Synergistic Geometry Processing: from Robust Geometric Modeling to Efficient Physical Simulations

by

Zhongshi Jiang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September, 2022

Professor Daniele Panozzo

Acknowledgements

Abstract

The numerical solution of partial differential equations (PDEs) portrays how a craft would behave even before it is created. In practice, a typical product begins its lifecycle from a digital specification of the shape and material. Computer simulation would then test the behavior to match the intent: the car hood's deformation under pressure or a pan's temperature when heated. In this case, a robust and accurate simulation could lift many iterations of manufacturing to the digital platform. In addition, as it is safer to conduct more stress experiments in a virtual environment, a reliable simulation also promotes safety, especially in designing biomechanical equipment.

There are various approaches to specify a shape, depending on the different geometry characteristics. Shape modeling evolves beyond smooth and regular objects like sculpture or furniture. Algorithm deduced structures have gained popularity since they deliver the promise of maximal durability with limited material. Downstream, computer simulation requires a discrete tessellation of the object. A notable example is a mesh: it contains a list of vertices for the spatial locations and a list of simple polygons or polyhedra. However, it is rarely the case where one mesh fits all. More degrees of freedom give a more accurate description of the desired shape, but it inevitably means more computational resources are needed to handle the computation.

Different representations between design methods and within various adaptations of simulations, even for the same shape, are not reliably correlated. In the general case, the material, attribute, or experiment setting cannot be robustly re-used on another representation without laborious manually fixing. Such a pipeline demands the engineer or designer to understand precisely the properties of the specific numerical methods of choice. It also prohibits an automatic and robust pipeline to compute optimal structure for the desired physical behavior.

My Ph.D. research's central thesis is to develop algorithms that reliably associate different representations of discrete geometry entities at diverse design and simulation stages. Instead of re-assigning properties at various phases of the design-simulation pipeline, or whenever the required budget is different, a good association between geometry at different stages gives more freedom to the design and development of automatic and adaptive solvers. The adaptation can be bi-directional: on the one hand, with coarse and concise tessellation, the algorithm utilizes the information associated with the original design to obtain more faithful simulations. On the other hand, in early-stage prototyping, and automatic data generation for deep learning, the principles I established allows for simplifying the shape and retaining the settings, providing a fast answer. In addition, I use the same principle to design the first large-scale and robust approach for curved mesh representation, which provides accurate representation as well as efficient physical simulations.

Besides the algorithm design, I also explore the underpinning theory to provide guarantees. Theoretical inquiry not only gives us a better understanding of digital geometry and shapes but also lays the foundation for suitable applications. On the other hand, we perform extensive numerical validations with the implemented software, involving tens of thousands of complex different geometry shapes. Finally, I also release the program implementation and detailed specifications to be open source and accessible.

Contents

List of Figures

List of Tables

1 Introduction

2 Preliminaries

3 | Bijective Projection in the Shell

3.1 Introduction

Triangular meshes are the most popular representation for discrete surfaces, due to their flexibility, efficiency, and direct support in rasterization hardware. Different applications demand different meshes, ranging from extremely coarse for collision proxies, to high-resolution and high-quality for accurate physical simulation. For this reason, the adaptation of a triangle mesh to a specific set of criteria (surface remeshing) is a core building block in geometry processing, graphics, physical simulation, and scientific computing.

In most applications, the triangular mesh is equipped with attributes, such as textures, displacements, physical properties, and boundary conditions (Figure 3.1). Whenever remeshing is needed, these properties must be transferred on the new mesh, a task which has been extensively studied in the literature and for which robust and generic solutions are still lacking (Section 4.2). Defining a continuous bijective map, more precisely, a homeomorphism where the inverse is also continuous, between two geometrically close piecewise-linear meshes of the same topology is a difficult problem, even in its basic form, when one of these meshes is obtained by adapting the other in some way (e.g., coarsening, refining, or improving triangle shape).a Common approaches to this problem are Euclidean projection [Jiao and Heath 2004], parametrization on a common domain [Praun et al. 2001; Kraevoy and Sheffer 2004; Lee et al. 1998], functional maps [Ovsjanikov et al. 2012], and generalized barycentric coordinates [Hormann and Sukumar 2017]. However, the problem is not fully solved, as all existing methods, as we discuss in greater detail in Section 4.2, often fail to achieve bijectivity and/or sufficient quality of the resulting maps when applied to complex geometries. Our focus is on correspondences between meshes obtained during a remeshing procedure, instead of solving the more general problem of processing arbitrary mesh pairs.

In this work, we propose a general construction designed to enable attribute mapping between geometrically close (in a well-defined sense) meshes by jointly constructing: (1) a shell $\mathcal{S}$ around triangle mesh $\mathcal{T}$ spanned by a set of prisms, inducing a volumetric vector field $\mathcal{V}$ in its interior and (2) a projection operator $\mathcal{P}$ that bijectively maps surfaces inside the shell to $\mathcal{T}$, as long as the dot product of the surface face normals and $\mathcal{V}$ is positive (we call such a surface a *section* of $\mathcal{S}$). Given a surface mesh $\mathcal{T}$ and its shell $\mathcal{S}$, it is now possible to exploit the bijection induced by $\mathcal{P}$ in many existing remeshing algorithms by adding to them an additional constraint ensuring that the generated surface is a section of a given shell.

As long as the generated mesh is a section, the projection operator $\mathcal{P}$ can be used to transfer application-specific attributes. At a higher level, the middle surface of our shell can be seen as

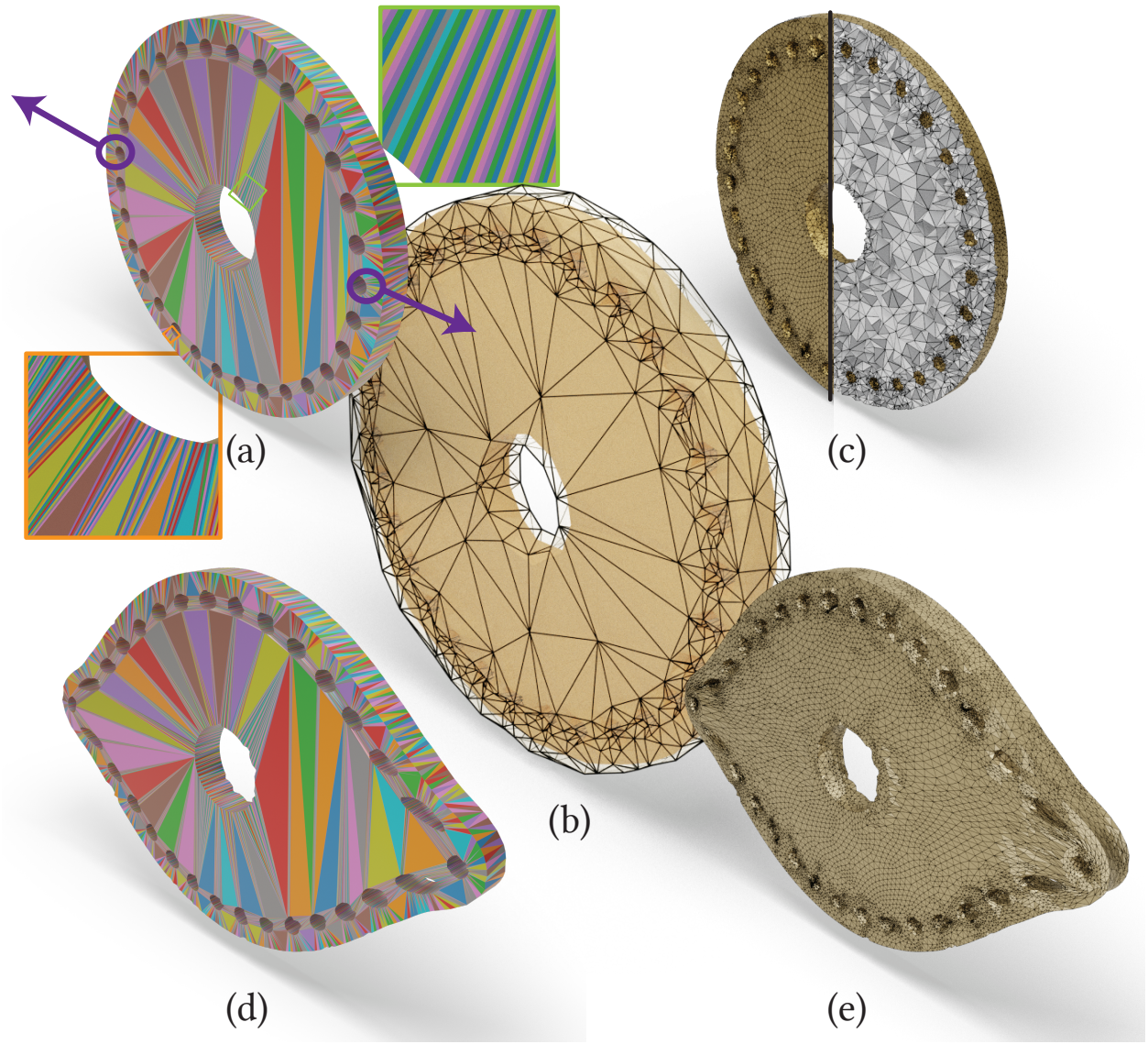


Figure 3.1: A low-quality mesh with boundary conditions (a) is remeshed using our shell (b) to maintain a bijection between the input and the remeshed output. The boundary conditions (arrows in (a)) are then transferred to the high-quality surface (c), and a non-linear elastic deformation is computed on a volumetric mesh created with TetGen (e). The solution is finally transferred back to the original geometry (d). Note that in this application setting both surface and volumetric meshing can be hidden from the user, who directly specifies boundary conditions and analyses the result on the input geometry.

a common parametrization domain shared by sections within the shell: differently from other methods that map the triangle meshes to a disk, region of a plane, a canonical polyhedron, or orbifolds, our construction uses an explicit triangle mesh embedded in ambient space as the common parametrization domain. This provides additional flexibility since it is adaptive to the density of the mesh and naturally handles models with high genus, while being numerically stable under floating-point representation (and exact if evaluated with rational arithmetic). The downside is that it is defined only for sections contained within the shell. The construction and optimization of our shell, and corresponding bijective mapping, is computationally more expensive than remeshing-only methods: our algorithms takes seconds to minutes on small and medium sized models, and might take hours on the large models in our tests.

We evaluate the robustness of the proposedapproach by constructing shells for a subset of the models in Thingi10k [Zhou and Jacobson 2016] and in ABC [Koch et al. 2019] (Section 3.4). We also integrate it in six common geometry processing algorithms to demonstrate its practical applicability (Section 3.5):

1. *Proxy.* The creation of a proxy, high-quality remeshed surface to solve PDEs (e.g., to compute geodesic distances or deformations), avoiding the numerical problems caused by a low-quality input in commonly used codes. Bijective projection operators associated with a shell enable us to transfer boundary conditions to the proxy mesh, compute the solution on the proxy, and then transfer the solution back to the original geometry.

2. *Boolean operations.* The remeshing of intermediate results of Boolean operations, to ensure high-quality intermediate meshes while preserving a bijection to transfer properties between them.

3. *Displacement Mapping.* The automatic conversion of a dense mesh into a coarse approximation and a regularly sampled displacement height map. Our method generates a bijection that allows us to bake the geometric details in a displacement map.

4. *Tetrahedral Meshing.* The conversion of a surface mesh of low quality into a high-quality tetrahedral mesh, with bijective correspondence.

5. *Geometric Textures.* Generation of complex topological structures using volumetric textures mapped to the volumetric parametrization of a simplified shell defined by $\mathcal{P}$. Our analysis on the initial shell also complements the literature on shell maps.

6. *Nested Cages.* A robust approach to generate a coarse approximation of a surface for collision checking, cage-based deformation, or multigrid approaches.

Our contributions are:

1. An algorithm to build a prismatic shell and the corresponding projection operator around an orientable, manifold, self-intersection free triangle mesh with arbitrary quality;

2. A new definition of bijective maps between *close-by* discrete surfaces;

3. A reusable, reference implementation provided at `https://github.com/jiangzhongshi/bijective-projection-shell`.

3.2 Related Works

We review works in computer graphics spanning both the realization of maps for attribute transfer (Section 3.2.1), and the explicit generation of boundary cages (Section 3.2.2), which are closest to our work.

3.2.1 Attribute Transfer

Transferring attributes is a common task in computer graphics to map colors, normals, or displacements on discrete geometries. The problem is deeply connected with the generation of UV maps, which are piecewise maps that allow to transfer attributes from the planes to surfaces (and composition of a UV map with an inverse may allow transfer between surfaces). We refer to [Floater and Hormann 2005; Sheffer et al. 2006; Hormann et al. 2007] for a complete overview, and we review here only the most relevant works.

Projection Modifying the normal field of a surface has roots in computer graphics for Phong illumination [Phong 1975], and tessellation [Boubekeur and Alexa 2008]. Orthographic, spherical, and cage based projections are commonly used to transfer attributes, even if they often leads to artifacts, due to their simplicity [Community 2018; Nguyen 2007]. Projections along a continuously-varying normal field has been used to define correspondences between neighbouring surfaces [Kobbelt et al. 1998; Lee et al. 2000; Panozzo et al. 2013; Ezuz et al. 2019], but it is often discontinuous and non-bijective. While the discontinuities are tolerable for certain graphics applications (and they can be reduced by manually editing the cage), these approaches are not usable in cases where the procedure needs to be automated (batch processing of datasets) or when bijectivity is required (e.g., transfer of boundary conditions for finite element simulation). These types of projection may be useful for some remeshing applications to eliminate surface details [Ebke et al. 2014], but it makes these approaches not practical for reliably transferring attributes. Our shell construction, algorithms, and associated projection operator, can be viewed as guaranteed continuous bijective projection along a field.

Common Domains A different approach to transfer attributes is to map both the source and the target to a common parametrization domain, and to compose the parametrization of the source domain with the inverse parametrization of the target domain to define a map from source to target. In the literature, there are methods that map triangular meshes to disks [Tutte 1963; Floater 1997], region of a plane [Maron et al. 2017; Aigerman et al. 2015, 2014; Schüller et al. 2013; Smith and Schaefer 2015; Rabinovich et al. 2017; Jiang et al. 2017; Weber and Zorin 2014; Campen et al. 2016; Müller et al. 2015; Gotsman and Surazhsky 2001; Surazhsky and Gotsman 2001; Zhang et al. 2005; Fu and Liu 2016; Litke et al. 2005; Schmidt et al. 2019], a canonical coarse polyhedra [Kraevoy and Sheffer 2004; Praun et al. 2001], orbifolds [Aigerman and Lipman 2015; Aigerman et al. 2017; Aigerman and Lipman 2016], Poincare disk [Springborn et al. 2008; Stephenson 2005; Kharevych et al. 2006; Jin et al. 2008], spectral basis [Ovsjanikov et al. 2012; Shoham et al. 2019; Ovsjanikov et al. 2017], and abstract domains [Kraevoy and Sheffer 2004; Schreiner et al. 2004; Pietroni et al.

2010]. While these approaches allow mappings between completely different surfaces, this is a hard problem to tackle in full generality fully automatically, with guarantees on the output (even some instances of the problem of global parametrization, i.e. maps from a specific type of almost everywhere flat domains to surfaces, lack a fully robust automatic solution).

Our approach uses a coarse triangular domain embedded in ambient space as the parametrization domain, and uses a vector field-aligned projection within an envelope to parametrize close-by surfaces bijectively to the coarse triangular domain. Compared to the methods listed above, our approach has both pros and cons. Its limitation is that it can only bijectively map surfaces that are similar to the domain, but on the positive side, it: (1) is efficient to evaluate, (2) guarantees an exact bijection (it is closed under rational computation), (3) works on complex, high-genus models, even with low-quality triangulations, (4) less likely to suffer from high distortion (and the related numerical problems associated with it), often introduced by the above methods. We see our method not as a replacement for the fully general surface-to-surface maps (since it cannot map surfaces with large geometric differences), but as a complement designed to work robustly and automatically for the specific case of close surfaces, which is common in many geometry processing algorithms, as well as serve as a foundation for generating such close surfaces (e.g., surface simplification and improvement, see Section 3.5)

Attribute Tracking In the specific context of remeshing or mesh optimization, algorithms have been proposed to explicitly track properties defined on the surface [Garland and Heckbert 1997; Cohen et al. 1997; Dunyach et al. 2013] after every local operation. By following the operations in reverse order, it is possible to resample the attributes defined on the input surface. These methods are algorithm specific, and provide limited control over the distortion introduced in the mapping. Our algorithm provides a generic tool that enables any remeshing technique to obtain such a map with minimal modifications.

3.2.2 Shell Generation

The generation of shells (boundary layer meshes) around triangle meshes has been studied in graphics and scientific computing.

Envelopes Explicit [Cohen et al. 1996, 1997] or implicit [Hu et al. 2016] envelopes have been used to control geometric error in planar [Hu et al. 2019a], surface [Guéziec 1996; Hu et al. 2017; Cheng et al. 2019], and volumetric [Hu et al. 2018, 2019b] remeshing algorithms. Our shells can be similarly used to control the geometric error introduced during remeshing, but they offer the advantage of providing a bijection between the two surfaces, enabling to transfer attributes between them without explicit tracking [Cohen et al. 1997]. We show examples of both surface and volumetric remeshing in Section 3.5. Also, [Barnhill et al. 1992; Bajaj et al. 2002] utilize envelopes for function interpolation and reconstruction, where our optimized shells can be used for similar purposes.

Shell Maps 2.5D geometric textures, defined around a surface, are commonly used in rendering applications [Wang et al. 2003, 2004; Porumbescu et al. 2005; Peng et al. 2004; Lengyel et al. 2001; Chen et al. 2004; Huang et al. 2007; Jin et al. 2019]. The requirement is to have a thin shell around the surface that can be used to map an explicit mesh copied from a texture, or a volumetric density field used for ray marching. Our shells are naturally equipped with a 2.5D parametrization that can be used for these purposes, and have the advantage of allowing users to generate coarse shells which are efficient to evaluate in real-time. The bijectivity of our map ensures that the volumetric texture is mapped in the shell without discontinuities. We show one example in Section 3.5.

Boundary Layer Boundary layers are commonly used in computational fluid dynamics simulations requiring highly anisotropic meshing close to the boundary of objects. Their generation is considered challenging [Aubry et al. 2015, 2017; Garimella and Shephard 2000]. These methods generate shells around a given surface, but do not provide a bijective map suitable for attribute transfer.

Collision and Animation Converting triangle meshes into coarse cages is useful for many applications in graphics [Sacht et al. 2015], including proxies for collision detection [Calderon and Boubekeur 2017] and animation cages [Thiery et al. 2012]. While not designed for this application, our shells can be computed recursively to create increasingly coarse nested cages. We hypothesize that a bijective map defined between all surfaces of the nested cages could be used to transfer forces from the cages to the object (for collision proxies), or to transfer handle selections (for animation cages). [Botsch et al. 2006; Botsch and Kobbelt 2003] uses a prismatic layer to define volumetric deformation energy, however their prisms are disconnected and only used to measure distortion. Our prisms could be used for a similar purpose since they explicitly tesselate a shell around the input surface.

3.2.3 Robust Geometry Processing

The closest works, in terms of applications, to our contribution are the recent algorithms enabling black-box geometry processing pipelines to solve PDEs on meshes *in the wild*.

[Dyer et al. 2007; Liu et al. 2015] refines arbitrary triangle meshes to satisfy the Delaunay mesh condition, benefiting the numerical stability of some surface based geometry processing algorithms. These algorithms are orders of magnitude faster than our pipeline, but, since they are refinement methods, cannot coarsen dense input models. While targeting a different application, [Sharp et al. 2019] offers an alternative solution, which is more efficient than the extrinsic techniques [Liu et al. 2015] since it avoids the realization of the extrinsic mesh (thus naturally maintaining the correspondence to the input, but limiting its applicability to non-volumetric problems) and it alleviates the introduction of additional degrees of freedom. [Sharp and Crane 2020] further generalizes [Sharp et al. 2019] to handle non-manifold and non-orientable inputs, which our approach currently does not support.

TetWild [Hu et al. 2018, 2019b] can robustly convert triangle soups into high-quality tetrahedral meshes, suitable for FEM analysis. Their approach does not provide a way to transfer boundary

Figure 3.2: Overview of our algorithm. We start from a triangle mesh, find directions of extrusion, build the shell, and optimize to simplify it.

conditions from the input surface to the boundary of the tetrahedral mesh. Our approach, when combined with a tetrahedral mesher that does not modify the boundary, enables to remesh low-quality surface, create a tetrahedral mesh, solve a PDE, and transfer back the solution (Figure 3.1). However, our method does not support triangle soups, and it is limited to manifold and orientable surfaces.

3.2.4 Isotopy between surfaces

[Chazal and Cohen-Steiner 2005; Chazal et al. 2010] presents conditions for two sufficiently smooth surfaces to be isotopic. Specifically, the projection operator is a homeomorphism. [Mandad et al. 2015] extends this idea to make an approximation mesh that is isotopic to a region. However, they did not realize a map suitable for transferring attributes.

3.3 Method

Our algorithm (Figure 3.2) converts a self-intersection free, orientable, manifold triangle mesh $\mathcal{T} = \{V_\mathcal{T}, F_\mathcal{T}\}$, where $V_\mathcal{T}$ are the vertex coordinates and $F_\mathcal{T}$ the connectivity of the mesh, into a shell composed of generalized prisms $\mathcal{S} = \{(B_\mathcal{S}, M_\mathcal{S}, T_\mathcal{S}), F_\mathcal{S}\}$, where $B_\mathcal{S}$, $M_\mathcal{S}$, $T_\mathcal{S}$ are bottom, middle, and top surfaces of the shell, consisting of bottom, middle, and top triangles of the prisms, and $F_\mathcal{S}$ is the connectivity of the prisms (Figure 3.3). The algorithm initially generates a shell $\mathcal{S}$ whose middle surface $M_\mathcal{S}$ has the same geometry as the input surface $\mathcal{T}$ (possibly with refined connectivity), and then optimizes it while ensuring that $\mathcal{T}$ is contained inside and projects bijectively to $M_\mathcal{S}$. The shell induces a volumetric vector field $\mathcal{V}$ and a projection operator $\mathcal{P}$ in the interior of each of its prisms (Section 3.3.1). This output can be used directly in many geometry processing tasks, as we discuss in detail in Section 3.5.

