: Example of a non-manifold surface mesh (left) which is automatically repaired by our algorithm (right second).

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 3.26) is directly usable for FEA, further editing, or fabrication. As a reference point, the original implementation of TetWild takes 215 minutes with a default envelope size. Another challenging model we tested contains complex thin structures coming from architecture (Figure 3.27). The method in [Masoud 2016; Ghomi et al. 2018] 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

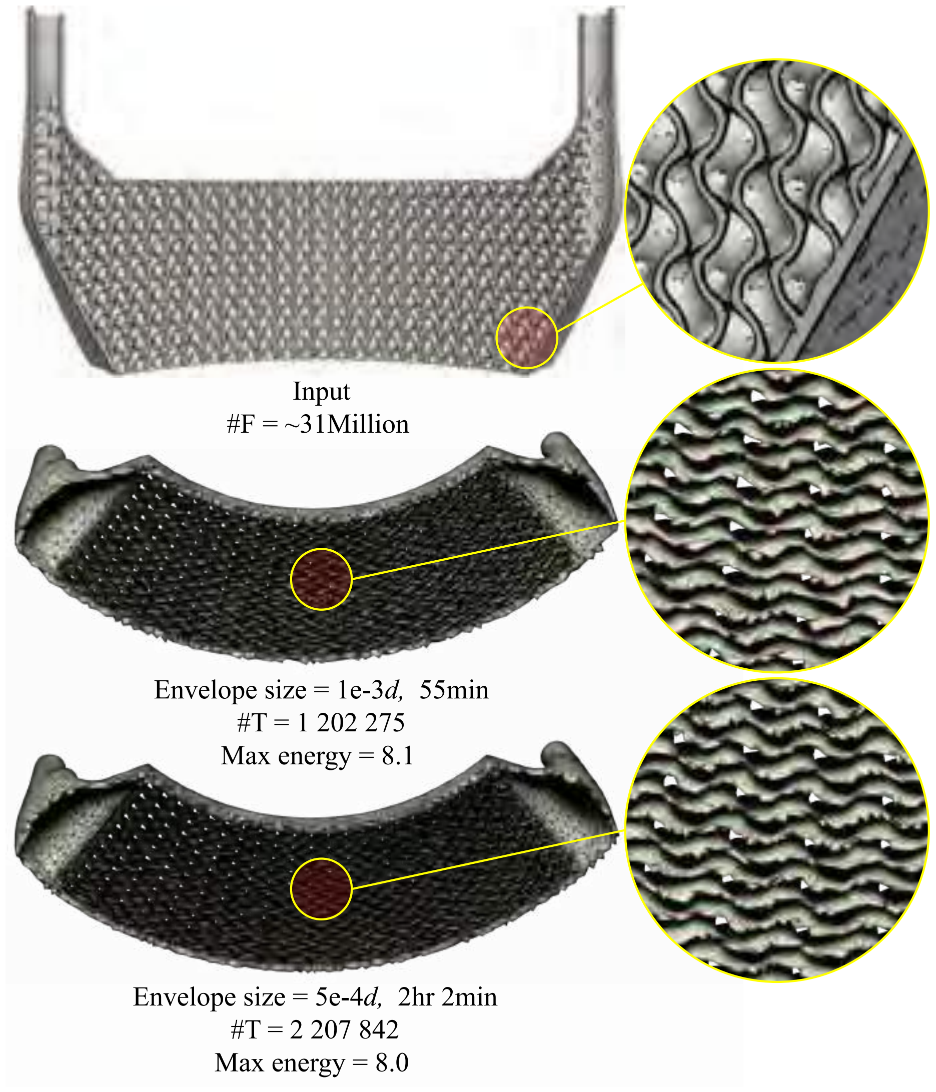


Figure 3.26: Meshing a complex model with 93 million vertices and 31 million faces with different envelope sizes (top). The input mesh contains degenerate triangles and severe self-intersections. Our output tetrahedral meshes are in geometric high quality with either default envelope size (middle) or half envelope size (bottom).

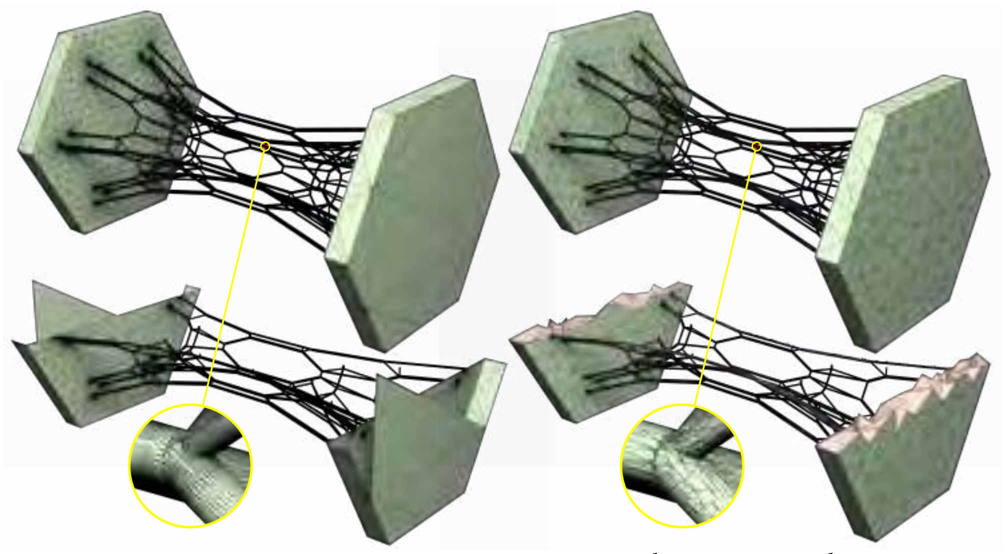


Figure 3.27: Example of an architectural application with 80 999 self-intersecting faces. The cylinders in the input are intersecting with each other as shown in the closeup. fTetWild successfully cleaned and tetrahedralized this input. Here we stop mesh optimization when maximum energy reaches 50.

mesh.

3.5.2 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 3.3.6) 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 (Figures 3.28, 3.29), providing robust (but slower) Boolean operations on imperfect geome-

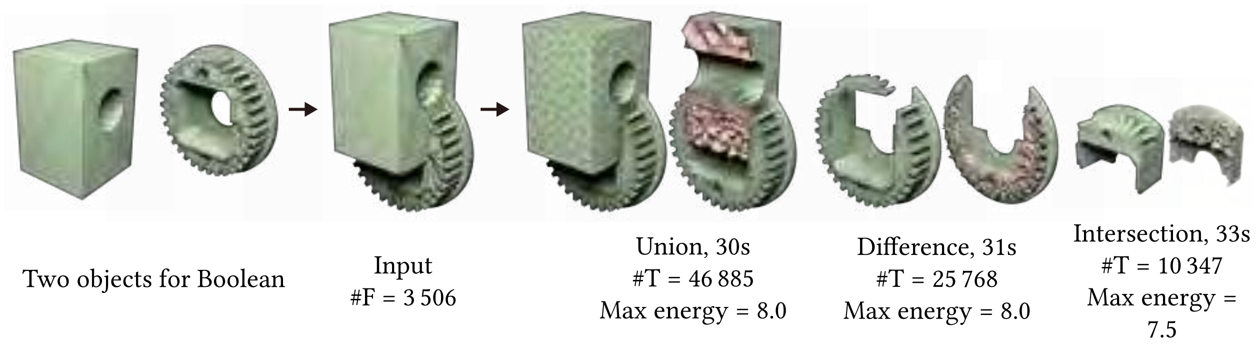


Figure 3.28: Three Boolean operations computed on non-manifold, self-intersecting, and non-PWN input surface meshes. The left are two objects for Boolean operation. The middle is the input surface mesh of fTetWild. The right are our output meshes after computing the union, difference, and intersection between the two objects. The average max AMIPS energy of outputs and average time of different operations are with small variance.

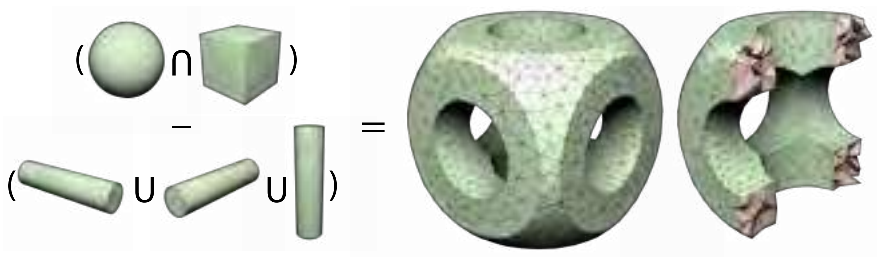


Figure 3.29: Four Boolean operations among 5 objects. fTetWild takes 34s and products output with #T = 8 060 and max energy = 7.2.

tries. The output is a tetrahedral mesh, which can be useful in downstream applications, and its boundary is a high quality surface triangular mesh.

3.5.3 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 3.30).

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

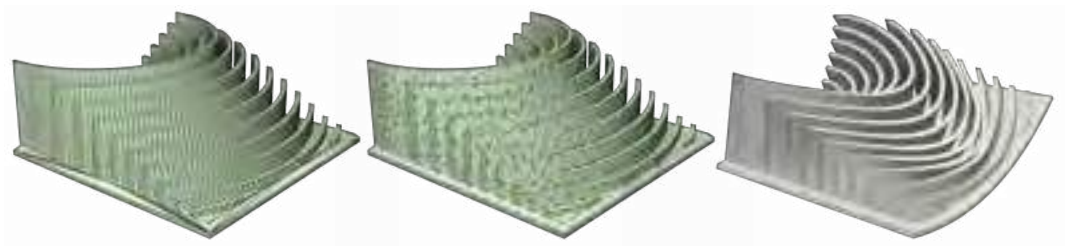


Figure 3.30: Example of non-linear elastic deformation of a body (right).

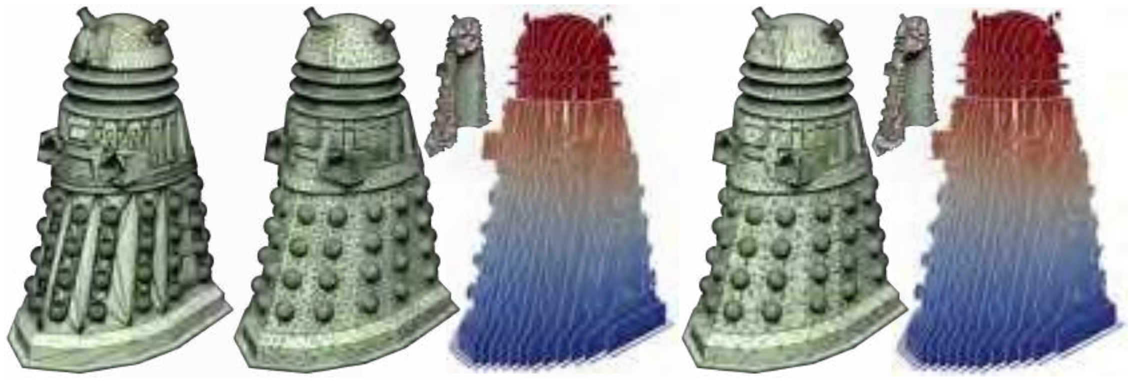


Figure 3.31: Two different stopping criteria of our algorithm. The full optimization (middle) improves the mesh to high quality, while using the criterion in [Schneider et al. 2018] (right) results in lower mesh quality but faster meshing and smaller mesh size. The color shows the solution of the volumetric Laplace equation.

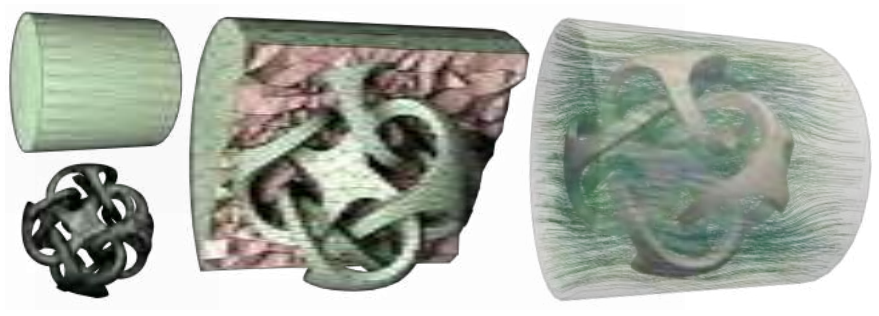


Figure 3.32: Streamlines of a fluid (right) moving in a cylindrical pipe (left top) with a complicated obstacle (left bottom) in the center. The background mesh (middle) is obtained by subtracting the obstacle from a cylinder using our method.

the mesh optimization early in our algorithm (thus reducing the meshing time) without affecting the quality of the simulation, Figure 3.31.

We use the Boolean difference (Section 3.5.2) to generate the background mesh required for simulating the fluid flow on a cylindrical tube containing an obstacle (Figure 3.32).

3.6 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. 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 3.33). 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

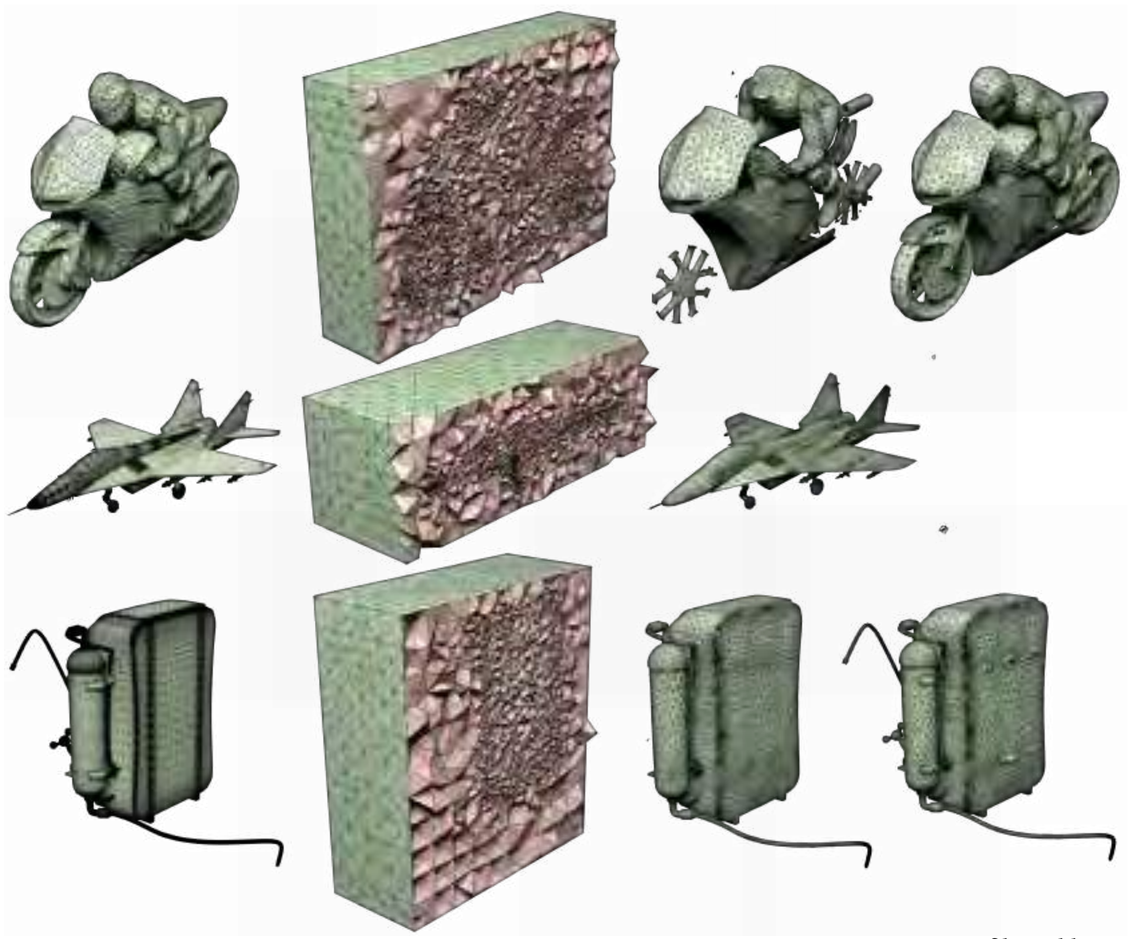

Input
Output
Output filtered by flood fill
Output filtered by winding number

Figure 3.33: 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.

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.

4 | TriWild: Robust Triangulation with Curve Constraints

Based on TetWild, we design a robust algorithm TriWild for generating coarse 2D planar curved triangle meshes that preserves input curved shapes with fewer elements and higher accuracy compared with linear meshes.

