Declarative Specification for Unstructured Mesh Editing Algorithms

ZHONGSHI JIANG*, New York University, USA and Meta, USA
JIACHENG DAI, New York University, USA
YIXIN HU, Pixel Labs, Tencent America, USA
YUNFAN ZHOU, New York University, USA
JEREMIE DUMAS and QINGNAN ZHOU, Adobe Research, USA
GURKIRAT SINGH BAJWA, DENIS ZORIN, and DANIELE PANOZZO, New York University, USA
TESEO SCHNEIDER, University of Victoria, Canada

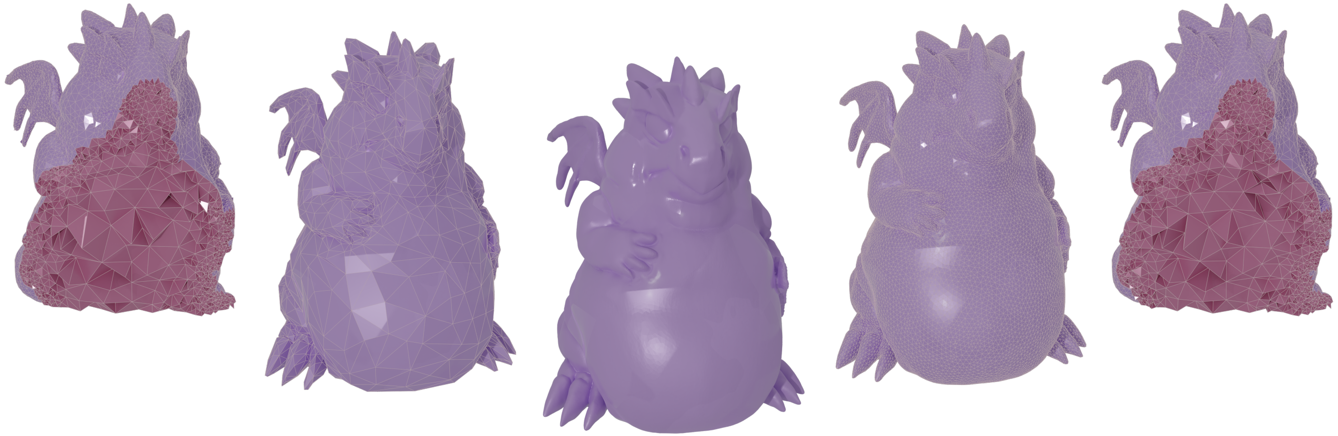

Fig. 1. Example of four different mesh editing algorithms implemented with our library. With our framework, users can implement different flavor of mesh editing with built in robustness, and readily available parallelism. From left to right: harmonic triangulation, QSlim, the input, isotropic remeshing, and robust tetrahedral mesh generation.

We introduce a novel approach to describe mesh generation, mesh adaptation, and geometric modeling algorithms relying on changing mesh connectivity using a high-level abstraction. The main motivation is to enable easy customization and development of these algorithms via a declarative specification consisting of a set of per-element invariants, operation scheduling, and attribute transfer for each editing operation.

We demonstrate that widely used algorithms editing surfaces and volumes can be compactly expressed with our abstraction, and their implementation within our framework is simple, automatically parallelizable on shared-memory architectures, and with guaranteed satisfaction of the prescribed invariants. These algorithms are readable and easy to customize for specific use cases.

We introduce a software library implementing this abstraction and providing automatic shared memory parallelization.

CCS Concepts: • **Mathematics of computing** → **Mesh generation**; • **Computing methodologies** → *Shape modeling*.

ACM Reference Format:
Zhongshi Jiang, Jiacheng Dai, Yixin Hu, YunFan Zhou, Jeremie Dumas, Qingnan Zhou, Gurkirat Singh Bajwa, Denis Zorin, Daniele Panozzo, and Teseo Schneider. 2022. Declarative Specification for Unstructured Mesh Editing Algorithms. *ACM Trans. Graph.* 41, 6, Article 251 (December 2022), 14 pages. https://doi.org/10.1145/3550454.3555513

*This work is primarily done while ZJ is at NYU

Authors' Address: ZJ, JD, YZ, GSB, DZ, DP, Courant Institute of Mathematical Sciences, New York University, 60 5th Avenue, New York, NY 10011, USA. YH,.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0730-0301/2022/12-ART251 $15.00
https://doi.org/10.1145/3550454.3555513

1 INTRODUCTION

Unstructured triangular and tetrahedral meshes are widely used in graphics, engineering, and scientific computing due to their flexibility to represent objects with complex boundaries. Such unstructured meshes find their usage in modeling and rendering 3D objects and scenes, discretizing partial differential equations for physical simulation, collisions detection and response, path planning in robotics, and many other applications.

An unstructured mesh is usually stored in a custom data-structure supporting a set of local operations to add, remove, or change its elements and their properties. A major research effort has been invested in exploring different data-structures and evaluating their generality and efficiency (Section 2), which led to the development of mesh

libraries such as CGAL [The CGAL Project 2020], VCG/meshlab [Cignoni et al. 2021], OpenMesh [Botsch et al. 2002], libigl [Jacobson et al. 2016], PMP [Sieger and Botsch 2019], and OpenVolumeMesh [Kremer et al. 2013]. Commonly, mesh-editing algorithms (i.e., algorithms based on operations changing the connectivity of a mesh) are tightly coupled with a data-structure and its API, and porting an algorithm from one library to another is a major engineering effort. Code relying on local operations is also inherently error prone, as it usually involves keeping track of properties attached to mesh elements as the mesh itself changes due to the local operations. Parallelizing code using a mesh data structure is also challenging, due to race conditions when multiple threads attempt to change the same region of the mesh.

At a high-level, it is common practice to describe a mesh editing algorithm as a sequence of topological and geometrical editing operations. We argue that this approach is unnecessarily low-level, as it exposes the algorithm designer to technical problems that can be handled automatically by changing the abstraction level. It also makes it challenging to use or customize mesh editing algorithms in larger projects (such as their use in physical simulation for adaptive refinement), as low-level data structure details percolate in the entire code-base. A particularly difficult challenge in these algorithms is to ensure that a set of conditions (such as manifoldness, being free from self-intersections, minimal quality, maximal geometrical approximation) hold after each operation is applied. This is usually tackled by simulating each operation for the purpose of checking these conditions, an error-prone process that needs to be carefully designed for each pair of operation and condition. The additional presence of attributes attached to vertices, edges, or faces further complicates these problems.

We propose a different way to describe mesh-editing algorithms on simplicial manifold meshes, using a declarative specification instead of a more traditional procedural approach. Instead of focusing on what the algorithm does, we ask the user to specify what are the requirements that the desired mesh should have. We divide these requirements into two groups: invariants and desiderata. The former is a description of hard requirements on the mesh (for example, no inverted elements or no self-intersections) and the latter is a set desirable properties (such as good quality). A mesh editing algorithm is then described as: (1) a set of per-element invariants (for example, all elements should have correct orientation), (2) a measure for the desiderata (for example, element quality), (3) a set of application-specific attributes attached to mesh elements (e.g., vertex coordinates), and how they are affected by local operations, and (4) a schedule of operation types. We show that many existing algorithms for mesh generation, remeshing, and parametrization, can be concisely expressed in this form (Section 4), which we denote IDAS (Invariant-Desiderata-Attributes-Schedule).

The IDAS specification has been designed with four goals:

(1) **Modularity:** The connectivity of the mesh is abstracted from the user, which can only navigate the mesh using a high-level abstraction based on a cell tuple [Brisson 1989]. This reduces the learning curve for new users, as they only need to learn a navigation API to implement algorithms in IDAS. It will also allow IDAS programs to benefit from continuous progress in data-structure design, as the data structure will be swappable without requiring downstream code changes in the high level IDAS code. This is in stark contrast with existing mesh libraries, which tend to be very invasive in the user code relying on them due to the close connection between navigation, mesh editing, and property management.
(2) **Usability:** The user code handles, at all times, a valid mesh: the library simulates each operation transparently allowing the user to navigate on a valid mesh before and after every operation, dramatically simplifying the logic required to define invariants and desiderata. Properties on the mesh are also similarly abstracted, allowing to attach attributes on every simplex independently on the data structure used for implementing the specification.
(3) **Efficiency:** The specification purposely requires only definitions of properties on individual elements. This feature allows runtimes for IDAS program to parallelize the computation (Section 4) without requiring special attention from a user. We demonstrate that automatic parallelization of mesh editing algorithms is possible on multi-core architectures.
(4) **Robustness:** The IDAS specification moves the majority of the robustness issues typical of meshing algorithm on the runtime used to execute a IDAS program instead of the IDAS code itself. This simplifies the development of robust algorithms: for example, the user invariants are guaranteed to be enforced during processing, as the runtime will automatically check them on every modified element. As long as the user provides correct code for the invariant (for example to check for area positivity of an element using a predicate), then the runtime ensures that the invariants will be satisfied for all elements.

Given an algorithm in IDAS form, we design an algorithm and runtime library to realize it, with guarantees on satisfying the invariant and a best effort to maximize the desiderata. Our library exploits shared memory parallelism without any additional effort required from users in the algorithm specification.