We first introduce the definition of our projection operator $\mathcal{P}$ and the conditions required for

Figure 3.3: Example of the top (left, outer) and bottom (right, inner) surface of the prismatic shell.

bijectivity of its restrictions to sections of the shell (Section 3.3.1). We then define shell validity (Section 3.3.2), present our algorithm for creating an initial shell (Section 3.3.3) and optimizing it to decrease the number of prisms (Section 3.3.4). To simplify the exposition, we initially assume that our input triangle mesh does not contain *singular points* (defined in Section 3.3.3) and boundary vertices, and we explain how to modify the algorithm to account for these cases in sections 3.3.5 and 3.3.6.

3.3.1 Shell and Projection

Let us consider a single generalized prism Δ in a prismatic layer $\mathcal{S}$ (Figure 3.4 left). The generalized prism Δ is defined by the position of the vertices of three triangles, one at the top, with coordinates t_1, t_2, t_3, one at the bottom, with coordinates b_1, b_2, b_3, and one in the middle, implicitly defined by a per-vertex parameter $\alpha_i \in [0, 1]$, with coordinates $m_i = \alpha_i t_i + (1 - \alpha_i) b_i, i = 1, 2, 3$. We will call the top (bottom) "half" of the prism *top (bottom) slab* (we refer to Appendix **??** for an explanation on why we need two slabs). For brevity, we will refer to a generalized prism as a prism.

Decomposition in Tetrahedra We decompose each prism Δ into 6 tetrahedra (3 in the top slab and 3 in the bottom one, Figure 3.4 middle), using one of the patterns in [Dompierre et al. 1999, Figure 4]. The patterns are identified by the orientation (rising/falling) of the two edges cutting the side faces of the prism. While, for a single prism, any decomposition would be sufficient for our purposes, we need a consistent tetrahedralization between neighboring prisms to avoid inconsistencies in the projection operator. To resolve this ambiguity, we use the technique proposed in [Garimella and Shephard 2000]: we define a total ordering over the vertices of the

Figure 3.4: A prism Δ (left) is decomposed into 6 tetrahedra (middle, for clarity, we only draw the 3 tetrahedra of the top slab). Each tetrahedron has a constant vector field in its interior (pointing toward the top surface), which is parallel to the only pillar of the prism that contains the point.

middle surface of Δ (naturally, we use the vertex id) and split (for each half of the prism) the face connecting vertices v_1 and v_2 with a rising edge if $v_1 < v_2$ and a falling edge otherwise.

Forward and Inverse Projection We define a piecewise constant vector field $\mathcal{V}$ inside the decomposed prism, by assigning to each tetrahedron $\boldsymbol{T}_j^\Delta$, $j = 1, \dots, 6$, the constant vector field defined by the only edge of $\boldsymbol{T}_j^\Delta$ which is a *oriented pillar* of Δ connecting the bottom surface to the top surface passing through the middle surface). That is, for any $p \in \boldsymbol{T}_j^\Delta$

$$\mathcal{V}(p) = t_i - b_i, \tag{3.1}$$

where i is the index of the vertex corresponding to the pillar edge of $\boldsymbol{T}_j^\Delta$. Note that $\mathcal{V}$ is constant on each tetrahedron and might be discontinuous on the boundary: we formally define the value of $\mathcal{V}$ on the boundary as any of the values of the incident tetrahedra. This choice does not affect our construction. There is exactly one integral (poly-)line passing through each point of the prism if all the decomposed tetrahedra have positive volumes (Theorem 3.2). This allows us to define the *projection operator* $\mathcal{P}(p)$ for a point $p \in \Delta$ as the intersection of the integral line $f_p(t)$ of the vector field $\mathcal{V}$ passing through p, with the middle surface of Δ (Figure 3.5). Intuitively, we can project any mesh that does not fold in each prism (Figure 3.6) to the middle surface. Formally, we introduce the following definition, to describe this property in terms of the triangle normals of the mesh.

With a slight abuse of the notation, for meshes and collections of prisms A and B, we use $A \cap B$ to denote the intersection of their corresponding geometry.

prism-tex/figs/projection.pdf

Figure 3.5: A point p (left) is traced through $\mathcal{V}$ inside the top part of the shell. A ray with p as origin and $\mathcal{V}$ as direction is cast inside the orange tetrahedron (middle). The procedure is repeated (on the blue tetrahedron) until the ray hits a point in the middle surface (right).

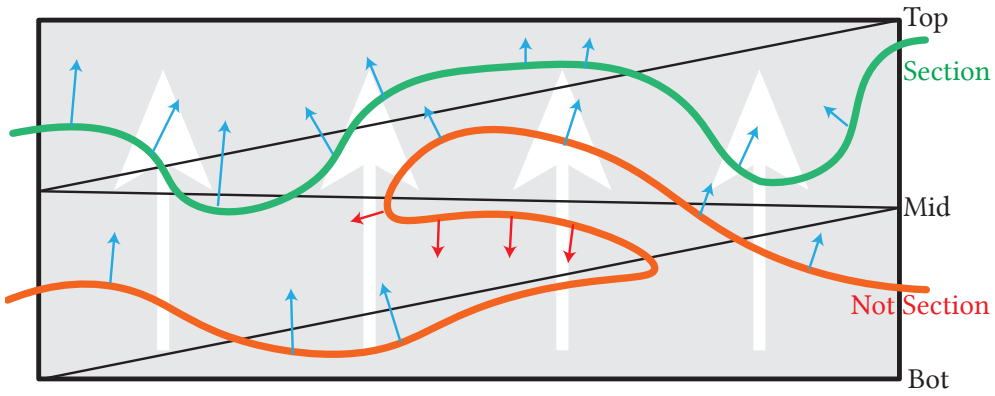


Figure 3.6: A 2D illustration for the normal dot product condition. The blue arrows agrees with the background vector field (white arrows), while the red arrows do not agree.

Definition 3.1. A *section* $\hat{\mathcal{T}}$ of a prism Δ is a manifold triangle mesh whose intersection with Δ is a simply connected submesh $\hat{\mathcal{T}} \cap \Delta$ whose single boundary loop is contained in the boundary of Δ, excluding its top and bottom surface, and such that for every point $p \in \hat{\mathcal{T}} \cap \Delta$ the dot product between the face normal $n(p)$ and the vector field $\mathcal{V}(p)$ is strictly positive. Similarly, a triangle mesh $\hat{\mathcal{T}}$ is a *section of a shell* $\mathcal{S}$, if it is a section of all the prisms of $\mathcal{S}$.

Note that this definition implies that all sections are contained inside the shell. Additionally, the definition implies that the section does not intersect with either bottom or top surface. However, our definition allows for the bottom or top surface to self-intersect. The intersection of the shell does not invalidate the *local* definition of projection since it is defined per prism. Allowing intersections is crucial to an efficient implementation of our algorithm since it allows us to take advantage of a static spatial data structure in later stages of the algorithm (Section 3.3.4).

Theorem 3.2. *If all 6 tetrahedra T_j^Δ in a decomposition of a prism Δ have positive volume, then the projection operator $\mathcal{P}$ defines a bijection between any section $\hat{\mathcal{T}}$ of Δ and the middle triangle of the prism (M in Figure 3.7).*

Proof. We prove this theorem in Appendix **??**. □

Figure 3.7: The composition $\mathcal{P}_{\mathcal{S}_2}^{-1}(\mathcal{P}_{\mathcal{S}_1}(x))$ with $x \in \mathcal{S}_1$, of a direct and an inverse projection operator defines a bijection between two sections $\mathcal{S}_1$ and $\mathcal{S}_2$.

The *inverse projection operator* $\mathcal{P}^{-1}$ is defined for a section $\hat{\mathcal{T}}$ as the inverse of the forward projection restricted to $\hat{\mathcal{T}}$. It can be similarly computed by tracing the vector field in the opposite direction, starting from a point in the middle surface of the prism. Note that, differently from the inverse Phong projection [Panozzo et al. 2013; Kobbelt et al. 1998], whose solution depends on the root-finding of a quadric surface, our shell has an explicit form for the inverse and does not require a numerical solve. The combination of forward and inverse projection operators allows to bijectively map between any pair of sections, independently of their connectivity (Figure 3.7). An interesting property of our forward and inverse projection algorithm, which might be useful for applications requiring a provably bijective map, is that our projection could be evaluated exactly using rational arithmetic.

3.3.2 Validity Condition

Shell, projection operator, section definitions, and the bijectivity condition (Theorem 3.2) are dependent on a specific tetrahedral decomposition, which depends on vertex numbering.

To ensure that our shell construction is independent from the vertex and face order, we define the validity of a shell by accounting for all 6 possible tetrahedral decompositions [Dompierre et al. 1999, Figure 4].

Definition 3.3. We say that a prismatic shell $\mathcal{S}$ is *valid with respect to a mesh* $\hat{\mathcal{T}}$ if it satisfies two conditions for each prism.

- I1 *Positivity.* The volumes of 24 tetrahedra (Appendix **??**) corresponding to 6 tetrahedral decompositions are positive.
- I2 *Section.* $\hat{\mathcal{T}}$ is a section of $\mathcal{S}$ for all 6 decompositions.

If a shell is valid, from I1, I2, then by Theorem 3.2, it follows that any map between sections induced by the projection operator $\mathcal{P}$ is bijective.

I2 ensures that the input mesh is a *valid section* independently from the decomposition, that is, we require the dot product to be positive with respect to all *three* pillars of a prism inside the convex hull. An interesting and useful side effect of this validity condition is that it ensures the bijectivity of a natural nonlinear parametrization of the prism interior (Appendix **??**, Figure 3.25).

3.3.3 Shell Initialization

We now introduce an algorithm to compute a *valid* prismatic shell $\mathcal{S} = \{(B_\mathcal{S}, M_\mathcal{S}, T_\mathcal{S}), F_\mathcal{S}\}$ with respect to a given triangle mesh $\mathcal{T} = \{V_\mathcal{T}, F_\mathcal{T}\}$ such that $\mathcal{T}$ is geometrically identical to the middle surface of $\mathcal{S}$. We assume that the faces of the triangle mesh are consistently oriented.

Extrusion Direction. The first step of the algorithm is the computation of an extrusion direction for every vertex of $\mathcal{T}$. These directions are optimized to be pointing towards the *outside* of the triangle mesh (which we assume to be orientable), that is, they must have a positive dot product with the normals of all incident faces. More precisely, for a vertex v, we are looking for a direction d_v such that $d_v \cdot n_f > 0$ for each adjacent face f with normal n_f. We can formulate this as the optimization problem

$$\begin{aligned} &\max_{d_v} \min_{f \in N_v} n_f \cdot d_v, \\ &\text{s.t.} \quad n_f \cdot d_v \geq \epsilon, \qquad \forall f \in N_v \\ &\qquad \|d_v\|^2 = 1. \end{aligned} \tag{3.2}$$

A solution, if it exists, can be found solving the following quadratic programming problem (Appendix **??**)

$$\begin{aligned} \min \quad & \|x\|^2, \\ \text{s.t.} \quad & \mathbf{C}x \geq 1, \end{aligned} \tag{3.3}$$

with $d_v = x/\|x\|$ and $\mathbf{C}$ the matrix whose rows are the normals n_f of the faces in the 1-ring N_v of vertex v. Solutions not satisfying $\|x\| \leq 1/\epsilon$ needs to be discarded (Appendix **??**). The QP can be solved with an off-the-shelf solver[Stellato et al. 2017; Cheshmi et al. 2020], and in particular it can be solved exactly [Gärtner and Schönherr 2000] to avoid numerical problems. Note that the Problem (3.2) is studied in a similar formulation in [Aubry and Löhner 2008] but their solution requires tolerances in multiple stages of the algorithm to handle cospherical point configurations.

The admissible set of (3.2) might be empty for a vertex v, that is, no vector d_v satisfies $\mathbf{C}d_v \geq \epsilon$. In this case we call v a *singularity*. For example, Figure 3.12 shows a triangle mesh containing a singularity: there exist no direction whose dot product with the adjacent face normals is positive. To simplify the explanation, we assume for the remainder of this section that $\mathcal{T}$ does not contain singularities and also that it does not contain boundaries: we postpone their handling to sections 3.3.5 and 3.3.6.

Proposition 3.4. *Let $\mathcal{T}$ be a closed (without boundary) triangle mesh without singularities and $\mathcal{N}$ be a per-vertex displacement field satisfying $\mathbf{C}\mathcal{N}_i > 0$ for every vertex v_i of $\mathcal{T}$. Then there exist a*

strictly positive per-vertex thickness δ_i *such that vertices* t_i *and* b_i *obtained by displacing* v_i *by* δ_i *in the direction of* $\mathcal{N}_i$ *and in the opposite direction, define a shell that satisfies invariant I1.*

Proof. We provide a proof in Appendix **??**. □

Initial Thickness We first show that a strictly positive per-vertex thickness δ exists for a shell $\mathcal{S}$ with $\mathcal{T}$ as its middle surface, and then discuss a practical algorithm to realize it.

Theorem 3.5. *Given a closed, orientable, self-intersection free triangle mesh* $\mathcal{T}$ *such that for all its vertices Problem* (3.2) *has a solution, a shell* $\mathcal{S}$ *exists such that* $\mathcal{T}$ *is the middle surface and there exist a strictly positive per-vertex thickness* δ*.*

Proof. We provide a proof in Appendix **??**. □

To find a valid per-vertex thickness δ_i to construct the top surface, we initially cast a ray in the direction of $\mathcal{N}$ for each vertex and measure the distance to the first collision with $\mathcal{T}$ (and cap it to a user-defined parameter δ_{max} if no collision is found). An initial top mesh is built with this extrusion thickness; then we test whether any triangle in the top surface intersects $\mathcal{T}$, through triangle-triangle overlap test [Guigue and Devillers 2003], and iteratively shrink δ_i in this triangle by 20% until we find a thickness that prevents intersections between the input and the top surface. Analogously, we build the bottom shell along the opposite direction. Note that the thickness of a vertex for the top and bottom surface can be different.

Validity of the Initial Shell Proposition 3.4 and Theorem 3.5 ensure that the initial shell constructed using a displacement field $\mathcal{N}$ obtained by solving (3.3), satisfies property I1. However, $\mathcal{T}$ might not formally satisfy the conditions for being a section of $\mathcal{S}$, despite being identical to the middle surface of $\mathcal{S}$, due to our definition of the projection operator $\mathcal{P}$. The reason for this can be seen in Figure 3.8. After the initialization, the middle surface is the input mesh. Thus every prism Δ corresponds to a triangle T_Δ of $\mathcal{T}$. The intersection $\Delta \cap \mathcal{T}$, required to check if $\mathcal{T}$ is a section of Δ (Definition 3.1) contains points from both T_Δ and its 1-ring neighborhood.

The projection operator (and thus the definition of the section) is based on all tetrahedral decompositions of Δ; it is possible that multiple tetrahedra (with different vector field values in $\mathcal{V}$) overlap on the boundary of Δ.

Depending on the dihedral angles in the mesh $\mathcal{T}$, it is possible that the dot product between one of the pillars (orange in Figure 3.8) and the face normals of some of the triangles from 1-ring neighborhood (green in Figure 3.8) is negative. While it may seem that this problem could be addressed by changing the definition so that each edge is assigned to one of the incident triangles, so that the field direction only in one incident tetrahedron needs to be considered, this problem is more significant than it may seem, as it leads to instability under small perturbations (e.g., due to floating-point rounding of coordinates). Such small perturbations can change the set of triangles intersecting a prism and thus violate the validity of the shell and, consequently, the bijectivity of $\mathcal{P}$. We propose instead to refine $\mathcal{T}$, without changing its geometry, so that the shell corresponding to the refined mesh satisfies I2 (i.e., its middle surface is a section).

Figure 3.8: The vector field aligned with the pillar edge (orange) has a negative dot product with the green triangle normal; the tetrahedron with this field direction meets the green triangle at the red vertex. As a consequence, the mid-surface is not a section. After topological beveling, the shell becomes valid since the dot product between the green normal and the new pillar (purple) is positive.

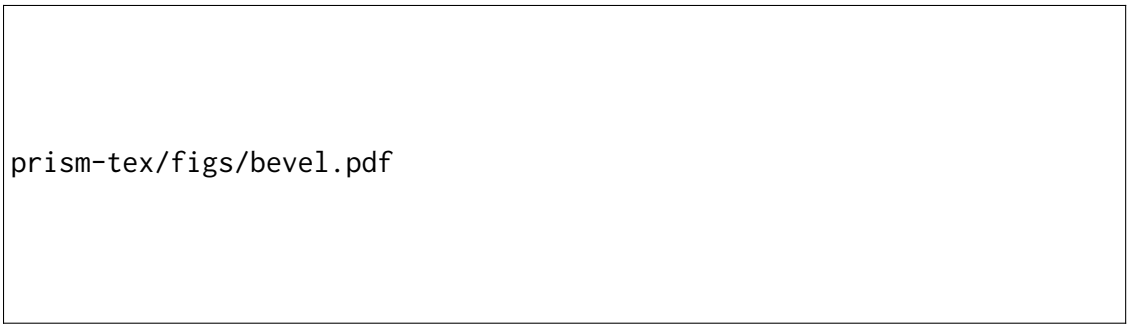


Figure 3.9: The beveling patterns used to decompose prisms for which $\mathcal{T}$ is not a section.

Topological Beveling. We identify a prism Δ_v for which I2 does not hold and use a beveling pattern [Coxeter 1973; Conway et al. 2016; Hart 2018] to decompose Δ_v in a way that $\mathcal{T}$ becomes a section for all 6 decompositions (I2). We refer to this operation as *topological beveling*, as it does not change the geometry of the mesh, only its connectivity (Figure 3.10). We use the pattern in Figure 3.9a for Δ_v, and we use the other two patterns (b) and (c) on the adjacent prisms to ensure valid mesh connectivity. The positions of the vertices are computed using barycentric coordinates (we used $t = 0.2$, i.e., the orange dot is at $1/5$ of the horizontal edge), and the normals of the newly inserted vertices are copied from the closest vertex (in Figure 3.9, the internal vertices have the normal of the triangle corner with the same color).

Theorem 3.6. *Suppose $\mathcal{T}$ is the middle surface of $\mathcal{S}$, and neither $T_\mathcal{S}$ or $B_\mathcal{S}$ intersects with $\mathcal{T}$. After topological beveling, I2 holds, that is, $\mathcal{T}$ is a section of the shell $\mathcal{S}$ for all 6 decompositions.*

Proof. We provide a proof in Appendix **??**. □

Output The output of this stage is a valid shell with respect to $\mathcal{T}$ (Section 3.3.2), that is, it satisfies I1 and I2.

Figure 3.10: Our algorithm refines the input model (left) with beveling patterns (middle) to ensure the generation of a valid shell. The subsequent shell optimizations gracefully remove the unnecessary vertices (right).

3.3.4 Shell Optimization

During shell optimization, we perform local operations (Figure 3.11) on a valid shell to reduce its complexity and increase the quality. Before applying every operation, we check the validity of the operation to ensure that: (1) the resulting middle surface will be manifold [Dey et al. 1999] and (2) the shell will be valid with respect to $\mathcal{T}$ (to ensure a bijective projection). We forbid any operation that does not pass these checks. We would like to remark that, while there are different choices to guide the shell modification, we experimentally discovered that allowing shell simplification and optimization consistently leads to thicker shells with a richer space of sections.

Theorem 3.7. *Let $\mathcal{S}$ be a valid shell with respect to a mesh $\mathcal{T}$ and let $C = \{\Delta_i\}_{i\in I} \subset \mathcal{S}$ be a collection of prisms such that the middle surface M_C of C is a simply connected topological disk.*

Let $\mathbb{O}$ be an operation replacing C with a new collection of prisms $C' = \{\Delta'_i\}_{i\in I'}$, preserving both geometry and connectivity of the sides of the prism collection C, and ensuring that $M_{C'}$ is a simply connected topological disk.

If these three assumptions hold:

1. *property I1 holds for C',*

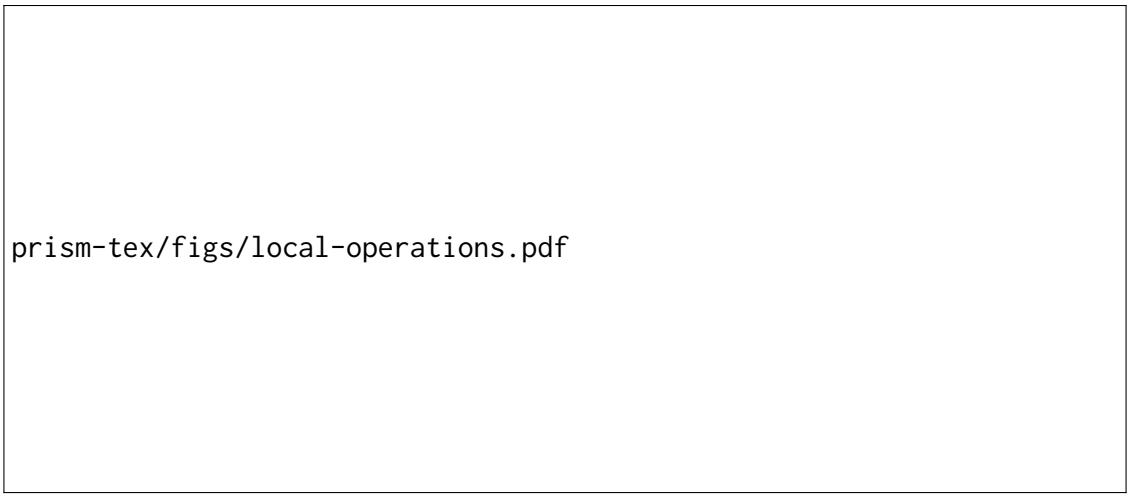


Figure 3.11: Different local operations used to optimize the shell. Mesh editing operations translate naturally to the shell setting. For vertex smoothing, we decompose the operation into 3 intermediate steps: pan, zoom, and rotate.