4.1 Introduction

Triangle meshing is at the core of a large fraction of two-dimensional computer graphics and computer aided engineering applications, most commonly, used to solve PDEs or optimization problems on 2D domains, in the context of physical simulation, geometric modeling, animation and nonphotorealistic rendering. Major efforts have been invested in robustly generating meshes with linear edges with good geometric quality. However, the restriction to linear meshes makes precise reproduction of simple curved shapes, such as a Bézier curve, impossible independently of the resolution used, resulting in artifacts and/or excessive refinement in applications ranging from physical simulation to nonphotorealistic rendering. Curved meshes, i.e. meshes with curved edges, are an effective solution to this problem: the idea is to use curved triangles instead of linear ones, providing significantly superior geometric approximation of a shape using a mesh of a particular size. In most cases, the lower triangle count leads to an overall more efficient

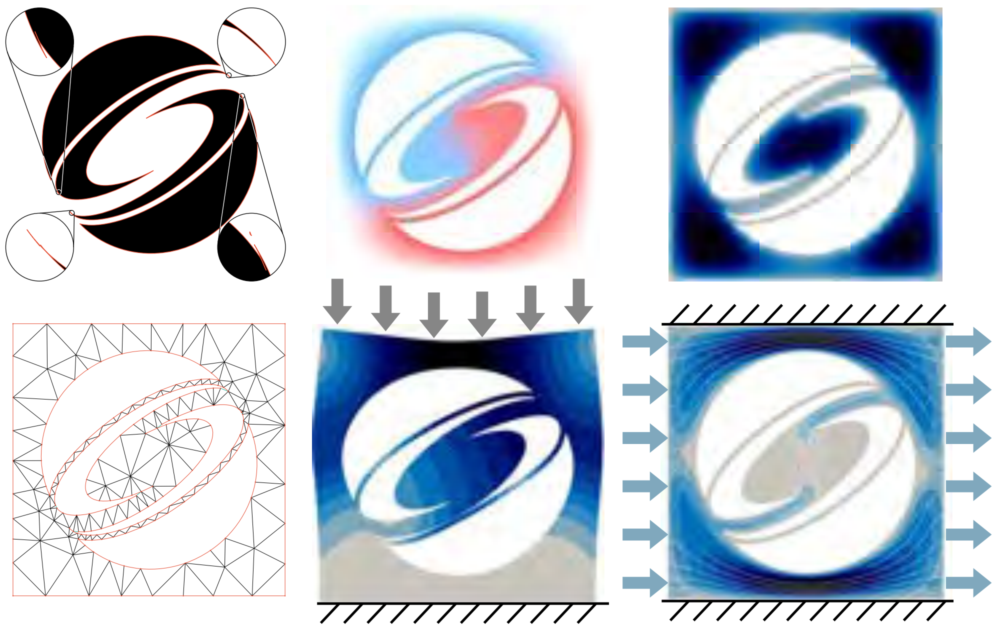


Figure 4.1: The official ACM SIGGRAPH logo (www.siggraph.org/about/logos) is converted into a curved triangle mesh. We use the mesh to compute diffusion curves (Laplace), inflate surface (bilaplace), deform elastic bodies (Neo-Hooke), and simulate fluid flow (Stokes). Note that the imperfections in the input (shown in the closeups) are automatically healed by our method.

simulation for a given desired accuracy [Ciarlet and Raviart 1972; Scott 1973, 1975; Braess 2007]. A simple 2D example is shown in Figure 4.2, which has a geometric error of 2% of the overall area when 236 linear triangles are used, and the error can be reduced to numerical zero with the same number of curved triangles with a cubic Lagrangian geometric map (Figure 4.2). While the use of curved meshes is well established in the FEM literature (with a few applications in graphics [Mezger et al. 2009; Boyé et al. 2012]), the automatic generation of these meshes is rarely considered, and the few existing methods we tested have a high failure rate on real-world examples (Section 4.5).

We propose a robust 2D meshing algorithm, *TriWild*, to generate high-quality curved meshes on complex 2D geometries. Our algorithm takes as input a 2D scene described as an SVG file (a soup of basic curved primitives, such as circles, ellipses, and Bézier curves), and automatically produces an analysis-ready, high-quality curved mesh. The algorithm starts by sanitizing the input curves and resampling them based on their curvature. This step is crucial, since "dirty" input is extremely common (see for a representative example the official SVG of the SIGGRAPH logo in Figure 4.1) and the preservation of degenerate features will inevitably lead to overrefined meshes not usable in downstream applications (Figure 4.2). The sanitized features are then meshed using a novel curved meshing algorithm that creates an initial linear mesh, curves its edges, and then maps each curved triangle to a reference domain (i.e., computes a *geometric map*). The algorithm internally uses rational coordinates for robustness and outputs a triangular mesh composed of cubic Bézier triangles with positive Jacobian in floating points coordinates. The created meshes are coarse, represent the input curves with high fidelity, and are directly usable to solve discrete PDEs.

We stress test TriWild on a large SVG collection, which is challenging even for existing robust linear triangular meshers. Our algorithm is able to handle even the most complex cases and produces meshes directly usable in FEM simulations. We also demonstrate the practical applicability of our algorithm in four common graphics applications: (1) color interpolation to create diffusion curves vector graphics [Orzan et al. 2008; Boyé et al. 2012], (2) viscous flow simulation in complex geometries [Stenberg 1984], (3) simulation of elastic deformations [Mezger et al. 2009], and (4) conversion of planar meshes into 3D models using surface inflation [Joshi and Carr 2008; Sýkora et al. 2014]. A reference implementation of our algorithm and a set of scripts to reproduce the results in the paper are provided at https://github.com/wildmeshing/TriWild.

4.2 Related Work

Our method creates a linear triangulation and then bends the edges of the mesh to create a curved triangulation. Both types of triangulations have received significant attention in the meshing literatures.

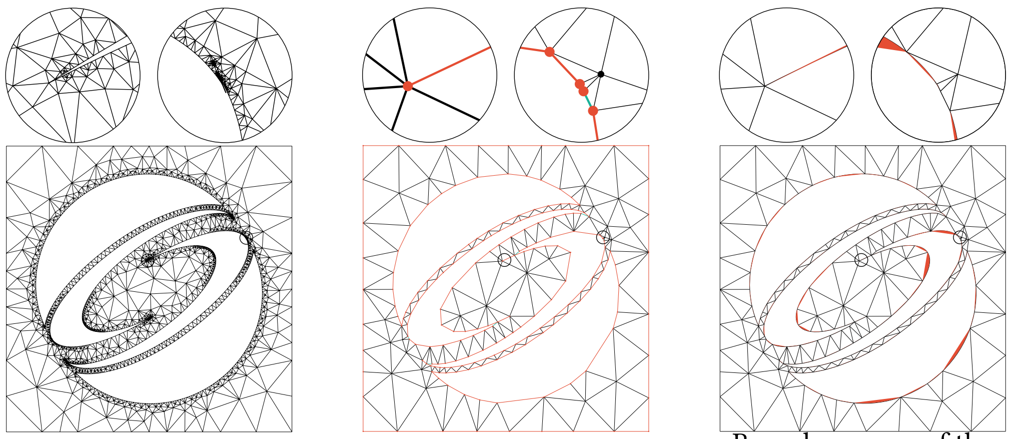


Figure 4.2: Comparison of Conforming Delaunay Triangulation (left) and of our linear output (middle and right) corresponding to the model in Figure 4.1. The curved mesh in Figure 4.1 has only 236 triangles and it approximates the input exactly, without any geometric error, while the CDT linear mesh requires around 4 thousand triangles to have a comparable visual quality.

4.2.1 Linear triangulations

Linear unstructured meshing in 2D is an old problem (e.g., [MacNeal 1949][1] or [Frederick et al. 1970]). There have been many papers, surveys and books written on this topic (e.g., [Cheng et al. 2012b; Shewchuk 2012]). Existing works can be broadly categorized by their primary methodology and corresponding strengths and weaknesses.

Advancing front methods generate a triangle mesh by growing a mesh in a flood-filling manner, typically growing inward from a given boundary [George 1971; Sadek 1980; Peraire et al. 1987]. While attractive because initial triangles placed near the starting regions can be high-quality and boundary-preservation is often trivial, the mesh quality typically gets progressively worse as the front continues. This culminates in a slew of issues when multiple fronts meet, where bad triangles are hard to avoid.

[1]In his PhD thesis, MacNeal [1949] physically created a triangle mesh on drawing paper and, by measuring angles with a protractor, solved the 2D Poisson equation using the now famous cotangent formula.

Grid/Quad-tree methods are in some sense a complement to advancing front methods. These methods begin with a grid of high-quality triangles everywhere and then adjust triangles near the domain boundary [Yerry and Shephard 1983; Baker et al. 1988; Bern et al. 1994]. The methods are fast and can obtain good *average* quality since most triangles in the interior will have perfectly regular shape. However, these methods struggle to achieve accurate boundary preservation without sacrificing *worst-case* quality at boundary triangles.

Delaunay methods are arguably the most widely used, in particular, the open source Triangle program by Shewchuk [1996]. Based on rigorous and well understood theory (e.g., [Aurenhammer 1991; Shewchuk 1999; Cheng et al. 2012b; Aurenhammer et al. 2013]), triangulations boast good mathematical properties stemming from all or most triangles fulfilling the local Delaunay criteria. Categorizations of Delaunay methods generally split according to how they deal with one-dimensional line segment constraints. *Conforming* methods iteratively add points along constraints to pure Delaunay triangulation until each segment is covered by a union of Delaunay edges. While the number of necessary inserted points is bounded (e.g., Bishop [2016] proved by $O(n^{2.5})$ for an n-vertex input segment graph), the output meshes can be prohibitively over-dense near input features. To avoid over-refinement, a preprocessing guided by a user tolerance could be done to merge or re-align problematic features before applying conforming Delaunay [Busaryev et al. 2009]. In contrast, *constrained* methods relax the Delaunay requirement for input segments. This relaxation prevents an explosion in the vertex count, but introduces difficulty maintaining quality and robustness near features. We compare directly to the Delaunay triangulation libraries Triangle [Shewchuk 1996] and CGAL [Boissonnat et al. 2002], both of which implement conforming and constrained methods.

Improvement methods attempt to increase the aggregate or worst-case quality of triangles in an existing mesh by local connectivity changes or vertex displacements [Canann et al. 1996, 1993; Lipman 2012]. The Optimal Delaunay Triangulation family choose a metric that harmonizes with Delaunay methods and their duality with Voronoi diagrams [Chen and Xu 2004]. While strategies exist to encourage these methods to huge input domain boundaries [Alliez et al. 2005b; Feng et al. 2018], they require a good, boundary-preserving initial starting point and generally do not support internal features. Our method follows the strategy of Hu et al. [2018] to create such an initial starting point for boundary and internal linear or curved features. We optimize the conformal AMIPS energy [Fu et al. 2015; Rabinovich et al. 2017] to measure and improve mesh quality.

4.2.2 Curved Triangulations

While linear meshes are predominant, curved meshes see frequent use in visual computing [Bargteil and Cohen 2014; Mezger et al. 2009] and engineering analysis [Xue et al. 2005; Bertrand et al. 2014b,a]. Meshes with curvilinear triangles offer a higher-order boundary approximation, enabling higher-accuracy simulations for smaller meshes [Babuška and Guo 1988, 1992; Bassi and Rebay 1997; Luo et al. 2001; Oden 1994; Hughes et al. 2005; Sevilla et al. 2011; Zulian et al. 2017]. To the best of our knowledge, all existing methods for constructing curved triangulations begin by creating a linear triangulation and then curve triangles to align with a curvilinear feature/boundary constraints. Our method is no exception.

Also to the best of our knowledge, all existing methods have been tested on small collections of comparatively simple models, and none of them can handle real-world, imperfect models (see Section 4.5.3 for our study on robustness of linear meshing methods, which are strictly simpler than high order methods). Our method is the first that has been tested on tens of thousands of real-world inputs.

Direct methods split features into shorter curves and then create incident triangles on the interior of the domain (similarly to linear advancing front methods) [Dey et al. 1999] or directly fit or snap high-order nodes of a curved mesh (based on minimum distances) [Ghasemi et al. 2016].

The curved triangles are represented using Lagrange polynomials, [Dey et al. 1999], quadratic or cubic Bézier polynomials [George and Borouchaki 2012; Lu et al. 2013; Luo et al. 2002], or NURBS [Engvall and Evans 2017]. To capture the high-order curve or surface, while most techniques assume an isometric mapping between each element of the linear mesh and the corresponding high-order piece by evenly interpolating the parameters of linear vertices for high-order nodes, [Shephard et al. 2005; Sherwin and Peiró 2002] take into account the anisotropic property of the to-be-curved region to compute the high order node parametric positions accordingly.

Deformation methods start with a linear mesh and elevate the degree of triangles to (so far, straight) high-order finite elements. By treating the triangulated domain as an elastic object, the mesh is deformed to curve triangles to match the input features. Different physical models have been employed, such as linear [Abgrall et al. 2014, 2012; Dobrzynski and El Jannoun 2017] and non-linear [Persson and Peraire 2009; Moxey et al. 2016; Poya et al. 2016] elasticity.

Distortion Metric, Inversion, and Intersections. Optimization methods are usually used as a post-processing step to attempt to untangle the inverted triangles created during the curving process and to improve their quality. Inverted triangles can be identified by extending the notion of area [Knupp 2000] to high-order function geometric maps [Engvall and Evans 2018; Johnen et al. 2013; Poya et al. 2016; Roca et al. 2012]. Various untangling strategies have been proposed, including geometric smoothing and connectivity modifications [Cardoze et al. 2004; Dey et al. 1999; Gargallo Peiró et al. 2013; George and Borouchaki 2012; Lu et al. 2013; Luo et al. 2002; Peiró et al. 2008; Shephard et al. 2005]. The mesh is then improved by optimizing various quality measures [Dobrzynski and El Jannoun 2017; Geuzaine et al. 2015; Roca et al. 2012; Ruiz-Gironés et al. 2017, 2016a,b; Stees and Shontz 2017; Karman et al. 2016; Toulorge et al. 2016; Ziel et al. 2017]. However, none of these methods can guarantee to produce an inversion-free curved mesh.

A different approach [Persson and Peraire 2009; Ruiz-Gironés et al. 2017] consists of initializing the optimization from a feasible inversion-free mesh, and preventing flips during deformation.

Our method follows this approach, but unlike previous methods we do not sacrifice input feature preservation. Another issue is mesh overlaps due to intersections of curved boundary segments (not necessarily incurring flipped triangles). To the best of our knowledge, our method is the first to deal with this problem explicitly (Section 4.3.1).

Representations of curved meshes vary: B-spline, implicit functions, and subdivision surfaces are typical high-order surface representations [Bruno and Pohlman 2003]. A few works assume the input being either an implicit function and fit B-spline patches [Peiró et al. 2008], or a linear boundary only and then try to obtain the high-order domain through the optimization of a linear mesh according to physical boundary conditions [Feng et al. 2018; Moxey et al. 2016; Poya et al. 2016; Ruiz-Gironés et al. 2017; Ziel et al. 2017]. The majority of the proposed works focus on polynomials, Bézier, and B-splines with a low degree (usually quadratic and cubic) [Dey et al. 1999; George and Borouchaki 2012; Lu et al. 2013; Luo et al. 2002; Peiró et al. 2008; Geuzaine et al. 2015; Johnen et al. 2013; Toulorge et al. 2013]. While Engvall and Evans [2017] show that exactly capturing NURBS patches is possible, their curved meshing algorithm is only applicable to clean CAD models, i.e. orientable, watertight, manifold, and without intersections, which are rare in practice. To the best of our knowledge, [Ruiz-Gironés et al. 2016b] is the first attempt on 2d meshing curved inputs with interior gaps. Since imperfect geometries are commonplace [Beall et al. 2003], all the existing curved meshing techniques are impractical in an automatic pipeline.

Curved Triangle Meshing Software. There are few 2D meshing software supporting curved triangles. To the best of our knowledge, the only ones available are in the Matlab Partial Differential Equation Toolbox [MATLAB Partial Differential Equation Toolbox 2018], GMSH [Geuzaine and Remacle 2009], and NekTar++ [Cantwell et al. 2015; Turner et al. 2018]. However, all of them have strict input requirements which are rarely met by vector drawings in the wild. Matlab only supports a CSG tree of circles, ellipses, and rectangles, greatly limiting its applicability. GMSH and NekTar++ both target the tessellation of domains specified in STEP format, using the Open-

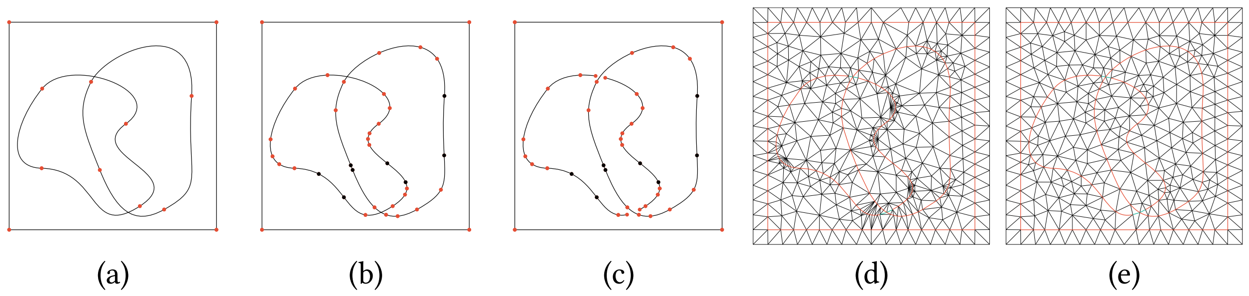


Figure 4.3: Overview of the pipeline of our algorithm. The input piecewise-Bézier curves (a) are split at inflection points (black) and at optimal position to limit the total curvature (b), and finally all intersections are removed (c). This concludes the feature preprocessing and the features are first linearly meshed (d) and finally a curved mesh is obtained (e).

CASCADE engine to create initial linear triangle meshes on the interior of a parametric patch. Neither supports open or self-intersecting curves, which are extremely common (in our dataset, this accounts for 99.95% of the inputs).

4.3 Method

Our curvilinear meshing algorithm (Figure 4.3) is divided into three stages: (1) analysis, filtering, and rounding of the input features (Section 4.3.1), (2) generation of a piece-wise linear initialization (Section 4.4.1), and (3) quality optimization and curving (4.4.2). The three stages are designed to work together, but can be used independently: for example, step (1) could be used as pre-processing for other linear triangle meshing pipelines to increase their robustness by sanitizing invalid inputs.