To demonstrate the generality and effectiveness of our approach, we provide IDAS formulations for five popular mesh editing algorithms (Section 4, Figure 1): (1) shortest edge collapse [Hoppe 1996] (decimation for triangle meshes), (2) QSlim [Garland and Heckbert 1997], (3) isotropic triangle meshing [Botsch and Kobbelt 2004] (remeshing for triangle meshes), (4) harmonic triangulations [Alexa 2019] (quality improvement for 3D volumes), and (5) robust tetrahedral mesh generation [Hu et al. 2019b] (conversion of surface meshes to volumetric meshes). The IDAS formulation closely resemble the textual description of the algorithms in the corresponding papers: it is compact, readable, and easy to adapt for requirements of specific applications. As an example, we show that modifying (1) and (2) to guarantee a maximal geometric error is straightforward. Despite its generality, IDAS implementations executed using our library are comparable or faster than state of the art implementations in open-source software: the overhead due to the framework generality is more than compensated by the automatic parallelization (Section 4).

We believe our contribution is an important step to allow researchers and practitioners to effectively develop new mesh-editing algorithms, shielding the designer of mesh editing algorithms from many of the robustness and correctness challenges plaguing previous low-level approaches, by moving these components inside the runtime environment. It will also allow mesh editing algorithms to be used more easily in larger systems, as they can be tailored to requirements of a specific application with minimal programming effort.

We provide an open-source implementation [1] of our library and of the five mesh editing algorithms as additional material.

2 RELATED WORK

2.1 Mesh Data Structures

Efficient data structures for representing solid geometry have been an intriguing research topic since the early days of computer graphics [Requicha 1980]. As a result, there is a large variety of mesh data structure designs, where they are each optimized for different usage scenarios. Index-array-based mesh data structure encodes each element as a list of vertex indices on its boundary. It is simple and memory efficient, but neighborhood query and local operations are not directly supported. Graph-based mesh data structures, including half-edge [Mäntylä 1987], winged-edge [Baumgart 1972], quad-edge [Guibas and Stolfi 1985], cell-tuple [Brisson 1989], etc., view meshes as graphs, where each element contains links to its adjacent elements. This design allows for efficient local query and update, making it ideal for algorithms like mesh simplification [Garland and Heckbert 1997]. Linear-algebra-based mesh data structures, such as [DiCarlo et al. 2014; Mahmoud et al. 2021; Zayer et al. 2017], encode adjacency information as sparse matrices. This design elegantly reduces neighborhood query and local operations to sparse matrix computations, which are highly optimized for modern parallel computing architecture. Closely related, is the concept of generalized combinatorial maps [Dufourd 1991; Lienhardt 1994], and the CGoGN library [Kraemer et al. 2014] provide an efficient implementation which includes parallel traversal of the mesh. By design, mesh data structures provide a low level interface to manipulate vertices, edges, faces, and tetrahedra. Different designs differ vastly in API and implementation details, making it hard to port algorithm from one data structure to another. In contrast, our framework decouples mesh data structure choice from algorithm specification, providing the flexibility of switching the underlying data structure in a seamless manner.

2.2 Domain specific languages in graphics

Our abstraction model of mesh processing algorithms draw inspiration from domain specific languages (DSL) in graphics. For dense regularly structured data such as images, Halide [Ragan-Kelley et al. 2013] popularized the idea of decoupling image processing operations from low level scheduling tasks. Similar abstraction that separates algorithm description from low level data structure and/or parallel architecture can also be found in other DSLs such as Simit [Kjolstad et al. 2016] for simulation over triangle meshes, Taco [Kjolstad et al. 2017] for dense and sparse tensor algebra, Taichi [Hu et al. 2019a] for simulation over sparse volumetric data, and Penrose [Ye et al. 2020] for generating diagrams from math notation.

[1] https://github.com/wildmeshing/wildmeshing-toolkit

2.3 Parallel Meshing

To meet the demand of generating large meshes, a number of popular mesh generation algorithms have been redesigned to leverage modern parallel computing hardware, both in a shared memory and distributed memory setting. Typically a divide-and-conquer strategy is adopted where a mesh is partitioned to run local processing operations on each subdomain in parallel. There are two key challenges involved: (1) how to handle operations involving elements shared by multiple partitions; (2) how to ensure load stay balanced across different processors as the mesh evolves.

One way to mitigate both challenges is to ensure mesh is partitioned into similar sized patches with high area to boundary ratio. A large number of partitioning strategies are available, including clustering-based approaches [Mahmoud et al. 2021], spacial-hierarchy-based approach [Lo 2012; Loseille et al. 2017], space-filling-curve-based approach [Borrell et al. 2018; Marot et al. 2019], and general purpose graph partitioning [Karypis and Kumar 1998]. Many variations of space-filling curves have also been used to construct mesh partitions [Aluru and Sevilgen 1997; Chrisochoides 2006]. To handle potential conflicts that may arise at partition boundaries, various synchronization strategies have been proposed [Chrisochoides 2006; Chrisochoides and Nave 2003; Okusanya and Peraire 1996] to minimize the amount of communication.

After generating the submeshes, some methods allow each compute node to work on them independently without synchronization. Once all threads are done, the meshes are merged [Blelloch et al. 1999; Chen 2010; Cignoni et al. 1993; Funke and Sanders 2017]. However these methods require complicated merge steps since the tetrahedra in the intermediate boundaries may not align. There are some techniques that compromise the Delaunay condition in some cases, so that the merging operation can be simpler [Lachat et al. 2014]. To avoid the tricky merge operations, other parallel strategies maintain a single complete Delaunay tetrahedralization and use synchronization techniques to avoid race conditions when working on a partition boundary [Chrisochoides and Nave 2003; Okusanya and Peraire 1997]. The parallel constrained Delaunay meshing algorithm [Chew et al. 1997] cleverly defines the boundary and edge constrains to reduce the variable and unpredictable communication patterns. Some other techniques use locks for handling conflicts and data races [Batista et al. 2010; Blandford et al. 2006; Foteinos and Chrisochoides 2011].

Another set of methods use recursive divide-and-conquer techniques for parallel implementation on shared memory machines [Marot and Remacle 2020]. All threads independently work on the internal parts of the mesh and skip the operations at the boundary. After this phase, processing of only the boundary elements becomes the new problem. This technique is then recursively used until all the mesh elements are processed. A similar set of techniques use clever space-filling curves for re-partitioning the mesh boundaries after each recursive phase [Aluru and Sevilgen 1997; Chrisochoides 2006].

Since the submesh boundaries are the main areas of concern, some methods entirely avoid any operations on these boundaries while ensuring the correctness of the result [Galtier and George 1996; Linardakis and Chrisochoides 2006]. These methods precompute the domain separators such that their facets are Delaunay admissible. This completely eliminates synchronization overheads, but only applies for Delaunay meshing.

Another conflict handling strategy is to simply reject the offending operations and try executing them later with a new domain partitioning [Marot et al. 2019]. This reject-and-repartition strategy may not guarantee algorithm termination, thus special care is needed to handle this case.

As the domain mesh evolves, keeping load balanced across processors becomes critical. Typically, this is done by periodically repartitioning the updated mesh. Zhou et al. [2012] proposes a predictive load balancing method to keep partitions balanced. Marot et al. [2019] uses simple rescaling of the space-filling curve to repartition the domain.

In this work, we are targeting only shared-memory parallelism, thus making the problem of reducing communications between processors less relevant. We use a graph-based space partitioning technique [Karypis and Kumar 1998] due to its simplicity and availability as open-source code (METIS), but we use it only to reduce the risk of conflicts. To avoid conflicts, we use a shared memory locking mechanism. This approach is only possible for shared-memory parallelism but has the major advantage of not requiring rebalancing and to respect, to a certain degree, the execution order prescribed by the user-code. This approach is possible thanks to the availability of efficient parallel atomic instructions, and parallel libraries based on them (oneTBB).

2.4 Scope of Mesh Editing

Mesh Generation. Tetrahedral meshing algorithms heavily rely on mesh editing operations. The most common approaches are Delaunay methods [Alliez et al. 2005a; Bishop 2016; Boissonnat et al. 2002; Boissonnat and Oudot 2005; Busaryev et al. 2009; Chen and Xu 2004; Cheng et al. 2008, 2012; Chew 1989; Cohen-Steiner et al. 2002; Dey and Levine 2008; Du and Wang 2003; Jamin et al. 2015; Murphy et al. 2001; Remacle 2017; Ruppert 1995; Shewchuk 1996, 1998, 2002; Si 2015; Si and Gartner 2005; Si and Shewchuk 2014; Tournois et al. 2009], which strive to generate meshes satisfying the Delaunay condition, grid methods [Bern et al. 1994; Bridson and Doran 2014; Bronson et al. 2013; Doran et al. 2013; Labelle and Shewchuk 2007; Molino et al. 2003; Yerry and Shephard 1983], which start from a regular lattice or with a hierarchical space partitioning and optionally intersect the background mesh with the input surface, and front-advancing methods [Alauzet and Marcum 2014; Cuilliere et al. 2013; Haimes 2014; Sadek 1980], which insert one element at a time, growing the volumetric mesh (i.e. marching in space), until the entire volume is filled .

These algorithms rely on local operations on mesh data-structures, and benefit from our framework to simplify the implementation and gain automatic parallelization. We discuss an implementation of one the more recent algorithms [Hu et al. 2019b, 2018] in Section 4. Note that some of these algorithms use local operation that are not implemented yet (such as 5-6 swap), but they could be added to our framework.

Constrained Meshing. Downstream applications often require meshes to satisfy either quality (avoidance of zero volume elements) or geometric (distance to the input surface) constraints. For example, Mandad et al. [2015] creates a surface approximation within a tolerance volume, the TetWild algorithms [Hu et al. 2019b, 2018] use an envelope [Wang et al. 2021] to restricts the geometry of the boundary of the tetrahedral mesh, [Brochu et al. 2012] adds constraints to local remeshing to avoid interpenetrations in simulations, and [Gumhold et al. 2003] extends mesh simplification [Garland and Heckbert 1999; Popović and Hoppe 1997] to ensure a non self-intersecting result.