2. *the top and bottom surfaces of C' do not intersect $\mathcal{T}$ ($T_{C'} \cap \mathcal{T} = B_{C'} \cap \mathcal{T} = \emptyset$),*
3. *the dot product condition $n(p) \cdot \mathcal{V}(p) > 0$ is satisfied for all points $p \in \mathcal{T} \cap \Delta'_i$ for all pillars of every prism Δ'_i of C',*

then, $\forall i \in I'$, $\mathcal{T} \cap \Delta'_i$ is a simply connected topological disk. In other words, $\mathcal{T}$ is a section of the new shell $\mathcal{S}'$ obtained by applying the operation $\mathbb{O}$ to $\mathcal{S}$.

Proof. We prove this theorem in Appendix **??**. □

We note that assumption (2) in Theorem 3.7 prevents the input surface from crossing the bottom/top surface, thus avoiding it to move in the interior of a region covered by more than one prism.

Our local operations (satisfying the definition of $\mathbb{O}$ in Theorem 3.7) are translated from surface remeshing methods [Dunyach et al. 2013] since our shell can be regarded as a triangle mesh (middle surface) extruded through a displacement field $\mathcal{N}$. All the local operations described below directly change the middle surface, and consequently affect the extruded shell. After every operation, the middle surface is recomputed by intersecting $\mathcal{T}$ with the edges of the prisms in $\mathcal{S}$.

Shell Quality We measure the quality of the shell $\mathcal{S}$ using the MIPS energy [Hormann and Greiner 2000] of its middle surface M_S. For each triangle T of the middle surface, we build a local reference frame, and compute the affine map J_T transforming the triangle into an equilateral reference triangle in the same reference frame. The energy is then measured by

$$\sum_{T \in M_S} \frac{\mathrm{tr}(J_T^T J_T)}{\det(J_T)}.$$

This energy is invariant to scaling, thus allowing the local operations to coarsen the shell whenever possible while encouraging the optimization to create well-shaped triangles. Good quality of the middle surface decreases the chances, for the subsequent operations, to violate the shell invariants.

Shell Connectivity Modifications. We translate three operations for triangular meshes to the shell settings (Figure 3.11 top). Edge collapse, split, and flip operations can be performed by simultaneously modifying the top and bottom surfaces and retrieve the positions for the middle surface through the intersection. We only accept the operations if they pass the invariant check.

Vertex Smoothing. Due to the additional degree of freedom on vertex-pairs (position, direction, and thickness), we decompose the smoothing operations into three components (Figure 3.11 bottom). *Pan* moves the positions of the top and bottom vertex at the same time, minimizing the MIPS quality of the middle surface. Neither the thickness or direction will be changed. *Rotate* re-aligns the local direction to be the average of the neighboring ones while keeping the position of the middle vertex fixed. *Zoom* keeps the direction and position of the middle vertex, and set the thickness of both top and bottom to be 1.5 times of the neighbor average, capped by the input target thickness.

Invariant Check We use exact orientation predicates [Shewchuk 1997] to make sure all the prisms satisfy positivity (I1). Further, we ensure that the original surface $\mathcal{T}$ is not intersecting with the bottom and top surface, except at the prescribed singularities. The check is done using the triangle-triangle overlap test [Guigue and Devillers 2003], accelerated using a static axis-aligned bounding box tree constructed from $\mathcal{T}$. To accelerate the checks for normal condition, for each prism Δ_i, we maintain a list triangles overlapping with its convex hull (an octahedron), and check their respective normals against all the three pillars of Δ_i. These three checks ensure that the three conditions in Theorem 3.7 are satisfied. Note that the vertex smoothing operation is continuous, in the sense that any point between the current position and the optimal one improves the shell. We, however, handle it as a discrete operation to check our conditions: we attempt a full step, and if I1 is not satisfied, we perform a bisection search for a displacement that does. We avoid bisection for the other two conditions since they are expensive to evaluate.

Projection Distortion An optional invariant to maintain (not necessary for guaranteeing bijectivity, but useful for applications), is a bound on the maximal distortion $\mathcal{D}_{\mathcal{P}}(\Delta)$ of $\mathcal{P}$ for a prism Δ. We measure it as the maximal angle between the normals of the set C containing the faces of $\mathcal{T}$ intersecting Δ and $\mathcal{V}$:

$$\mathcal{D}_{\mathcal{P}}(\Delta) = \max_{p \in C} \angle(n_p, \mathcal{V}(p)),$$

where $\angle$ is the unsigned angle in degrees. This quantity is bounded from below by the smallest dihedral angle of $\mathcal{T}$, making it impossible to control exactly. However, we can prevent it from increasing by measuring it and discarding the operations that increase it. In our experiments, we use a threshold of 89.95 degrees.

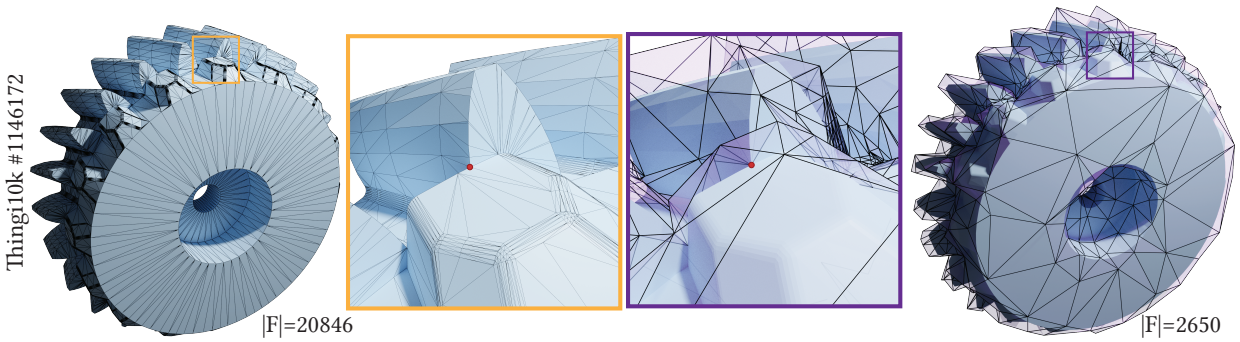


Figure 3.12: A model with 48 singularities and a close up around one (left). Our shell is pinched around each of them without affecting other regions (right).

Scheduling and Termination Our optimization algorithm is composed of two nested loops. The outer loop repeats a set of local operations until the face count between two successive iteration decreases by less than 0.01%. In the inner loop we: (1) flip every edge of $\mathcal{S}$ decreasing the MIPS energy and avoiding high and low vertex valences [Dunyach et al. 2013]; (2) smooth all vertices which include pan, zoom, and rotate; and (3) collapse every edge of $\mathcal{S}$ not increasing the MIPS energy over 30. Note that for every operation, we check the invariants, the projection distortion, and manifold preservation and reject any operation violating them. After the outer iteration terminates (i.e. the shell cannot be coarsened anymore), we further optimize the shell with 20 additional iterations of flips and vertex smoothing.

3.3.5 Singularities

Singularities, i.e. vertices of $\mathcal{T}$ for which the constraints set of problem (3.2) is empty, are surprisingly common in large datasets (Figure 3.12 shows an example). For instance, in our subset of Thingi10k [Zhou and Jacobson 2016], although only 0.01% vertices are singular, 8% of the models have at least one singular point. This has been recently observed as a limitation for the construction of nested cages [Sacht et al. 2015, Appendix A], and it is a well-known issue when building boundary layers [Aubry et al. 2015, 2017; Garimella and Shephard 2000]. There are two main situations that give rise to singular points. The first one naturally generates a singular point when more than two ridge-lines meet (e.g., figures 3.12 and 3.30), thus making the point a feature point. The second one is a pocket-like mesh artifact, often produced as a result of mesh simplification (Figure 3.13).

While singularities might seem fixable by applying local smoothing or subdivision as a pre-process, it is not desirable in the case of a feature point, and is likely to introduce self-intersection or more serious geometric inconsistency. Therefore, due to the above reasons and observing that they are very uncommon, we propose to extend our theory (Appendix **??**) and algorithm to handle isolated singularities by *pinching* the thickness of the shell. Note that in the rare case where two singular points are sharing the same edge, they are *automatically* separated by our topological beveling.

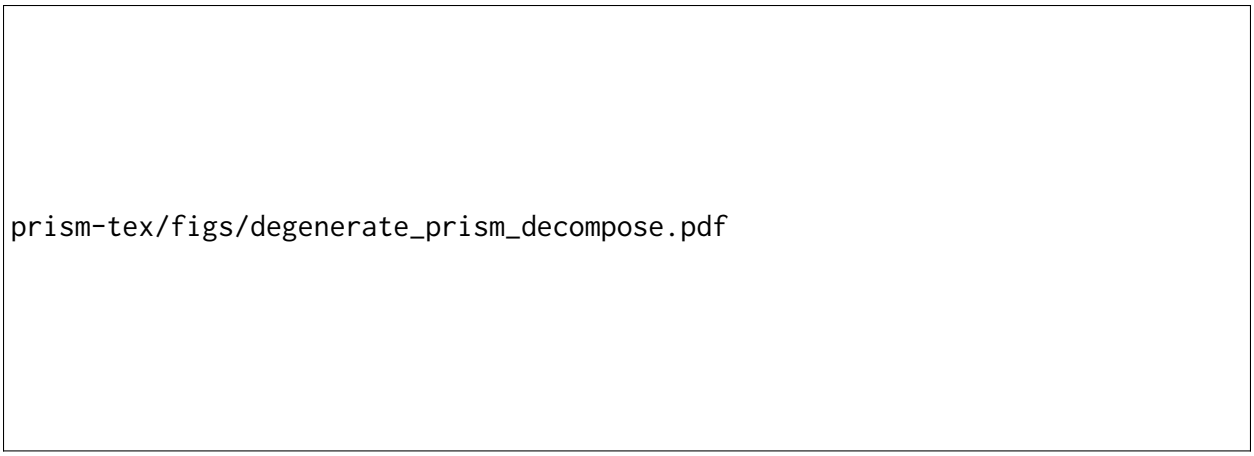
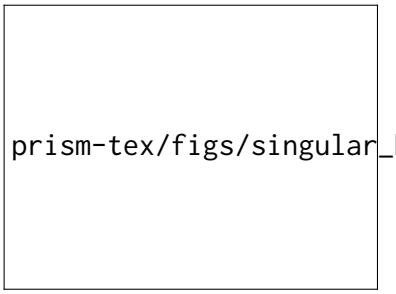


Figure 3.13: Left to right: examples of a singularity as a feature point and as a meshing artefact; and illustration of the degenerate prism with a singularity (red) and its tetrahedral decomposition made of only four tetrahedra.

Pinching. We extend our definition of the shell by allowing it to have zero thickness on singularities, thus tessellating the degenerate prism with 4 instead of 6 tetrahedra (Figure 3.13). We further remark that these isolated points must be excluded from Definition 3.1. In the implementation, this requires to change the intersection predicates to skip the singular vertices. With this change, the singularity becomes a trivial point of the projection operator $\mathcal{P}$, and the rest of our shell can still be used in applications without further changes. Since singularities tends to be isolated (they are usually located at the juncture of multiple sharp features), this solution has minimal effects on applications: for example, when our shell is used for remeshing, pinching the shell at singularities will freeze the corresponding isolated vertices while allowing the rest of the mesh to be freely optimized.

The topological beveling algorithm is changed most significantly: for singularities, there is no pillar to copy from. In this case, we apply an additional edge split, to use the pattern in the inset (with the singularity marked by a white dot) in the one-ring neighborhood of the singularity. The newly inserted vertices lie either inside a triangle (uncircled red and orange dots), or in the interior of an edge (circled red and orange dots). Therefore, we assign to the orange vertices the average normal of the two adjacent triangles, and to red the pillar of the connected orange one.



Additionally, the edges connecting singularities will always be beveled/split after beveling. Therefore no prism will contain more than one singular point. We discuss the technical extensions for our proofs to shells with pinched prisms in Appendix **??**.

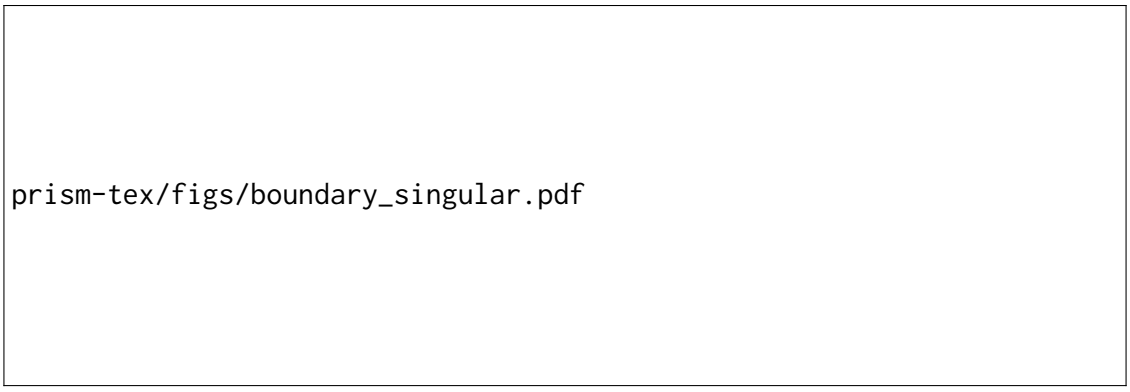


Figure 3.14: AA' is a direction with positive dot product with respect to all its neighboring faces. However, no valid shell can be built following that direction.

3.3.6 Boundaries.

We introduced our algorithm, assuming that the input mesh does not have boundaries. We will now extend our construction to handle this case, which requires minor variations to our algorithm.

For some vertices on the boundary, it might be impossible to extrude a valid shell (Figure 3.14), even if problem (3.2) has a solution, as Theorem 3.5 does not apply in its original form. We identify such cases by connecting every edge in the 1-ring neighborhood of the boundary vertex to the extruded point and check if they collide with the existing 1-ring triangles (e.g., the triangle $A'AB$ intersects the existing input triangle in Figure 3.14). If it is the case, we consider this vertex as a singularity, and we pinch the shell. Note that this is an extremely rare case and, in our experiments, we detected it only for models where the loss of precision in the STL export introduces rounding noise on the boundary.

Once we pinch all boundary singularities, our construction extends naturally to the boundary. The only necessary modification is in the shell optimization (Section 3.3.4), where we skip all operations acting on boundary vertices to maintain the bijectivity of the induced projection operator (Figure 3.15). We thus freeze these vertices and never allow them to move or be affected by any other modification of the shell. Note that, in certain applications, it might also be useful to freeze additional non-boundary vertices to ensure that these remain on the middle surface during optimization (e.g., to exactly represent a corner of a CAD model).

3.4 Results

Our algorithm is implemented in C++ and uses Eigen [Guennebaud et al. 2010] for the linear algebra routines, CGAL [The CGAL Project 2020] and Geogram [Lévy 2015] for predicates and spatial searching, and libigl [Jacobson et al. 2016] for basic geometry processing routines. We run our experiments on cluster nodes with a Xeon E5-2690 v2 @ 3.00GHz. The reference implementation used to generate the results is attached to the submission and will be released as an open-source project.

prism-tex/figs/open-hand.pdf

Figure 3.15: An example of a mesh with boundary.

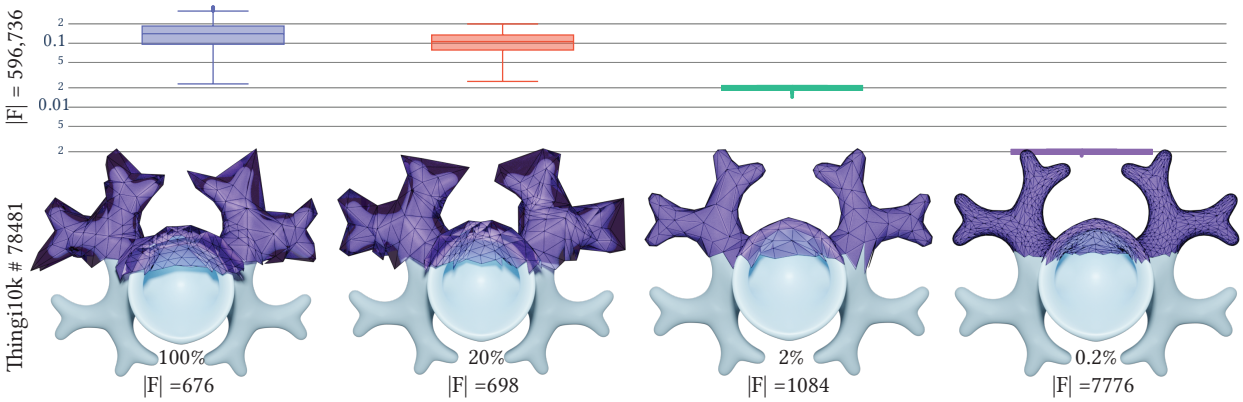


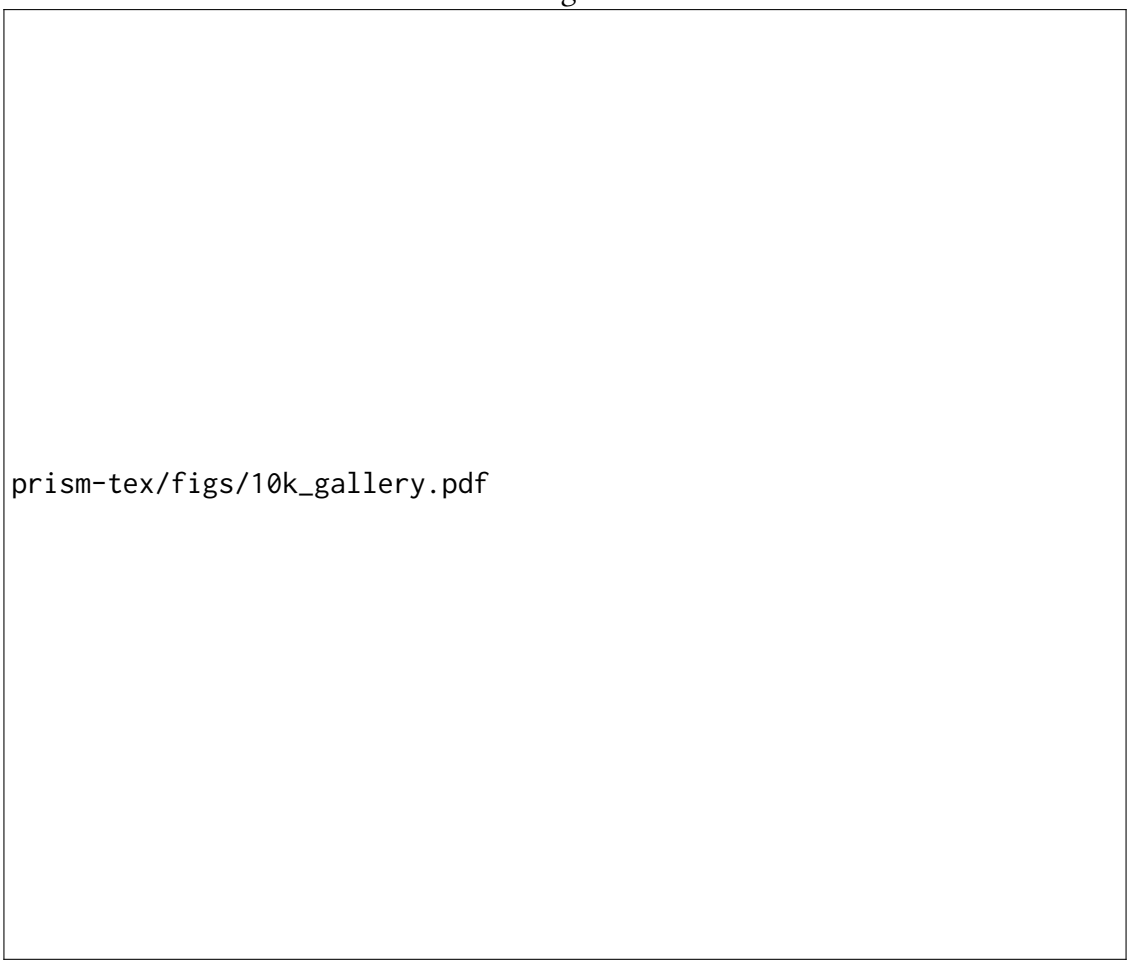
Figure 3.16: The effect of different target thickness on the number of prisms |F| of the final shell, and distribution of final thickness (shown as the box plots on the top).

Robustness For each dataset, we selected the subset of meshes satisfying our input assumptions: intersection-free, orientable, manifold triangle meshes without zero area triangles (tested using a numerical tolerance 10^{-16}). We test self-intersections by two criteria: a ball of radius 10^{-10} around each vertex does not contain non-adjacent triangles; and all the dihedral angles are larger than 0.1 degrees.

We tested our algorithm on two datasets: (1) Thingi10k dataset [Zhou and Jacobson 2016] containing, after the filtering due to our input assumptions, 5018 models; and (2) the first chunk of for the ABC dataset [Koch et al. 2019] with 5545 models. The only user-controlled parameter of our algorithm is the target thickness of our shell; in all our experiments (unless stated otherwise), we use 10% of the longest edge of the bounding box. In Figure 3.16, we show how the target thickness influences the usage of the shell: a thicker shell provides a larger class of sections, thus accommodates more processing algorithms, while a thinner one offers a natural bound on the geometric fidelity of the sections.