4.3.1 Input preprocessing and output

Commonly used 2D meshing algorithms and software make strong, often implicit assumptions about the quality of the input, usually requiring no self-intersection, no degeneracy, and no small angle between intersecting segments. However, these conditions are rarely met in real-world data, and their violation often results in either a meshing failure or over-refinement in the affected

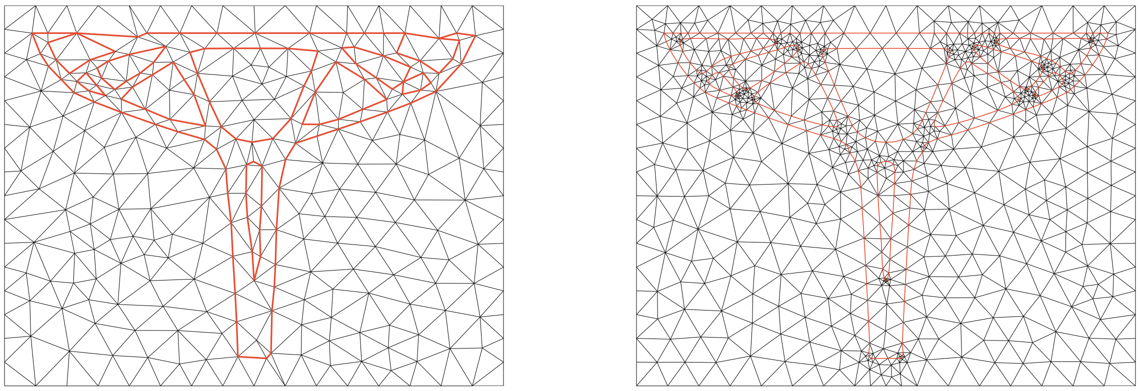


Figure 4.4: Traditional meshing tolerance may lead to jaggy boundary even in simple configurations (left). Preserving features removes the problem (right).

regions (Figure 4.2). In both cases, it might not be possible to solve PDEs in these domains, which then requires manual interaction to clean up the problematic regions. A possible solution is using a meshing tolerance [Mandad et al. 2015b; Hu et al. 2018], i.e. allowing a controlled geometric error in the created mesh: if the tolerance is sufficiently large, these algorithms will automatically remove small features, preventing over-refinement and numerical problems due to imperfections in the input. However, this comes at the cost of approximating the input, which is particularly problematic around straight features, which might become jaggy (Figure 4.4).

We propose a different approach: we analyze the input curves to identify a subset (*primary feature curves*) that can be represented with a curved triangle mesh with a user-desired target edge-length l, and approximate the rest (*secondary feature curves*) with a piecewise linear mesh within an ϵ meshing tolerance.

Input Description. Our input is a feature soup $\mathcal{F} = \{\mathcal{P}, C\}$ of 2D isolated points ($\mathcal{P}$) and 2D cubic Bézier curves (C) representing the features of the scene. The parameters of the primitives are provided in double floating points.

We obtain the input feature soups by converting a subset of the SVG 2.0 standard (points, circles, ellipses, curves, and straight lines) into their cubic Bézier representation (replacing circles and ellipses with the Bézier approximation used by Adobe Illustrator) and we output our results in the gmsh format [Geuzaine and Remacle 2009].

Next, we define primary and secondary feature curve sets. The algorithms for obtaining these are described in Section 4.4. The primary feature curve set satisfies two conditions: bounded curvature and μ-separation.

Bounded Curvature. Let l_m be the user-controlled minimal edge length of the final mesh. Now consider a circle with curvature higher than $2/l_m$: since a triangle of this size cannot fit inside the circle it will be impossible to represent such a feature without refining more than what the user prescribed. We thus discard all the parts of curves whose local curvature is higher than $2/l_m$, and denote these new feature set as $\mathcal{F}_l$.

μ-separation. Ideally, we would like to preserve all features in $\mathcal{F}_l$. However, this might be impossible if we want to represent the output at a given resolution. A counterexample is simple to construct: take as input two parallel segments at a distance d. A triangle mesh that exactly represents both segments must contain at least one triangle between them, and the area of this triangle will tend to 0 as d tends to 0, leading to a possible inversion of the triangle due to floating point rounding errors. Similarly, two feature endpoints that are at a distance d between each other will force the insertion of two vertices in the final mesh, corresponding to the two endpoints, that are also at a distance d. As d tends to zero, rounding errors might flip triangles that are in the neighbourhood. Even if we could somehow perturb the floating point coordinates to prevent inversions, the quality and size of these triangles will make the resulting mesh unusable for most downstream applications.

To avoid this problem, we formally identify these cases defining a local validity condition for sets of planar curves.

Definition 4.1. Let $\mathcal{F} \in \mathbb{R}^2$ be a collection of planar piecewise Bézier curves. The *μ-separated* set $\mathcal{F}_\mu$, is the subset of $\mathcal{F}_l$, such that for any point p on a feature f, the μ ball centered at p contains only a single connected component of f and no other feature curves or connected components of f.

This definition ensures that no pair of points in the *μ-separated* set $\mathcal{F}_\mu$ can be closer than μ, except if they belong to the same curve and are connected by a part of the curve fully contained inside a μ ball. For convenience, we will refer to $\mathcal{F}_\mu$ as the *primary features*, and its complement $\mathcal{F} \setminus \mathcal{F}_\mu$, i.e. the parts of features that have been filtered out because of their high curvature or because of the μ-separation, as *secondary features*.

The parameters μ (feature envelope) and l_m (minimal edge length) control the desired level of approximation: a small μ will lead to denser meshes with more features tagged as primary and preserved more accurately in the final mesh (Figure 4.6). Similarly, a small ϵ (boundary envelope, Section 4.4.1) will produce secondary feature which better approximate the input secondary curves. We also use a numerical tolerance ϵ_m to account for rounding errors in the control points of the input curves.

Output Description. The output of our algorithm is a valid triangular mesh of a bounding box containing all input curves. Its edges are line segments or cubic Bézier curves, satisfying the following properties: (1) the edges do not intersect each other or the boundary of the bounding box, and (2) the edges in one-to-one correspondence with the primary features are within μ distance from their assigned feature (Section 4.4). While we do not have any formal guarantee on the preservation or approximation of the secondary features (Figure 4.3 (e) green edges), our algorithm strive to preserve them if the resolution allows and if they are not too close to primary features (which are given priority).

The coordinates of the output mesh vertices and control points for the curved edge are represented in double floating point representation. We use cubic Bézier curves for the edges since they can induce *volumetric* geometric map defined with cubic Lagrange triangles which, restricted to edges, correspond to the Bézier curve. We decide to use Lagrange bases for the map, since

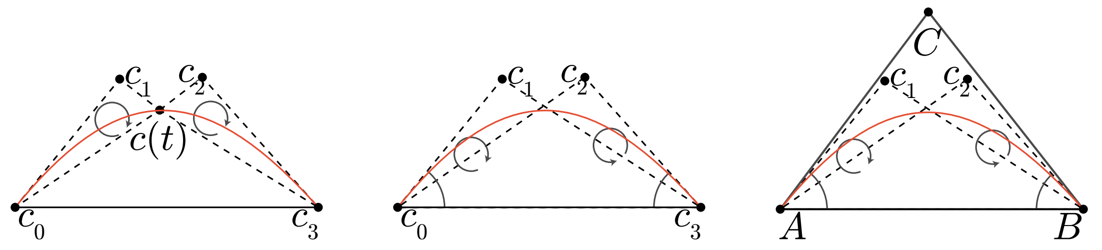


Figure 4.5: Notation for Lemma 4.2, Lemma 4.3, and Theorem 4.4.

they are ubiquitously used in curved meshing applications and are supported by most FEM systems [Geuzaine and Remacle 2009; Ansys 2018; Abaqus 2018].

Note that, during our optimization we not only produce curved meshes but also try to produce a bijective geometric map expressed with Lagrange bases. Our algorithm could be adapted to produce curved meshes with NURBS edges, which would allow to reproduce ellipses and circles exactly; we leave this extension as a future work.

4.3.2 Mesh Quality for Triangle Curving.

Our algorithm is based on a necessary and sufficient condition on the mesh quality (minimal angle) of a linear mesh, that ensures that its edges can be bent into Bézier curves without self-intersections. We will use this condition as a criteria to guide the discrete resampling of the input curves, which will be used to create an initial linear triangle mesh.

Lemma 4.2. *Let $c(t)$, $t \in [0,1]$, be a cubic Bézier curve with control points c_0, c_1, c_2, c_3, no self-intersection, and constant-sign curvature less than π, i.e. no inflection points (Figure 4.5, left). Then, for any $t \in [0,1]$, $c(t)$ is on the same side of line segments $[c_0, c_1]$ and $[c_2, c_3]$.*

Proof. Because of the constant-sign curvature, the rotation of the vector $c'(t)$ is always positive (or negative) prohibiting the curve to change direction. Additionally, since the curvature is less than π, the two segments $[c_0, c_1]$ and $[c_2, c_3]$ are in the same side of the line passing trough $[c_0, c_3]$. Therefore, for the curve to cross the edge $[c_0, c_1]$ and match the tangent at the end point it requires to a full turn which can be only achieved with a self intersection since degree 3 Bézier cannot spiral. $[c_2, c_3]$ follows by symmetry. □

Lemma 4.3. *Let $c(t)$, $t \in (0,1)$ be a cubic Bézier curve with control points c_0, c_1, c_2, c_3, without self-intersection, constant-sign curvature (no inflection points), and total curvature α strictly smaller than π (Figure 4.5, middle). Then the signed angles between the vectors $\vec{c_0, c_3}$ and $\vec{c_0, c_1}$ and $\vec{c_2, c_3}$ and $\vec{c_0, c_3}$ turns in opposite directions, and are smaller than α.*

Proof. The fact that two angles between the segments turn in opposite directions follows from Lemma 4.2: any point on the curve is on the right (and left side) of the tangent and in particular the intersection between the segments $[c_3, c_1]$ or $[c_0, c_2]$. Finally, since the total curvature is the integral of the curvature (which is constant-sign) is smaller than α, any tangent angles is smaller than α, and in particular the two at the endpoints which correspond to the angles $\angle c_1 c_0 c_3$ and $\angle c_0 c_3 c_2$. □

Theorem 4.4. *Let ABC be a triangle with minimal angle α, and $c(t)$, $t \in (0,1)$ be a cubic Bézier curve with control points A,c_1,c_2,B, no self-intersection, constant-sign curvature and total curvature α strictly smaller than π (Figure 4.5, right). Then c(t) does not intersect the edges AC and CB, for any $t \in \{0..1\}$.*

Proof. Since that quadrilateral is contained in the triangle ABC, it follows that the curve will not intersect it. □

Theorem 4.4 provides a direct connection between linear mesh quality (measured as the minimal angle over the triangles of a linear mesh) and the maximal turning angle of the feature curve assigned to one of linear mesh edges. We will fix a minimal angle $\alpha = 10$ degrees that the meshing algorithm will optimize for, and refine the input features to ensure that the angle between the tangents at the endpoints and corresponding edges does not exceed α. Note that this condition is only necessary but not sufficient to ensure a positive Jacobian of the Lagrangian geometric map assigned to the triangle, and it is thus only a very effective heuristic (Section 4.5).

4.4 Feature Processing for Curved Meshing

We introduce an algorithm to compute primary and secondary features from the set $\mathcal{F}$ of input features. This algorithm uses standard double floating point precision. The algorithm has six parameters: μ the feature envelope (distance between primary features), ϵ the boundary envelope (desired accuracy for secondary features, described below), ϵ_m (small number used to account for roundoff errors), α (desired minimal angle in the target mesh), l (desired edge length of the target mesh), and l_m (minimal edge length of the target mesh). In our experiments, we use μ =1e-3d, $\epsilon = 2\mu$, ϵ_m = 1e-8, α = 10 degrees, $l = d/20$, and l_m =1e-4d with d the diagonal of the bounding box. It first splits the feature curves into elementary pieces (*discrete curve sections*) discarding some not satisfying primary feature constraints, and then classifies the remaining ones.

Removal of Degenerate Curves. We convert all degenerate curves which have all their control points too close together (i.e., contained in a circle of radius ϵ_m), into a feature point computed as the average of the control points. The rationale behind this choice is that any such curve will be too small to be represented with a mesh of target resolution and we thus opt to represent it as a single point in the output mesh.

We also identify all Bézier curves corresponding to straight segments by least-square fitting a line to the control points, and marking them as straight if the sum of distances of the control points to the line is smaller than ϵ_m. These features are replaced by line segments to avoid numerical problems in the next steps and to simplify the point to feature queries.

Inflection Point Split. We split each curve $c_i \in C$ at its inflection points, decomposing it into at most three curves with constant-sign curvature, which is a crucial property required by Theorem 4.4. The inflection points are computed explicitly by finding the parametric coordinates for which the curvature changes sign

$$\kappa \|c'(t)\|^3 = \det(c(t)', c(t)'') = 0, \tag{4.1}$$

and ensuring that $c(t)' \neq 0$.

Total Curvature Split. While each part of the curve has constant-sign curvature by construction, its total curvature is not bounded, potentially creating self-intersections in the curving step (Theorem 4.4). We thus recursively subdivide the curves until the turning number is smaller than 180 degrees to prevent self-intersections (Theorem 4.4). The optimal splitting point, which halves the total curvature, corresponds to the point in which the tangent is the average of the two endpoints tangents. This condition can be formulated as a *quadratic* equation in t

$$\det\left(\frac{c'(0) + c'(1)}{2}, c'(t)\right) = 0.$$

Note that when the angle between $c'(0)$ and $c'(1)$ is larger than π, we need to use minus the average tangent.

Curve Resampling. We discretize the resulting curves by sampling them recursively, splitting them in half at each step, until two conditions are satisfied: (1) the segments are shorter than the user-desired target edge length l and (2) the polyline approximation is within μ distance from the Bézier curve. From now on, each curve section will be denoted with the term *discrete curve section.*

Primary and Secondary Feature Tagging. We now compute a discrete version of a μ-separated soup of features, directly using the polyline approximating the curves. We greedily traverse all the discrete curve sections, and for each one we mark it as primary, check the μ-separated condition, and discard conservatively all the segments that violate it, marking them as secondary. The

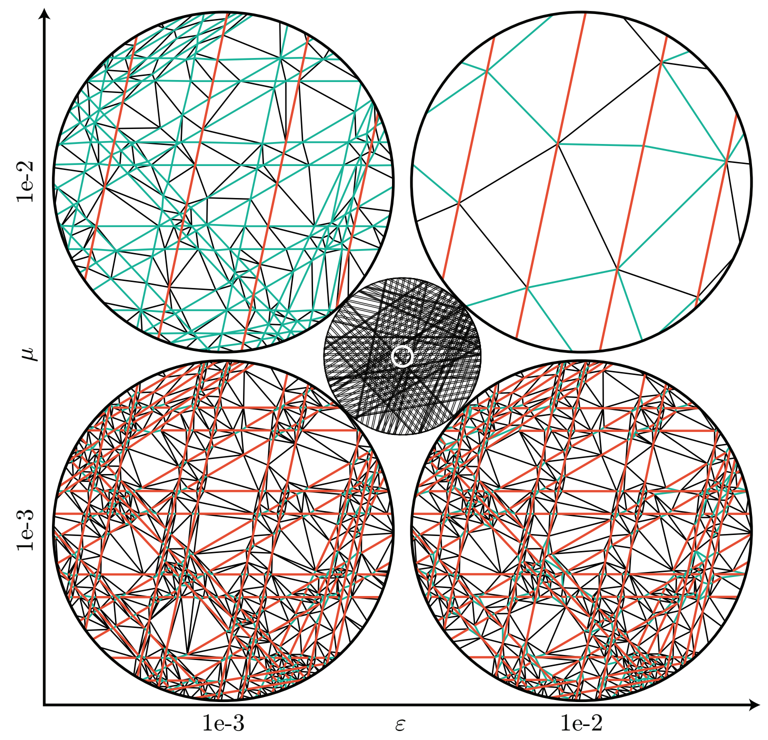


Figure 4.6: Effect of the feature envelope μ and boundary envelope ϵ on the closeup (white circle in the middle) of an input mesh (center). The larger μ and ϵ (top right corner) are, the less primary (red) and secondary (green) features the final mesh will have. By enlarging μ (y-axis from bottom to top) more primary features will be present in the final result. A similar effect is obtained by enlarging ϵ (x-axis from left to right).

output of this stage is a valid μ-separated set features.

Secondary Feature Simplification. A second pass is used to prune close secondary features: every pair of feature endpoints closer than μ corresponding to secondary features are collapsed at their barycenter. This step is not strictly necessary, but it dramatically reduces the number of triangles generated by the BSP subdivision step, improving the running times in challenging models with thousands of self-intersections.

4.4.1 Linear Meshing

Similarly to many existing curved meshing algorithms (Section 4.2), we create an initial valid linear mesh and then curve its edges to match the feature curves. However, our pipeline differs from existing approaches: (1) we interleave mesh curving and quality mesh improvement and (2) we never allow the curved triangles to get inverted, i.e. we keep the Jacobian of their geometric map positive.

Generation of a Valid Linear Triangle Mesh. To construct an initial linear triangle mesh, we implement a 2D version of the TetWild algorithm proposed in [Hu et al. 2018] , trivially adapting all the steps to their 2D counterparts, which are both simpler and much more efficient than the volumetric version. We thus denote our algorithm as TriWild.

We use the same hybrid geometric kernel proposed in [Hu et al. 2018], using rational coordinates to avoid numerical problems in the first phase, and rounding them to floating point coordinates during quality optimization. The algorithm requires a parameter ϵ to control the size of boundary envelope. In Figure 4.6, we compared the results of different combinations of envelope ϵ and feature envelope μ. The influence of targeted edge lengths l and minimal edge length l_m on final output is shown in Figure 4.7. For the sake of brevity, we describe here the extensions required to handle the input curved features and we refer to [Hu et al. 2018] for the complete description of the algorithm.

Feature Invariants. Similarly to TetWild, TriWild preserves two invariants during the mesh generation and optimization: (1) secondary feature edges need to stay within the ϵ envelope, and (2) triangles cannot be inverted. We add two additional invariant for the primary feature edges: (3) the integrated curvature of the part of the parametric curve associated with every triangle edge must be smaller than α, and (4) primary feature vertices are allowed to only move on the curve. We simply discard all operations that violate any invariant, and we use our preprocessing

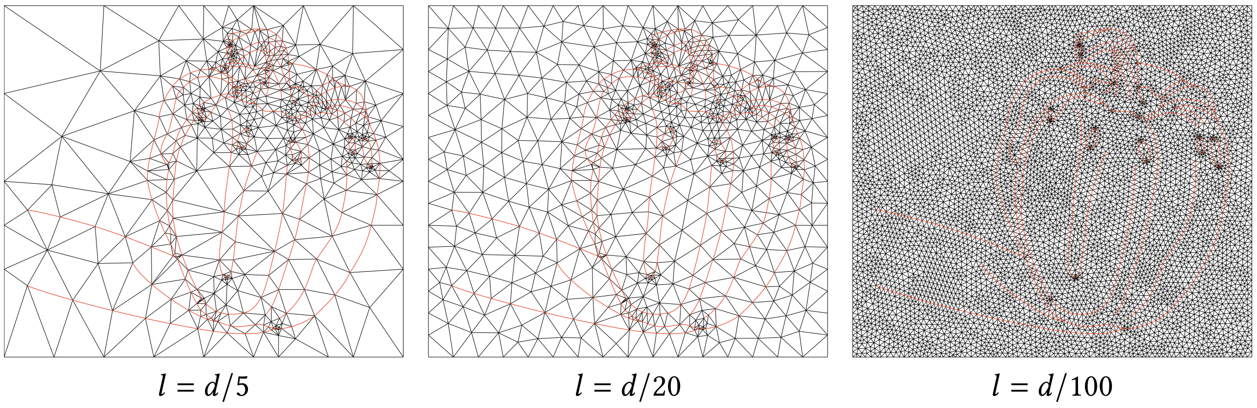


Figure 4.7: Effect of the targeted edge length l on the output mesh.