These criteria are explicitly modeled as invariants in our framework, and they can be easily swapped in and out existing implementations, as we demonstrate in Section 4.

Mesh Improvement. Mesh improvements modifies an existing mesh by changing its connectivity and position of the vertices to improve the quality of its elements [A. Freitag and Ollivier-Gooch 1998; Alexa 2019; Alliez et al. 2005b; Canann et al. 1996, 1993; Chen and Xu 2004; Feng et al. 2018; Hu et al. 2018; Klingner and Shewchuk 2007; Lipman 2012]. We show in Section 4 a reimplementation of [Alexa 2019] in IDAS form.

Dynamic Remeshing and Adaptive Mesh Refinement (AMR). Simulations involving large deformations are common in computer graphics, and if the surface or volume deformed is represented by a mesh, it is inevitable that after a large deformation the quality of the elements will deteriorate, and the mesh will have to be updated. Additionally, it is often required to concentrate more elements in regions of interest whose location is changing during the simulation, for example to capture a fold in a cloth simulation, or a fracture in a brittle material. These two challenges are tackled in elastoplastic and viscoplastic simulations [Bargteil et al. 2007; Hutchinson et al. 1996; Wicke et al. 2010; Wojtan and Turk 2008], in fluid simulations [Ando et al. 2013; Chentanez et al. 2007; Clausen et al. 2013; Klingner et al. 2006; Misztal and Bærentzen 2012], in cloth simulation [Bender and Deul 2013; Li and Volkov 2005; Narain et al. 2013, 2012; Pfaff et al. 2014; Simnett et al. 2009; Villard and Borouchaki 2002], and fracture simulation [Busaryev et al. 2013]. All these algorithms could benefit from our contribution, to simplify their implementation and obtaining speedup due to the automatic parallelization offered by our approach.

A different approach is discussed in [Grinspun et al. 2002], where the refinement is performed on the basis to avoid the difficulties with explicit remeshing. However, this approach cannot coarsen a dense input, and also cannot increase the quality of elements, making it usable only for specific scenarios [Grinspun et al. 2002]. Our approach aims at lowering the barrier for integrating explicit remeshing algorithms in simulation applications, thus allowing to directly use standard simulation methods on adaptive meshes without having to pay the high implementation cost for the mesh generation.

When remeshing is paired with algorithms simulating contacts that do not tolerate interpenetrations (for example [Li et al. 2020]),

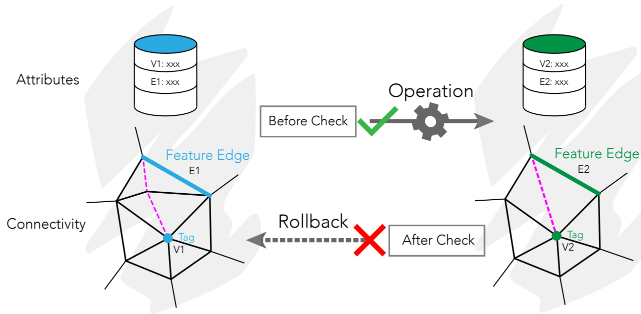

Fig. 2. Overview of the components behind our specification. A mesh is represented trough its topology, implemented by our library, and a list of user provided attributes. Before an operation is attempted, we explicitly perform a pre-check, and, if successful, we generate a mesh (attributes and topology). At this point, we trigger the after check to validate the operation (e.g., check if the newly generate mesh has positive volume). In case the after check fails, we *automatically* rollback the operation and restore the mesh to its previous *valid* state.

it is necessary to ensure that adaptive remeshing does not break this invariant. This can be achieved adding non-penetration constraints to each local mesh editing operations, as proposed in [Brochu et al. 2012]. Our framework is ideal for developing such methods, as additional constraints can be added to existing mesh editing algorithms with minimal modifications, as we demonstrate in Section 4.

Parametrization. Conformal mesh parametrization algorithms adapt the mesh during optimization, as a a fixed triangulation restricts the space of metrics realizable [Campen et al. 2019; Campen and Zorin 2017a,b; Gu et al. 2018a,b; Luo 2004; Springborn 2020; Sun et al. 2015]. Two very recent works [Campen et al. 2021; Gillespie et al. 2021] introduce robust algorithms based on Ptolemy flips to compute conformal maps satisfying a prescribed metric.

All these methods require changing the mesh connectivity of a triangle mesh, and could thus benefit from our framework to simplify their implementation and parallelize the mesh editing operations.

Mesh Arrangements/Boolean Operations. Boolean operations are basic algorithms often used in geometry processing applications. Recently, [Zhou et al. 2016] proposed a robust way to compute them by constructing a space arrangement, and then filtering the result using the generalized winding number [Jacobson et al. 2013]. A similar approach, using an approximated meshing algorithm, has been extended in [Hu et al. 2019b], using a tetrahedral mesher to create the initial arrangement. The reimplementation of TetWild introduced in this paper (Section 4) can be extended for a similar purpose.

3 METHOD

Our declarative specification is designed to remove the burden of low-level management of the mesh connectivity and attributes, allowing an algorithm designer to focus only on high-level requirements. The design consists of five components (Figure 2).

3.1 Mesh editing components

Operation Rollback. It is common to perform mesh editing to improve a given energy functional, such as mesh quality or element size. However, due to the discrete and combinatorial nature of the operations, it is not possible to use standard smooth optimization techniques to reduce the energy (e.g., Newton's method or line-search to ensure that the energy decreases). Instead, in such settings, the energy can only be evaluated before and after every operation to measure the operation's effect. This paradigm is commonly implemented using an ad-hoc energy evaluation that "simulates" the operation only for the purpose of measuring the energy change. This simulation is complex (especially in 3D), and error-prone, as not only the connectivity changes, but the energy likely depends on properties attached to mesh elements, which needs to be updated accordingly.

We propose instead to make this process opaque to the user, providing to the user-code an explicit copy of the mesh (and up to date attributes) *before and after* the operation is performed to allow an easy and reliable energy evaluation. The correctness and efficiency of this process is handled by the runtime. This reduces the complexity of mesh editing considerably in our experience, as it makes them more similar to traditional finite difference approaches where the energy is evaluated on different points on the domain to approximate its derivative.

Explicit Invariants. It is common to have a set of desiderata on the mesh that needs to be satisfied, such as avoiding triangle insertions or self-intersections. Given the complexity of a mesh editing algorithm it is difficult to ensure that they are satisfied, as these conditions needs to be checked after every operation is applied (and they often depend on attributes too, such as vertex positions).

We propose to make these invariants explicit, and delegate to the library the task of ensuring that they are checked after every mesh modifications, and after the input is loaded. In this way, not only the code is simpler, but it is much easier to ensure correctness, as the checks are handled transparently by the library.

Explicit Attribute Update. Mesh attributes are usually handled by low-level meshing libraries, allowing to attach them to the desired mesh element (vertex, edge, face, triangle, or tetrahedra). However, the handling of attributes after a local operation is performed is usually a responsibility of the user code, as it is dependent on the application.

We propose to make this process more explicit, requiring the user to provide the rules on how to update attributes after operation in a high-level specifications, and delegating the actual update to the library. This makes the specification more direct and less error-prone, and allows users to write algorithms without having to know the low-level details on how the local mesh operations work.

Parallel Scheduling. The type and scheduling of local operations is crucial in mesh editing algorithms. It usually involves maintaining a priority queue of operations, which is updated after every local operation.

We provide a direct way of controlling the operations performed and how the queue is updated. In the library, we can then distribute the work automatically on multiple threads, hiding from the

user code the complexity of performing mesh editing operations in parallel and ensuring that race conditions are avoided.

Abstract Mesh Navigation. Both invariant and attribute updates require navigating a mesh. Instead of relying on data-structure specific navigation, we favor the use of the cell tuple abstraction [Brisson 1989]. This allows the specification to be independent of the mesh data structure used in the library. The `Tuple` stores four indices (three for surface meshes), vertex, edge, face, and tetrahedron and provides a single function per index, called `switch`, to change one index while keeping the other indices fixed. For instance `switch_vertex` changes the vertex index while keeping edge, face, and tetrahedron fixed which has the effect of selecting the opposite vertex on an edge.

3.2 Declarative Specification

Our API provides two abstractions: a `TetMesh` (and `TriMesh` class for 2D) (Algorithm 1), and a `Scheduler` (Algorithm 2).

Mesh Classes. Both the `TetMesh` and `TriMesh` classes provide the basic local operations (e.g., edge split or collapse) and, for each operation, their corresponding *before* and *after* methods. The mesh class is responsible of implementing the operations changing the topology, and the application code must *only* override the before and after methods to update attributes. The before method has a view of the mesh before the operation, and can thus navigate it to cache local attributes, while the after method has a view after the operation is performed, and it is responsible for updating attributes. In the simple case of regular subdivision of a triangle mesh, the `split_before` caches the coordinates of the two edge endpoints, and the `split_after` computes the position of the newly inserted vertex by averaging them.

In addition, the mesh class provides a method, which can be overridden by the user code, that *automatically* verifies user-provided invariants (e.g., maintain positive elements' volume). All user-provided methods return a Boolean status to notify the mesh classes if the operation fails; in case it does, our API rolls back the operation and restores the topology to the previous valid state. As the connectivity and attributes management is handled by the class, this ensures that, in case of failure of the operation, the mesh will go back to a valid state.

Our API provides the standard local operations: edge collapsing, edge/face swapping, edge splitting, and smoothing. We also provide an additional, non standard, operation: triangle insertion. This operation is an enhanced version of splitting where multiple edges, faces, and tetrahedra are subdivided to represent an input triangle provided as input. This operation is useful to compute mesh arrangements, and it is also used in meshing algorithms [Hu et al. 2019b].