Our algorithm successfully creates shells for all 5018 models for Thingi10k and 5545 for ABC. We show a few representative examples of challenging models for both datasets in Figure 3.17,

Thingi10k



ABC

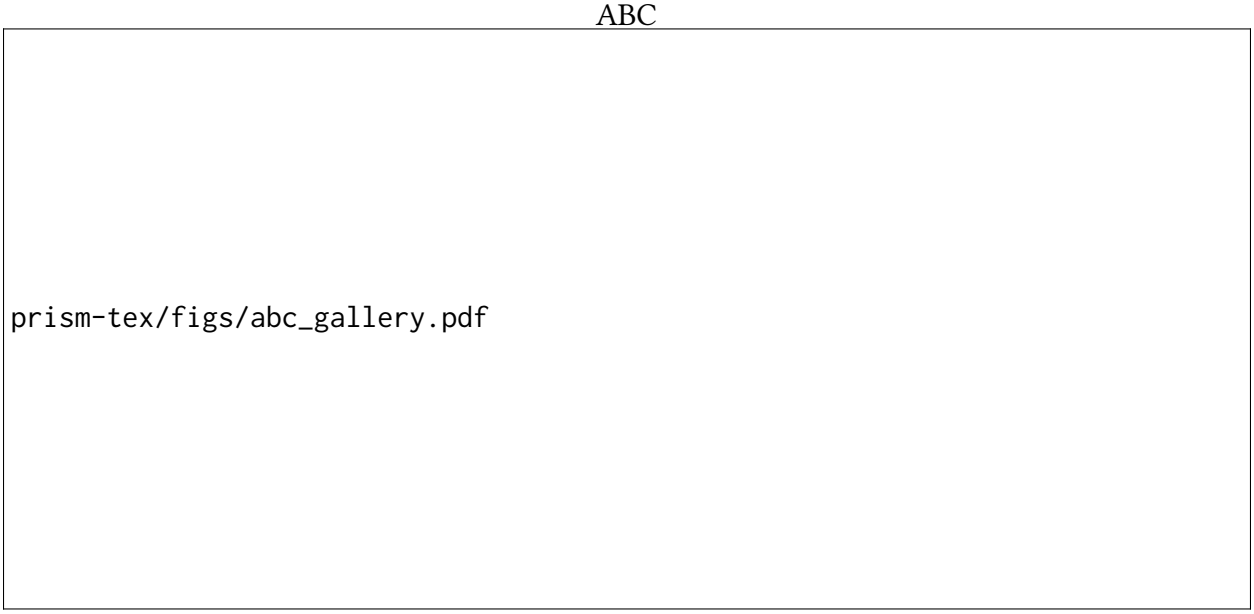


Figure 3.17: Gallery of shells built around models from Thingi10k [Zhou and Jacobson 2016] and ABC [Koch et al. 2019].

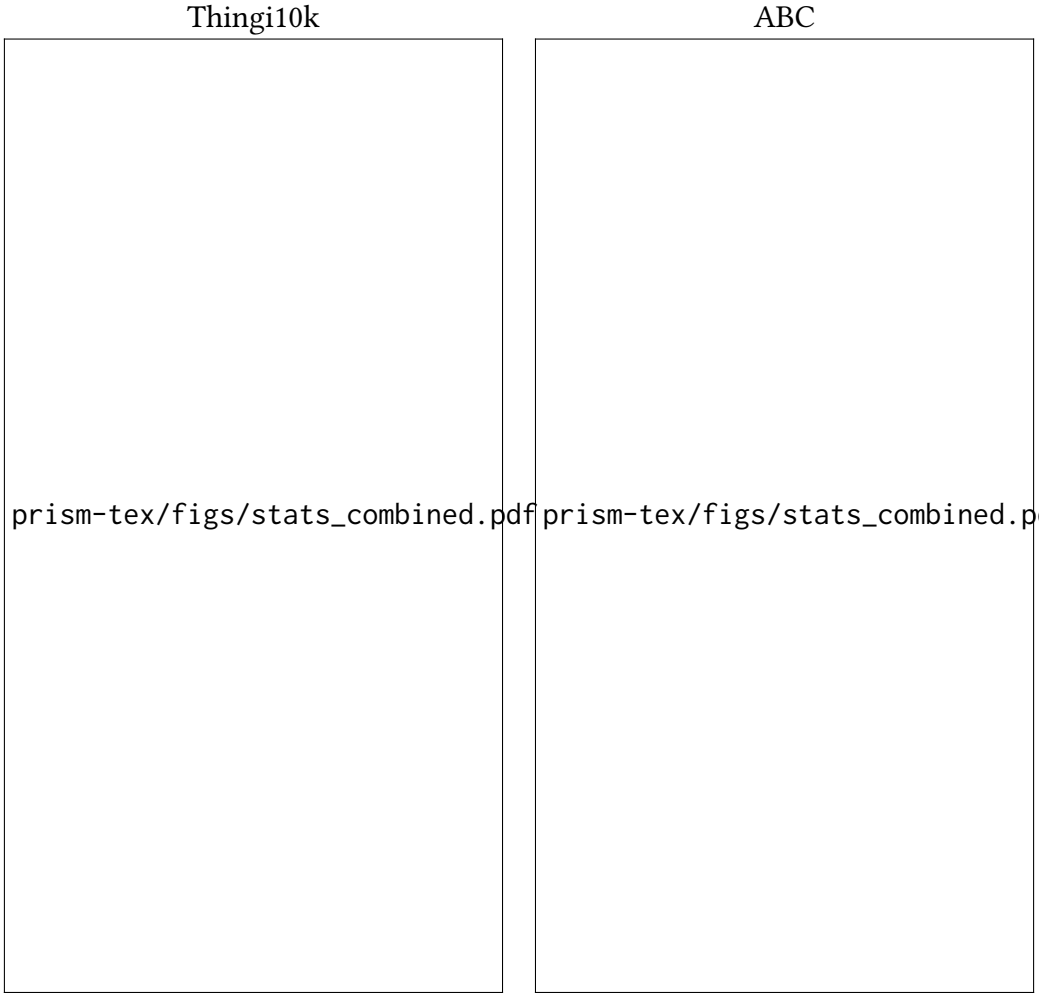


Figure 3.18: Statistics of 5018 shells in Thingi10k dataset [Zhou and Jacobson 2016] (left) and 5545 in ABC Dataset [Koch et al. 2019] (right).

including models with complicated geometric and topological details. In all cases, our algorithm produces coarse and thick cages, with a bijective projection field defined.

We report as a scatter plot the number of output faces, the timing, and the memory used by our algorithm (Figure 3.18). In total, the number of prisms generated by our algorithm is 7% and 2% of the number of input triangles for the Thingi10k and ABC dataset respectively and runs with no more than 4.7 GB of RAM. The generation and optimization of the shell takes 5min and 59s in average and up to 8.6 hours for the largest model. 50% of the meshes finish in 3 minutes and 75% in 6 minutes and 15 seconds.

Comparison to Simple Baselines In Figure 3.19 we compare to two baseline methods based on [Garland and Heckbert 1998]. For each method, we generate a coarse mesh, uniformly subdivide it for visualization purposes, and query the corresponding spatial position on the original input

Figure 3.19: The *UV based method* cannot simplify the prescribed seams, and introduces self-intersections. The projection induced by the *Naive Cage* method is not continuous and not bijective; it leads to visible spikes in the reconstructed geometry.

to form the subdivided mesh. The *UV based* method is a conventional way of establishing correspondence in the context of texture mapping. However, the robust generation of a UV atlas satisfying a variety of user constraints is still an open problem. We use the state-of-the-art methods [Li et al. 2018; Jiang et al. 2017] to generate a low-distortion bijective parametrization, and use seam-aware decimation technique [Liu et al. 2017] to generate the coarse mesh. Due to the complex geometry and the length of the seam (Figure 3.19 second figure), the simplification is not able to proceed beyond the prescribed seam while maintaining the bijectivity, making the pipeline inadequate especially for building computational domains.

We also set up a baseline of the *Naive Cage* method by creating a simplified coarse mesh with [Garland and Heckbert 1998] and use Phong projection to establish the correspondence [Kobbelt et al. 1998; Panozzo et al. 2013]. Such attribute transfer is not guaranteed to be bijective; some face may not be projected (Figure 3.19 third image). With our method, we can generate a coarse mesh while having a low-distortion bijective projection (Figure 3.19 fourth image).

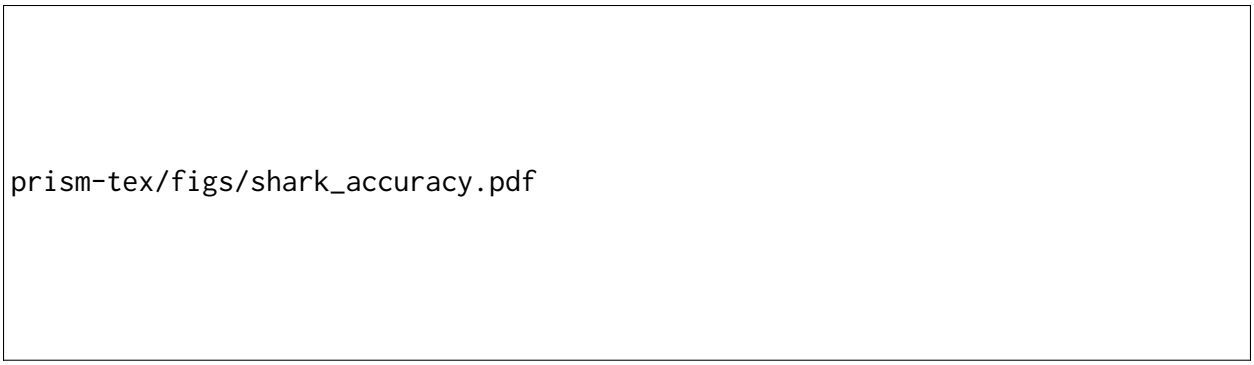


Figure 3.20: Our projection is three orders of magnitude more accurate than the baseline method, and bijectively reconstructs the input vertex coordinates.

Numerical accuracy To evaluate the numerical error introduced by our projection operator when implemented with floating-point arithmetic, we *transfer* the vertices of the input mesh to the middle surface and *inverse transfer* them from the middle surface back to the input mesh. We measure the Euclidean distance with respect to the source vertices (Figure 3.20). We compare the same experiment with the Phong projection [Kobbelt et al. 1998]. This alternative approach exhibits distance errors up to 10^{-5} even after ruling out the outliers for which the method fails due to its lack of bijectivity. The maximal error of our projection is on the order of 10^{-8}; this error could be completely eliminated (for applications requiring an exact bijection) by implementing the projection operator and its inverse using rational arithmetic.

3.5 Applications

Using our shell $\mathcal{S}$, we implement the following predicates and functions:

- is_inside(p): returns true if the point $p \in \mathbb{R}^3$ is inside $\mathcal{S}$.
- is_section($\mathcal{T}$): returns true if the triangle mesh $\mathcal{T}$ is a section of $\mathcal{S}$.
- $\mathcal{P}(p)$: returns the prism id (pid), the barycentric coordinates (α, β) in the corresponding triangle of the middle surface, and the relative offset distance from the middle surface (h, which is -1 for the bottom surface, and 1 for the top surface) of the projection of the point p.
- $\mathcal{P}^{-1}(\text{pid}, \alpha, \beta, h)$ is the inverse of $\mathcal{P}(p)$.
- $\mathcal{P}_{\mathcal{T}}(\text{tid}, \alpha, \beta) = \mathcal{P}(q)$, where q is the point in the triangle tid of the mesh $\mathcal{T}$, with barycentric coordinates α, β.
- $\mathcal{P}_{\mathcal{T}}^{-1}(\text{pid}, \alpha, \beta)$ is the inverse of $\mathcal{P}_{\mathcal{T}}$.

As explained in Section 3.3, our shell may self-intersect and we opted to simply exclude the overlapping regions. In practice this affects only the function is_inside(p) which needs to check if p is contained in two or more non-adjacent prisms.

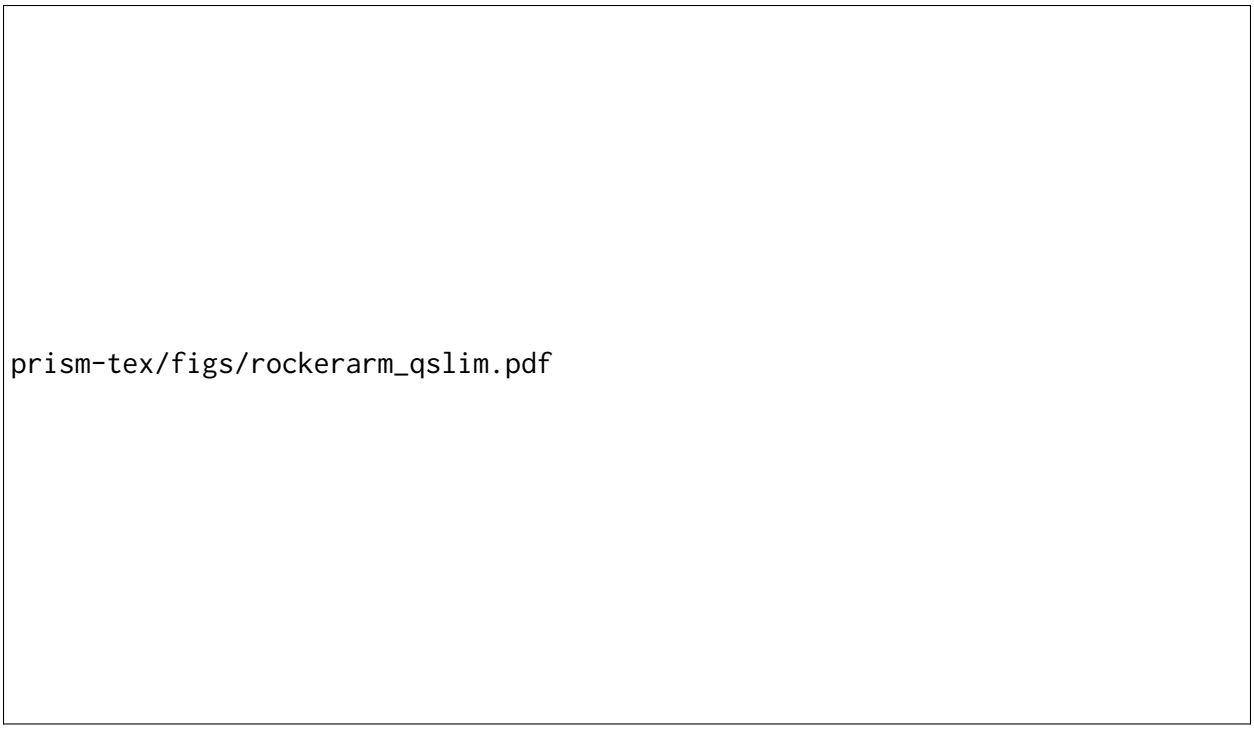


Figure 3.21: We attempt to simplify *rockerarm* (top left) from 20088 triangles to 100. QSlim [Garland and Heckbert 1998] succeeds in reaching the target triangle count (top right) but generates an output with a self-intersection (red) and flipped triangles (purple). With our shell constraints (bottom left), the simplification stagnates at 136 triangles, but the output is free from undesirable geometric configurations. Note that both examples use the same quadratic error metric based sceduling [Garland and Heckbert 1998].

These functions are sufficient to implement all applications below, demonstrating the flexibility of our construction and how easy it is to integrate in existing geometry processing workflows.

Remeshing We integrated our shell in the meshing algorithm proposed in [Dunyach et al. 2013] by adding envelope checks ensuring that the surface is a section after every operation. After simplification, we can use the projection operator to transfer properties between the original and remeshed surface (e.g., figures 3.22, 3.23). Since the remeshed surface is a section, a very practical side effect of our construction is that the remeshed surface is *guaranteed to be free of self-intersections*, As shown in Figure 3.21, the constraints enforced through our shell prevents undesirable geometric configurations (intersections, pockets, or triangle flips).

Proxy A particularly useful application of our shell is the construction of proxy domains for the solutions of PDEs on low-quality meshes. Additionally, by specifying the target thickness parameter (Figure 3.16), we are able to bound the geometry approximation error to the input as well. Using our method, we can (1) convert a low-quality mesh to a proxy mesh with higher quality and desired density, (2) map the boundary conditions from the input to the proxy using

Figure 3.22: The heat method (right top) produces inaccurate results due to a poor triangulation (left top). We remesh the input with our method (left bottom), compute the solution of the heat method [Crane et al. 2013] on the high-quality mesh (middle bottom) and transfer the solution back to the input mesh using the bijective projection (bottom right). This process produces a result closer to the exact discrete geodesic distance [Mitchell et al. 1987] (top middle, error shown in the histograms).

the bijective projection map, (3) solve the PDE on the proxy (which is a standard mesh), and (4) transfer back the solution on the input surface (Figure 3.22). Our algorithm can be directly used to solve volumetric PDEs. by calling an existing tetrahedral meshing algorithm between steps 2 and 3 (figures 3.1, 3.23). In this case, we control the geometric error by setting the target thickness (we use 2% of the longest edge of the bounding box).

Boolean Operations The mesh arrangements algorithm enables the robust and exact (up to a final floating-point rounding) computation of Boolean operations on PWN meshes [Zhou et al. 2016]. However, the produced meshes tend to have low triangle quality that might hinder the performance of downstream algorithms. By interleaving a remeshing step performed with our algorithm after every operation, we ensure high final quality and more stable runtime. The

Figure 3.23: *Knot* simplified with our method and tetrahedralized with TetGen. The original model is converted to 1,503,428 tetrahedra (left) while the simplified surface is converted to only 176,190 tetrahedra (right).

composition of the bijections enables us to transfer properties between different nodes of the CSG tree (Figure 3.24).

Displacement Mapping The middle surface of the shell is a coarse triangular mesh that can be directly used to compress the geometry of the input mesh, storing only the coarse mesh connectivity and adding the details using normal and displacement maps (Figure 3.25). A common way to build such displacement is to project [Kobbelt et al. 1998; Collins and Hilton 2002] the dense mesh on the coarser version. As shown in Appendix **??**, our method guarantees that this natural projection is also bijective as long as the coarse mesh is a section. This alleviates the loss of information even on challenging geometry configurations, and our shell can thus be used to automate the creation of projection cages and displacement maps.

Geometric Textures. The inverse projection operator provides a 2.5D parametrization around a given mesh and can be used to apply a volumetric texture (Figure 3.26). Note that we build the volumetric texture on the simplified shell, while still being able to bijectively transfer the texture coordinates.

3.6 Variants

Input with Self-Intersections Up to this point, we assumed that our input meshes are without self-intersections. This requirement is necessary to guarantee a bijection between any section (e.g.,

Figure 3.24: The union of two meshes is coarsened through our algorithm, while preserving the exact correspondence, as shown through color transfer.

the input mesh) and the middle surface. Such bijection is essential for a key target application, the transfer of boundary conditions for solving PDEs on meshes or mesh-bounded domains.

However, our method can be easily extended to meshes containing self-intersections, broadening the class of meshes it can be applied to, at the cost of making the resulting shell usable in fewer application scenarios: for example, if it is used for remeshing, it will likely generate a new surface that still contains self-intersections.

If $\mathcal{T}$ contains self-intersections, our algorithm can be trivially extended to generate a shell which will be *locally injective*, and the bijectivity of the mapping between sections still holds but with respect to the immersion. The only change required is to modify the invariance checks (Section 3.3.4): we have to replace the global intersection check with checking whether local triangles overlap with the current prisms. Figure 3.27 shows an example of a mesh $\mathcal{T}$ with self-intersections, the generated shell, and the isolines of geodesic transferred on the coarser middle surface.

Figure 3.25: Top: an input mesh decimated to create a coarse base mesh, and the details are encoded in a displacement map (along the normal) with correspondences computed with Phong projection [Kobbelt et al. 1998]. Bottom: the decimation is done within our shell while using the same projection as above. With our construction, this projection becomes bijective (Appendix **??**), avoiding the artifacts visible on the ears of the bunny in the top row.

Figure 3.26: Two volumetric chainmail textures (right) are applied to a shell (bottom left) constructed from the *Animal* mesh (top left). The original UV coordinates are transferred using our projection operator.

Figure 3.27: An example of a shell built around a self-intersecting mesh.

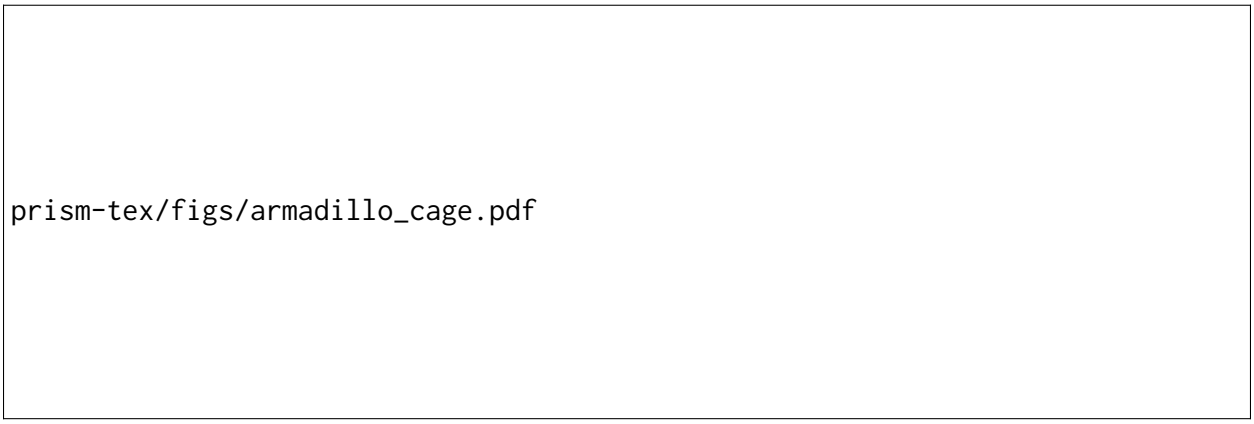


Figure 3.28: The *Armadillo* model with four nested cages. We create a shell from the original mesh, and then rerun our algorithm on the outer shell to create the other three layers. Note that all layers are free of self-intersections, and we have an explicit bijective map between them.

Resolving Shell Self-Intersections For certain applications it might be preferable to have a shell whose top and bottom surfaces do not self-intersect: for example, in the construction of nested cages [Sacht et al. 2015] (useful for collision proxies and animation cages), we want to iteratively build nested shells while ensuring no intersections between them (Figure 3.28). With a small modification, our algorithm can be used to generate nested cage automatically and robustly, with the additional advantage of being able to map any quantity bijectively across the layers and to the input mesh. In contrast, [Sacht et al. 2015] does not provide guarantees on the success (e.g., the reference implementation of [Sacht et al. 2015] fails on Figure 3.19, probably due to the presence of a singularity).

To resolve the self-intersections of the top (bottom) surface, we identify the regions covered by more than one prism by explicitly testing intersections between the tetrahedralized prisms, accelerated using [Zomorodian and Edelsbrunner 2000]. For every detected prism, we reduce the thickness by 20%, and iterate until no more intersections are found. Differently from the procedure in Section 3.3.3, where reducing the thickness of the shell always maintains the validity of the shell, at this stage, the shrinking of the shell may make the shell invalid, since $\mathcal{T}$ may not be contained

Figure 3.29: After optimization, the shell may self-intersect (left). Our postprocessing can be used to extract a non-selfintersecting shell, which is easier to use in downstream applications (right).