(Section 4.3.1) to ensure that the invariants hold for the initial triangle mesh generated after BSP subdivision.

Feature Handling. TriWild iterates between four local operations for mesh improvement: (1) edge splitting, (2) edge collapsing, (3) edge swapping, and (4) vertex smoothing. Their behaviour and implementation are identical to their 3D counterpart in TetWild. The only exceptions are the vertices and edges corresponding to primary features: (1) when a feature edge is split, we place the inserted vertex in the middle of the parametric curve attached if it does not introduce inverted triangles, otherwise we place it in the middle of the linear edge, (2) when we collapse edges involving an endpoint of a curve, we always keep the endpoint in the same position, (3) we disallow swaps on primary and secondary feature edges, and (4) we restrict smoothing of vertices attached to a feature to lie on the feature itself (Section 4.4.3). During all the operations, we explicitly keep track of the parametric position of all the vertices lying on primary features.

Vertex Projection. While unconditionally robust, triangle meshing using a BSP subdivision has the unfortunate side effect of potentially refining some of the edges corresponding to input features. This is problematic if the features are curved, since the inserted vertices will lie close, but not exactly on the feature. We thus add an additional step in the mesh improvement that moves every feature vertex as close as possible to its assigned feature (Figure 4.8) to enforce the fifth

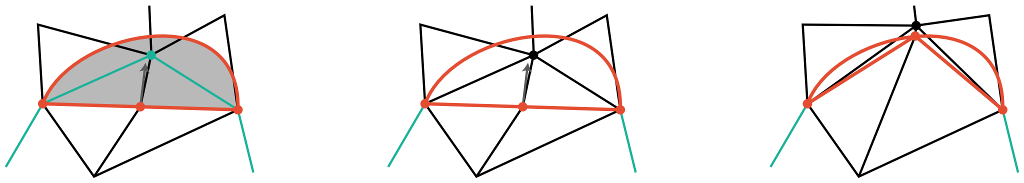


Figure 4.8: The green vertex prevents the new orange point in the feature to move to the curve without creating inverted triangles (first image). We un-mark it so it is free to move (second feature) until is pushed away and the feature vertex is snapped to its feature (last image).

invariant. For each such vertex, we compute the closest point on the feature (Section 4.4.3), and move as close as possible to it, while not violating any of the 3 invariants above. These vertices are not allowed to move further away from their closest point on the features. As the vertices move toward their target position, the rest of the mesh follows them since the smoothing operations strive to keep the quality high, eventually allowing these vertices to snap to the feature. While rare, it is possibly that some vertices in the region between the linear and the curved feature (Figure 4.8) cannot move due to Invariant (1). We thus delete the tagging for any secondary feature in these areas to allow the primary feature vertices to be snapped.

Termination Criteria. The quality optimization terminates when the minimal angle of the mesh is larger than α or the AMIPS energy is smaller than 10 (default value of TetWild), which is a good heuristic for curving the linear triangles (Theorem 4.4). Note that this condition does not guarantee that the elements will not be inverted during curving, but makes it less likely. We also stop the optimization if the maximum of iterations is reached. We have experimentally observed that for most models it is possible to stop the optimization prematurely, without affecting the quality in noticeable ways, and we thus used lower thresholds (10 degrees, AMIPS energy 30, max 80 iterations) for the large scale stress tests to reduce the overall running time at the cost of mesh quality.

4.4.2 Curvilinear Mesh Optimization

The result of the previous stage is a linear mesh which edges are assigned to a primary feature and which vertices have the corresponding parametric values. We now aim to construct a per-triangle bijective geometric map expressed in Lagrange form which, restricted to edges, corresponds to the curve. This curvilinear mesh optimization iteratively warps the edges of the linear mesh while optimizing its quality with local operations. Note that our optimization only bends the feature edges.

Splitting and Curving To simplify all operations applied on a curved mesh, we split every triangle with more than one edge assigned to a curved feature until it has only one edge. We assign the cubic Lagrangian geometric map

$$g(x,y) = \sum_{i=0}^{9} v_i \ell_i(x,y)$$

(see Section 4.4.3 for the definition of $\ell_i(x,y)$) to each triangle with a feature edge (the first 3 v_i are the vertices of the triangle) and we compute its coefficients v_i by evaluating the curve $c(t)$ at $1/3$ and $2/3$ for the curved edge. For the remaining two linear edges, we sample them linearly, and use the average first 9 v_i for the position of the central node. This procedure is not guaranteed to produce a valid geometric map due to 2 reasons: (1) the solution might not exist (avoiding intersections between the boundary segments of a Lagrangian triangle is not a sufficient condition to ensure an overall positive Jacobian), and (2) even if it exists it might not be in the simple form we just described. We thus skip problematic triangles in this stage (assigning them a linear geometric map) and we improve the quality of the mesh further, increasing the probability that the geometric map fit will produce a bijective map.

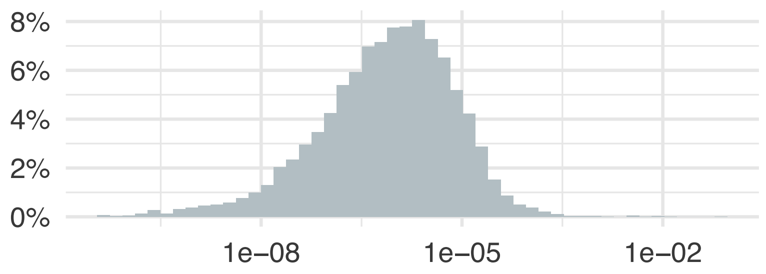


Figure 4.9: Maximal least square fitting error with respect to d on a log scale.

Curved Mesh Optimization. The mesh quality is improved using the same local operations used in Section 4.4.1, but with less constraints, since it is not necessary anymore to enforce the total curvature invariant. Since the mesh is now using a non-linear geometric map, the Jacobian is no longer constant in each triangle, and checking for validity is more expensive [Geuzaine et al. 2015]. Note that each operation on a curved triangle potentially modifies the curved edges and requires to update the Lagrange coefficients v_i to *always* match the associated feature.

Termination Criteria and Least Squares Fitting. The optimization terminates when the user-controlled quality threshold is achieved (AMIPS Energy 20), or after a user-controlled number of iterations (default 10) is reached. If there are curved triangles which are still impossible to curve exactly using the simple Lagrange basis (i.e., the Jacobian of the geometric mapping is negative), we do a best-effort and fit them only in a least square sense by optimizing for the Lagrangian coefficients v_i that best approximate the boundary curve (minimize the distance between the two curves) under the constraint of having a positive Jacobian. This happens to 44.7% meshes of all our outputs and 0.18% faces per mesh (those with fitting) in average. The overall max error in our experiments is 6.99e-2d (mean 1.87e-5d, std 6.99e-4, Figure 4.9), indicating a faithful reproduction of the input features.

4.4.3 Technical Detail

Curved AMIPS For a feature vertex $p = c(t)$ where c is the feature curve and t is the parametric value, we let $E(p)$ be the traditional AMIPS energy which is minimized using Newton method, thus we requires gradient and hessian of E. For the smoothing on the features the minimization becomes univariate:

$$\min_{t\in(0,1)} E(c(t)).$$

We now need gradient and hessian of $E(c(t))$ with respect to t which can be easily obtained by the chain rule:

$$E'(c(t)) = \langle(\nabla E)(c(t)), c'(t)\rangle,$$

and

$$E''(c(t)) = (c'(t))^T\, H_E((c(t))\, c'(t) + \langle(\nabla E)(c(t)), c''(t)\rangle.$$

Note that since $E(c(t))\colon \mathbb{R} \longrightarrow \mathbb{R}$ the gradient and hessian are scalars (while for $E(x, y)\colon \mathbb{R}^2 \longrightarrow \mathbb{R}$ they are tensorial).

Point Cubic Bezier Distance The parametric value $t^\star$ of the closest point to p in the curve c can be find as

$$t^\star = \arg\min_{t\in(0,1)} \|c(t) - p\|^2,$$

which, by the first order condition leads to the following equation

$$\frac{\mathrm{d}}{\mathrm{d}t}\|c(t) - p\|^2 = 0.$$

Since $c(t)$ is a polynomial of order 3, the squared norm becomes of order 6, which implies that the first order condition equation is of order 5. Therefore it can be written as

$$\frac{\mathrm{d}}{\mathrm{d}t}\|c(t) - p\|^2 = at^5 + bt^4 + ct^3 + dt^2 + et + f = 0.$$

Fining roots of this polynomial is a challenging task, which we solve by computing the eigenvalues λ_i, $i = 0, \dots, 4$ of the 5×5 companion matrix

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & f/a \\ 1 & 0 & 0 & 0 & e/a \\ 0 & 1 & 0 & 0 & d/a \\ 0 & 0 & 1 & 0 & c/a \\ 0 & 0 & 0 & 1 & b/a \end{pmatrix}.$$

For each *real* eigenvalue λ_i we compute $\|c(\lambda_i) - p\|^2$ and select $t^\star$ as the λ_i with smallest distance in the interval $0, 1$. If no λ_i are in interval, we know that $t^\star$ will be either zero or one, which can be decided by evaluating the distance from the endpoints. Note that we opted for this solution for simplicity, a more efficient alternative is Bezier clipping [Sederberg and Nishita 1990].

Cubic Lagrange Bases The ten Lagrange bases are defined as

$$\begin{aligned}
\ell_0 &= -\frac{1}{2}(3y - 1 + 3x)(y - 1 + x)(3y - 2 + 3x) & \ell_1 &= \frac{1}{2}x(9x^2 - 9x + 2) \\
\ell_2 &= \frac{1}{2}y(9y^2 - 9y + 2) & \ell_3 &= \frac{9}{2}x(x + y - 1)(3x + 3y - 2) \\
\ell_4 &= -\frac{9}{2}x(3x^2 + 3xy - 4x - y + 1) & \ell_5 &= \frac{9}{2}xy(3x - 1) \\
\ell_6 &= \frac{9}{2}xy(3y - 1) & \ell_7 &= -\frac{9}{2}y(3xy - x + 3y^2 - 4y + 1) \\
\ell_8 &= \frac{9}{2}y(x + y - 1)(3x + 3y - 2) & \ell_9 &= -27xy(x + y - 1)
\end{aligned}$$

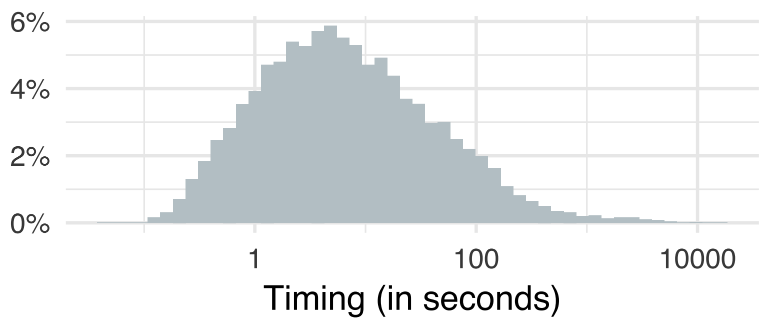


Figure 4.10: Timing of our curved pipeline on a log scale.

4.5 Results

We implemented our algorithm in C++, using Eigen [Guennebaud et al. 2010] for linear algebra routines. The source code of our reference implementation is available at `https://github.com/wildmeshing/TriWild`. The experiments were performed on cluster nodes with 2 Xeon E5-2690v4 2.6GHz CPUs and 250GB memory, each with 64GB of reserved memory, and allowed for a maximum running time of 6 hours.

To test our curved mesh generation we crawled 19,686 SVG images from openclipart.org, a free SVG image repository. For each SVG image we extract the set of features $\mathcal{F}$ which are used in our algorithm. Figure 4.11 shows some examples of the created curved meshes. Within the time limit of 6 hours we successfully created 19,685 curved meshes with only one failure due to large input size. Figure 4.10 summarizes the running time of our curved pipeline. Since we always reject operations (included the fitting) introducing inverted elements, the only possible failure is inherited by TetWild: some vertices might fail to be rounded into floating points. However, this never happened in our experiments.

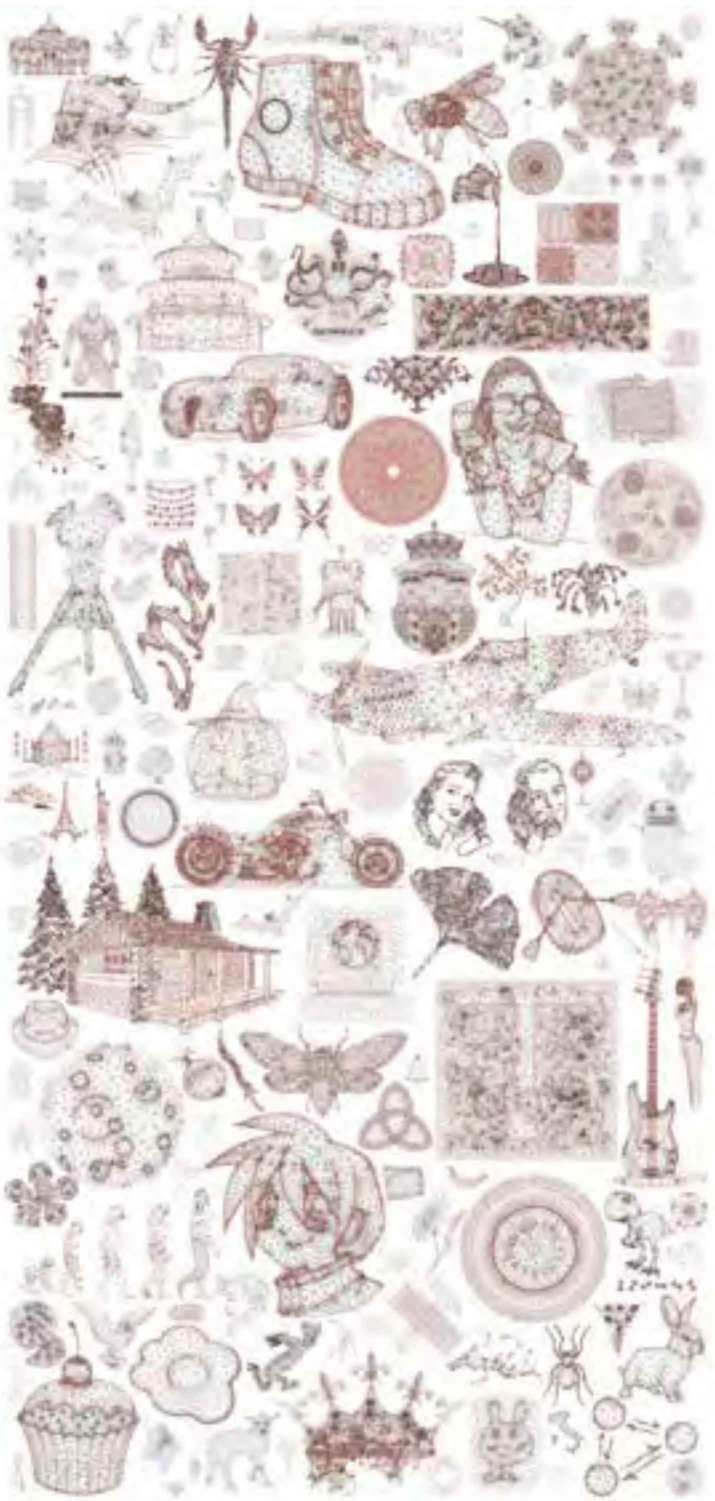


Figure 4.11: Random selected examples of curved meshes obtained with our method on the Openclip dataset.

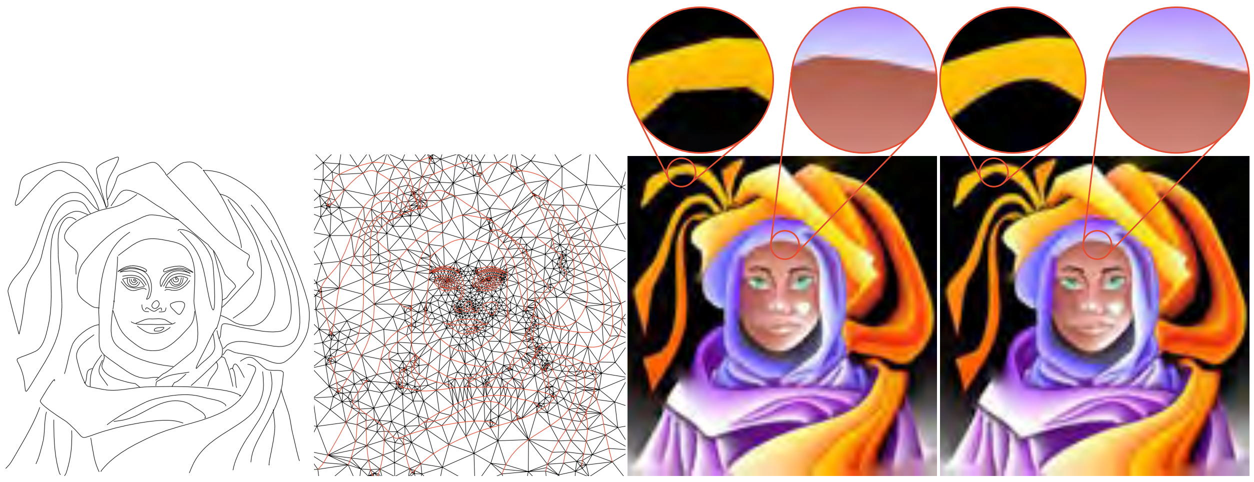


Figure 4.12: Example of diffusion curves for a linear (second right) and curved (most right) mesh. The most left is the input and the second is the our curved output.