Since the `TetMesh` class only handles topology, the operation requires the list of edges and tetrahedron the input triangle intersects. Internally it subdivides all of them, and generates a valid tetrahedral mesh using the connectivity table in [Hu et al. 2019b]. The before operation provides the user the list of faces that will be changed by the operation, allowing the user code to explore the mesh and

Algorithm 1 API of our `TetMesh` class.

```
class TetMesh
{
public:

    bool split_edge(const Tuple& t,
        std::vector<Tuple>& new_tets);
    bool collapse_edge(const Tuple& t,
        std::vector<Tuple>& new_tets);
    bool swap_edge(const Tuple& t,
        std::vector<Tuple>& new_tets);
    bool swap_face(const Tuple& t,
        std::vector<Tuple>& new_tets);
    bool smooth_vertex(const Tuple& t);

    bool insert_triangle(
        const std::vector<Tuple>& intersected_tets,
        const std::vector<Tuple>& intersected_edges);

protected:

    bool invariants(const std::vector<Tuple>& tets);

    bool split_before(const Tuple& t);
    bool split_after(const Tuple& t);

    bool collapse_before(const Tuple& t);
    bool collapse_after(const Tuple& t);

    bool swap_edge_before(const Tuple& t);
    bool swap_edge_after(const Tuple& t);

    bool swap_face_before(const Tuple& t);
    bool swap_face_after(const Tuple& t);

    bool smooth_before(const Tuple& t);
    bool smooth_after(const Tuple& t);

    bool insert_triangle_before(
        const std::vector<Tuple>& faces);
    bool insert_triangle_after(
        const std::vector<Tuple>& faces,
        const std::vector<std::vector<Tuple>>& new_f);
};
```

Algorithm 2 API of our Scheduler.

```
template <class Mesh>
struct Scheduler
{
    function<double
        (const Mesh&, Op op, const Tuple&)>
        priority = ...;

    function<vector<pair<Op, Tuple>>
        (const Mesh&, Op, const vector<Tuple>&)>
        renew_neighbor_tuples = ...;

    function<vector<size_t>
        (Mesh&, const Tuple&)>
        lock_vertices = ...

    function<bool(const Mesh&)>
        stopping_criterion = ...;

    function<bool
        (const Mesh&, tuple<double, Op, Tuple>& t)>
        should_process = ...;

    size_t num_threads = ...;
    size_t max_retry_limit = ...;
    size_t stopping_criterion_checking_frequency = ...;

    bool operator()
        (Mesh& m,
        const vector<pair<Op, Tuple>>& ops);
};
```

cache attributes, while the after provides a mapping between the old faces and any newly inserted face in the mesh.

Scheduler. The second part of our API is the Scheduler that is responsible to control the order of the individual operations and then execute a list of operations. The main purpose of the scheduler is to abstract the operation order and hide parallelization details from the user. Our scheduler provides customizable callbacks, including,

- *Priority* to order the local mesh edit operations.
- *Renew neighbor tuples* that is invoked after a successful operation, to add newly created tuples and operations into the queue.
- *Lock vertices* that provides information on the affected region for the operations, and avoiding conflicts.
- *Stopping criterion* that is checked periodically to terminate the program if certain criterion is met. For example, number of vertices, or quality criterion.

3.3 Implementation.

We implement a runtime for our specification in C++, using Intel oneTBB for parallelization.

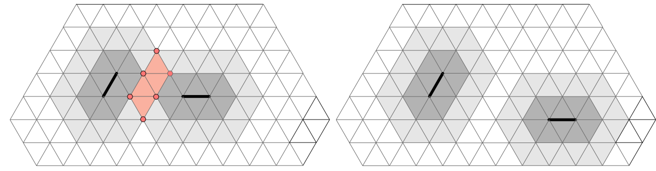

Fig. 3. Example of the locking region for two edges. In the example the operation requires locking a two-ring neighborhood (e.g., for the edge collapse operation). If the two edges are sufficiently far (right) both operations can be safely executed in parallel. When the two edges are close (left) the operations might fail acquiring the mutexes in the shared area.

Data Structure. We opt for an indexed data structure, where we explicitly represent the vertices and the simplex of higher dimension (triangle for 2D, tetrahedra for 3D). Each vertex explicitly stores a list of incident simplices, and each simplex stores a sorted list of its vertices. While not the most efficient option for navigation, this data structure makes the implementation of local operations much simpler.

Parallelization. To avoid conflicts between local operations working on the same part of the mesh, we introduce a synchronization mechanism using locks.

Each mesh vertex is associated with a mutex. Whenever a thread wants to access (read/write) any attribute stored in a vertex, edge, triangle, or tetrahedron it must first acquire a lock on *all* the vertices of the tetrahedron containing the element(s) storing the attribute (Figure 3). For example, if a thread wants to read a value on an edge of a 3D mesh, it first needs to acquire a lock on all vertices of all the tetrahedra containing that edge. This mechanism is used also for mesh navigation, and for updating the mesh connectivity.

At a first look, this mechanism might seem cumbersome and expensive. However, we rely on asynchronous, tentative lock acquisition operations. We *try* to acquire the lock, and give up and release all previously acquired locks if the lock is already taken by another thread. These operations are efficient on modern hardware and dramatically improve the performance, while avoiding deadlocks: the locking region is dependent on the local operation, and for all the operation we implement (edge split, collapse, flip, and vertex smoothing), the two-ring neighborhood is sufficient to prevent a deadlock. A downside is that an operation might be skipped due to impossibility of acquiring a mutex. These operations are retried for several times (by default 10 times) and run serially if the still do not succeed. Before performing any local operation, we try to acquire the lock on vertices in the 1-ring or 2-ring of the vertex involved in the operation. For example, a vertex smoothing operation requires acquiring the 1-ring vertex neighborhood of the smoothed vertex, while an edge-collapse operation on an edge (v_1, v_2) requires acquiring the lock on the 2-ring vertex neighborhood of both v_1 and v_2 (Figure 3).

Finally, since we partition the input mesh using Morton Encoding [Karras 2012], the amount of conflicts (and skipped operations) is low.

3.4 Example: Shortest Edge Collapse

We show how the library is used in a classical example, shortest edge collapse. In this case, we add a 3D position to every vertex as a vertex attribute (by default, there are no attributes attached to mesh elements). For every attribute and for every operations we plan to use in the scheduler, we need to provide a function that updates such attribute (Algorithm 3). In the `collapse_before` function, we cache the two vertex coordinates associated with the collapsed edge represented by `Tuple t`. In the `collapse_after` function, we generate a new vertex in the middle of the two endpoints of the collapsed edge.

Algorithm 3 Overridden methods in `TriMesh` sub-class to implement shortest edge collapse.

```
//Save two vertices attached to edge t
bool collapse_before(const Tuple& t)
{
    cache.v1p = verts[t.vid()];
    cache.v2p = verts[switch_vertex(t).vid()];
    return true;
}

//Generate a new point
bool collapse_after(const Tuple& t)
{
    verts[t.vid()] = (cache.v1p + cache.v2p) / 2.0;
    return true;
}
```

Equipped with the 3D position attribute, which at this point will be automatically kept up to date by the library, we can now schedule the collapse operation (Algorithm 4). For shortest edge collapse, we want to attempt to collapse all edges, prioritizing the shortest ones, until we reach a fixed number of collapses `n_collapse`: in the code we registering the operation type (`ops`), specify how to update the queue after an operation (`renew`) by adding all the neighbouring edges, and specifying the edge length as a priority (`priority`). Note that the outdated elements in the queue that are affected by a local operation are automatically invalidated using a tagging mechanism on the tuples which is opaque to the user.

4 APPLICATIONS

To showcase the generality and effectiveness of our approach, we implement five popular mesh editing algorithms in our framework, and compare them with reference implementations. Overall, the performance of our method are competitive for surface applications, but the overhead due to the approach generality is higher in 3D, leading to higher running time.

Shortest Edge Collapsing. The simplest algorithm for simplifying a triangle edge is shortest edge collapse [Hoppe 1996], which performs a series of collapse operations prioritizing the shorter edges. The algorithm requires only one local operation, edge collapse. A common criteria for termination is reaching a desired number of mesh elements. We compare our implementation with the "decimate" implementation in libigl [Jacobson et al. 2016]. The serial

Algorithm 4 Scheduler setup for the schedule shortest edge collapse.

```
//Collect edges attacched to tris
vector<Tuple> new_edges_after(const vector<Tuple>& tris)
{
    vector<Tuple> new_edges;
    for (auto t : tris) {
        for (auto j = 0; j < 3; j++) {
            new_edges.push_back(
                tuple_from_edge(t.fid(), j));
        }
    }
    return new_edges;
}

bool collapse_shortest(int n_collapses)
{
    //Register operations
    auto ops = vector<pair<Op, Tuple>>();
    for (auto& l : get_edges())
        ops.emplace_back("edge_collapse", l);

    //After a successful operation,
    //we append all new edges
    auto renew = [](auto& m, auto op, auto& tris) {
        auto edges = m.new_edges_after(tris);
        auto optup = vector<pair<Op, Tuple>>();
        for (auto& e : edges)
            optup.emplace_back("edge_collapse", e);
        return optup;
    };

    //priority in which we collapse
    auto priority = [](auto& m, auto op, const Tuple& e) {
        const auto v1 = m.verts[e.vid()];
        const auto v2 = m.verts[e.switch_vertex(m).vid()];
        auto len2 = (v1 - v2).squaredNorm();
        return -len2;
    };

    //Set the functions to the scheduler
    Scheduler executor;
    executor.renew_neighbor_tuples = renew;
    executor.priority = priority;
    executor.stopping_criterion_checking_frequency =
                n_collapses;
    //We stop only when we perform n_collapses
    executor.stopping_criterion =
                [](auto& m) { return true; };
    //Run the executor
    executor(*this, ops);
}
```

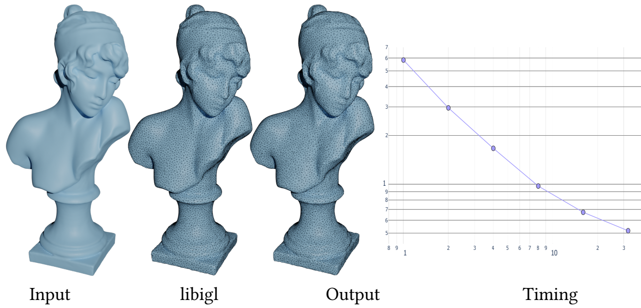

Fig. 4. Comparison of our parallel implementation (32 threads) of shortest edge collapse (scalability plot on the right, from 1 to 32 threads) of a model with 281, 724 faces with the serial version in libigl. Both libigl and our output have 28, 168 faces and comparable edge length (1.058 for libigl versus 1.061 for ours). Our serial method runs in 4.9s (5.84s on a single thread, 0.52s with 32 thread, leading to a speedup of 11×), while libigl runs in 2.74s.