Figure 3.30: We generate a pinched shell for a model with a singularity (left). Optionally, we can complete the shell using our Boolean construction.

anymore in $\mathcal{S}$. Whenever this happens, we perform one step of red-green refinement [Bank et al. 1983] on the regions we wish to thin, and we iterate until we succeed. This procedure is guaranteed to terminate since, on the limit of the refinement, the middle surface will be geometrically identical to $\mathcal{T}$, and thus Theorem 3.5 holds. In the worst case, the procedure terminates when the size of triangles on the middle surface is comparable to the input; then no refinement is required to shrink below the minimum separation of the input. Figure 3.29 shows how the intersecting shell between the legs of the camel can be shrunk to generate an intersection-free shell.

Pinching Alternative. For certain applications, such as boundary layer meshing, it is necessary to have a shell with non-zero thickness everywhere, including at singularities, and it is tolerable to lose bijectivity at the vicinity of a singularity. For these cases, we propose a Boolean construction to *fill* the shell around singularities, and to extend the projection operator $\mathcal{P}$ inside these regions. That is, every point in the filled region will project to the singularity.

Without loss of generality, let us assume that $\mathcal{T}$ has a single singularity (Figure 3.30). We initially construct a pinched shell, with zero thickness at the singularity, construct a valid shell (Section 3.3), and then perform a corefinement [Loriot et al. 2020] between a tetrahedron (centered at the singularity and whose size is smaller than the minimal thickness of the neighboring vertices) and the shell. The result of the corefinement operation (Figure 3.30 middle) consists of triangles belong to the tetrahedron, or the shell surface. The remaining part of the tetrahedron is a star-shaped polyhedron with the singularity in its kernel, and sharing a part of its boundary with the shell. This polyhedron can be easily tetrahedralized by connecting its triangulated boundary faces (one of them is highlighted in red in Figure 3.30 middle) with the singularity. For every point p in these tetrahedra, the projection operator $\mathcal{P}$ projects p to the singularity. The remaining triangles are divided into two groups: the triangles with only one new vertex complete the degenerate prisms (one of them is highlighted in blue in Figure 3.30 middle), while the others map to the edges they are attached to.

3.7 Limitations

Currently, our algorithm is limited to manifold and orientable surfaces: its extension to non-manifold and/or non-orientable meshes is a potential venue for future work. With such an extension, the integration of shells with robust tetrahedral meshers [Hu et al. 2018, 2019b] would allow to solve PDEs on imperfect triangle meshes without ever exposing the user to the volumetric mesh, allowing them to directly work on the boundary representation to specify boundary conditions and to analyze the solution of the desired PDE. Since we rely on the additional checks for the bijective constraints, our method is slower than classical surface mesh adaptation algorithms and it is not suitable for interactive applications.

Integrating our approach into existing mesh processing algorithms might lose some of their guarantees or properties since our shell might prevent some local operations. Other surface processing algorithms guarantee some properties under some regularity assumptions on the input, which might not hold when our bijective constraints are used. For example, QSlim [Garland and Heckbert 1997] might not be able to reach the desired target number of vertices and [Dey and Ray 2010] might not be able to achieve the bounded aspect ratio. A practical limitation is that integrating our approach into existing remeshing or simplification implementations requires code level access.

Our shell is ideal for triangle remeshing algorithms employing incremental changes: not every geometry processing algorithm requiring a bijective map can use our construction. For example, it is unclear how isosurface-extraction methods [Hass and Trnkova 2020] could use our shell or how global parametrization algorithms [Kraevoy and Sheffer 2004; Schreiner et al. 2004; Alliez et al. 2003; Bommes et al. 2013] could benefit from our method since they already compute a map

to a common domain.

3.8 Concluding Remarks

We introduce an algorithm to construct shells around triangular meshes and define bijections between surfaces inside the shell. We proposed a robust algorithm to compute the shell, validated it on a large collection of models, and demonstrated its practical applicability in common applications in graphics and geometry processing.

We believe that many applications in geometry processing could benefit from bijectively mapping spatially close surfaces, and that the idea of using an explicit mesh as a common parametrization domain could be extended to the more general case of computing cross-parametrizations between arbitrary surfaces. To foster research in this direction, we will release our reference implementation as an open-source project.

4 Bijective and Coarse High-Order Tetrahedral Meshes

4.1 Introduction

Piecewise linear approximations of surfaces are a popular representation for 3D geometry due to their simplicity and wide availability of libraries and algorithms to process them. However, dense sampling is required to faithfully approximate smooth surfaces. Curved meshes, that is, meshes whose element's geometry is described as a high-order polynomial, are an attractive alternative for many applications, as they require fewer elements to achieve the same representation accuracy of linear meshes. In particular, curved meshes have been shown to be effective in a variety of simulation settings in mechanical engineering, computational fluid dynamics, and graphics. Despite their major benefits, they are not as popular as linear meshes: We believe that one of the main reason for their limited usage is the lack of an automatic, robust way of constructing them.

While robust meshing algorithm exists for volumetric linear tetrahedral meshing, there are few algorithms for curved meshes, and even fewer of them having either a commercial or open-source implementation (Section 4.2). Only a few algorithms work directly on arbitrary triangle meshes (most of them require the input geometry to be either a CAD file or an implicit function), and

curve_meshing_in_shell_tex/figs/teaser.pdf

Figure 4.1: Our pipeline starts from a dense *linear* mesh with annotated features (green), which is converted in a curved shell filled with a high-order mesh. The region bounded by the shell is then tetrahedralized with linear elements, which are then optimized. Our output is a coarse, yet accurate, curved tetrahedral mesh ready to be used in FEM based simulation. Our construction provides a bijective map between the input surface and the boundary of the output tetrahedral mesh, which can be used to transfer attributes and boundary conditions.

none of them can reliably process a large collection of 3D models.

We propose the first robust and automatic algorithm to convert dense piecewise linear triangle meshes (which can be extracted from scanned data, volumetric imaging, or CAD models) into coarse, curved tetrahedral meshes equipped with a bijective map between the input triangle mesh and the boundary of the tetrahedral mesh. Our algorithm takes advantage of the recently proposed bijective shell construction [**?**] to allow joint coarsening, remeshing, and curving of the dense input mesh, which we extend to support feature annotations. Our outputs are guaranteed to have a self-intersection free boundary and the geometric map of every element is guaranteed to be bijective, an important requirement for FEM applications.

The key ingredient of our algorithm is the separation of the curved volumetric meshing problem into a near-surface, surface curving problem, followed by a restricted type of linear volumetric meshing.

We believe that our curved meshing algorithm will enable wider adoption of curved meshes, as it will provide a way to automatically convert geometric data in multiple formats into a coarse tetrahedral mesh readily usable in finite element applications. To showcase the benefits of our approach we study 2 settings: (1) we show that a coarse proxy mesh can be used to compute non-linear deformations efficiently and transfer them onto a high-resolution geometry, targeting real-time simulation (Figure 4.1), and (2) we show that our meshes are ready to use in downstream FEM simulations.

We validate the reference implementation of our approach on a large collection of more than 8000 geometrical models, which will be released as an open-source project to foster the adoption of curved meshes in academia and industry.

Our contributions are:

- An algorithm to convert dense piecewise linear meshes into coarse curved volumetric meshes while preserving a bijective map with the input and bounding the approximation error.
- An extension of the bijective shell construction algorithm to support feature annotations.
- An algorithm for conforming tetrahedral meshing without allowing refinement on the boundary, but allowing internal Steiner points.
- A large-scale dataset of high-order tetrahedral meshes.

4.2 Related Works

We review the literature on the generation of unstructured and structured curved meshes. We also review the literature on boundary-preserving linear meshing, as it is an intermediate step of our algorithm.

4.2.1 Curved Tetrahedral Mesh Generation

High-order meshes are used in applications in graphics [???] and engineering analysis [?] where it is important to reduce the geometric discretization error [?????], while using a low number of degrees of freedom. The creation of high-order meshes is typically divided into three steps: (1) linear meshing of the smooth input surface, (2) curving of the linear elements to fit the surface, and (3) optimization to heal the elements inverted during curving. We first cover steps 2 and 3, and postpone the overview of linear tetrahedral algorithms to Section 4.2.3.

Direct methods. Direct methods are the simplest family of curving algorithms, as they explicitly interpolate a few points of the target curved surface or project the high-order nodes on the curved boundary [???????]. The curved elements are represented using Lagrange polynomials, [??], quadratic or cubic Bézier polynomials [???], or NURBS [??]. [??] further optimizes the high-order node distribution according to geometric quantities of interest, such as length, geodesic distance, and curvature. In the case where no CAD information is available, [?], [?] use smooth reconstruction to compute high-order nodes and perform curving.

Deformation methods Deformation methods consider the input linear mesh as a deformable, elastic body, and use controlled forces to deform it to fit the curved boundary. Different physical models have been employed such as linear, [????], and (variants of) non-linear elasticity [???]. A comparison between different elasticity and distortion energies is presented in [???].

Direct and deformation methods have been tested on small collections of simple models, and, to the best of our knowledge, none of them can provide guarantees on the validity of the output or has been tested on large collections of models. There are also no reference implementations we could compare against.

Inversions and Intersections. Most of these methods introduce inverted elements during the curving of the high-order elements. Inverted elements can be identified by extending Jacobian metrics for linear elements [??] to high-order ones [??Johnen et al. 2013; ?; ?]. Various untangling strategies have been proposed, including geometric smoothing and connectivity modifications [??????????????????????????]. None of these techniques can guarantee to remove the inverted elements.

An alternative approach is to start from an inversion-free mesh and slowly deform it [??], explicitly avoiding inversions at the cost of possibly inaccurate boundary reproductions. These methods cannot however guarantee that the boundary will not self-intersect. Our approach follows a similar approach but uses a geometric shell to ensure element validity and prevention of boundary self-intersections.

Curved Optimal Delaunay Triangulation [?] generalize optimal delaunay triangulation paradigm to the high-order setting, through iteratively update vertices and connectivity. Their algorithm starts with a point cloud sampled from triangle meshes. However, the success of the method depends on the choice of final vertex number and sizing field, where insufficient vertices may result in broken topology or invalid tetrahedral meshes.

Software Implementation. Despite the large literature on curved meshing generation, there are very few implementations available.

Nektar [?] is a finite element software with a meshing component, which can generate high-order elements. We do not explicitly compare as their documentation (Section 4.5.1.5 Mesh Correction[1]) states that the algorithm is not fully automatic and not designed to process robustly large collections of models. Gmsh [?] is an open source software that supports the curved meshing of CAD models, but it does not support dense linear meshes as input. Despite the difference in the input type, we provide a comparison with Gmsh in Section 4.6.2, as it is the only method that we could run on a large collection of shapes.

To the best of our knowledge, the commercial software that support curved meshing (Pointwise [??]) are also requiring a CAD model as input.

Animation Curved tetrahedral meshes have also gained popularity in the context of fast animation. With fewer degrees of freedom and preserved geometric fidelity, [?] observe the benefit of quadratic tetrahedra in the pipeline of physically based animation. [?] further investigate the transfer problem when using curved meshes as a proxy.

4.2.2 Curved Structured Mesh Generation

The use of a hexahedral mesh as a discretization for a volume, allows to naturally define C^k splines over the domain, which can be used as basis functions for finite element methods: this idea has been pioneered by IsoGeometric Analyisis (IGA), and it is an active research area. The generation of volumetric, high-order parametrizations that conform to a given input geometry is an extremely challenging problem [??]. Most of the existing methods rely on linear hexahedral mesh generation, which is on its own a really hard problem for which automatic and robust solutions to generate coarse meshes are still elusive [??????] due to the inherently global nature of the problem. The current state of the art for IGA meshing is a combination of manual decomposition of the volume and semi-automated geometrical fitting [??].

In contrast, our approach is automatic, i.e. can automatically process thousands of models without any manual intervention, while providing explicit guarantees on both the validity of the elements and the maximal geometric error. Its downside is the C^0 continuity of the basis on the elements' interfaces. However, it is unclear to us if this is a real limitation: in many common settings in computer graphics and mechanical engineering (Poisson problems, linear and non-linear elasticity) the higher smoothness offered by spline functions does not make noticeable difference experimentally [?] (and it actually leads to much worse conditioning of the system matrix, which is problematic for iterative solvers), and we thus believe that curved tetrahedral meshing is a very exciting alternative as it dramatically simplifies both the meshing and fitting of high-order elements.

[1] `https://doc.nektar.info/userguide/5.0.0/user-guidese17.html`

4.2.3 Boundary Preserving Tetrahedral Meshing

We refer to [Hu et al. 2018] for a detailed overview of linear tetrahedral meshing, and we focus here only on the techniques that target boundary preserving tetrahedral meshing.

The most popular linear tetrahedral meshing methods are based on *Delaunay refinement* [???], i.e. the insertion of new vertices at the center of the circumscribed sphere of the worst tetrahedron in term of radius-to-edge ratio. This approach is used in the most popular tetrahedral meshing implementations [??], and, in our experiments, proved to be consistently successful as long as the boundary is allowed to be refined. A downside of these approaches is that a 3D Delaunay mesh, unlike the 2D case, might still contain "sliver" tetrahedra, thus requiring mesh improvement heuristics [????]. [?] discusses this issue in detail and provides a different formulation to avoid it without the use of a postprocessing. [?] introduces an approach that does not allow insertion of Steiner points, making it not suitable for generic polyhedra domains.

There are many variants of Delaunay-based meshing algorithms including *Conforming Delaunay tetrahedralization* [??], *constrained Delaunay tetrahedralization* [????], and *Restricted Delaunay tetrahedralization* [???].

To the best of our knowledge, all these methods are designed to allow some modifications of the input surface (either refinement, resampling, or approximation). One exception is the constrained Delaunay implementation in TetGen [?] that allows disabling any modification to the boundary. However, this comes at the cost of much lower quality and potential robustness issues, as we show in Appendix **??**.

A different tetrahedral meshing approach has been proposed in [Hu et al. 2018], and its variants [??], where the problem is relaxed to generate a mesh that is close to the input to increase robustness. However, these approaches are not directly usable in our setting, as we require boundary preservation.

Due to these issues, we propose a novel boundary-preserving tetrahedral meshing algorithm specifically tailored for the shell mesh generated by our curved meshing algorithm.

4.2.4 Curved Surface Fitting

There are many algorithms for fitting curved *surfaces* to dense 3D triangle meshes. The most popular approaches fit spline patches, usually on top of a quadrangular grid. Since generating quadrilateral meshes is a challenging problem for which robust solutions do not exist yet, we refer to [?] for an overview, and only review in this section algorithms for unstructured curved mesh generation, which are more similar to our algorithm. We note that the focus of our paper is volumetric meshing: while we generate an intermediate curved surface mesh, this is not the goal of our algorithm, especially since the generated surface is only C^0 on edges.

[?] fits a smooth surface represented by a point cloud to a curved triangle mesh based on a subdivision surface scheme and an interleaving mesh simplification and fitting pipeline that preserves sharp features. The algorithm does not provide explicit correspondence to the input: they are defined using distance closest point, which is not bijective far from the surface.

[?] converts dense irregular polygon meshes of arbitrary topology into coarse tensor product B-spline surface patches with accompanying displacement maps. Based on the work [?] that

fits triangle surface meshes with Bézier patches, [?] fits triangle surface meshes with high-order B-spline quadrilateral patches and adaptively subdivide the patches to reduce the fitting error. These methods produce smooth surfaces but do not have feature preservation.

Another related topic is the definition of smooth parametric surfaces interpolating triangle meshes. We refer to [?] for an overview of subdivision methods and discuss here the approaches closer to our contribution.

[?] proposed to use triangular Bézier patches to define smooth surfaces over arbitrary triangle meshes ensuring tangent plane continuity by relaxing the constraint of the first derivatives at the input vertices. Following Hahmann's work, [?] presents a more complete pipeline: perform QEM simplifications, trace the coarse mesh onto the dense one and perform parameterization relaxing and smoothing. Then it fits a hierarchical triangular spline [?] to the surface. More recent work [?] approximates the triangulation of an implicit surface with a G^1 surface. These schemes are usually designed to interpolate existing meshes rather than simplifying a dense linear mesh into a coarse curved mesh and are thus orthogonal to our contribution.

4.3 Shell Preliminaries

We briefly overview [?] as our work uses and extends it. [?] introduces bijective projection shells (which we will abbreviate as shells in the rest of the paper), a new geometry processing tool to perform mesh editing while preserving a close-by bijective map to the input surface.

Shell. The shell is defined by three trianglulated surface meshes $\overline{\mathcal{S}} = \{(B_s, V_s, T_s), F_s\}$ sharing the same mesh connectivity F_s, where B_s, V_s, and T_s are the vertices of the bottom, middle, and top surface respectively. Each triangle in F_s corresponds to a *generalized prism*, defined by connecting the corresponding triangles in the three surfaces with straight edges, called *pillars*. Each prism P has three vertices $v_i \in V_s, t_i \in T_s, b_i \in B_s, i = 1, 2, 3$ on the middle, top, and bottom surface, respectively. Each pillar decomposes into a top $h_i^T = t_i - v_i, i = 1, 2, 3$ and bottom $h_i^B = b_i - v_i, i = 1, 2, 3$ slab, and each slab can be canonically decomposed into 3 tetrahedra, and each tetrahedron contains a constant vector field aligned with the pillars it is connected with [?, Figure 4]. The vector field is used to define a *projection operator* Π within the shell.

Projection Operator. For every prism P, the projection operator Π_P is defined as the tracing of the piece-wise constant vector field V inside the decomposed prism, by assigning to each tetrahedron $T_j^P, j = 1, \ldots, 6$, the constant vector field defined by the only edge of T_j^P which is one of the oriented pillars $h_i^T, h_i^B, i = 1, 2, 3$ ([?, Figure 4]). That is, $\forall p \in T_j^P$

$$\Pi_P(p) = h_i^k,$$

where i is the index of the vertex corresponding to the pillar edge of T_j^P, and k is either the top or bottom surface. The shell projection operator Π is defined as the operator whose restriction to P, $\Pi|_P$ is Π_P, and it defines a bijective map between every pair of specific triangle meshes contained in the shell, called *sections*.

Section. A triangle mesh is a section if it is contained within the shell, and if the dot product of the normal of each of its triangles with the vector field in each of the overlapping tetrahedra is positive. The projection operator Π defines a bijective map between any pair of sections if the shell is *valid*.

Shell Validity. [?] defines a shell $\overline{\mathcal{S}}$ to be *valid* with respect to an input mesh $\mathcal{M}$ if it satisfies two conditions:

1. The volumes of all possible tetrahedral decomposition of a prism (24 of them) are positive.
2. $\mathcal{M}$ is a section for all possible tetrahedral decompositions. That is, the input mesh is contained within the shell, and the dot product between the mesh's normals and the shell's pillar is positive.

Singularity. Singular vertices are a special geometric configuration, where the neighboring triangles of a specific vertice admits a conflicting set of normals. [?] extends the shell construction to allow such cases. Around the (isolated) singular vertices, the prisms are pinched to become generalized pyramids, composed of two tetrahedra instead of three.

The algorithm to build the shell creates an initial valid extrusion, potentially thin and dense, and then iteratively uses the shell local operations (i.e., vertex smoothing, edge collapse, edge split, and edge flip) [?, Section 3.4] to improve its quality while preserving the validity.

4.3.1 Variation from the Original Algorithm

To extend the shell formulation in [?] to accommodate for feature preservation (Section 4.5), we modify the definition of a valid section [?, Definition 3.1], by relaxing the intersection between a triangle and a prism in the discrete case. That is, we do not consider the prism to be intersecting a triangle if they share only one vertex of the triangle; we also ignore the intersection if they intersect on a feature edge on opposite sides. The bijectivity and validity condition of the shell projection trivially holds.

4.4 Curved Tetrahedral Mesh Generation

Input. The input of our algorithm is a collection of oriented manifold, watertight, self-intersection-free triangle mesh $\mathcal{M} = (V, F)$, and a set of points $p_i \in \mathcal{P}$ (possibly empty) on the surface of $\mathcal{M}$ (Figure 4.2, left) where the distance bound ε is prescribed. The collection $\mathcal{M}$ must be consistently oriented such that it is the boundary of an oriented 3-manifold. A set of edges can also be optionally provided as annotated features (Section 4.5).

Figure 4.2: Input triangle mesh $\mathcal{M}$ and points $\mathcal{P}$. Output curved tetrahedral mesh $\mathcal{T}^k$ and bijective map ϕ^k.

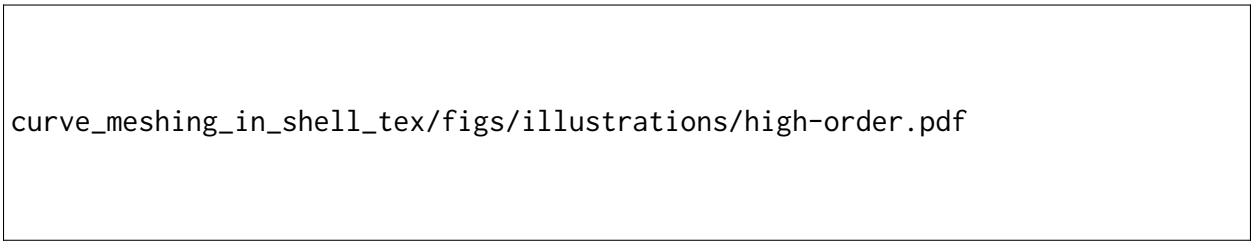


Figure 4.3: Lagrange nodes on the reference element $\hat{\tau}$ for different $k = 1, 2, 3$ and example of geometric mapping g.