4.5.1 Applications

The main application of meshing is simulation, which aims to solve a partial differential equation (PDE) for an unknown function u defined over a domain Ω subject to some boundary constraints. The role of the function u depends on the application, for instance in elasticity it is displacement, while for fluids it is velocity. One of the typical numerical methods to solve PDEs is the finite element method (FEM). As the name suggests, the first essential step of a FEM consist of meshing the domain (thus creating the triangles) which can be done with both linear or curved meshes. In fact, for FEM simulations, one requires only a *bijective map* from the reference triangle (unit right-angle triangle) to the actual physical triangle, without any assumptions about the linearity of the map. Changing the *geometric* order of the triangles (e.g., from linear to cubic) only changes the local assembler but not affects the size and sparsity of the final linear system. In fact, it only requires a slightly higher quadrature order, leading to similar performance as linear triangles.

The main advantage of high order geometries is that we can use coarse meshes, which leads to faster simulations, without introducing any geometrical error on the boundary description. We now demonstrate the effectiveness of our curved meshes for different applications using different

PDEs. We focus our applications to standard PDEs used in graphics, and remark that other more complex equations might benefit more from high-order geometries [Bassi and Rebay 1997]. All experiments are done on cubic meshes (the geometric map used is cubic Lagrange polynomials) with quadratic Lagrange polynomials as basis functions to represent the solution. When we compare against meshes with straight edges, we replace the cubic geometric map with a linear one. We stress that replacing the linear geometric map with a cubic one produces systems with *exactly* the same size and sparsity, the only difference are the entries of the matrices, which are slightly more expensive to compute for the high-order map.

Laplacian The simplest, and by far the most popular in graphics, PDE is the Laplace equations

$$\Delta u = f, \quad u = g \quad \text{for } x \in \partial\Omega,$$

where $\partial\Omega$ is the boundary of the domain. A popular graphics application of such equation is diffusion curves [Orzan et al. 2008; Boyé et al. 2012]. A set of curves is augmented with 2 colors, one on the left and one on the right. The final image is generated by diffusing the colors from the input colored curves. In other words, the mesh is cut open along the curves, the left and right colors play the role of boundary conditions, and the solution is the color at every pixel. We generate a curved mesh with our method from the annotated curves in [Orzan et al. 2008] and run the diffusion simulation with and without curved triangles, Figure 4.12. We clearly see that, at the same resolution, the curved mesh provides a superior result, avoiding approximation artifacts on the diffusion curves. The runtime for the linear mesh is 0.074s, and switching to cubic geometric map only adds 0.016 seconds.

Bi-Laplacian For many graphics applications, the Bi-Laplacian equation is preferred since it produces smoother results around the boundary.

$$\Delta^2 u = 0, \quad u = g \quad \text{for } x \in \partial\Omega,$$

To solve this PDE we require mixed finite elements [Monk 1987] or C^1 basis functions [Boyé et al. 2012], we opted for the former since it is simpler to implement. Among many application of the bi-Laplacian, we picked surface inflation [Joshi and Carr 2008; Sýkora et al. 2014]. The idea is elegant: the mesh is fixed at the curves (zero boundary conditions) and a force is applied to the curves to "lift" the mesh. We follow the Rèpousse [Joshi and Carr 2008] construction but instead of using the curvature to inflate we use a constant value. Figure 4.13 shows the results for the same mesh with and without curved triangles: similarly as before, the curved mesh do not exhibit any visible artifacts, and the runtime is only slightly (0.2%) higher for the curved mesh.

Elasticity In Figure 4.14, we study the difference of an elastic deformation using the Neo-Hookean elasticity model, where the stress σ relationship is not linear with respect to the displacement u:

$$-\mathrm{div}(\sigma[u]) = f \quad \sigma[u] = \mu(F[u] - F[u]^{-T}) + \lambda \ln(\det F[u]) F[u]^{-T},$$

where $F[u] = \nabla u + I$ and λ, μ are material parameters. Note that due to the non-linearity of the PDE, the solution method requires a non-linear solver such as the Newton method. We use a simple setup, a square domain with a circular hole in the middle is hanged on the top (zero Dirichlet) and gravity is applied. Even in the context of non-linear PDE the overhead of curved triangles is negligible (curved 0.0975s, linear 0.0857s) and curved meshes produce results closer to the ground truth solution computed on a dense mesh, Figure 4.14.

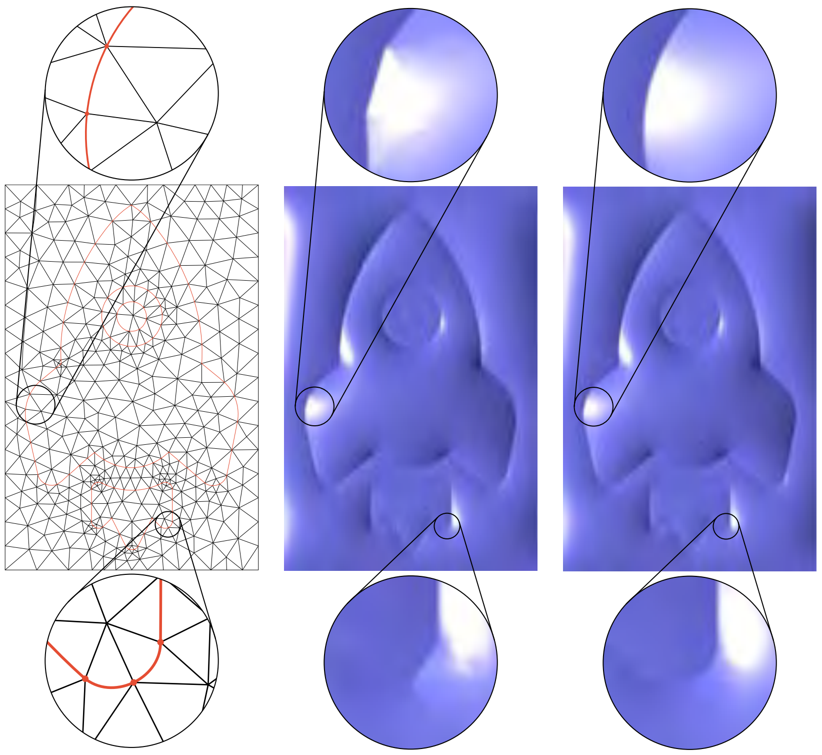


Figure 4.13: Example of inflation on a linear (middle) and curved (right) mesh of the same resolution, the input mesh is shown in the (left).

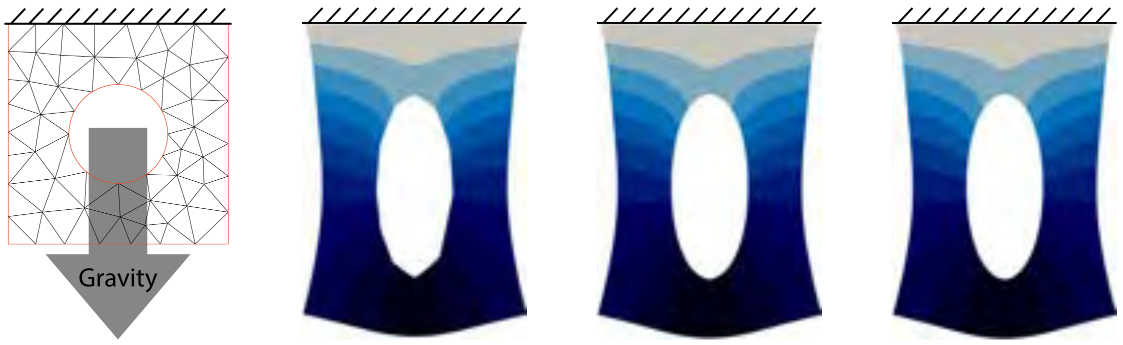


Figure 4.14: Example of elastic deformation of linear (second figure) and curved (third figure) triangles. A solution on a dense mesh is also provided for reference (fourth figure). Note how the curved mesh provides a much more accurate solution than the linear mesh, especially around the hole. The coarse mesh used in both simulations is shown in the first figure. The color shows the y-displacement.

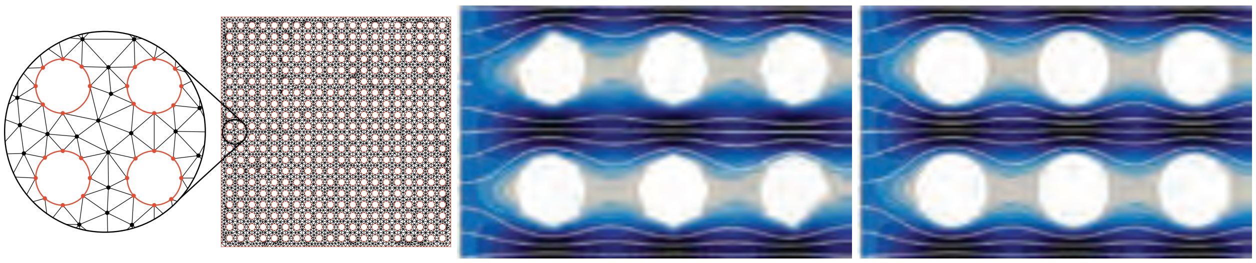


Figure 4.15: Example of Stokes fluid simulation for small circular obstacles (left) on linear (middle) and curved (right) triangles. The color shows the norm of the velocity of the fluid, and the lines are the stream lines.

Stokes The final application we consider is fluid simulation, where we are looking for the fluid velocity u of a fluid. We solve the Stokes equation

$$-\mu\Delta u + \nabla p = 0$$
$$-\mathrm{div} u = 0,$$

where p is the pressure and μ is the viscosity of the fluid. The experiment consists of a pipe filled with small circular obstacles with a fluid passing through (Figure 4.15 left), a setup used for studying cancer cell migration within interstitial tissues [Panagiotakopoulou et al. 2016]. In other words, we have a constant non-zero boundary condition on the left and right side of the domain (in and out flow velocities) and zero velocity on the rest of the boundary (i.e., the top and bottom, and the obstacles). Figure 4.15 shows a close-up of the results of the simulation for curved and linear meshes. Because of the poor approximation of the linear mesh, the behaviour of the fluid is asymmetric and unnatural. While simulation on linear mesh takes 2.65s, it takes 2.70s on curved mesh where the difference is small.

4.5.2 Comparison with Curved Meshers

To the best of our knowledge only three existing available software allow to generate curved meshes that preserve input curve features: the Matlab PDE toolbox [MATLAB Partial Differential

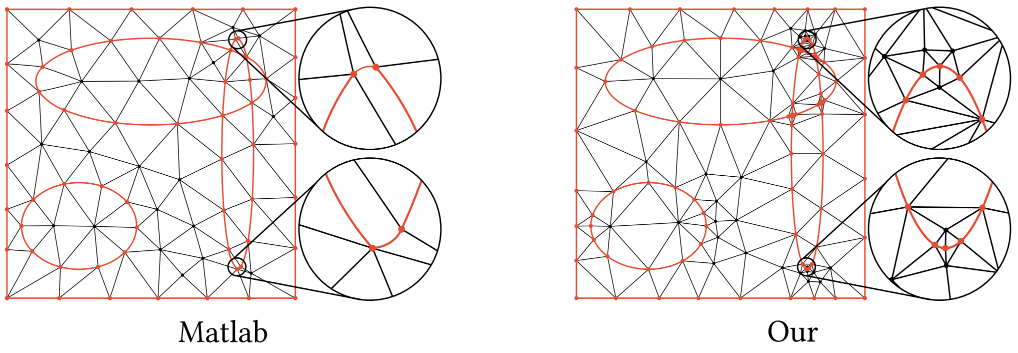


Figure 4.16: Comparison between Matlab (left) and our (right) mesh for a simple set of ellipses.

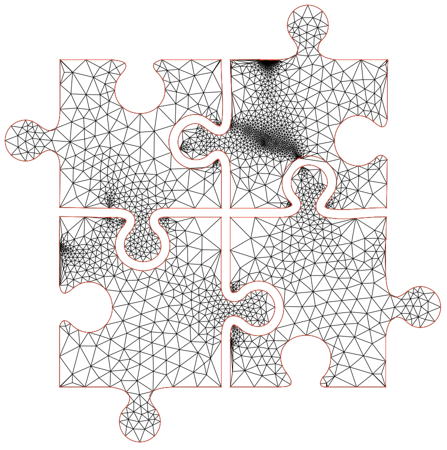
Equation Toolbox 2018], GMsh [Geuzaine and Remacle 2009], and NekTar++ [Cantwell et al. 2015]. The main difference between these software and our solution is that they require a CSG tree (Matlab) or a boundary curve which clearly defines an interior domain (GMsh, NekTar++). With these solutions it is impossible to mesh a set of *open* input curves, thus limiting the applications, for instance diffusion curves or inflation (Section 4.5.1) would be impossible. We manually created 2 simple examples and provided a representative comparison to one method for each category.

Matlab According to the Matlab documentation[2], the 2D meshing package exploits constructive solid geometry (CSG), which uses a set of solid blocks: square, rectangle, circle, ellipse, and polygon. This short list of primitives limits the applications considerably, for instance it is not possible to represent even closed polybezier curves. Another major limitation of the toolbox is that it supports only linear and quadratic triangles, leading, for instance, to poor approximations of circular arcs. To test Matlab we setup a simple example, a unit square with 3 ellipses curves added[3] and produced the SVG for our method, Figure 4.16. Note that our method constructs a dense high-quality mesh around the high-curvature tip of the ellipse while Matlab resorts to large low quality triangles.

While constructing this example we discover 2 problems in the Matlab mesher: if we shrink

[2]`https://www.mathworks.com/help/pde/geometry.html`

[3]The Matlab experiment consists of: a unit square at the origin, an ellipse centered at $(0.25, 0.25)$ with semiaxes 0.2 and 0.15, an ellipse centered at $(0.5, 0.75)$ with semiaxes 0.4 and 0.15, and an ellipse centered at $(0.8, 0.5)$ with semiaxes 0.055 and 0.4. The four primitives are just added.



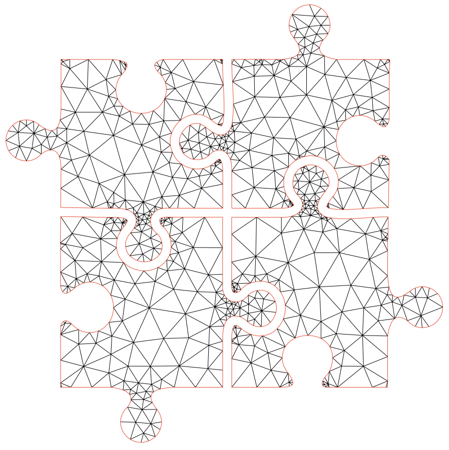


Figure 4.17: Comparison of creating a curved cubic mesh for a puzzle piece with GMsh (left) and our method (right). GMsh mesh contains 19 inverted triangles, whereas ours has none and it can be directly used in downstream applications.

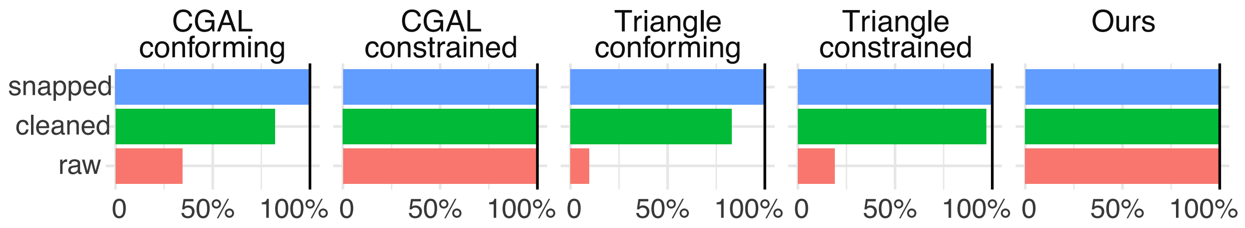


Figure 4.18: The success rate of generating a triangulation over the raw, cleaned and snap rounded Openclipart data set.

the last ellipse first semi-axis from 0.055 to 0.054 it produces an error and cannot generate any valid output; by shrinking it even more it goes in infinite loop, while our method is unaffected by these changes.

GMsh GMsh requires the curves to define a closed domain. Since GMsh exactly reproduces the input boundaries, it overrefines near the defects (Figure 4.17). Additionally, it relies on an "un-tangling" strategy: it first generates a possibly invalid curved mesh, and then tries to "un-invert" the triangles. For the example in Figure 4.17, the untangling fails and the mesh still contains 19 inverted triangles, making it unsuitable for FEM simulations.