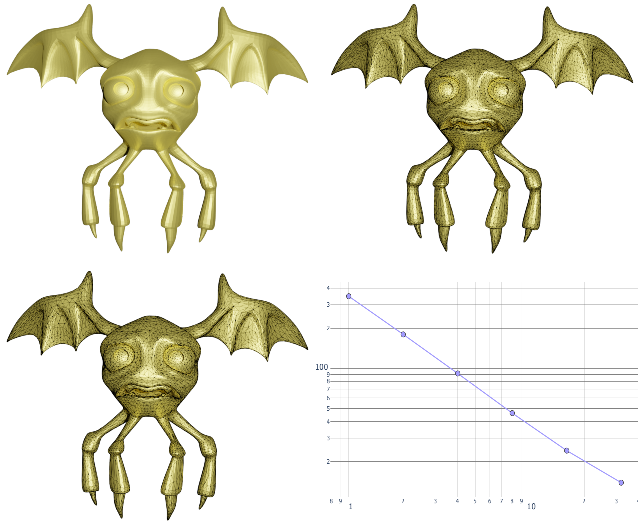

Fig. 5. Comparison of our parallel implementation of QSlim with the serial version in libigl for a model with 1, 909, 755 faces. Top right, the libigl output has 17, 891 faces and takes 41.26s. Bottom left, our output has 17, 906 faces and runs in 306.59s. Our implementation scales well: 347.59s with one thread and 13.88s with 32 (25× speedup).

libigl implementation is comparable when running the algorithms serially, and our parallel implementation is up to 9 times faster when using 32 threads (Figure 4).

QSlim. We use our framework to implement QSlim [Garland and Heckbert 1997]. QSlim collapse edges based on the planarity of the two adjacent faces measured with an error quadric. The algorithm continues to collapse until it reaches a target number of edges. We compare our implementation with the QSlim implementation in libigl [Jacobson et al. 2016]. The serial libigl implementation is 8 times faster than our implementation, due to their direct manipulations of elements in the queue with each collapse. But our parallel implementation is twice as fast when using 16 threads (Figure 5).

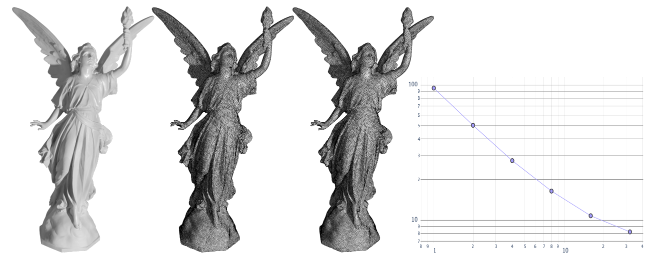

Fig. 6. Example of uniform remeshing a model with 2, 529, 744 triangles (left) with the same target edge length. Middle, [Möbius and Kobbelt 2010] remeshes it to 78, 322 faces in 31.96 seconds. On the right is our 32-thread implementation, which generates 71, 640 triangles in 8.2 seconds (78.34s serial, 95.0s for a single thread, leading to a speedup of 11×). The difference in density of the meshes is due to differences in the detail of the implementation, which makes the two methods reach an average vertex valence of 5.999, and a similar target edge length (differ 0.01% of the bounding box diagonal length) with a different element budget.

Isotropic Remeshing. We implemented the widely used algorithm for isotropic remeshing proposed in [Botsch and Kobbelt 2004]. This algorithm alternates edge collapse, edge flips, edge splits, and tangential smoothing to obtain a mesh that is isotropic (i.e. all elements have the same size) and where all triangles are close to equilateral. The process is guided by a user-provided target edge length L, and terminates when no local operation leads to either an improvement in the desired edge lengths or an improvement in vertex valence [Botsch and Kobbelt 2004].

In Figure 6, we compare our implementation of [Botsch and Kobbelt 2004] with the implementation in OpenFlipper [Möbius and Kobbelt 2010]. The OpenFlipper implementation is 2.5 times faster when running on a single thread, and our implementation becomes faster after 4 threads are used (Figure 6).

Harmonic Triangulations. The *harmonic triangulations* algorithm has been introduced as an alternative to sliver exudation in the Delaunay tetrahedralization pipeline to efficiently reduce sliver tetrahedra. The original paper [Alexa 2019] proposes to use both flip and smoothing operations.

The code provided by the authors implements a reduced version of the algorithm proposed in the paper, restricting the optimization to 3-2 edge swap operations. We thus implemented both a reduced version for a fair comparison (Figure 7) and a complete version. Our more generic framework is twice as slower than the hand-optimized code written by the authors when running serially, and it is 2 times faster when running on 32 threads (Figure 7).

Tetrahedral Meshing. The TetWild algorithm is a tetrahedral meshing algorithm with minimal input requirements: given an input triangle soup, it can generate a tetrahedral mesh which approximates its volume. We take inspiration from the original algorithm introduced in [Hu et al. 2019b, 2018] with a few modifications: (1) we use the insertion operation [Hu et al. 2019b] (using rational coordinates) as a replacement for their BSP partitioning, as this simplifies the implementation, (2) we use the envelope proposed in [Wang et al.

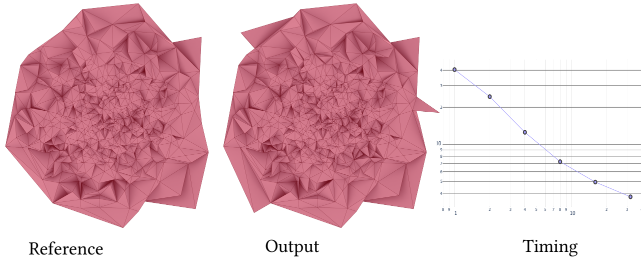

Fig. 7. Example of *Harmonic Triangulations* starting with one million Gaussian distributed random points. Both our and the reference implementation reach a similar target number of tetrahedra (5.9 million for the reference and 6.1 million for ours, due to a difference in operation ordering) and a similar Mean Harmonic Index (0.547 for the reference and 0.554 for ours). Our method takes 3.82s with 32 threads (15.49s serial, 40.49s on a single thread, speedup of 11×), while the reference serial implementation takes 6.37s.

2020] instead of sampling, and (3) we use 2-3 face swapping, 3-2 and 4-4 edge swapping operations, to simplify the implementation. We show results on two models in Figure 8: the results are very similar to the original implementation, and our version is 2 times faster when using 8 threads. We experimentally observe that our framework scales well up to 8 threads, after that the algorithm becomes slower. This is because, as we increase the number of threads and partitions, the frequent conflict in tetrahedral mesh edge operations affects the parallel performance. We believe that this observation might be useful for the future design of high performance concurrent mesh generation algorithms. We also measure the qualities of resulting tetrahedral meshes in Figure 9. Note that due to various different implementation design, our performance and quality may be better or worse compared with TetWild[Hu et al. 2019b]. Fine tuning our performance and a large scale comparision is beyond the scope of this work.

4.1 Parallelization

Enabling the parallelization mechanism introduces a minor slowdown as visible in the difference between the pure serial and one thread timings on our applications, due to the additional cost of allocating mutexes and to acquire them. Additionally, due to the nature of parallelization, our concurrent implementation is not deterministic. The differences between different runs are however minor: we run uniform remeshing on the model in Figure 6 five times and obtained an average vertex valence of 5.999 with a standard deviation of 9.885×10^{-7}. The average Hausdorff distance is 0.5% of the size of the bounding box diagonal compared to the serial result, with a standard deviation of 0.2%. The algorithm scales well in all 5 applications (figures 4, 5, 6, 7, 8), obtaining a scaling speedup (see the timing breakdown and the additional overhead in Table 1). We would like to remark that thanks to our specification and our runtime, the serial and parallel implementation of the five algorithms above is almost identical.