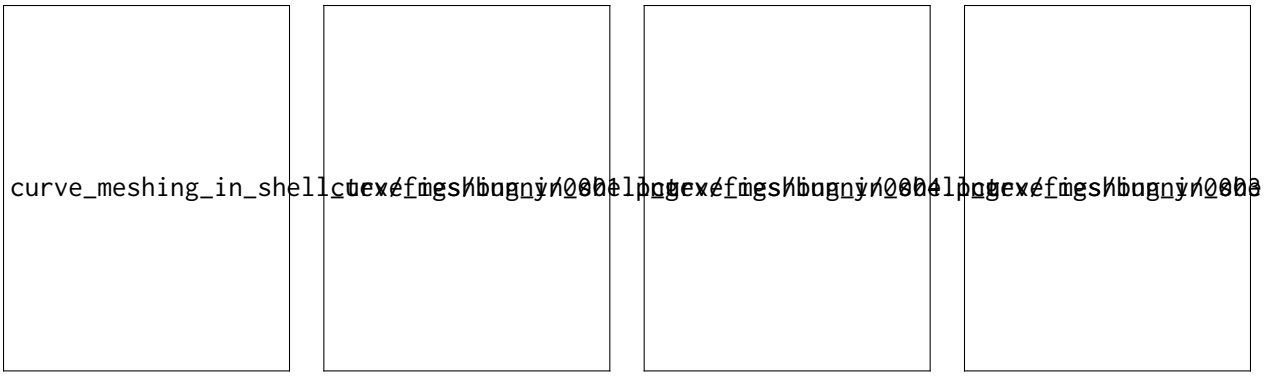
Output. The output of our algorithm is a tetrahedral mesh $\mathcal{T}^k = (V^k, T^k)$ of order k. Formally, each tetrahedron $\tau \in T^k$ is defined through the *geometric map* from the reference tetrahedron $\hat{\tau}$,

$$g^\tau = \sum_{j=1}^{n} c_j^\tau L_j(\hat{u}, \hat{v}, \hat{w}), \tag{4.1}$$

where $\hat{u}, \hat{v}, \hat{w}$ are the local coordinates of a point in $\hat{\tau}$, c_j^τ are the control points for a tetrahedron τ, and L_j are polynomial bases (typically Lagrange bases). For two tetrahedra τ_1 and τ_2 of T^k sharing a face F, the restriction of the maps g^{τ_i}, $i = 1, 2$, to F coincide. Figure 4.3 shows the position of the control points c_j on the reference element for $k = 1, 2, 3$ for the Lagrange bases. We call the tetrahedralization of a curved mesh $\mathcal{T}^k$ *positive* if $\det(J_{g^\tau}) > 0$ everywhere on every τ. In particular, for $k = 1$, since g is affine, J_{g^τ} is constant and the positivity reduces to the positive orientation of the vertices [Shewchuk 1997].

Note that, while bijectivity of the geometric map g^τ implies positivity, the reverse is not true. Therefore, our algorithm not only checks for $\det(J_{g^\tau}) > 0$, but also ensures that the boundary $\partial\mathcal{T}^k$ does not intersect; we show in Appendix **??** that these two conditions guarantee the bijectivity of g^τ. Furthermore, our algorithm ensures that the distance from any point in $\mathcal{P}$ to $\partial\mathcal{T}^k$ (the surface of $\mathcal{T}^k$) is smaller than an user-controlled parameter ε.

We are not assuming anything on $\mathcal{P}$: a sparse set of points will generate a mesh less faithful to the input geometry, while a dense sampling computed for instance with Poisson disk sampling [**?**]



0 points. 200 points. 500 points. 1000 points.

Figure 4.4: Effect of the choice of the set $\mathcal{P}$ on the output.

will prevent the surface from deviating too much (Figure 4.4).

Our algorithm guarantees that the tetrahedralization is positive and that $\partial\mathcal{T}^k$ does not have self-intersections. It also aims at coarsening $\mathcal{T}^k$ as much as possible while striving to obtain a good geometric quality. To reliable fit $\partial\mathcal{T}^k$ to $\mathcal{M}$ we require a bijective map

$$\phi^k : \mathcal{M} \to \partial\mathcal{T}^k$$

from the input $\mathcal{M}$ to the surface of $\mathcal{T}^k$ (Figure 4.2 right). Our algorithm also generates this map and exposes it as an output for additional uses, such as attribute transfer. Note that, since we build upon the shell construction in [**?**], we also guarantee $\partial\mathcal{T}^k$ is homeomorphic and topology-preserving with respect to $\mathcal{M}$ (Figure 4.5).

To simplify the explanation, we use the bar $\overline{\ }$ to represent quantities on the *straight* linear shell, the tilde $\widetilde{\ }$ for the *curved* shell, and hat $\hat{\ }$ for the reference elements (e.g., $\overline{P}$ is the prism on the straight coarse shell, $\widetilde{P}$ is the curved prism, and $\hat{P}$ is the prism on the reference configuration).

Definition 4.1. We call a curved mesh $\mathcal{T}^k$ and its mapping ϕ^k to $\mathcal{M}$ *valid* if it satisfies the following conditions:

1. ϕ^k is bijective;
2. the distance between any $p \in \mathcal{P}$ and $\partial\mathcal{T}^k$ is less than ε;
3. $\mathcal{T}^k$ is positive (i.e., every geometric map g^τ has positive Jacobian's determinant).

Overview. Our algorithm starts by creating a valid mesh (i.e., it satisfies 4.1), then it performs local operations (Appendix **??**) to improve $\mathcal{T}^k$ (i.e., coarsen it and improve its quality) while ensuring all the conditions remain valid with respect to local modification. To achieve this goal, our algorithm uses two stages: (1) curved shell generation and (2) tetrahedral mesh generation and optimization.

Figure 4.5: Our algorithm maintains free of intersection even on challenging models, without the need of setting adaptive threshold.

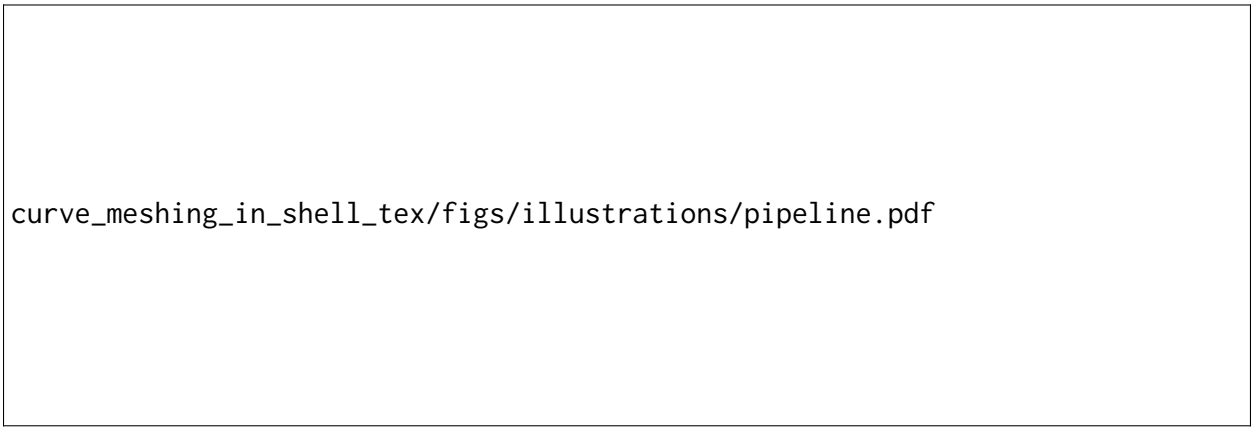


Figure 4.6: Overview of curved mesh generation pipeline.

Stage 1: Curved Shell Construction. In the first stage (Section 4.4.1) we extend the shell construction of [?] by combining the shell projection Π with a high-order mapping

$$\psi^k : \widetilde{\mathcal{S}} \to \overline{\mathcal{S}}.$$

We start form the *valid* shell $\overline{\mathcal{S}}$ constructed from the input mesh $\mathcal{M}$ (i.e., $\mathcal{M}$ is a section $\overline{\mathcal{S}}$). We call $\overline{\mathcal{S}}$ a *projection shell* and call the prismatic projection Π. Together with the construction of $\overline{\mathcal{S}}$, we build an order k *curved prismatic shell* $\widetilde{\mathcal{S}}$ that defines a curved layer around $\mathcal{M}$ and a *bijective* map ψ^k between $\widetilde{\mathcal{S}}$ and $\overline{\mathcal{S}}$ (Figure 4.6, first three figures) that ensures that the distance between $\mathcal{M}$ and $\partial\mathcal{T}^k$ is smaller than ε (Section 4.4.2). That is, $\phi^k(p) < \varepsilon$ for any $p \in \mathcal{P}$. (Note that we do not require $\mathcal{M}$ to be a section of $\widetilde{\mathcal{S}}$.)

To facilitate the volumetric meshing in the next stage (Section 4.4.3), we restrict the top and bottom surface of $\widetilde{\mathcal{S}}$ to be linear (independently from the order of ψ^k). The final output of this first stage is a high-order volumetric shell, a bijective mapping $\phi^k = \Pi \circ \psi^k$, and a *positive* tetrahedralization of $\widetilde{\mathcal{S}}$ with flat boundary. In other words, the tetrahedralization of $\widetilde{\mathcal{S}}$ satisfies 4.1.

Stage 2: Tetrahedral Mesh Generation. In the second stage (Section 4.4.3) we use boundary-conforming tetrahedralization to connect the top and bottom surface of $\widetilde{\mathcal{S}}$ with a background tetrahedral mesh, thus generating a *positive* order k tetrahedralization $\mathcal{T}^k$ of a bounding box around the input, which we can further optimize with local operations to improve its quality (Figure 4.6, last two figures).

To ensure that our first condition is satisfied, we define the mapping ϕ^k as a composition of several mappings, which we ensure are bijective. For the second condition, we initialize our construction with ϕ^k as the identity, and thus, the distance at the sample points is zero. After every operation, we recompute the distance and "undo" the operation if the distance becomes larger than ε. To ensure that the last condition holds, we rely on checking if all prisms (linear and curved) have positive geometric mapping, which ensures that they can be tetrahedralized with a positive tetrahedralization. Ensuring the condition while coarsening $\mathcal{M}$ allows us to generate a coarse *curved* tetrahedral mesh $\mathcal{T}^k$ and the *bijective* map ϕ^k to the input mesh $\mathcal{M}$.

4.4.1 High-order Shells

To simplify the explanation, we first focus on the case where $\varepsilon = \infty$, that is, we aim at generating an as-coarse-as-possible curved mesh. Note that, the trivial solution (i.e., a single tetrahedron) is not necessary a valid $\mathcal{T}^k$ since it would be impossible to build the bijective mapping ϕ^k.

The output of [?] is a coarse shell $\overline{\mathcal{S}}$ with a piecewise linear middle surface. To curve it, we construct a shell $\widetilde{\mathcal{S}}$ and the bijective map ψ^k while constructing $\overline{\mathcal{S}}$. The shell $\widetilde{\mathcal{S}}$ is constructed warping every prism $\widetilde{P}$ of $\widetilde{\mathcal{S}}$ with ψ^k. Since we define ϕ^k as $\Pi \circ \psi^k$, and Π satisfies the first two conditions 4.1, we only need to ensure that ψ^k is bijective, for ϕ^k to be bijective. We define the mapping $\psi^k = \overline{\omega} \circ (\widetilde{\omega}^k)^{-1}$ through two cross-parametrization maps from the reference prism $\hat{P}$:

$$\overline{\omega} : \hat{P} \to \overline{P}, \qquad \widetilde{\omega}^k : \hat{P} \to \widetilde{P}.$$

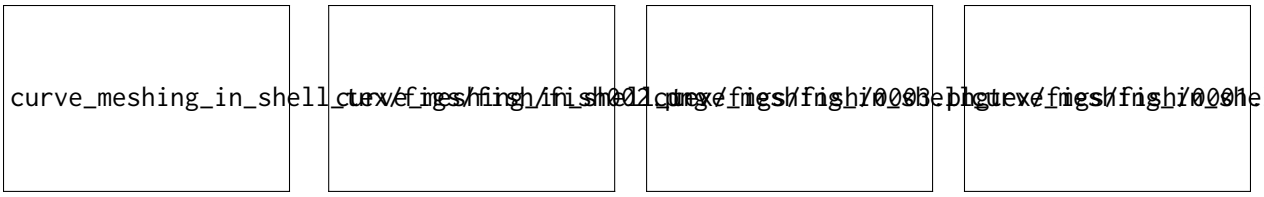


Figure 4.7: A model simplified with different distances.

Both mappings $\overline{\omega}$ and $\widetilde{\omega}^k$ are defined as the tensor product between the base triangular mapping (high-order for $\widetilde{\omega}^k$) and pillar's barycentric heights. That is, for a prism $\widetilde{P}$

$$\widetilde{\omega}^k(\hat{u}, \hat{v}, \hat{h}) = \sum_{j=1}^{n} c_j^P L_j(\hat{u}, \hat{v}) \left((1 - \hat{u} - \hat{v}) h_1^{\widetilde{P}} + \hat{u} h_2^{\widetilde{P}} + \hat{v} h_3^{\widetilde{P}}\right) \hat{h},$$

where $\hat{u}, \hat{v}, \hat{h}$ are the barycentric coordinates in the reference prism, c_j the control points of a triangle, $h_i^{\widetilde{P}}$ the three pillars heights, and L_j a triangle polynomial basis. By ensuring that ψ^k is bijective, we guarantee that any curved tetrahedralization of a prism $\widetilde{P}$ will be a valid tetrahedralization of $\widetilde{\mathcal{S}}$.

We note that to decouple the following tetrahedral mesh generation and the curved shell generation, we ensure that $\widetilde{\omega}^k$ maps the top and bottom face of the curved prism $\widetilde{P}$ to a linear triangle.

After each local operation, we generate samples $\hat{s}_i, i = 1, \ldots, m$ on the parametric base of the prism $\hat{P}$ and use $\Pi^{-1} \circ \overline{\omega}$ to map $\hat{s}_i$ back to $\mathcal{M}$ and $\widetilde{\omega}^k$ to map them to $\partial\mathcal{T}$. Using the mapped point we solve

$$\min_{c_i} \sum_{i=1}^{m} \|(\Pi^{-1} \circ \overline{\omega})(\hat{s}_i) - \widetilde{\omega}[c_i]^k(\hat{s}_i)\|_2^2,$$

where c_i are the control points of $\widetilde{\omega}^k$. As $\widetilde{\omega}[c_i]^k(\hat{s}_i)$ is a linear function of c_i. This is a quadratic optimization problem. The control points of the top and bottom surface are fixed to ensure that $\widetilde{\omega}^k$ maintains the two surfaces as linear. We validate the bijectivity of $\widetilde{\omega}^k$ by checking positivity of the determinant [Johnen et al. 2013] and that the top and bottom surfaces are intersection free. The intersection is simplified in our case since fast and exact algorithms[Guigue and Devillers 2003] are available since the top and bottom surfaces stay linear.

4.4.2 Distance Bound

In the previous section, we explained how to generate a curved shell $\widetilde{\mathcal{S}}$ that satisfies 4.1. To ensure that the middle surface of $\widetilde{\mathcal{S}}$ has a controlled distance from the points in $\mathcal{P}$, we interleave a distance check in the construction of $\widetilde{\omega}^k$ after each local operation. Formally, after every local operation we use the mapping ϕ^k to map every point $p_i \in \mathcal{P}$ to $\widetilde{p}_i = \phi^k(p_i)$ a point on the coarse curved middle surface of $\widetilde{\mathcal{S}}$ and, if $\|p_i - \widetilde{p}_i\| \geq \varepsilon$ we reject the operation. The initial shell is trivially

Figure 4.8: Two dimensional overview of the five steps of our boundary preserving tetrahedral meshing algorithm.

a valid initialization as ϕ^k is identity and thus, the distance is zero. Note that, $\widetilde{p}_i$ is not necessarily the closest point to p_i on $\partial\mathcal{T}$, thus $\|p_i - \widetilde{p}_i\|$ is an upper bound on the actual pointwise distance. Figure 4.7 shows the effect of the distance bound on the surface; a small distance will lead to a denser mesh with more details, while a large one will allow for more coarsening.

4.4.3 Tetrahedral Meshing

The outcome of the previous stage is a curved tetrahedralization of $\widetilde{\mathcal{S}}$ that closely approximates $\mathcal{M}$ with linear ("flat") boundaries. We now consider the problem of filling its interior (and optionally its exterior) with a tetrahedral mesh, a problem known as *conforming boundary preserving* tetrahedralization.

Several solution exists for this problem (Section 4.2.3) and the most common implementation is TetGen [**?**]. Most algorithms refine the boundary, which allows deriving bounds on the quality of the tetrahedral mesh. However, in our setting, this is problematic, as any change will have to be propagated to the curved shell. To avoid coupling the volumetric meshing problem with the curved shell coarsening, while technically possible it is very challenging to implement robustly, we opt for using a tetrahedral meshing algorithm that preserves the boundary exactly. Not many algorithms support this additional constraint, the only one with a public implementation is the widely used TetGen algorithm. However, we discovered that, when this option is used, it suffers from robustness issues, which we detail in Appendix **??**. To solve this problem in our specific setting, we propose in the following five step algorithm (Figure 4.8) taking advantage of the availability of a shell, based on the TetWild [Hu et al. 2018] algorithm.

Step 1. To generate a boundary preserving linear mesh, we first exploit the shell to extrude the bottom surface B (and top T) further by a positive (potentially small) constant δ such that the newly extruded bottom surface B_e (and top T_e) does not self-intersect. The space between B and B_e (and between T and T_e) consists of prisms divided into positive tetrahedra.

Step 2. Then we insert B_e and T_e in a background mesh $\mathcal{B}$ generated following TetWild algorithm ([Hu et al. 2018, Section 3.1]), that is, we use the triangle of B_e and T_e as the input triangle meshes

for the first stage of the TetWild algorithm, which inserts them into a background mesh $\mathcal{B}$, so that each input triangle is a union of faces of refinement of $\mathcal{B}$.

We interrupt the algorithm after the binary space partitioning (BSP) subdivision (and before the TetWild mesh optimization [Hu et al. 2018, Section 3.2]) to obtain a positive tetrahedral mesh in rational coordinates with a surface with the same geometry of B_e and T_e, but possibly different connectivity as TetWild might refine it during the BSP stage.

Step 3. Our original goal was to compute a mesh conforming to B and T, but we could not do it directly with TetWild as they might be refined. We now replace the mesh generated by TetWild between B_e and T_e with another one conforming to B and T. To achieve this, we delete all tetrahedra between B_e and T_e, and insert the surfaces B and T, which will "float" in the empty space between B_e and T_e. We now want to fill the space between B and B_e with positive tetrahedra conforming to the surfaces B and T.

Step 4. Every prism P, made by a bottom triangle B^T and a bottom extruded triangle B_e^T and its corresponding bottom extruded refined triangle $B_e'^T \in \mathcal{B}$, can be tetrahedralized without refining B: That is, we first decompose the prism B^T, B_e^T in tetrahedra (always possible by construction), then refine every tetrahedron touching $B_e'^T$. By repeating the same operation on the space between T and T_e we will have a positive linear boundary conforming tetrahedral mesh of B and T.

Step 5. The tetrahedra generated in the previous step will have rational coordinates and will also likely have low quality. To round the coordinates to floating-point representation and to improve their quality, we use the mesh optimization stage of TetWild, with the minor variant of keeping the vertices and edges on B and T frozen. Note that the vertices in B and T are already roundable to floating-point representation, as they were part of the input.

Curved Tetrahedral Mesh Optimization After generating the conforming linear tetrahedral mesh. we stitch it with the tetrahedralized $\widetilde{\mathcal{S}}$ to obtain a valid output mesh $\mathcal{T}^k$ (Definition 4.1). However, its quality might be low, in particular in the curved region, as $\widetilde{\mathcal{S}}$ can be thin with large triangles. To improve the quality of $\mathcal{T}^k$ we adapt the local operation of a linear pipeline to our curved settings. We propose three local operations: smoothing, collapse, and flip. Since the surface of $\mathcal{T}^k$ is already coarse and of high quality, as part of the definition of ϕ^k, we prevent any local operation from changing it. We validate every local operation (i.e., check the positivity of $\mathcal{T}^k$) using the convex-hull property [Johnen et al. 2013] and reject the operation if it is violated. Our local operations are prototypical, and we leave as future work a more comprehensive study of curved mesh optimization.

Smoothing. As for the linear case, we compute the total energy of a vertex v by summing up the energies of the tetrahedra adjacent to it, which we compute on 56 regularly sampled points. We then perform gradient descent for all high-order nodes in the one-ring neighborhood of v. That is, we collect all edge nodes, face nodes, and cell nodes of the one-ring neighborhood of

Figure 4.9: Input triangle mesh with features and output curved mesh with feature preserved equipped with bijective map ϕ^k.

v. Differently from the linear case, the optimization is expensive since the nodes neighborhood typically contains hundreds of nodes.

Collapse and Swap The collapse and swap are the same as in a linear mesh, and we place the high-order nodes of the newly created face on the linear flat face.

4.5 Feature Preserving Curved Shell

Input We enhance the input to additionally include a set of feature edges $f_i \in \mathcal{F}$ and feature vertices $v_i \in \mathcal{V}$ such that no triangle in F has more than one feature edge (Figure 4.9 left). (This property can be satisfied on any generic mesh by performing 1-to-3 refinement on every triangle with more than one feature edge).

Output Since the input has features, the output curved mesh $\mathcal{T}^k$ will also have curved feature edges $f_i^k \in \mathcal{T}^k$, feature vertices $v_i^k \in \mathcal{V}^k$, and the bijective map ϕ^k preserves features by bijectively mapping $\mathcal{F}$ to $\mathcal{F}^k$ and $\mathcal{V}$ to $\mathcal{V}^k$. Our method makes no assumption on the topology and "quality" of the features. If the features are reasonable, it will produce a high-quality mesh, while if the features are close our algorithm will preserve them and result in smaller triangles on the surface. (Figure 4.10).

The previous construction generates valid curved tetrahedral meshes and the bijective map ϕ^k based on the construction of [?]. However, the shell construction cannot coarsen features: the authors suggest freezing them. For instance, when performing an edge collapse on the feature, the new coarse edge (orange) will not map to the feature (green) anymore (Figure 4.11). To ensure feature preservation we extend Definition 4.1.

Definition 4.2. We call a curved mesh $\mathcal{T}^K$ and its mapping ϕ^k from $\mathcal{M}$ *valid and feature preserving* if they are valid (Definition 4.1) and ϕ^k bijectively maps $\mathcal{F}$ to $\mathcal{F}^k$ and $\mathcal{V}$ to $\mathcal{V}^k$

As for the non-feature preserving case, we always aim to maintain a valid feature preserving $\mathcal{T}^k$.