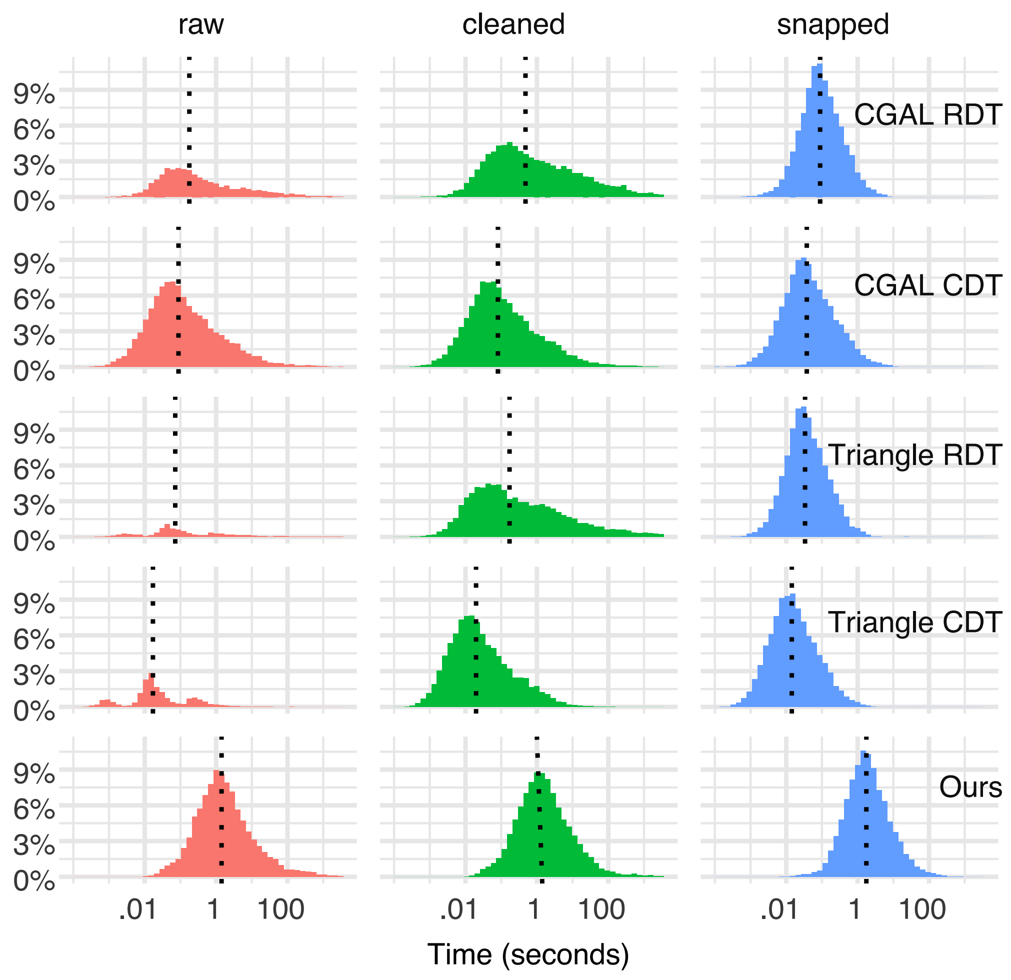


Figure 4.19: The total running time of all triangulation algorithms.

4.5.3 Linear Meshing Comparison

Our algorithm can be used to robustly generate traditional linear triangles meshes, by simply loading a collection of points and line segments, and marking all the segments as secondary features (Section 4.4). We compare extensively with two popular open source 2D meshing libraries, implementing the current state of the art triangulation algorithms: Triangle [Shewchuk 1996] and CGAL's 2D Triangulation module [Boissonnat et al. 2002]. Both libraries are capable of taking a set of (potentially intersecting) segments as constraints and generating either a constrained Delaunay triangulation (CDT) or a conforming Delaunay triangulation (RDT) as output. We compare our results with both CDT and RDT results generated by these libraries on three sets of input: (1) raw input, (2) cleaned input, and (3) snapped input. Raw input contains piecewise linear approximations of 19,686 SVG images crawled from openclipart.org The clean input is obtained by (1) removing duplicated vertices, (2) removing duplicated edges, and (3) removing degenerate edges. Note that intersecting segments are not fixed in the cleaning process, since all methods supports them. Lastly, the snapped input is the output of iteratively snap rounding [Goodrich et al. 1997] the raw input using ϵ as the pixel size. For linear comparison, we limit the maximum computing resources for each input to 1 hour running time and 16 GB memory.

Success Rate The first, and simplest, metric is to check if the algorithm successfully generated a non-empty output. Figure 4.18 compares all methods on the three types of inputs. For snapped inputs (where there is no intersections), all methods succeeds nearly 100%. However on cleaned inputs, RDT from both CGAL and Triangle fails due to the presence of intersecting segments. Only our approach and CGAL CDT are robust enough against the rampant presence of intersecting and degenerate segments from the raw input. It is interesting to observe that the single model that produces invalid output with Triangle RDT is not the same one that produces an invalid output for CGAL RDT.

Figures 4.19 and 4.20 shows the total running time and the total number of triangles generated

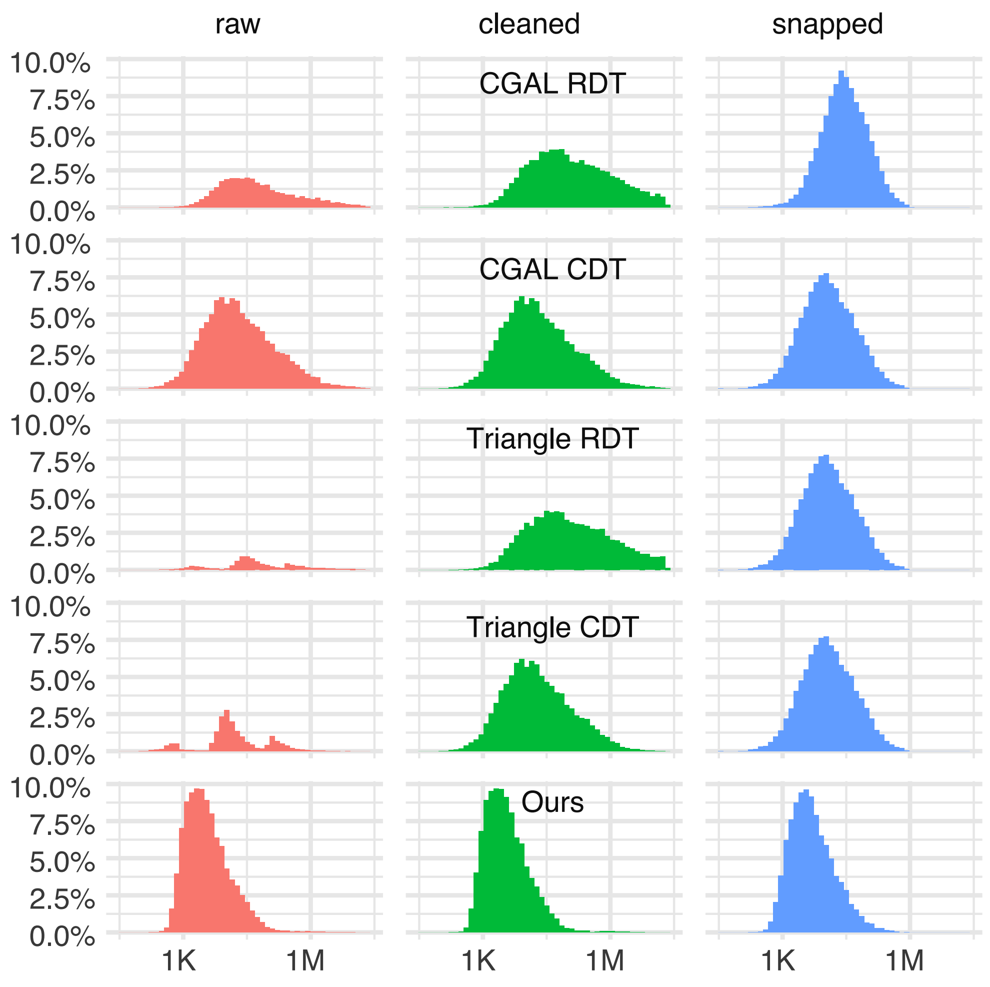


Figure 4.20: Comparison of the number of triangles generated by each method on a log scale.

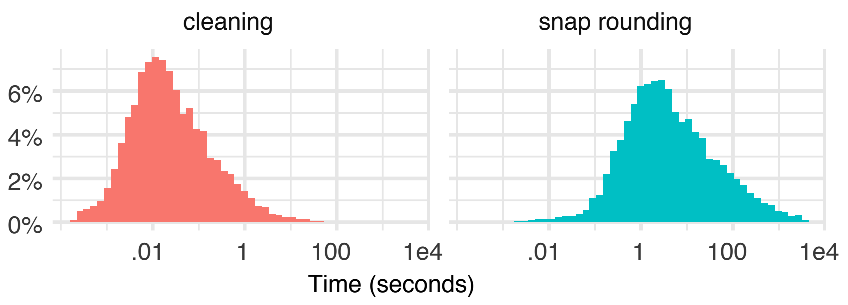


Figure 4.21: The running time of two preprocessing algorithms we used. Note that snap rounding is much more expensive than triangulation in general.

Table 4.1: Total Number of Failed Results for All Methods.

	Triangle		CGAL		Ours*
	CDT	RDT	CDT	RDT	
raw	16010 (67)	17806 (1)	3743 (3585)	12945 (1)	13 (0)
cleaned	645 (24)	3329 (0)	524 (457)	3560 (0)	83 (0)
snapped	5 (0)	5 (0)	5 (0)	38 (0)	5 (0)

Note: The failure refers to no output or output with inverted triangles. Numbers in parenthesis represent the number of output triangulations that contains inversions. *While our algorithm occasionally timed out due to limited computing resources, we have validated that it can always succeed with a larger epsilon and 64G memory.

by all methods on all three sets of inputs. Although slower, our algorithm along with CGAL's CDT are the only methods not requiring input preprocessing to be robust. Note that the preprocessing step can be very expensive (Figure 4.21) and will likely dominate the running time when paired with a triangle meshing algorithm that requires clean input. Note that all timing plots are using logarithmic scale for x-axis.

Correctness and Quality Table 4.1 lists the number of invalid triangulations generated by each method, i.e. triangulation containing one or more inverted triangles. We use exact predicates to check for triangle inversions [Shewchuk 1997]. Our algorithm generates inversion-free triangulation for all inputs, while both Triangle and CGAL produce invalid outputs occasionally. Figure 4.22 illustrates one of the many triangulation quality measures [Shewchuk 2002c]: min edge to max edge ratio. Our algorithm produces more well-shaped triangles than CDT, but slightly worse that RDT. The number (Figure 4.20) and quality (Figure 4.22) of the triangles generated by our method is mostly independent from the preprocessing done to the input, suggesting that our algorithm is stable to small perturbation in the input. This is not the case for other methods, which exhibit major differences in both number of elements and quality.

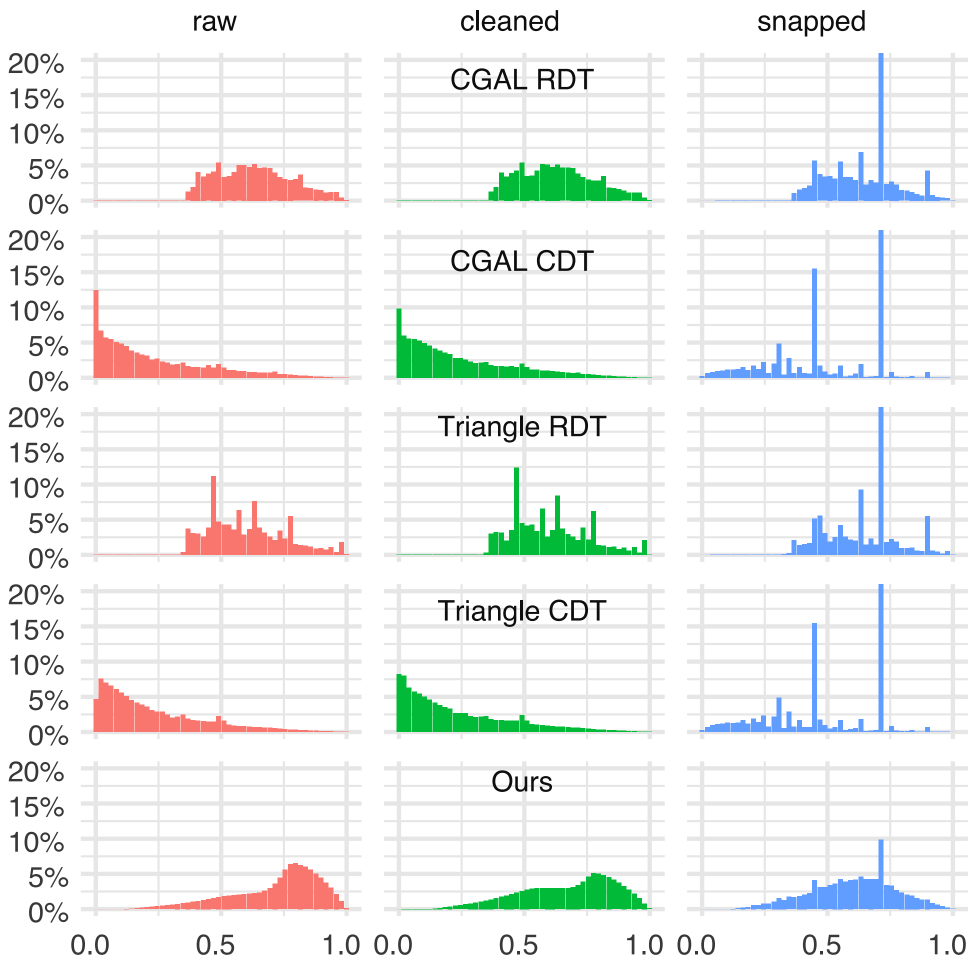


Figure 4.22: Edge ratio distribution on 1,000 randomly sampled outputs for each method.

4.6 Limitations and Concluding Remarks

We introduce an algorithm to create curved triangular meshes preserving features from a large collection of real-world SVG drawings. We demonstrated that our algorithm supports many applications in computer graphics, and compares favorably against existing triangle meshers, producing coarser meshes due to its native ability to filter out small scale features that are smaller than a user-controlled epsilon.

The main limitation of our algorithm is that it does not guarantee to exactly preserve intersections between curves. While our current algorithm produces visually pleasing results in these cases, we would like to explore the use of an exact Bezier arrangement [Wein et al. 2018] of the input features, to address this issue. A second interesting direction for future work is the creation of meshes with geometric maps of arbitrary degree or with rational Bezier edges, able to exactly reproduce circular arcs. While extending our algorithm will require minor modifications, their use in FEM is unclear, since there are no standard elements that reproduce them. Compared to other linear triangle meshers, our algorithm is around 10 times slower. We believe that parallelization of the mesh optimization stage would provide a noticeable performance boost.

We expect our contribution and our reference implementation to have a large impact in computer graphics and mechanical engineering, by considerably lowering the efforts required to build curved (and linear) meshes and use them for FEM simulations.

5 | Conclusion

This dissertation explains the reason why we regard the triangular and tetrahedral meshing problem as a black-box problem: Meshing is a fundamental problem in many fields, including computer graphics, mechanical engineering, and scientific computing, and meshers should be able to process real-world boundary representations of objects. However, the boundary representation acquired from modeling or scanning procedure often has artifacts, like self-intersection, small gaps, non-manifoldness, and etc., which are not handled by previous methods.

We propose envelope-based triangulation and tetrahedralization algorithms, WildMeshing, that deal with both 2D and 3D problems: Triangulating the area of a 2D shape and tetrahedralizing the volume of a 3D shape. Despite being in different dimensions, these two problems share many similarities. So we formulate three core principles for both WildMeshing algorithms: (1) Robustness: test and verify the algorithm on large public datasets; (2) Complete automation: no complex parameter tuning and the default parameter should work on any input; And (3) high-quality output: the algorithm should generate high-quality meshes approximating the input surface within a user-defined distance threshold. We purposely make no assumption about the input and work on imperfect input in the wild.

Compared with existing state-of-the-art methods, we formulate the problem in a different way: Instead of trying to preserve the input geometry exactly, we approximate the geometry with a controlled approximation error, which is more practically useful since most of the real-world geometries are anyhow imperfect and thus exact preservation is of unclear utility.

Our linear tetrahedral meshing algorithm, TetWild and fTetWild, share a similar goal but have different guarantees: TetWild has the guarantee to preserve all the input faces in the user-defined envelope but does not guarantee to produce a floating-point output mesh, while fTetWild guarantees to produce a floating-point output mesh, but does not guarantee to preserve all the input faces in the user-defined envelope. fTetWild has the added advantage of combining the robustness of TetWild with a running time comparable to Delaunay-based methods.

The different guarantees and performance make the algorithms complementary: TetWild can be extended to solve problems that require exactly preserving input geometry since it can generate an initial tetrahedral mesh that preserves the input exactly in rational coordinates. fTetWild foregoes this feature to improve running time and avoid both the higher memory usage linked to rational coordinates and the implementation complexity of having to handle polyhedral meshes, and it is thus easier to extend as it requires only basic data structures for simplicial meshes.

Since both TetWild and fTetWild handle sharp features in a soft way: they are present in the output, but their vertices could be displaced, causing a smooth surface to be uneven within the surface tolerance envelope. In this scenario, we introduce a curved meshing algorithm, TriWild, to create curved 2D planar triangular meshes preserving input feature curves. We demonstrate that our output meshes support many applications in computer graphics, and compare favorably against existing triangle meshers due to the fact that TriWild is able to generate coarser meshes with input curved geometry preserved. Extending TriWild to 3D is a major challenge and an avenue for future work.

WildMeshing is a radically different approach to geometric computing: we change the problem statement to at the same time achieve reasonable running times and strong guarantees on the output, and we approach verification of correctness in an experimental way, complementing formal guarantees with experimental evidence of success on a large collection of real-world 3D models. We provide a new way of verifying algorithm experimentally that uses a large-scale

Table 5.1: Guarantees and limitations of WildMeshing algorithms

Algorithm	Guarantees	Limitations
TetWild	1. Always terminate. 2. Represent all input triangles in the envelope.	1. Output might be in rational coordinates. 2. Have no bound on quality. 3. Produce piece-wise linear approximation of the input.
fTetWild	1. Always terminate. 2. Output in floating-point.	1. Output might not approximate all the input triangles in the envelope. 2. Have no bound on quality. 3. Produce piece-wise linear approximation of the input.
TriWild	1. Always terminate. 2. Represent all input triangles in the envelope.	1. Output might be in rational coordinates. 2. Have no bound on quality. 3. Produce piece-wise linear approximation of the input that satisfies a set of assumptions.

public real-world dataset on the scale of ten thousand instead of hundreds as the benchmark to test the robustness of a research algorithm. The large collection of clean models generated by WildMeshing can benefit the other research works that need clean meshes. This shifts the paradigm of algorithm verification in the graphics community.

5.1 Summary

In table 5.1 we summarize the guarantees and limitations of WildMeshing algorithms.

By analyzing the pros and cons of the two variants of TetWild, we identified 8 open questions, which we would like to explore in the future:

1. Is it possible to guarantee a floating-point output in the TetWild algorithm? This seems to hold experimentally, but we lack formal proof.

2. Is it possible to guarantee the insertion of all the input triangles in fTetWild? This also holds experimentally, but we lack formal proof.

3. Can we extend TriWild to 3D tetrahedralization and preserve the input surface using a

high-order approximation in the output tetrahedral mesh?