An inevitable drawback of parallelization is that the algorithms cannot efficiently preserve ordering. For instance, in shortest edge collapse, every thread will try to collapse edges in its own partition

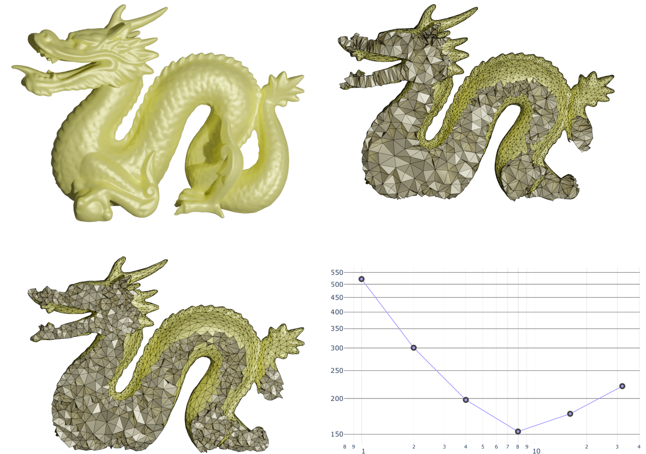

Fig. 8. Tetrahedralize a surface with 856, 294 faces. Original TetWild (top right) generates a mesh with 56, 761 tetrahedra in 287.58s; our reimplementation (bottom left) generates a mesh with 44, 866 tetrahedra in 153.33s with 8 threads (452.42s serial, 521.47s on a single thread, speedup of 3.4×). The difference in number of tetrahedra is likely due to the different order of scheduling of operations due to the partitioning.

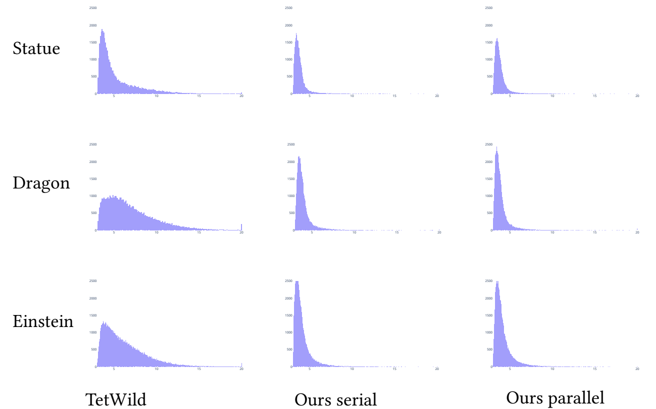

Fig. 9. Histogram of the AMIPS energy for the tetrahedra by tetrahedralization the models from in Figure 4, 8, 11. In the dragon model (Figure 8), our re-implementation produces higher quality meshes on average, likely due to the different operation scheduling. However, the parallel output has a small number of elements with low quality (e.g., AMIPS energy over 400) on this model when stopped at 10 iterations. We believe this is due to the postponement of operations when threading conflicts arise, as this effect disappears if we allow more iterations.

	Serial	Single thread	Overhead	# Mutex locked	Per lock overhead
Figure 4	4.733	5.377	0.64	2269085	2.8e-07
Figure 6	78.573	108.261	30	110768681	2.7e-07
Figure 7	13.047	42.232	29	64803789	4.5e-07

Table 1. Timings of the serial version compared with the single threaded version, which has additional overhead due to the unnecessary locking/unlocking.

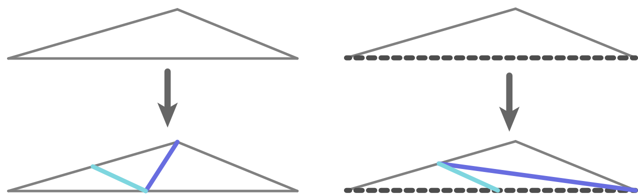

Fig. 10. Example of splitting the longest edge on the bottom of the triangle. On the left, the bottom edge can be split first with the introduction of the dark blue edge; next, the left edge is split (added the light blue edge) leading to a decreasing maximum edge-length. On the right, the bottom edge is locked (illustrated by a dashed line) ; when the bottom edge is unlocked in the next iteration, the edge length decrease is less effective.

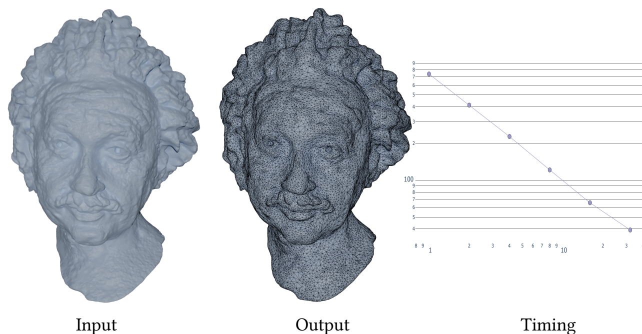

Fig. 11. Shortest edge collapse with envelope containment of a model with 857, 976 faces. Our method successfully generates a mesh with 71, 298 faces in 37.49s with 32 threads (731.32s serial, 725.34s on a single thread, speedup of 20×).

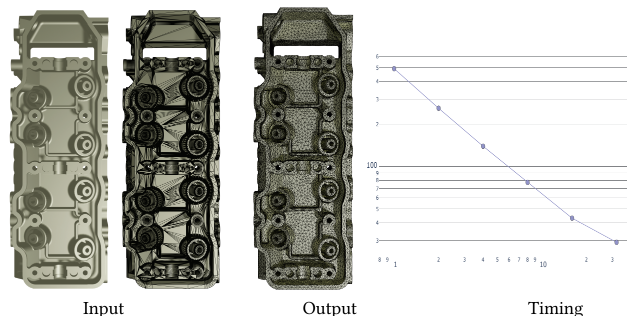

Fig. 12. Uniform remeshing with envelope containment check of a model with 198, 918 faces. Our method produces a mesh with 68, 202 faces in 29.11s with 32 threads and 493.54s for a single thread (483.68s for the serial version) leading to a speedup of 16×.

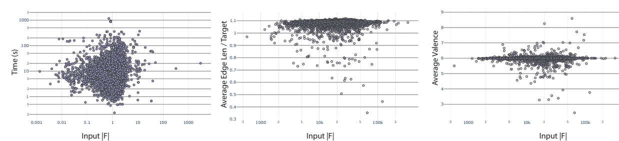

Fig. 13. Timings, target edge length ratio, and valence for every model in the dataset. Most models finish within a minute. The target edge length ratio measures how well our algorithm simplifies the meshes to reach the desired edge length, with an optimal value of 1. Since uniform remeshing strives to generate regular meshes, for most models our algorithm is able to obtain the optimal valence of 6.

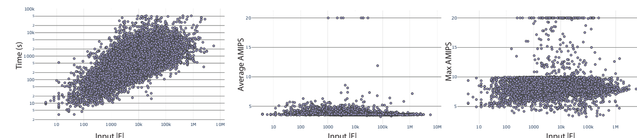

Fig. 14. Timing, max and average AMIPS energy (capped at 20) for maximum 25 iterations of tetrahedral meshing. Most models finish withing 20 minutes with only a few taking up to a day. Even by limiting the iterations to 25, most models reach an average AMIPS energy lower than 10, with optimal value at 3.

independently from the others. If one of these collapses append on the partition's interface, the thread will need to acquire a lock. In case of failure the collapse is postponed to a later stage thus not respecting the order (Figure 10). This is a rare event that is more problematic for fast operations.

4.2 Algorithm Modifications

A major motivation to invent and develop this declarative language is enabling easy customization of meshing algorithms. As an example, we add an additional termination criteria to the shortest edge collapse and uniform surface refinement. Integrating the envelope check is straightforward with our approach, as it only requires adding the envelope check to the invariants. We use the open-source library proposed in [Wang et al. 2020], which allows to directly specify the maximal allowed surface deviation. The envelope adds a noticeable computational cost, which is ameliorated by our parallel implementation (figures 11 and 12).

4.3 Large-scale dataset validation

To validate our framework we run our reimplementation of uniform remeshing and tetrahedral meshing on the Thingi10k dataset [Zhou and Jacobson 2016]. We run all experiments serially on an individual node of an HPC cluster an Intel Xeon Platinum 8268 24C 205W 2.9GHz Processors limiting the runtime to 15 hours.

For uniform remeshing, Figure 13 shows the time, average edge length normalized by the target, and average valence of isotropic remeshing on the ten thousand models. Most of our models finish within 10 seconds with only a few requiring more than a minute. For almost all meshes, the algorithm succeeds at reaching the target edge length and valence of 6.

For TetWild we limit the number of iterations to 25 (Figure 14). We note that within the 15 hours limit only 2.5% models did not finish, and after 25 iterations 3% of the models still have some rational coordinates. Among the successful models, most finish within 20 minutes and succeed in achieving high-quality meshes (only 8 models have an average AMIPS energy larger than 10).

5 CONCLUDING REMARKS

This paper introduces a new declarative specification for mesh algorithms based upon five guiding principles (Section 3.1) to allow an easier implementation, while at the same time obtaining competitive performances and exploiting parallel hardware. We note that the principles are based on our experience in mesh optimization and not necessarily the only or optimal choices. A more formal justification would be interesting to explore in the future.

Using this specification, we implement five popular mesh editing algorithms covering mesh generation and optimization on surfaces and volumes, which can be easily adapted for other use cases: we demonstrated that integrating an envelope check requires only a few lines of code.

The library we implemented supports shared memory parallelism, which leads to a good scaling on the machines we tested it on. We believe an exciting venue for future work would be the implementation of a library for our specification targeting MPI to distribute the computation over an HPC cluster. Having access to such a library would allow our five mesh editing applications to run on a distributed environment with minimal or no changes.

ACKNOWLEDGMENTS

This work was supported in part through NYU IT High Performance Computing resources, services, and staff expertise. This work was also partially supported by the NSF CAREER award under Grant No. 1652515, the NSF grants OAC-1835712, OIA-1937043, CHS-1908767, CHS-1901091, NSERC DGECR-2021-00461 and RGPIN 2021-03707, a Sloan Fellowship, a gift from Adobe Research and a gift from Advanced Micro Devices, Inc. The authors thank all the reviewers for their feedback.