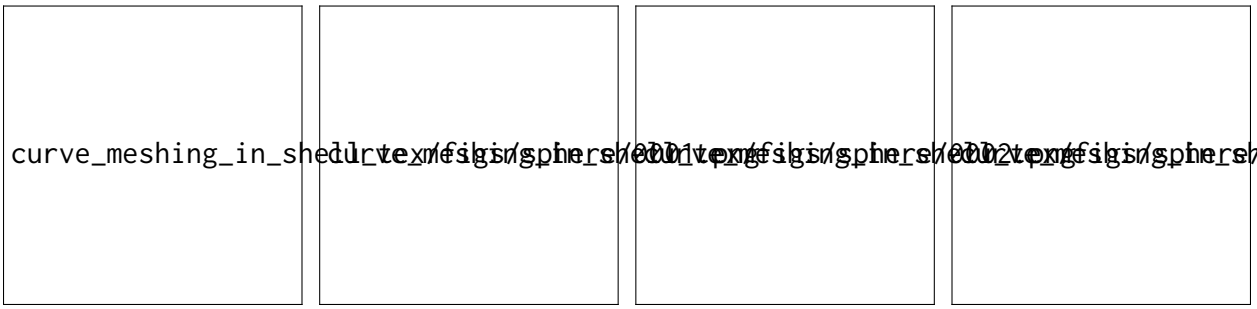


Figure 4.10: A sphere with different marked features (green). As we increase the number of features our algorithm will preserve them all but the quality of the surface suffers.

Figure 4.11: Input feature (green) is not preserved after traditional shell simplification.

Figure 4.12: Overview of the construction of the first stage of our pipeline.

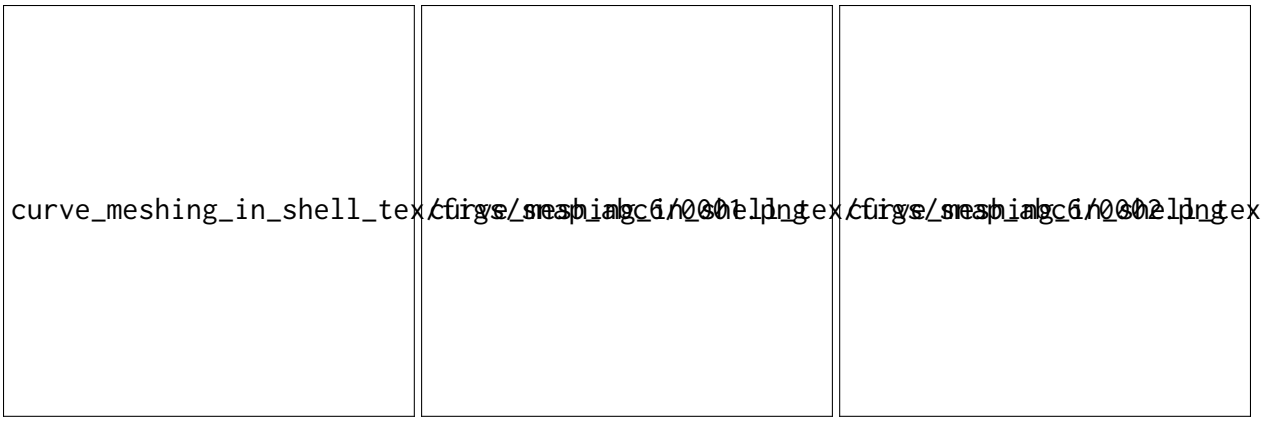


Figure 4.13: The input mesh has feature edges snapped, to create a valid shell, as well as the curved mesh.

Figure 4.14: The input edges feature (green) are grouped together in poly-lines and categorized in graph (left) and loops (right). For every graph we add the nodes (blue) to the set of feature vertices.

To account for features, we propose to change the prismatic map Π, that is, we only need to change the first stage. This is done by snapping the input features (Section 4.5.1). That is, we modify $\mathcal{M}$ to "straighten" the feature to ensure that the coarse prismatic projection preserves them and construct a mapping β between the straight mesh $\overline{\mathcal{M}}$ and $\mathcal{M}$ (Figure 4.13).

The outcome is a *valid* shell $\overline{\mathcal{S}}$ with respect to the straight surface $\overline{\mathcal{M}}$ (i.e., $\overline{\mathcal{M}}$ is a section $\overline{\mathcal{S}}$) that preserve features, the prismatic projection Π, and the bijective map β that can be directly used in the curved pipeline (Section 4.4). That is, the mapping ϕ^k will be defined as $\phi^k = \beta \circ \Pi \circ \psi^k$.

As for the non-preserving feature pipeline we ensure that our conditions are always met, starting from a trivial input and rejecting operations violating them. Our goal is to modify the input mesh $\mathcal{M}$ and create $\overline{\mathcal{M}}$ by moving its vertices. In such a way, the mapping β is simply barycentric. To guarantee bijectivity of ϕ^k we need to ensure that all mappings composing it are bijective, in particular β. To ensure that β is bijective it is enough that $\overline{\mathcal{M}}$ is self-intersection free (guaranteed by the shell construction) and that all its triangles have positive area. By straightening the features of $\mathcal{M}$ we ensure that the edges of the prism will map to the feature. Thus, ϕ^k will be feature preserving.

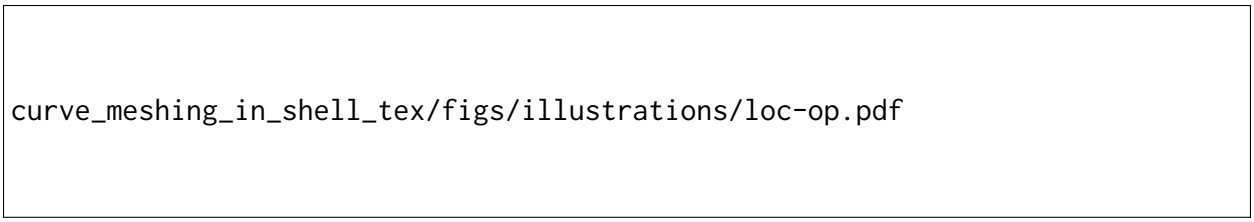


Figure 4.15: Illustration of smoothing on a feature.

Feature Grouping. The first step of our pipeline consists of grouping successive edges $f_i \in \mathcal{F}$ into poly-lines and identifying two categories: loops and graphs (Figure 4.14). For every graph, we identify its nodes and add them as feature vertices. In other words, we add to $\mathcal{V}$ all the end-points and junction of poly-lines.

4.5.1 Feature straightening.

To allow feature coarsening, we propose to straighten $\mathcal{M}$ to ensure that all features are collinear. In other words, we build, together with the shell $\overline{\mathcal{S}}$, a mesh $\overline{\mathcal{M}} = (\overline{V}, F)$ (i.e., a mesh with the same connectivity F of $\mathcal{M}$) and features $\overline{\mathcal{F}}$ such that every triangle of $\overline{\mathcal{M}}$ has a positive area, $\overline{\mathcal{M}}$ is a section of $\overline{\mathcal{S}}$, and the features in $\overline{\mathcal{F}}$ are collinear. In such a way, the mapping β is trivially defined as piecewise affine ($\overline{\mathcal{M}}$ and $\mathcal{M}$ share the same connectivity) and is locally injective as long as all the triangles on $\overline{\mathcal{M}}$ have positive areas. Note that the bijectivity of β follows from the fact that the shell prevents self-intersections of $\overline{\mathcal{M}}$.

To construct a mesh $\overline{\mathcal{M}}$ with straight features, we start with $\overline{\mathcal{M}} = \mathcal{M}$ (in the beginning all prisms of $\overline{\mathcal{S}}$ cover at most one feature edge). Let $\overline{f}^1 = \{\overline{f}^1_i\}, i = 1, \dots, n$ and $\overline{f}^2 = \{\overline{f}^2_i\}, i = 1, \dots, k$ two chains of feature edge belonging to the same feature $\overline{f} \in \overline{\mathcal{F}}$. For every local operation acting on a feature $\overline{f}_1$ and $\overline{f}_2$ we first construct the new feature $\overline{f}^n = \{\overline{f}^n_i\}, i = 1, \dots, k$ such that the segments $(\overline{f}^n_i, \overline{f}^n_{i+1})$ are collinear and their length is proportional to (f_i, f_{i+1}) (the feature vertices in the input mesh $\mathcal{M}$), that this we use arc-length cross parameterization from f to $\overline{f}^n$ (Figure 4.15 show an example of smoothing a feature). Moving vertices of $\overline{f}^n$ will also move the vertices of $\overline{\mathcal{M}}$ thus, straighten the mesh as the local operations proceed. After the construction of $\overline{f}^n$ we check if the newly constructed $\overline{\mathcal{M}}$ is still a section of $\overline{\mathcal{S}}$ and if the triangles modified by the straightening have areas larger than ϵ. In practice, we choose $\epsilon = 10^{-10}$: a smaller value would lead to numerical instabilities and a larger one to less straightening.

Note that not all features can be straight; for instance, if a triangle has three feature vertices (the snapped feature will result in a degenerate triangle, thus, β will not be bijective) or if the snapping flips the normal ($\overline{\mathcal{M}}$ will no longer be a section of $\overline{\mathcal{S}}$). Both are extremely rare cases in our dataset.

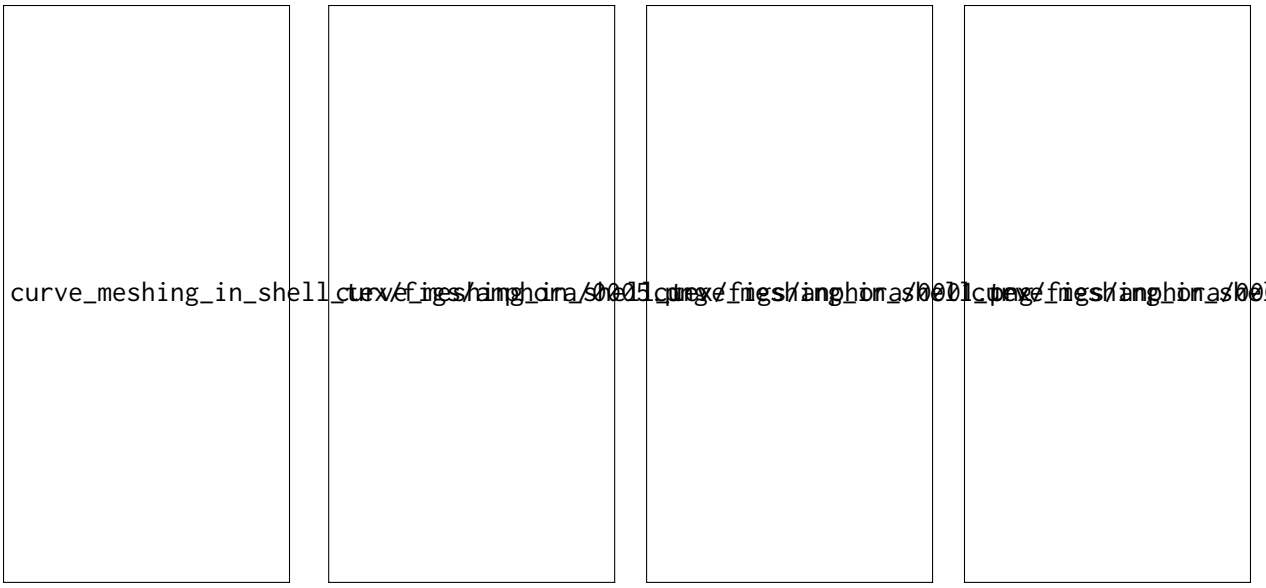


Figure 4.16: Curved meshes of different order. The additional degrees of freedom allows for more coarsening.

4.6 Results

Our algorithm is implemented in C++, using Eigen [Guennebaud et al. 2010] for the linear algebra routines, CGAL [The CGAL Project 2020] and Geogram [Lévy 2015] for predicates and geometric kernel, libigl [Jacobson et al. 2016] for basic geometry processing routines, and meshio [**?**] for converting across the different formats. We run our experiments on cluster nodes with Intel Xeon Platinum 8268 CPU 2.90GHz. The reference implementation and the data used to generate the results will be released as an open-source project.

To simplify the exposition, all meshes presented in this section are quartic meshes ($k = 4$). Our method is flexible and, for lower k, it will generate denser meshes (Figure 4.16).

4.6.1 Large Scale Validation.

We tested the robustness and quality of the result produced by our algorithm on three datasets: (1) Thingi10k dataset [Zhou and Jacobson 2016] containing 3574 models without features; (2) the first chunk of the ABC dataset [Koch et al. 2019] with 5328 models with features marked from the STEP file and (3) the CAD dataset [**?**] containing 106 models with semi-manual features.

Note that the original datasets contain more models since, for each of them, we selected meshes satisfying our assumptions: intersection-free (using the same strategy as in [**?**] with a distance tolerance of 10^{-6} and dihedral angle of $2°$) oriented, manifold triangle meshes, smallest triangle area larger than 10^{-8}.

Our method has only the geometry accuracy parameter ε, which we set to 1% of the longest

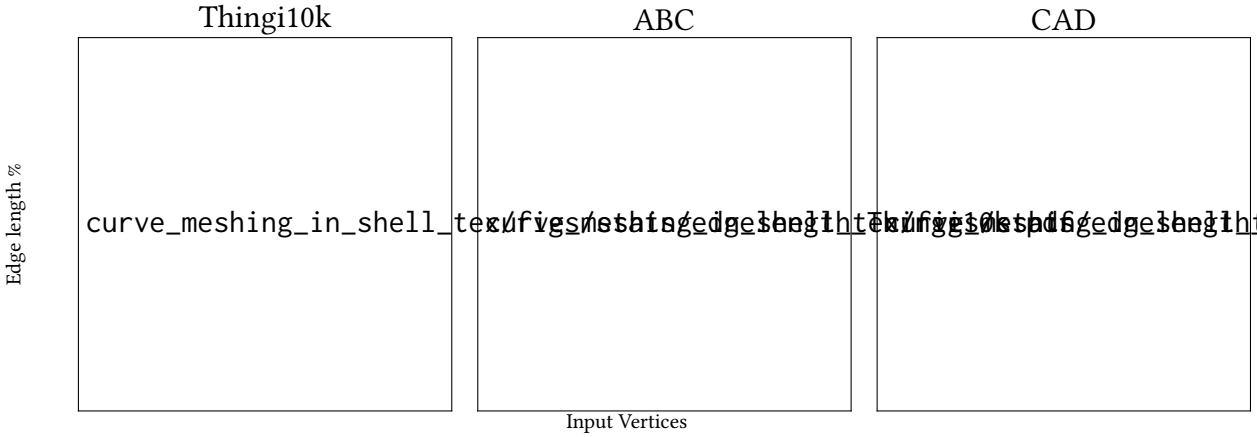


Figure 4.17: Relative average edge length (with respect to longest bounding box edge of each model) of our curved meshes versus number of input vertices.

Figure 4.18: Within the same distance bound (10^{-3} of the longest bounding box side), our method generates a coarser high order mesh, compared to the linear counterpart generated by fTetWild.

bounding box edge, and the point set $\mathcal{P}$ which we set as the input vertices V. With this basic setup, our algorithm aims to produce the coarsest possible mesh while preserving features and striving to generate high-quality meshes. Our algorithm successfully generates curved meshes for 3527 for Thingi10k, 5268 for the ABC, and all for CAD dataset within 12 hours; by allowing more time, all models but 3 can be successfully processed. The 3 failures are due to models with a small one-tetrahedra component that "move" inside the shell as it grows. This is an implementation choice: we use collision detection instead of continuous collision detection for efficiency reasons.

Our method successfully generates coarse meshes whose average edge length is 10% of the model size while preserving features (Figure 4.17). Figure 4.18 shows how our method successfully captures the features and coarsen the surface with curved elements, while many linear elements (generated with fTetWild [**?**]) are required to closely approximate the surface.

The output of our algorithm can be directly used in the simulation (Section 4.6.4) since we

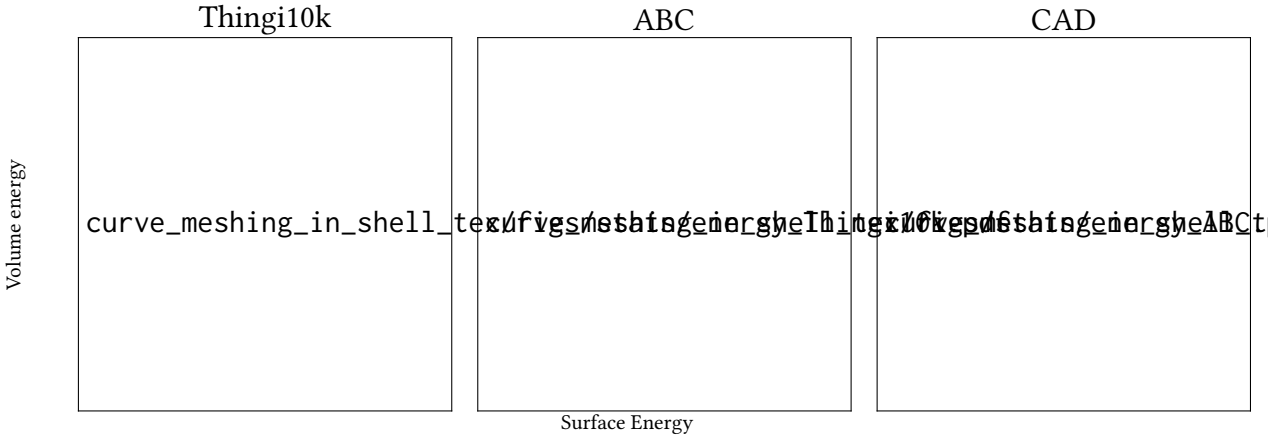


Figure 4.19: Surface and volume average MIPS energy of the output of our method (the CAD volume energy is truncated at 100, excluding 6 models).

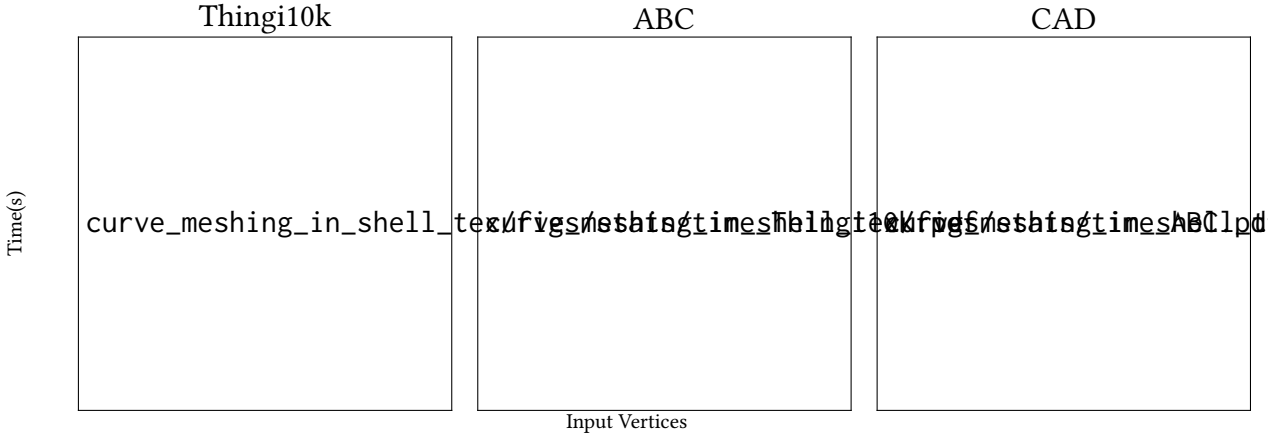


Figure 4.20: Timing of our algorithm versus the input number of vertices.

guarantee that the geometric mapping g is positive. To ensure good conditioning and performance of the numerical solver, we measure the MIPS energy [Hormann and Greiner 2000; Fu et al. 2015] of our output meshes (Figure 4.19).

Figure 4.20 shows the running time of our method with respect to the number of vertices. The running time of our algorithm is linear with respect to the number of input vertices; it takes around an hour for a model with around 10 thousand vertices.

4.6.2 Comparisons

Curved ODT [?] is, to the best of our knowledge, the only existing algorithm designed to convert dense triangle meshes into coarse, curved approximations. The input and output are the same as in our algorithm. However, their method does not provide a bijective map between the input and output, does not guarantee to preserve features, has no bound on the distance to the input surface, and does not guarantee that the elements are positive. While our algorithm has

Input | CODT with uniform (46249 elements) | CODT with LFS (38159 elements) | Our (2037 elements)

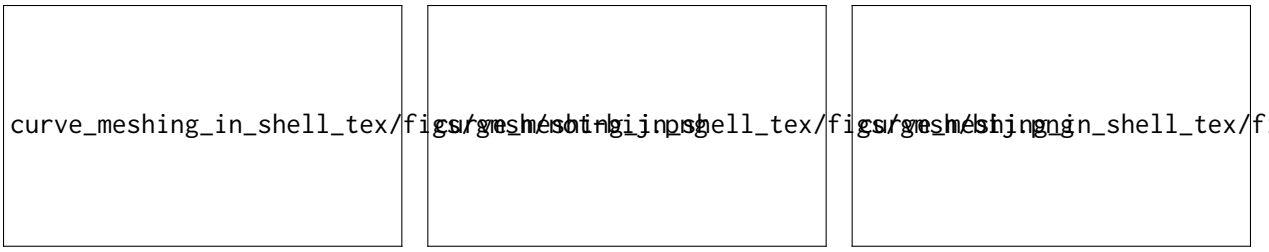
Figure 4.21: Compared with Curved ODT, our method does not rely on setting vertex number and sizing field, and can generate coarse valid results.

been designed to process large collections of data automatically, exposing only a few intuitive options to control the faithfulness to the input, the reference implementation of the curved ODT method provided to us by the authors requires the user to choose multiple *per-model* parameters to achieve good results, and the parameters have a strong effect on the quality and validity of the result (as shown in [**?**, Figure 16]). We thus restricted our comparison to only a small selection of models (see additional material) that the authors of [**?**] processed for us.