4. Although we have experimental evidence that the WildMeshing algorithms produce consistently high-quality outputs, we do not have a formal guarantee. Is it possible to provide a formal bound on quality for the current algorithms? If not can we develop a similar algorithm with a quality bound?

5. Our current algorithm output meshes while preserving the input boundary within a distance tolerance but does not preserve the topology of the input, which is especially useful for multi-material models (such as MRI images). Is it possible to extend the algorithms to preserve the input topology?

6. Our current algorithm does not produce a mapping between input and the boundary of the output, which is often required in Finite Element Analysis applications to transfer boundary conditions. Is it possible to change the surface tracking algorithm to produce a bijective mapping between the input surface and the boundary of the output?

7. Our current envelope check uses a conservative approximation of the L-infinity distance due to efficiency and robustness considerations. Can an exact check be developed for L-infinity or L-p distance?

8. We currently generate the whole mesh at once and do not have local remeshing. Can we extend the WildMeshing idea to remeshing applications where the input changes between iterations?

5.2 Future Work

Based on the WildMeshing algorithms, one natural extension of TriWild is to automatically and reliably convert CAD models composed of a collection of smooth primitives (feature points, elementary curves and patches, NURBS curves and patches) into a tetrahedral mesh suitable for finite element analysis and a corresponding triangle mesh suitable for processing using standard geometry processing, animation, and rendering algorithms. We would like to make minimal assumptions on the input geometry, tolerating gaps, self-intersections, and inconsistent orientations in the input primitives and provide, whenever possible, a map to transfer properties from the original CAD primitives to the output simplicial mesh.

The challenges in meshing CAD models come from the modeling procedure where the input patches are intersected with each other and then trimmed to form a certain shape desired by the designer. The trim on a patch is a curve in 2D parametric space that approximates the intersection curve of the patch with other patches in 3D physical space. Thus, the trims of the same intersection on different patches may not align. In the meantime, the computation of the intersection could be numerically unstable in floating-point and could be ambiguous using different CAD geometric kernels. It could also introduce high algebraic degree curves that are impractical and thus soon approximated by fixed degree curves during the CAD modeling procedure.

To avoid the notorious trimming, we propose an idea for converting CAD models into meshes: we independently convert each untrimmed patch into a high-quality triangle mesh and then introduce an algorithm, based on the TetWild tetrahedral meshing algorithm [Hu et al. 2018], to fill the entire space. After all the patches are represented in the tetrahedral mesh, we filter out the tetrahedra outside the surface and optionally recover only the surface of the CAD using the feature curves provided in the CAD.

We have implemented a preliminary algorithm for the idea that take as input a collection of points and bijective parametric curves and patches (Figure 5.1(1)) and output a tetrahedral mesh of the bounding box of the input with the input preserved. The part of an input primitive far from the others by a distance tolerance μ is preserved by a set of linear elements (edges and faces) with their vertices mapped to the primitive and the distance from the primitive smaller than ε_μ. The other part is approximated with the distance from the primitive smaller than ε. We first gener-

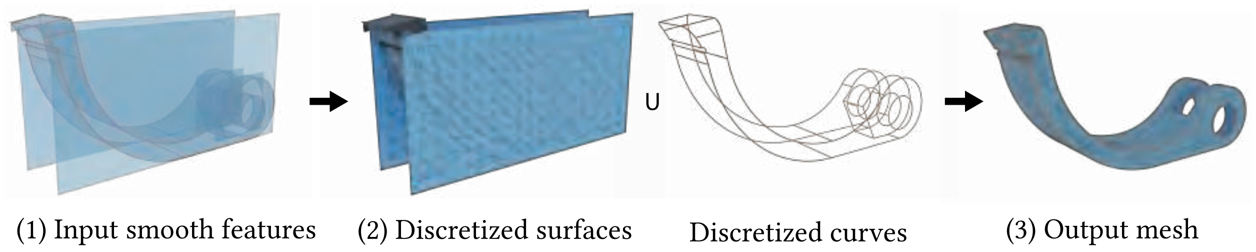


Figure 5.1: Pipeline of the algorithm. (1) The input features, where feature patches are in blue and feature curves are in orange. (2) The discretization of surfaces (blue) and the discretization of curves (orange). (3) The output tetrahedral mesh with the input feature preserved.

ate a discretization of the input, discretizing the curves into a set of segments and triangulating the patches (Figure 5.1(2)). We then tetrahedralize the domain while keeping track of the correspondence of the linear elements (vertices, edges, faces) and the input primitives. Optionally, we extract the interior volume of the shape from the tetrahedralized box (Figure 5.1(3)).

There are three main problems left to solve. (1) Our current algorithm does not guarantee to handle non-bijective input, e.g. input with self-intersection or singularities. We would like to design a method to handle non-bijective input correctly since it is quite common in the real world but hard to fix. (2) Our current algorithm estimates without error control the surface deviation from the output to the input. We plan to compute exactly or estimate with a controlled error the surface deviation. (3) Our current algorithm relies on the trimmed patches to extract the interior volume. We try to avoid trimming the patches and figure out a new way to extract the interior.

We aim to break barriers among different geometric computing kernels, allowing us to reliably convert them into triangle meshes, tetrahedral meshes, or point clouds used in many downstream applications without any user intervention. We believe it will also be an important tool for automating CAD design and simulation.

Bibliography

Abaqus (2018). Abaqus.

Abgrall, R., Dobrzynski, C., and Froehly, A. (2012). A method for computing curved 2D and 3D meshes via the linear elasticity analogy: preliminary results. Research Report RR-8061, INRIA.

Abgrall, R., Dobrzynski, C., and Froehly, A. (2014). A method for computing curved meshes via the linear elasticity analogy, application to fluid dynamics problems. *International Journal for Numerical Methods in Fluids*, 76(4):246–266.

Alauzet, F. and Marcum, D. L. (2013). A closed advancing-layer method with changing topology mesh movement for viscous mesh generation. In *22nd International Meshing Roundtable, IMR 2013, October 13-16, 2013, Orlando, FL, USA*.

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

Alliez, P., Cohen-Steiner, D., Yvinec, M., and Desbrun, M. (2005a). Variational tetrahedral meshing. In *ACM Transactions on Graphics (TOG)*, volume 24. ACM.

Alliez, P., Cohen-Steiner, D., Yvinec, M., and Desbrun, M. (2005b). Variational tetrahedral meshing. *ACM Trans. Graph.*, 24(3):617–625.

Ansys (2018). Ansys.

Attene, M. (2010a). A lightweight approach to repairing digitized polygon meshes. *Vis. Comput.*, 26(11):1393–1406.

Attene, M. (2010b). A lightweight approach to repairing digitized polygon meshes. *The Visual Computer*, 26(11):1393–1406.

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

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

Attene, M., Campen, M., and Kobbelt, L. (2013). Polygon mesh repairing: An application perspective. *ACM Comput. Surv.*, 45(2):15:1–15:33.

Attene, M., Giorgi, D., Ferri, M., and Falcidieno, B. (2009). On converting sets of tetrahedra to combinatorial and pl manifolds. *Computer Aided Geometric Design*, 26(8):850–864.

Aurenhammer, F. (1991). Voronoi diagrams&mdash;a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405.

Aurenhammer, F., Klein, R., and Lee, D.-T. (2013). *Voronoi Diagrams and Delaunay Triangulations.* WORLD SCIENTIFIC, River Edge, NJ, USA.

Babuška, I. and Guo, B. (1992). The h, p and h-p version of the finite element method; basis theory and applications. *Advances in Engineering Software*, 15(3):159–174.

Babuška, I. and Guo, B. Q. (1988). The h-p version of the finite element method for domains with curved boundaries. *SIAM Journal on Numerical Analysis*, 25(4):837–861.

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

Bargteil, A. W. and Cohen, E. (2014). Animation of deformable bodies with quadratic bézier finite elements. *ACM Trans. Graph.*, 33(3):27:1–27:10.

Barill, G., Dickson, N., Schmidt, R., Levin, D. I., and Jacobson, A. (2018). Fast winding numbers for soups and clouds. *ACM Transactions on Graphics*, 37(4):43:1–43:12.

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

Bartoň, M., Hanniel, I., Elber, G., and Kim, M.-S. (2010). Precise hausdorff distance computation between polygonal meshes. *Computer Aided Geometric Design*, 27(8):580–591. Advances in Applied Geometry.

Bassi, F. and Rebay, S. (1997). High-order accurate discontinuous finite element solution of the 2d euler equations. *Journal of Computational Physics*, 138(2):251–285.

Beall, M. W., Walsh, J., and Shephard, M. S. (2003). Accessing cad geometry for mesh generation. In *Proceedings of the 12th International Meshing Roundtable*, pages 33–42.

Bern, M., Eppstein, D., and Gilbert, J. (1994). Provably good mesh generation. *Journal of Computer and System Sciences*, 48(3):384–409.

Bernstein, G. (2013). Cork boolean library . https://github.com/gilbo/cork.

Bernstein, G. and Fussell, D. (2009). Fast, exact, linear booleans. pages 1269–1278, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Bertrand, F., Muñzenmaier, S., and Starke, G. (2014a). First-order system least squares on curved boundaries: Higher-order Raviart–Thomas elements. 52(6):3165–3180.

Bertrand, F., Muñzenmaier, S., and Starke, G. (2014b). First-order system least squares on curved boundaries: Lowest-order Raviart–Thomas elements. 52(2):880–894.

Bieri, H. and Nef, W. (1988). Elementary set operations with d-dimensional polyhedra. In *Proc. IWCGA*, pages 97–112, Berlin, Heidelberg. Springer-Verlag.

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

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

Boissonnat, J.-D. and Oudot, S. (2005). Provably good sampling and meshing of surfaces. *Graphical Models*, 67(5):405–451.

Bommes, D., Lévy, B., Pietroni, N., Puppo, E., Silva, C., Tarini, M., and Zorin, D. (2012). State of the art in quad meshing. In *Eurographics STARS*.

Botsch, M. and Kobbelt, L. (2004). A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '04, pages 185–192, New York, NY, USA. ACM.

Boyé, S., Barla, P., and Guennebaud, G. (2012). A vectorial solver for free-form vector gradients. *ACM Trans. Graph.*, 31(6):173:1–173:9.

Braess, D. (2007). *Finite Elements*. Cambridge University Press, third edition. Cambridge Books Online.

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

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

Brönnimann, H., Fabri, A., Giezeman, G.-J., Hert, S., Hoffmann, M., Kettner, L., Pion, S., and Schirra, S. (2017). 2D and 3D linear geometry kernel. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.11 edition.

Bronson, J. R., Levine, J. A., and Whitaker, R. T. (2012). Lattice cleaving: Conforming tetrahedral meshes of multimaterial domains with bounded quality. In *Proceedings of the 21st International Meshing Roundtable, IMR 2012, October 7-10, 2012, San Jose, CA, USA*, pages 191–209.

Bruno, O. P. and Pohlman, M. M. (2003). High order surface representation. *Topics in Computational Wave Propagation, Direct and Inverse Problems.*

Busaryev, O., Dey, T. K., and Levine, J. A. (2009). Repairing and meshing imperfect shapes with delaunay refinement. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, pages 25–33, New York, NY, USA. ACM.

Campen, M. and Kobbelt, L. (2010). Exact and robust (self-)intersections for polygonal meshes. 29(2):397–406.

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

Canann, S. A., Stephenson, M. B., and Blacker, T. (1993). Optismoothing: An optimization-driven approach to mesh smoothing. *Finite Elements in Analysis and Design*, 13(2):185–190.

Cantwell, C., Moxey, D., Comerford, A., Bolis, A., Rocco, G., Mengaldo, G., Grazia, D. D., Yakovlev, S., Lombard, J.-E., Ekelschot, D., Jordi, B., Xu, H., Mohamied, Y., Eskilsson, C., Nelson, B., Vos, P., Biotto, C., Kirby, R., and Sherwin, S. (2015). Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219.

Cardoze, D., Cunha, A., Miller, G. L., Phillips, T., and Walkington, N. (2004). A bézier-based approach to unstructured moving meshes. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 310–319, New York, NY, USA. ACM.

Carey, G. F. (1997). *Computational grids: generations, adaptation & solution strategies.* CRC Press.

Chen, L. and Xu, J.-c. (2004). Optimal delaunay triangulations. *Journal of Computational Mathematics*, pages 299–308.

Chen, X., Peng, D., and Gao, S. (2012). Svm-based topological optimization of tetrahedral meshes. In *Proceedings of the 21st International Meshing Roundtable, IMR 2012, October 7-10, 2012, San Jose, CA, USA*, pages 211–224.

Cheng, S.-W., Dey, T. K., Edelsbrunner, H., Facello, M. A., and Teng, S.-H. (2000). Silver exudation. *Journal of the ACM (JACM)*, 47(5):883–904.

Cheng, S.-W., Dey, T. K., and Levine, J. A. (2008). A practical delaunay meshing algorithm for a large class of domains. In *Proceedings of the 16th International Meshing Roundtable*, pages 477–494. Springer.

Cheng, S.-W., Dey, T. K., and Shewchuk, J. (2012a). *Delaunay mesh generation.* CRC Press.

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

Chew, L. P. (1989). Constrained delaunay triangulations. *Algorithmica*, 4.

Chew, L. P. (1993). Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the ninth annual symposium on Computational geometry*, pages 274–280. ACM.

Ciarlet, P. G. and Raviart, P.-A. (1972). Interpolation theory over curved elements, with applications to finite element methods. 1(2):217–249.

Cohen-Steiner, D., De Verdiere, E. C., and Yvinec, M. (2002). Conforming delaunay triangulations in 3d. In *Proceedings of the eighteenth annual symposium on Computational geometry*, pages 199–208. ACM.

Cuillière, J., François, V., and Drouet, J. (2012). Automatic 3d mesh generation of multiple domains for topology optimization methods. In *Proceedings of the 21st International Meshing Roundtable, IMR 2012, October 7-10, 2012, San Jose, CA, USA*, pages 243–259.

Dey, S., O'Bara, R. M., and Shephard, M. S. (1999). Curvilinear mesh generation in 3d. In *IMR*, pages 407–417. John Wiley & Sons.

Dey, T. K. and Levine, J. A. (2008). Delpsc: a delaunay mesher for piecewise smooth complexes. In *Proceedings of the twenty-fourth annual symposium on Computational geometry*, pages 220–221. ACM.

Dobrzynski, C. and El Jannoun, G. (2017). High order mesh untangling for complex curved geometries. Research Report RR-9120, INRIA Bordeaux, équipe CARDAMOM.

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

Doran, C., Chang, A., and Bridson, R. (2013). Isosurface stuffing improved: Acute lattices and feature matching. In *ACM SIGGRAPH 2013 Talks*, SIGGRAPH '13, pages 38:1–38:1, New York, NY, USA. ACM.

Douze, M., Franco, J.-S., and Raffin, B. (2015). QuickCSG: Arbitrary and faster boolean combinations of n solids. Technical Report 01121419, Inria Research Centre Grenoble, Rhone-Alpes.

Du, Q. and Wang, D. (2003). Tetrahedral mesh generation and optimization based on centroidal voronoi tessellations. *International journal for numerical methods in engineering*, 56(9):1355–1373.

Dunyach, M., Vanderhaeghe, D., Barthe, L., and Botsch, M. (2013). Adaptive Remeshing for Real-Time Mesh Deformation. In Otaduy, M.-A. and Sorkine, O., editors, *Eurographics 2013 - Short Papers*. The Eurographics Association.

Engvall, L. and Evans, J. A. (2017). Isogeometric unstructured tetrahedral and mixed-element bernstein-bézier discretizations. *Computer Methods in Applied Mechanics and Engineering*, 319:83–123.

Engvall, L. and Evans, J. A. (2018). Mesh quality metrics for isogeometric bernstein-bézier discretizations. *arXiv:1810.06975*.

Engwirda, D. (2016). Conforming restricted delaunay mesh generation for piecewise smooth complexes. *CoRR*.

Eppstein, D. (2001). Global optimization of mesh quality. *Tutorial at the 10th Int. Meshing Roundtable*.

Faraj, N., Thiery, J.-M., and Boubekeur, T. (2016). Multi-material adaptive volume remesher. *Comput. Graph.*, 58(C):150–160.

Farin, G. (2002). The morgan kaufmann series in computer graphics and geometric modeling. In Farin, G., editor, *Curves and Surfaces for CAGD*. Morgan Kaufmann, San Francisco, fifth edition edition.

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

Frederick, C. O., Wong, Y. C., and Edge, F. W. (1970). Two-dimensional automatic mesh generation for structural analysis. *International Journal for Numerical Methods in Engineering*, 2(1):133–144.

Freitag, L. A. and Ollivier-Gooch, C. (1997). Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40(21).

Fu, X. M., Liu, Y., and Guo, B. (2015). Computing locally injective mappings by advanced mips. *ACM Trans. Graph.*, 34(4):71:1–71:12.

Gao, X., Jakob, W., Tarini, M., and Panozzo, D. (2017). Robust hex-dominant mesh generation using field-guided polyhedral agglomeration. *ACM Trans. Graph.*, 36(4):114:1–114:13.

Gargallo-Peiró, A., Roca, X., Peraire, J., and Sarrate, J. (2013). Defining quality measures for validation and generation of high-order tetrahedral meshes. In *Proceedings of the 22nd International Meshing Roundtable, IMR 2013, October 13-16, 2013, Orlando, FL, USA*, pages 109–126.

Gargallo Peiró, A., Roca Navarro, F. J., Peraire Guitart, J., and Sarrate Ramos, J. (2013). High-order mesh generation on cad geometries. In *Adaptive Modeling and Simulation 2013*, pages 301–312. Centre Internacional de Mètodes Numèrics en Enginyeria (CIMNE).

George, J. A. (1971). *Computer Implementation of the Finite Element Method*. PhD thesis, Stanford University, Stanford, CA, USA. AAI7205916.

George, P. and Borouchaki, H. (2012). Construction of tetrahedral meshes of degree two. *International Journal for Numerical Methods in Engineering*, 90(9):1156–1182.

George, P. L., Borouchaki, H., and Saltel, E. (2003). 'ultimate' robustness in meshing an arbitrary polyhedron. *International Journal for Numerical Methods in Engineering*, 58(7):1061–1089.