REFERENCES

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

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

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

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

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

Srinivas Aluru and Fatih Erdogan Sevilgen. 1997. Parallel domain decomposition and load balancing using space-filling curves. In *Proceedings fourth international conference on high-performance computing*. IEEE, 230–235.

Ryoichi Ando, Nils Thürey, and Chris Wojtan. 2013. Highly Adaptive Liquid Simulations on Tetrahedral Meshes. *ACM Trans. Graph. (Proc. SIGGRAPH 2013)* (July 2013).

Adam W. Bargteil, Chris Wojtan, Jessica K. Hodgins, and Greg Turk. 2007. A Finite Element Method for Animating Large Viscoplastic Flow. In *ACM SIGGRAPH 2007 Papers* (San Diego, California) *(SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 16–es. https://doi.org/10.1145/1275808.1276397

Vicente HF Batista, David L Millman, Sylvain Pion, and Johannes Singler. 2010. Parallel geometric algorithms for multi-core computers. *Computational Geometry* 43, 8 (2010), 663–677.

Bruce G. Baumgart. 1972. *Winged Edge Polyhedron Representation.* Technical Report. Stanford, CA, USA.

Jan Bender and Crispin Deul. 2013. Adaptive cloth simulation using corotational finite elements. *Computers & Graphics* 37, 7 (2013), 820–829. https://doi.org/10.1016/j.cag.2013.04.008

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

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

Daniel K Blandford, Guy E Blelloch, and Clemens Kadow. 2006. Engineering a compact parallel Delaunay algorithm in 3D. In *Proceedings of the twenty-second Annual Symposium on Computational Geometry*. 292–300.

Guy E Blelloch, Gary L Miller, Jonathan C Hardwick, and Dafna Talmor. 1999. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica* 24, 3 (1999), 243–269.

Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. 2002. Triangulations in CGAL. *Computational Geometry* 22 (2002), 5–19.

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

Ricard Borrell, Juan Carlos Cajas, Daniel Mira, Ahmed Taha, Seid Koric, Mariano Vázquez, and Guillaume Houzeaux. 2018. Parallel mesh partitioning based on space filling curves. *Computers & Fluids* 173 (2018), 264–272.

Mario Botsch and Leif Kobbelt. 2004. A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. 185–192.

Mario Botsch, Stephan Steinberg, Stephan Bischoff, and Leif Kobbelt. 2002. Openmesh-a generic and efficient polygon mesh data structure. (2002).

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

E. Brisson. 1989. Representing Geometric Structures in d Dimensions: Topology and Order. In *Proceedings of the Fifth Annual Symposium on Computational Geometry* (Saarbruchen, West Germany) *(SCG '89)*. Association for Computing Machinery, New York, NY, USA, 218–227. https://doi.org/10.1145/73833.73858

Tyson Brochu, Essex Edwards, and Robert Bridson. 2012. Efficient Geometrically Exact Continuous Collision Detection. *ACM Trans. Graph.* 31, 4, Article 96 (jul 2012), 7 pages. https://doi.org/10.1145/2185520.2185592

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

Oleksiy Busaryev, Tamal K. Dey, and Joshua A. Levine. 2009. Repairing and Meshing Imperfect Shapes with Delaunay Refinement. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling* (San Francisco, California) *(SPM '09)*. ACM, 25–33.

Oleksiy Busaryev, Tamal K. Dey, and Huamin Wang. 2013. Adaptive Fracture Simulation of Multi-Layered Thin Plates. *ACM Trans. Graph.* 32, 4, Article 52 (jul 2013), 6 pages. https://doi.org/10.1145/2461912.2461920

Marcel Campen, Ryan Capouellez, Hanxiao Shen, Leyi Zhu, Daniele Panozzo, and Denis Zorin. 2021. Efficient and Robust Discrete Conformal Equivalence with Boundary. *ACM Trans. Graph.* 40, 6, Article 261 (dec 2021), 16 pages. https://doi.org/10.1145/3478513.3480557

Marcel Campen, Hanxiao Shen, Jiaran Zhou, and Denis Zorin. 2019. Seamless Parametrization with Arbitrary Cones for Arbitrary Genus. *ACM Trans. Graph.* 39, 1 (2019).

Marcel Campen and Denis Zorin. 2017a. *On Discrete Conformal Seamless Similarity Maps.* arXiv:1705.02422 [cs.GR]

Marcel Campen and Denis Zorin. 2017b. Similarity Maps and Field-Guided T-Splines: a Perfect Couple. *ACM Trans. Graph.* 36, 4 (2017).

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

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

Long Chen and Jin-chao Xu. 2004. Optimal Delaunay Triangulations. *Journal of Computational Mathematics* 22, 2 (2004), 299–308.

Min-Bin Chen. 2010. The merge phase of parallel divide-and-conquer scheme for 3d delaunay triangulation. In *International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 224–230.

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

Siu-Wing Cheng, Tamal K. Dey, and Jonathan Shewchuk. 2012. *Delaunay Mesh Generation.* Chapman and Hall/CRC, Boca Raton, Florida.

Nuttapong Chentanez, Bryan E. Feldman, François Labelle, James F. O'Brien, and Jonathan R. Shewchuk. 2007. Liquid Simulation on Lattice-Based Tetrahedral Meshes. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (San Diego, California) *(SCA '07)*. Eurographics Association, Goslar, DEU, 219–228.

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

L Paul Chew, Nikos Chrisochoides, and Florian Sukup. 1997. Parallel constrained Delaunay meshing. *ASME APPLIED MECHANICS DIVISION-PUBLICATIONS-AMD* 220 (1997), 89–96.

Nikos Chrisochoides. 2006. Parallel mesh generation. In *Numerical solution of partial differential equations on parallel computers*. Springer, 237–264.

Nikos Chrisochoides and Démian Nave. 2003. Parallel Delaunay mesh generation kernel. *Internat. J. Numer. Methods Engrg.* 58, 2 (2003), 161–176.

Paolo Cignoni, Fabio Ganovelli, et al. 2021. VCG Library.

P. Cignoni, C. Montani, R. Perego, and R. Scopigno. 1993. Parallel 3D Delaunay Triangulation. *Computer Graphics Forum* 12, 3 (1993), 129–142. https://doi.org/10.1111/1467-8659.1230129

Pascal Clausen, Martin Wicke, Jonathan R. Shewchuk, and James F. O'Brien. 2013. Simulating Liquids and Solid-Liquid Interactions with Lagrangian Meshes. *ACM Trans. Graph.* 32, 2, Article 17 (apr 2013), 15 pages. https://doi.org/10.1145/2451236.2451243

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

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

Tamal K. Dey and Joshua A. Levine. 2008. Delpsc: A Delaunay Mesher for Piecewise Smooth Complexes. In *Proceedings of the twenty-fourth annual symposium on Computational geometry - SCG '08*. ACM Press, New York, NY, USA, 220–221. https://doi.org/10.1145/1377676.1377712

Antonio DiCarlo, Alberto Paoluzzi, and Vadim Shapiro. 2014. Linear algebraic representation for topological structures. *Computer-Aided Design* 46 (2014), 269–274. https://doi.org/10.1016/j.cad.2013.08.044 2013 SIAM Conference on Geometric and Physical Modeling.

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

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

Jean-François Dufourd. 1991. An OBJ3 functional specification for boundary representation. In *Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications*. 61–72.

Leman Feng, Pierre Alliez, Laurent Busé, Hervé Delingette, and Mathieu Desbrun. 2018. Curved Optimal Delaunay Triangulation. *ACM Trans. Graph.* 37, 4 (2018), 61:1–61:16.

Panagiotis Foteinos and Nikos Chrisochoides. 2011. Dynamic parallel 3D Delaunay triangulation. In *Proceedings of the 20th International Meshing Roundtable*. Springer, 3–20.

Daniel Funke and Peter Sanders. 2017. Parallel d-D delaunay triangulations in shared and distributed memory. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 207–217.

Jérôme Galtier and Paul Louis George. 1996. Prepartitioning as a way to mesh subdomains in parallel. In *in 5th International Meshing Roundtable*. Citeseer.

Michael Garland and Paul Heckbert. 1999. *Quadric-Based Polygonal Surface Simplification*. Ph.D. Dissertation. USA. AAI9950005.

Michael Garland and Paul S Heckbert. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 209–216.

Mark Gillespie, Boris Springborn, and Keenan Crane. 2021. Discrete Conformal Equivalence of Polyhedral Surfaces. *ACM Trans. Graph.* 40, 4 (2021).

Eitan Grinspun, Petr Krysl, and Peter Schröder. 2002. CHARMS: A Simple Framework for Adaptive Simulation. *ACM Trans. Graph.* 21, 3 (jul 2002), 281–290. https://doi.org/10.1145/566654.566578

Xianfeng Gu, Ren Guo, Feng Luo, Jian Sun, and Tianqi Wu. 2018a. A discrete uniformization theorem for polyhedral surfaces II. *Journal of Differential Geometry* 109, 3 (2018), 431–466.

Xianfeng Gu, Feng Luo, Jian Sun, and Tianqi Wu. 2018b. A discrete uniformization theorem for polyhedral surfaces. *Journal of Differential Geometry* 109, 2 (2018), 223–256.

Leonidas Guibas and Jorge Stolfi. 1985. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi. *ACM Trans. Graph.* 4, 2 (apr 1985), 74–123. https://doi.org/10.1145/282918.282923

Stefan Gumhold, Pavel Borodin, and Reinhard Klein. 2003. Intersection free simplification. *International Journal of Shape Modeling* 9, 02 (2003), 155–176.

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

Hugues Hoppe. 1996. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 99–108.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019a. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.