Geuzaine, C., Johnen, A., Lambrechts, J., Remacle, J.-F., and Toulorge, T. (2015). *The Generation of Valid Curvilinear Meshes*, pages 15–39. Springer International Publishing, Cham.

Geuzaine, C. and Remacle, J.-F. (2009). Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331.

Ghasemi, A., Taylor, L. K., and Newman, III, J. C. (2016). Massively parallel curved spectral/finite element mesh generation of industrial cad geometries in two and three dimensions. *Fluids Engineering Division Summer Meeting*, (50299).

Ghomi, A. T., Bolhassan, M., Nejur, A., and Akbarzadeh, M. (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.

Goodrich, M. T., Guibas, L. J., Hershberger, J., and Tanenbaum, P. J. (1997). Snap rounding line segments efficiently in two and three dimensions. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 284–293. ACM.

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

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

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

Hachenberger, P. and Kettner, L. (2019). 3D boolean operations on nef polyhedra. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition.

Haimes, R. (2015). MOSS: multiple orthogonal strand system. *Eng. Comput. (Lond.)*, 31(3):453–463.

Hu, K., Yan, D. M., Bommes, D., Alliez, P., and Benes, B. (2017). Error-bounded and feature preserving surface remeshing with minimal angle improvement. *IEEE Transactions on Visualization and Computer Graphics*, 23(12):2560–2573.

Hu, Y., Schneider, T., Gao, X., Zhou, Q., Jacobson, A., Zorin, D., and Panozzo, D. (2019). Triwild: Robust triangulation with curve constraints. *ACM Trans. Graph.*

Hu, Y., Schneider, T., Wang, B., Zorin, D., and Panozzo, D. (2020). Fast tetrahedral meshing in the wild. *ACM Trans. Graph.*, 39(4).

Hu, Y., Zhou, Q., Gao, X., Jacobson, A., Zorin, D., and Panozzo, D. (2018). Tetrahedral meshing in the wild. *ACM Trans. Graph.*, 37(4):60:1–60:14.

Hughes, T. J., Cottrell, J. A., and Bazilevs, Y. (2005). Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. 194(39):4135–4195.

Jacobson, A., Kavan, L., and Sorkine-Hornung, O. (2013). Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH)*, 32(4):33:1–33:12.

Jacobson, A., Panozzo, D., et al. (2016). libigl: A simple C++ geometry processing library. http://libigl.github.io/libigl/.

Jakob, W., Tarini, M., Panozzo, D., and Sorkine-Hornung, O. (2015). Instant field-aligned meshes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA)*, 34(6).

Jamin, C., Alliez, P., Yvinec, M., and Boissonnat, J.-D. (2015). Cgalmesh: a generic framework for delaunay mesh generation. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):23.

Johnen, A., Remacle, J. F., and Geuzaine, C. A. (2013). Geometrical validity of curvilinear finite elements. *Journal of Computational Physics*, 233:359–372.

Joshi, B. J. and Ourselin, S. (2003). Bsp-assisted constrained tetrahedralization. In *Proceedings of the 12th International Meshing Roundtable, IMR 2003, Santa Fe, New Mexico, USA, September 14-17, 2003*, pages 251–260.

Joshi, P. and Carr, N. A. (2008). Repoussé: Automatic inflation of 2d artwork. In *Proceedings of the Fifth Eurographics Conference on Sketch-Based Interfaces and Modeling*, SBM'08, pages 49–55, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Karman, S. L., Erwin, J. T., Glasby, R. S., and Stefanski, D. (2016). High-order mesh curving using wcn mesh optimization. In *46th AIAA Fluid Dynamics Conference, AIAA AVIATION Forum*.

Klingner, B. M. and Shewchuk, J. R. (2007). Agressive tetrahedral mesh improvement. In *Proceedings of the 16th International Meshing Roundtable*, pages 3–23, Seattle, Washington.

Knupp, P. M. (2000). Achieving finite element mesh quality via optimization of the jacobian matrix norm and associated quantities. part ii, a framework for volume mesh optimization and the condition number of the jacobian matrix. *International Journal for Numerical Methods in Engineering*, 48(8):1165–1185.

Kovalsky, S. Z., Galun, M., and Lipman, Y. (2016). Accelerated quadratic proxy for geometric optimization. *ACM Trans. Graph.*, 35(4):134:1–134:11.

Labelle, F. and Shewchuk, J. R. (2007a). Isosurface stuffing: fast tetrahedral meshes with good dihedral angles. In *ACM Transactions on Graphics (TOG)*, volume 26, page 57. ACM.

Labelle, F. and Shewchuk, J. R. (2007b). Isosurface stuffing: Fast tetrahedral meshes with good dihedral angles. In *ACM SIGGRAPH 2007 papers on - SIGGRAPH '07*, page 57, New York, NY, USA. ACM Press.

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

Lipman, Y. (2012). Bounded distortion mapping spaces for triangular meshes. *ACM Trans. Graph.*, 31(4):108.

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169.

Lu, Q., Shephard, M. S., Tendulkar, S., and Beall, M. W. (2013). Parallel curved mesh adaptation for large scale high-order finite element simulations. In Jiao, X. and Weill, J.-C., editors, *Proceedings* *of the 21st International Meshing Roundtable*, pages 419–436, Berlin, Heidelberg. Springer Berlin Heidelberg.

Luo, X., Shephard, M. S., and Remacle, J.-F. (2001). The influence of geometric approximation on the accuracy of high order methods. *Rensselaer SCOREC report*, 1.

Luo, X., Shephard, M. S., Remacle, J.-F., O'Bara, R. M., Beall, M. W., Szabó, B. A., and Actis, R. (2002). p-version mesh generation issues. In *IMR*.

MacNeal, R. H. (1949). *The solution of partial differential equations by means of electrical networks.* PhD thesis, CalTech.

Magalhães, S. V., Franklin, W. R., and Andrade, M. V. (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*, page 44, New York, NY, USA. ACM, ACM.

Mandad, M., Cohen-Steiner, D., and Alliez, P. (2015a). Isotopic approximation within a tolerance volume. *ACM Trans. Graph.*, 34(4):64:1–64:12.

Mandad, M., Cohen-Steiner, D., and Alliez, P. (2015b). Isotopic approximation within a tolerance volume. *ACM Trans. Graph.*, 34(4):64:1–64:12.

Masoud, A. (2016). *3D Graphical Statics Using Reciprocal Polyhedral Diagrams.* PhD thesis, ETH Zruich, Stefano Franscini Platz 5, Zurich, CH, 8093.

MATLAB Partial Differential Equation Toolbox (2018). Matlab partial differential equation toolbox. The MathWorks, Natick, MA, USA.

Mezger, J., Thomaszewski, B., Pabst, S., and Straśer, W. (2009). Interactive physically-based shape editing. *Computer Aided Geometric Design*, 26(6):680–694. Solid and Physical Modeling 2008.

Misztal, M. K. and Bærentzen, J. A. (2012). Topology-adaptive interface tracking using the deformable simplicial complex. *ACM Trans. Graph.*, 31(3):24:1–24:12.

Molino, N., Bridson, R., and Fedkiw, R. (2003). Tetrahedral mesh generation for deformable bodies. In *Proc. Symposium on Computer Animation*.

Monk, P. (1987). A mixed finite element method for the biharmonic equation. *SIAM Journal on Numerical Analysis*, 24(4):737–749.

Moxey, D., Ekelschot, D., Keskin, Ü., Sherwin, S., and Peiró, J. (2016). High-order curvilinear meshing using a thermo-elastic analogy. *Computer-Aided Design*, 72:130–139. 23rd International Meshing Roundtable Special Issue: Advances in Mesh Generation.

Murphy, M., Mount, D. M., and Gable, C. W. (2001). A point-placement strategy for conforming delaunay tetrahedralization. *International Journal of Computational Geometry & Applications*, 11(06):669–682.

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

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

Oden, J. (1994). Optimal h-p finite element methods. *Computer Methods in Applied Mechanics and Engineering*, 112(1):309–331.

Orzan, A., Bousseau, A., Winnemöller, H., Barla, P., Thollot, J., and Salesin, D. (2008). Diffusion curves: A vector representation for smooth-shaded images. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)*, volume 27.

Owen, S. J. (1998). A survey of unstructured mesh generation technology. In *IMR*.

Panagiotakopoulou, M., Bergert, M., Taubenberger, A., Guck, J., Poulikakos, D., and Ferrari, A. (2016). A nanoprinted model of interstitial cancer migration reveals a link between cell deformability and proliferation. *ACS Nano*, 10(7):6437–6448. PMID: 27268411.

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

Pavic, D., Campen, M., and Kobbelt, L. (2010). Hybrid Booleans. 29:75–87.

Peiró, J., Sherwin, S. J., and Giordana, S. (2008). Automatic reconstruction of a patient-specific high-order surface representation and its application to mesh generation for cfd calculations. *Medical & Biological Engineering & Computing*, 46(11):1069–1083.

Peraire, J., Vahdati, M., Morgan, K., and Zienkiewicz, O. C. (1987). Adaptive remeshing for compressible flow computations. *J. Comput. Phys.*, 72(2):449–466.

Persson, P.-O. and Peraire, J. (2009). Curved mesh generation and mesh refinement using lagrangian solid mechanics. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*.

Poya, R., Sevilla, R., and Gil, A. J. (2016). A unified approach for a posteriori high-order curved mesh generation using solid mechanics. *Computational Mechanics*, 58(3):457–490.

Rabinovich, M., Poranne, R., Panozzo, D., and Sorkine-Hornung, O. (2017). Scalable locally injective mappings. *ACM Trans. Graph.*, 36(2):16.

Remacle, J. (2017). A two-level multithreaded delaunay kernel. *Computer-Aided Design*, 85:2–9.

Roca, X., Gargallo-Peiró, A., and Sarrate, J. (2012). Defining quality measures for high-order planar triangles and curved mesh generation. In Quadros, W. R., editor, *Proceedings of the 20th International Meshing Roundtable*, pages 365–383, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ruiz-Gironés, E., Gargallo-Peiró, A., Sarrate, J., and Roca, X. (2017). An augmented lagrangian formulation to impose boundary conditions for distortion based mesh moving and curving. *Procedia Engineering*, 203:362–374. 26th International Meshing Roundtable, IMR26, 18-21 September 2017, Barcelona, Spain.

Ruiz-Gironés, E., Roca, X., and Sarrate, J. (2016a). High-order mesh curving by distortion minimization with boundary nodes free to slide on a 3d cad representation. *Computer-Aided Design*, 72:52–64. 23rd International Meshing Roundtable Special Issue: Advances in Mesh Generation.

Ruiz-Gironés, E., Sarrate, J., and Roca, X. (2016b). Generation of curved high-order meshes with optimal quality and geometric accuracy. *Procedia Engineering*, 163:315–327. 25th International Meshing Roundtable.

Ruppert, J. (1995). A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of algorithms*, 18(3):548–585.

Sadek, E. A. (1980). A scheme for the automatic generation of triangular finite elements. *International Journal for Numerical Methods in Engineering*, 15(12):1813–1822.

Samet, H. (2005). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

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

Schönhardt, E. (1928). Über die zerlegung von dreieckspolyedern in tetraeder. *Mathematische Annalen*, 98(1):309–312.

Schweiger, M. and Arridge, S. (2016). Basis mapping methods for forward and inverse problems: Basis mapping methods. *International Journal for Numerical Methods in Engineering*, 109.

Scott, L. R. (1973). *Finite element techniques for curved boundaries.* PhD thesis, Massachusetts Institute of Technology.

Scott, R. (1975). Interpolated boundary conditions in the finite element method. 12(3):404–427.

Sederberg, T. and Nishita, T. (1990). Curve intersection using bézier clipping. *Computer-Aided Design*, 22(9):538–549.

Sederberg, T. W., Zheng, J., Bakenov, A., and Nasri, A. (2003). T-splines and t-nurccs. *ACM Trans. Graph.*, 22(3):477–484.

Sevilla, R., Fernández-Méndez, S., and Huerta, A. (2011). Nurbs-enhanced finite element method (nefem). 18(4):441–484.

Sheehy, D. R. (2012). New bounds on the size of optimal meshes. *Comput. Graph. Forum*, 31(5).

Shen, C., O'Brien, J. F., and Shewchuk, J. R. (2004). Interpolating and approximating implicit surfaces from polygon soup. In *Proceedings of ACM SIGGRAPH 2004*, pages 896–904. ACM Press.

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

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

Shephard, M. S., Flaherty, J. E., Jansen, K. E., Li, X., Luo, X., Chevaugeon, N., Remacle, J.-F., Beall, M. W., and O'Bara, R. M. (2005). Adaptive mesh generation for curved domains. *Applied Numerical Mathematics*, 52(2):251–271. ADAPT '03: Conference on Adaptive Methods for Partial Differential Equations and Large-Scale Computation.

Sherwin, S. and Peiró, J. (2002). Mesh generation in curvilinear domains using high-order elements. *International Journal for Numerical Methods in Engineering*, 53(1):207–223.

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

Shewchuk, J. R. (1996). Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In Lin, M. C. and Manocha, D., editors, *Applied Computational Geometry Towards Geometric Engineering*, pages 203–222, Berlin, Heidelberg. Springer Berlin Heidelberg.

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

Shewchuk, J. R. (1998). Tetrahedral mesh generation by delaunay refinement. In *Proceedings of the fourteenth annual symposium on Computational geometry*, pages 86–95. ACM.

Shewchuk, J. R. (1999). Lecture notes on delaunay mesh generation.

Shewchuk, J. R. (2002a). Constrained delaunay tetrahedralizations and provably good boundary recovery. In *IMR*, pages 193–204.

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

Shewchuk, J. R. (2002c). What is a good linear element? interpolation, conditioning, and quality measures. In *Proceedings of the 11th International Meshing Roundtable, IMR 2002, Ithaca, New York, USA, September 15-18, 2002*, pages 115–126.

Shewchuk, J. R. (2002d). What is a good linear finite element? - interpolation, conditioning, anisotropy, and quality measures. Technical report, In Proc. of the 11th International Meshing Roundtable.

Si, H. (2015a). Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2):11:1–11:36.

Si, H. (2015b). Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2):11:1–11:36.

Si, H. and Gärtner, K. (2005). Meshing piecewise linear complexes by constrained delaunay tetrahedralizations. In *IMR*, pages 147–163. Springer.

Si, H. and Shewchuk, J. R. (2014). Incrementally constructing and updating constrained delaunay tetrahedralizations with finite-precision coordinates. *Eng. Comput. (Lond.)*, 30(2):253–269.

Stees, M. and Shontz, S. M. (2017). A high-order log barrier-based mesh generation and warping method. *Procedia Engineering*, 203:180–192. 26th International Meshing Roundtable, IMR26, 18-21 September 2017, Barcelona, Spain.

Stenberg, R. (1984). Analysis of mixed finite element methods for the stokes problem: A unified approach. *Mathematics of Computation*, 42(165):9–23.

Sýkora, D., Kavan, L., Čadík, M., Jamriška, O., Jacobson, A., Whited, B., Simmons, M., and Sorkine-Hornung, O. (2014). Ink-and-ray: Bas-relief meshes for adding global illumination effects to hand-drawn characters. *ACM Trans. Graph.*, 33(2):16:1–16:15.

Takayama, K., Jacobson, A., Kavan, L., and Sorkine-Hornung, O. (2014). A simple method for correcting facet orientations in polygon meshes based on ray casting. *Journal of Computer Graphics Techniques*, 3(4):53–63.

Tang, M., Lee, M., and Kim, Y. J. (2009). Interactive hausdorff distance computation for general polygonal models. *ACM Trans. Graph.*, 28(3):74:1–74:9.

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

Toulorge, T., Geuzaine, C., Remacle, J.-F., and Lambrechts, J. (2013). Robust untangling of curvilinear meshes. *Journal of Computational Physics*, 254:8–26.

Toulorge, T., Lambrechts, J., and Remacle, J.-F. (2016). Optimizing the geometrical accuracy of curvilinear meshes. *Journal of Computational Physics*, 310:361–380.

Tournois, J., Wormser, C., Alliez, P., and Desbrun, M. (2009). Interleaving delaunay refinement and optimization for practical isotropic tetrahedron mesh generation. *ACM Transactions on Graphics*, 28(3):Art–No.

Turner, M., Peiró, J., and Moxey, D. (2018). Curvilinear mesh generation using a variational framework. *Computer-Aided Design*, 103:73–91. 25th International Meshing Roundtable Special Issue: Advances in Mesh Generation.

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

Wang, B., Schneider, T., Hu, Y., Attene, M., and Panozzo, D. (2020). Exact and efficient polyhedral envelope containment check. *ACM Trans. Graph.*, 39(4).

Wang, C. C. L. (2011). Approximate boolean operations on large polyhedral solids with partial mesh reconstruction. 17(6):836–849.

Weatherill, N. P. and Hassan, O. (1994). Efficient three-dimensional delaunay triangulation with automatic point creation and imposed boundary constraints. *International Journal for Numerical Methods in Engineering*, 37(12):2005–2039.

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

Xue, D., Demkowicz, L., et al. (2005). Control of geometry induced error in hp finite element (fe) simulations. i. evaluation of fe error for curvilinear geometries. *Int. J. Numer. Anal. Model*, 2(3):283–300.

Yerry, M. A. and Shephard, M. S. (1983). A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics and Applications*, 3(1):39–46.

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

Zhou, Q., Grinspun, E., Zorin, D., and Jacobson, A. (2016). Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)*, 35(4):39.

Zhou, Q. and Jacobson, A. (2016a). Thingi10K: A dataset of 10, 000 3D-printing models. *CoRR*, abs/1605.04797.

Zhou, Q. and Jacobson, A. (2016b). Thingi10k: A dataset of 10,000 3d-printing models, `https://ten-thousand-models.appspot.com`. Technical report, New York University.

Ziel, V., Bériot, H., Atak, O., and Gabard, G. (2017). Comparison of 2d boundary curving methods with modal shape functions and a piecewise linear target mesh. *Procedia Engineering*, 203:91–101. 26th International Meshing Roundtable, IMR26, 18-21 September 2017, Barcelona, Spain.

Zulian, P., Schneider, T., Kai, H., and Rolf, K. (2017). Parametric finite elements with bijective mappings. *BIT Numerical Mathematics*, pages 1–19.