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

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

Dave Hutchinson, Martin Preston, and Terry Hewitt. 1996. Adaptive Refinement for Mass/Spring Simulations. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation '96* (Poitiers, France). Springer-Verlag, Berlin, Heidelberg, 31–45.

Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. 2013. Robust Inside-Outside Segmentation Using Generalized Winding Numbers. *ACM Trans. Graph.* 32, 4, Article 33 (jul 2013), 12 pages. https://doi.org/10.1145/2461912.2461916

Alec Jacobson, Daniele Panozzo, C Schüller, O Diamanti, Q Zhou, N Pietroni, et al. 2016. libigl: A simple C++ geometry processing library, 2016.

Clement Jamin, Pierre Alliez, Mariette Yvinec, and Jean-Daniel Boissonnat. 2015. CGALmesh: A Generic Framework for Delaunay Mesh Generation. *ACM Trans. Math. Software* 41, 4 (10 2015), 1–24. https://doi.org/10.1145/2699463

Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. 33–37.

George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 943–948.

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, et al. 2016. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–21.

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

Bryan M. Klingner, Bryan E. Feldman, Nuttapong Chentanez, and James F. O'Brien. 2006. Fluid Animation with Dynamic Meshes. In *ACM SIGGRAPH 2006 Papers* (Boston, Massachusetts) *(SIGGRAPH '06)*. Association for Computing Machinery, New York, NY, USA, 820–825. https://doi.org/10.1145/1179352.1141961

Pierre Kraemer, Lionel Untereiner, Thomas Jund, Sylvain Thery, and David Cazier. 2014. CGoGN: N-dimensional meshes with combinatorial maps. In *Proceedings of the 22nd International Meshing Roundtable*. Springer, 485–503.

Michael Kremer, David Bommes, and Leif Kobbelt. 2013. OpenVolumeMesh–A versatile index-based data structure for 3D polytopal complexes. In *Proceedings of the 21st International Meshing Roundtable*. Springer, 531–548.

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

Cédric Lachat, Cécile Dobrzynski, and François Pellegrini. 2014. Parallel mesh adaptation using parallel graph partitioning. In *5th European conference on computational mechanics (ECCM V)*, Vol. 3. CIMNE-International Center for Numerical Methods in Engineering, 2612–2623.

Ling Li and Vasily Volkov. 2005. Cloth Animation with Adaptively Refined Meshes. In *Proceedings of the Twenty-Eighth Australasian Conference on Computer Science - Volume 38* (Newcastle, Australia) *(ACSC '05)*. Australian Computer Society, Inc., AUS, 107–113.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. 2020. Incremental Potential Contact: Intersection- and Inversion-free Large Deformation Dynamics. *ACM Trans. Graph. (SIGGRAPH)* 39, 4, Article 49 (2020).

Pascal Lienhardt. 1994. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal of Computational Geometry & Applications* 4, 03 (1994), 275–324.

Leonidas Linardakis and Nikos Chrisochoides. 2006. Delaunay decoupling method for parallel guaranteed quality planar mesh refinement. *SIAM Journal on Scientific Computing* 27, 4 (2006), 1394–1423.

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

SH Lo. 2012. Parallel Delaunay triangulation in three dimensions. *Computer Methods in Applied Mechanics and Engineering* 237 (2012), 88–106.

Adrien Loseille, Frédéric Alauzet, and Victorien Menier. 2017. Unique cavity-based operator and hierarchical domain partitioning for fast parallel generation of anisotropic meshes. *Computer-Aided Design* 85 (2017), 53–67.

Feng Luo. 2004. Combinatorial Yamabe flow on surfaces. *Communications in Contemporary Mathematics* 6, 05 (2004), 765–780.

Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. 2021. RXMesh: A GPU Mesh Data Structure. *ACM Trans. Graph.* 40, 4, Article 104 (jul 2021), 16 pages. https://doi.org/10.1145/3450626.3459748

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

Martti Mäntylä. 1987. *An introduction to solid modeling*. Computer Science Press, Inc.

Célestin Marot, Jeanne Pellerin, and Jean-François Remacle. 2019. One machine, one minute, three billion tetrahedra. *Internat. J. Numer. Methods Engrg.* 117, 9 (2019), 967–990.

Célestin Marot and Jean-François Remacle. 2020. Quality tetrahedral mesh generation with HXT. *arXiv preprint arXiv:2008.08508* (2020).

Marek Krzysztof Misztal and Jakob Andreas Bærentzen. 2012. Topology-Adaptive Interface Tracking Using the Deformable Simplicial Complex. *ACM Trans. Graph.* 31, 3, Article 24 (jun 2012), 12 pages. https://doi.org/10.1145/2167076.2167082

Jan Möbius and Leif Kobbelt. 2010. Openflipper: An open source geometry processing and rendering framework. In *International Conference on Curves and Surfaces*. Springer, 488–500.

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

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

Rahul Narain, Tobias Pfaff, and James F. O'Brien. 2013. Folding and Crumpling Adaptive Sheets. *ACM Trans. Graph.* 32, 4, Article 51 (jul 2013), 8 pages. https://doi.org/10.1145/2461912.2462010

Rahul Narain, Armin Samii, and James F. O'Brien. 2012. Adaptive Anisotropic Remeshing for Cloth Simulation. *ACM Trans. Graph.* 31, 6, Article 152 (nov 2012), 10 pages. https://doi.org/10.1145/2366145.2366171

T Okusanya and J Peraire. 1996. Parallel unstructured mesh generation. (1996).

T Okusanya and J Peraire. 1997. 3-D Parallel unstructured mesh generation. In *Proc. Joint ASME/ASCE/SES Summer Meeting*. Citeseer.

Tobias Pfaff, Rahul Narain, Juan Miguel de Joya, and James F. O'Brien. 2014. Adaptive Tearing and Cracking of Thin Sheets. *ACM Trans. Graph.* 33, 4, Article 110 (jul 2014), 9 pages. https://doi.org/10.1145/2601097.2601132

Jovan Popović and Hugues Hoppe. 1997. Progressive Simplicial Complexes. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., USA, 217–224. https://doi.org/10.1145/258734.258852

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

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

Aristides G. Requicha. 1980. Representations for Rigid Solids: Theory, Methods, and Systems. *ACM Comput. Surv.* 12, 4 (dec 1980), 437–464. https://doi.org/10.1145/356827.356833

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

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

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

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

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

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

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

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

Daniel Sieger and Mario Botsch. 2019. The Polygon Mesh Processing Library. http://www.pmp-library.org.

Timothy J. R. Simnett, Stephen D. Laycock, and Andy M. Day. 2009. An Edge-based Approach to Adaptively Refining a Mesh for Cloth Deformation. In *Theory and Practice of Computer Graphics*, Wen Tang and John Collomosse (Eds.). The Eurographics Association. https://doi.org/10.2312/LocalChapterEvents/TPCG/TPCG09/077-084

Boris Springborn. 2020. Ideal Hyperbolic Polyhedra and Discrete Uniformization. *Discrete & Computational Geometry* 64, 1 (2020), 63–108.

Jian Sun, Tianqi Wu, Xianfeng Gu, and Feng Luo. 2015. Discrete conformal deformation: algorithm and experiments. *SIAM Journal on Imaging Sciences* 8, 3 (2015), 1421–1456.

The CGAL Project. 2020. *CGAL User and Reference Manual* (5.0.3 ed.). CGAL Editorial Board. https://doc.cgal.org/5.0.3/Manual/packages.html

Jane Tournois, Camille Wormser, Pierre Alliez, and Mathieu Desbrun. 2009. Interleaving Delaunay Refinement and Optimization for Practical Isotropic Tetrahedron Mesh Generation. *ACM Transactions on Graphics* 28, 3 (07 2009), 1.

Julien Villard and Houman Borouchaki. 2002. Adaptive Meshing For Cloth Animation. In *In Proceedings of the 11th International Meshing Roundtable (IMR 2002*. 243–252.

Bolun Wang, Zachary Ferguson, Teseo Schneider, Xin Jiang, Marco Attene, and Daniele Panozzo. 2021. A Large-Scale Benchmark and an Inclusion-Based Algorithm for Continuous Collision Detection. *ACM Trans. Graph.* 40, 5, Article 188 (sep 2021), 16 pages. https://doi.org/10.1145/3460775

Bolun Wang, Teseo Schneider, Yixin Hu, Marco Attene, and Daniele Panozzo. 2020. Exact and Efficient Polyhedral Envelope Containment Check. *ACM Trans. Graph.* 39, 4 (July 2020).

Martin Wicke, Daniel Ritchie, Bryan M. Klingner, Sebastian Burke, Jonathan R. Shewchuk, and James F. O'Brien. 2010. Dynamic Local Remeshing for Elastoplastic Simulation. *ACM Trans. Graph.* 29, 4, Article 49 (jul 2010), 11 pages. https://doi.org/10.1145/1778765.1778786

Chris Wojtan and Greg Turk. 2008. Fast viscoelastic behavior with thin features. *ACM Trans. Graph.* 27, 3 (2008), 1–8. https://doi.org/10.1145/1360612.1360646

Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 144–1.

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

Rhaleb Zayer, Markus Steinberger, and Hans-Peter Seidel. 2017. A GPU-Adapted Structure for Unstructured Grids. *Comput. Graph. Forum* 36, 2 (may 2017), 495–507. https://doi.org/10.1111/cgf.13144

Min Zhou, Ting Xie, Seegyoung Seol, Mark S Shephard, Onkar Sahni, and Kenneth E Jansen. 2012. Tools to support mesh adaptation on massively parallel computers. *Engineering with Computers* 28, 3 (2012), 287–301.

Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–15.

Qingnan Zhou and Alec Jacobson. 2016. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797* (2016).