From our discussions with the authors, we observed that curved ODT generates a *valid* output when we provide (1) a sufficiently large number of vertices and (2) a good local feature size sizing field (LFS) [**?**] to efficiently spend the vertex budget in the regions with more geometrical details. Figure 4.21 shows an example of a model for which [**?**] fails to converge when using a uniform sizing field, while it succeeds when the sizing field is used.

In contrast, our algorithm can be run automatically on a large collection of geometrical models, it is guaranteed to have positive Jacobian (up to the use of floating-point predicates, Section 4.7), it preserves features, it automatically controls the density of the output depending on the desired user-provided distance threshold, and it provides a bijective map between the input mesh the boundary of the curved surface. For the model in Figure 4.21, our result contains 2037 elements, 18 times less than the curved ODT algorithm.

Gmsh [**?**] can only generate curved meshes from boundary representation (BRep), that is the input is not exactly the same as ours. To compare both algorithms, we start from the BRep and we generate a dense *linear* mesh that we use for our input. The Gmsh algorithm first constructs a curved mesh by fitting the high-order nodes to the BRep (possibly inverting elements) then performs mesh optimization to untangle them [**?**]; thus has no guarantee to generate positive meshes while preserving the surface (Figure 4.22, left). Additionally, Gmsh algorithm cannot control the distance from the input when the untangling allows the surface to move, and thus the surface is "wiggly" and denser than our result (Figure 4.22, center). We also observed that if the initial surface mesh is not dense enough, Gmsh closes holes and cannot generate a valid



Gmsh Fixed Surface | Gmsh Free Surface | Ours

Figure 4.22: Example of a BRep meshed with Gmsh where the optimization fails to untangle elements when fixing the surface. By allowing the surface to be modified, the mesh becomes "wiggly". Our method successfully generate a positive curved mesh.

Coarse Gmsh | Dense Gmsh | Ours

Figure 4.23: Example of a STEP file meshed with Gmsh where, due to the low mesh density, the tetrahedralization is not positive. Gmsh manages to generate a positive mesh by using a denser initial tessellation. Since our method starts from a dense mesh and coarsen it, it can successfully resolve the geometry.

tetrahedral mesh (Figure 4.23).

4.6.3 Flexibility

Since the input to our method is a triangle mesh, our method naturally supports a variety of input that can be easily converted into triangle meshes. For instance, $\mathcal{M}$ can be generated from marching an implicit surface or Catmull-Clark subdivision of a hand-made quad mesh (Figure 4.24). The bijective map ϕ^k is used, for instance, to transfer the geodesic distance field or color information from the input triangle mesh to the coarse curved mesh.

4.6.4 Applications

Large Scale Poisson To show that our meshes are ready for simulation, we solve for Poisson equation

$$\Delta u = f, \quad u|_{\partial\Omega} = g,$$

Figure 4.24: Our algorithm processes triangle meshes that can be extracted from different formats: an implicit microstructure geometry from [**?**] or a subdivision surface from [**?**]. The bijective map preserved on the surface allows taking advantage of the plethora of surface algorithms including polyhedral geodesic computation [Mitchell et al. 1987] and texture mapping.

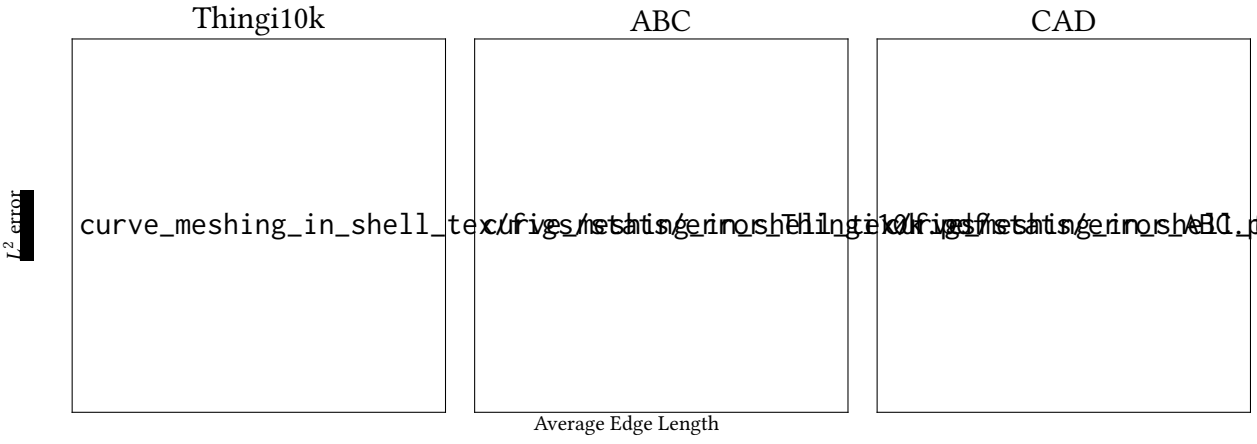
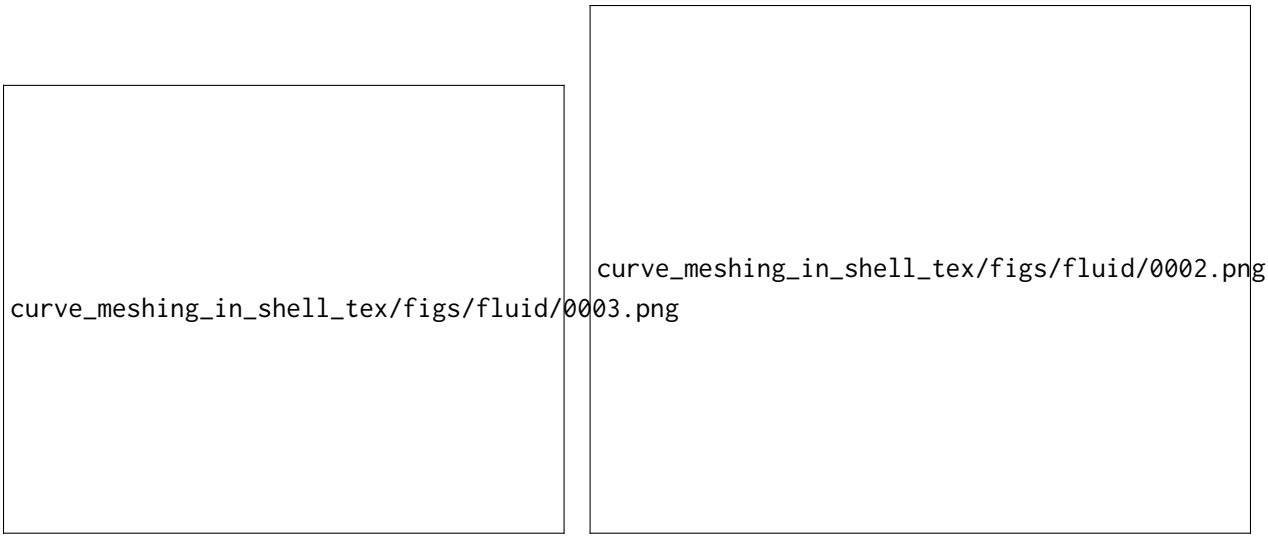


Figure 4.25: L^2 error of the solution of the Poisson equation with respect to model size on our three datasets.



Tetrahedral mesh Curved simulation

Figure 4.26: By meshing the region between a box and a complicated obstacle, we are able to perform non-linear fluid simulation on our curved mesh.

where Ω is the domain (i.e., the mesh), f is the right-hand side, and g are the Dirichlet boundary conditions. To simplify the setup and the error measurements, we use fabricated solutions [**?**]. That is, we choose the function u_{exact} to be

$$\begin{aligned} u_{\text{exact}}(x_1, x_2, x_3) = \\ 3/4\, e^{-((9x_1-2)^2+(9x_2-2)^2+(9x_3-2)^2)/4} + 3/4\, e^{-(9x_1+1)^2/49-(9x_2+1)/10-(9x_3+1)/10} \\ + 1/2\, e^{-((9x_1-7)^2+(9x_2-3)^2+(9x_3-5)^2)/4} - 1/5\, e^{-(9x_1-4)^2-(9x_2-7)^2-(9x_3-5)^2}, \end{aligned}$$

then we plug it in the equation to obtain f (g is simply u_{exact}). Figure 4.25 shows the L^2 error (average) distribution across our three datasets using our quartic meshes with quadratic approximation of u (i.e., we use superparametric elements).

High Accuracy Fluids Our curved meshes can be directly used to solve different partial differential equations (PDEs). For instance, by meshing the part outside the top shell and discarding the rest, we can generate a curved background mesh for the Navier-Stokes equation (Figure 4.26).

Fast Animation Our coarse curved meshes can be used as animation proxies as in [**??**]. We first compute an as-coarse-as-possible curved mesh (i.e., we set ε to infinity). Then we apply the boundary condition to simulate an elastic distortion of the curved mesh using linear elements. Finally, we use our bijective map φ^4 to map the displacement back to the input high-detailed surface mesh (Figure 4.1). The results are almost indistinguishable to a classical pipeline (i.e., mesh the input mesh), but the runtime is 400 times faster (8s versus over 50 minutes).

4.7 Limitations and Concluding Remarks

We introduce an automatic algorithm to convert dense triangle meshes in coarse, curved tetrahedral meshes whose boundary is within a user-controlled distance from the input mesh. Our algorithm supports feature preservation, and generates meshes with positive Jacobians and high quality, which are directly usable for FEM simulations.

Limitations. Our algorithm generates meshes with a C^0 geometric map. For most FEM applications, this is not an issue. However, for geometric modeling applications, where only the mesh boundary is used, the C^0 geometric map introduces normal discontinuities, which are undesirable. While the surface looks smooth from far away, plotting the reflection lines shows the discontinuity between the normals. We believe an exciting extension of our work would be to study the feasibility of using geometric maps that are C^1 [**?**] or C^2 [**?**]. A second limitation is that, in our implementation, the validity conditions (Definition 4.1) are currently checked using floating point arithmetic, using numerical tolerances to account for rounding errors. While our implementation works on a large collection of models, it is possible that it will fail on others due to the inexact validity predicates. We are not aware of exact predicates for these conditions, and we believe that developing them is an interesting, and challenging, venue for future work.

Future Work. Our current high order mesh optimization pipeline is preliminary, as it only supports vertex smoothing, collapse, and swap. Adding additional operators, allow them to exploit the curved geometric map, and optimizing the boundary could lead to a further increase in mesh quality. While simple at a high-level, this change will require to merge the two parts of our algorithm, a major implementation effort.

Conclusions. We believe that our work will foster adoption of curved meshes, and open the door to a new family of geometry processing algorithms able to take advantage of this highly compact, yet accurate, shape representation.

5 Declarative Specification for Unstructured Mesh Editing Algorithms

6 Conclusion

A Appendix

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero

dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Bibliography

Aigerman, N., Kovalsky, S. Z., and Lipman, Y. (2017). Spherical orbifold tutte embeddings. *ACM Trans. Graph.*, 36(4).

Aigerman, N. and Lipman, Y. (2015). Orbifold tutte embeddings. *ACM Trans. Graph.*, 34(6).

Aigerman, N. and Lipman, Y. (2016). Hyperbolic orbifold tutte embeddings. *ACM Trans. Graph.*, 35(6).

Aigerman, N., Poranne, R., and Lipman, Y. (2014). Lifted bijections for low distortion surface mappings. *ACM Trans. Graph.*, 33(4).

Aigerman, N., Poranne, R., and Lipman, Y. (2015). Seamless surface mappings. *ACM Trans. Graph.*, 34(4).

Alliez, P., De Verdire, E. C., Devillers, O., and Isenburg, M. (2003). Isotropic surface remeshing. In *2003 Shape Modeling International.*, pages 49–58. IEEE.

Aubry, R., Dey, S., Mestreau, E., and Karamete, B. (2017). Boundary layer mesh generation on arbitrary geometries. *International Journal for Numerical Methods in Engineering*, 112(2):157–173.

Aubry, R. and Löhner, R. (2008). On the 'most normal'normal. *Communications in Numerical Methods in Engineering*, 24(12):1641–1652.

Aubry, R., Mestreau, E., Dey, S., Karamete, B., and Gayman, D. (2015). On the 'most normal'normal—part 2. *Finite Elements in Analysis and Design*, 97:54–63.

Bajaj, C. L., Xu, G., Holt, R. J., and Netravali, A. N. (2002). Hierarchical multiresolution reconstruction of shell surfaces. *Computer Aided Geometric Design*, 19(2):89–112.

Bank, R. E., Sherman, A. H., and Weiser, A. (1983). Some refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing, Applications of Mathematics and Computing to the Physical Sciences*, 1:3–17.

Barnhill, R. E., Opitz, K., and Pottmann, H. (1992). Fat surfaces: a trivariate approach to triangle-based interpolation on surfaces. *Computer Aided Geometric Design*, 9(5):365–378.

Bommes, D., Lévy, B., Pietroni, N., Puppo, E., Silva, C., Tarini, M., and Zorin, D. (2013). Quad-mesh generation and processing: A survey. In *Computer Graphics Forum*, volume 32, pages 51–76. Wiley Online Library.

Botsch, M. and Kobbelt, L. (2003). Multiresolution surface representation based on displacement volumes. In *Computer Graphics Forum*, volume 22, pages 483–491. Wiley Online Library.

Botsch, M., Pauly, M., Gross, M. H., and Kobbelt, L. (2006). Primo: coupled prisms for intuitive surface modeling. In *Symposium on Geometry Processing*, number CONF, pages 11–20.

Boubekeur, T. and Alexa, M. (2008). Phong tessellation. In *ACM Transactions on Graphics (TOG)*, volume 27, page 141. ACM.

Calderon, S. and Boubekeur, T. (2017). Bounding proxies for shape approximation. *ACM Trans. Graph.*, 36(4).

Campen, M., Silva, C. T., and Zorin, D. (2016). Bijective maps from simplicial foliations. *ACM Trans. Graph.*, 35(4):74:1–74:15.

Chazal, F. and Cohen-Steiner, D. (2005). A condition for isotopic approximation. *Graphical Models*, 67(5):390–404.

Chazal, F., Lieutier, A., Rossignac, J., and Whited, B. (2010). Ball-map: Homeomorphism between compatible surfaces. *International Journal of Computational Geometry & Applications*, 20(03):285–306.

Chen, Y., Tong, X., Wang, J., Wang, J., Lin, S., Guo, B., Shum, H.-Y., and Shum, H.-Y. (2004). Shell texture functions. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 343–353. ACM.

Cheng, X.-X., Fu, X.-M., Zhang, C., and Chai, S. (2019). Practical error-bounded remeshing by adaptive refinement. *Computers & Graphics*, 82:163 – 173.

Cheshmi, K., Kaufman, D. M., Kamil, S., and Dehnavi, M. M. (2020). Nasoq: Numerically accurate sparsity-oriented qp solver. *ACM Transactions on Graphics*, 39(4).

Cohen, J., Manocha, D., and Olano, M. (1997). Simplifying polygonal models using successive mappings. In *Proceedings. Visualization'97 (Cat. No. 97CB36155)*, pages 395–402. IEEE.

Cohen, J., Varshney, A., Manocha, D., Turk, G., Weber, H., Agarwal, P., Brooks, F., and Wright, W. (1996). Simplification envelopes. In *Siggraph*, volume 96, pages 119–128.

Collins, G. and Hilton, A. (2002). Mesh decimation for displacement mapping. In *Eurographics (Short Papers)*.

Community, B. O. (2018). *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam.

Conway, J. H., Burgiel, H., and Goodman-Strauss, C. (2016). *The symmetries of things*. CRC Press.

Coxeter, H. S. M. (1973). *Regular polytopes*. Courier Corporation.

Crane, K., Weischedel, C., and Wardetzky, M. (2013). Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Transactions on Graphics (TOG)*, 32(5).

Dey, T. K., Edelsbrunner, H., Guha, S., and Nekhayev, D. V. (1999). Topology preserving edge contraction. *Publ. Inst. Math.(Beograd)(NS)*, 66(80):23–45.

Dey, T. K. and Ray, T. (2010). Polygonal surface remeshing with delaunay refinement. *Engineering with computers*, 26(3):289–301.

Dompierre, J., Labbé, P., Vallet, M.-G., and Camarero, R. (1999). How to subdivide pyramids, prisms, and hexahedra into tetrahedra. *IMR*, 99:195.

Dunyach, M., Vanderhaeghe, D., Barthe, L., and Botsch, M. (2013). Adaptive remeshing for real-time mesh deformation.

Dyer, R., Zhang, H., and Möller, T. (2007). Delaunay mesh construction.

Ebke, H.-C., Campen, M., Bommes, D., and Kobbelt, L. (2014). Level-of-detail quad meshing. *ACM Transactions on Graphics (TOG)*, 33(6):184.

Ezuz, D., Solomon, J., and Ben-Chen, M. (2019). Reversible harmonic maps between discrete surfaces. *ACM Trans. Graph.*, 38(2).

Floater, M. S. (1997). Parametrization and smooth approximation of surface triangulations. *Computer Aided Geometric Design*, 14:231–250.

Floater, M. S. and Hormann, K. (2005). Surface parameterization: a tutorial and survey. In *In Advances in Multiresolution for Geometric Modelling, Mathematics and Visualization*, pages 157–186. Springer Verlag.

Fu, X.-M. and Liu, Y. (2016). Computing inversion-free mappings by simplex assembly. *ACM Trans. Graph.*, 35(6):216:1–216:12.

Fu, X.-M., Liu, Y., and Guo, B. (2015). Computing locally injective mappings by advanced mips. *ACM Transactions on Graphics (TOG)*, 34(4):71.

Garimella, R. V. and Shephard, M. S. (2000). Boundary layer mesh generation for viscous flow simulations. *International Journal for Numerical Methods in Engineering*, 49(1-2):193–218.

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

Garland, M. and Heckbert, P. S. (1998). Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings Visualization'98 (Cat. No. 98CB36276)*, pages 263–269. IEEE.

Gärtner, B. and Schönherr, S. (2000). An efficient, exact, and generic quadratic programming solver for geometric optimization. In *Proceedings of the sixteenth annual symposium on Computational geometry*, pages 110–118.

Gotsman, C. and Surazhsky, V. (2001). Guaranteed intersection-free polygon morphing. *Computers & Graphics*, 25(1):67–75.

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

Guéziec, A. (1996). *Surface simplification inside a tolerance volume*. IBM TJ Watson Research Center.

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

Hart, G. W. (2018). Conway notation for polyhedra. *URL: http://www. gergehart. co/virtual-polyhedra/conway_notation. html*.

Hass, J. and Trnkova, M. (2020). Approximating isosurfaces by guaranteed-quality triangular meshes. In *Computer Graphics Forum*, volume 39, pages 29–40. Wiley Online Library.

Hormann, K. and Greiner, G. (2000). Mips: An efficient global parametrization method. Technical report, ERLANGEN-NUERNBERG UNIV (GERMANY) COMPUTER GRAPHICS GROUP.

Hormann, K., Lévy, B., and Sheffer, A. (2007). Mesh parameterization: Theory and practice. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, New York, NY, USA. ACM.

Hormann, K. and Sukumar, N., editors (2017). *Generalized Barycentric Coordinates in Computer Graphics and Computational Mechanics*. CRC Press, Boca Raton, FL.

Hu, K., Yan, D., 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, K., Yan, D.-M., Bommes, D., Alliez, P., and Benes, B. (2016). 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. (2019a). Triwild: robust triangulation with curve constraints. *ACM Transactions on Graphics (TOG)*, 38(4):52.

Hu, Y., Schneider, T., Wang, B., Zorin, D., and Panozzo, D. (2019b). Fast tetrahedral meshing in the wild. *arXiv preprint arXiv:1908.03581*.

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.

Huang, J., Liu, X., Jiang, H., Wang, Q., and Bao, H. (2007). Gradient-based shell generation and deformation. *Computer Animation and Virtual Worlds*, 18(4-5):301–309.

Jacobson, A., Panozzo, D., Schüller, C., Diamanti, O., Zhou, Q., Pietroni, N., et al. (2016). libigl: A simple c++ geometry processing library, 2016.

Jiang, Z., Schaefer, S., and Panozzo, D. (2017). Simplicial complex augmentation framework for bijective maps. *ACM Transactions on Graphics*, 36(6).

Jiao, X. and Heath, M. T. (2004). Overlaying surface meshes, part i: Algorithms. *International Journal of Computational Geometry & Applications*, 14(06):379–402.

Jin, M., Kim, J., Luo, F., and Gu, X. (2008). Discrete surface ricci flow. *IEEE Transactions on Visualization and Computer Graphics*, 14(5):1030–1043.

Jin, Y., Song, D., Wang, T., Huang, J., Song, Y., and He, L. (2019). A shell space constrained approach for curve design on surface meshes. *Computer-Aided Design*, 113:24–34.

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

Kharevych, L., Springborn, B., and Schröder, P. (2006). Discrete conformal mappings via circle patterns. *ACM Trans. Graph.*, 25(2):412–438.

Kobbelt, L., Campagna, S., Vorsatz, J., and Seidel, H.-P. (1998). Interactive multi-resolution modeling on arbitrary meshes. In *Siggraph*, volume 98, pages 105–114.

Koch, S., Matveev, A., Jiang, Z., Williams, F., Artemov, A., Burnaev, E., Alexa, M., Zorin, D., and Panozzo, D. (2019). Abc: A big cad model dataset for geometric deep learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Kraevoy, V. and Sheffer, A. (2004). Cross-parameterization and compatible remeshing of 3d models. *ACM Transactions on Graphics (TOG)*, 23(3):861–869.

Lee, A., Moreton, H., and Hoppe, H. (2000). Displaced subdivision surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 85–94.

Lee, A. W., Sweldens, W., Schröder, P., Cowsar, L. C., and Dobkin, D. P. (1998). Maps: Multiresolution adaptive parameterization of surfaces. In *Siggraph*, volume 98, pages 95–104.

Lengyel, J., Praun, E., Finkelstein, A., and Hoppe, H. (2001). Real-time fur over arbitrary surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 227–232. ACM.

Lévy, B. (2015). Geogram.

Li, M., Kaufman, D. M., Kim, V. G., Solomon, J., and Sheffer, A. (2018). Optcuts: joint optimization of surface cuts and parameterization. *ACM Transactions on Graphics (TOG)*, 37(6):1–13.

Litke, N., Droske, M., Rumpf, M., and Schröder, P. (2005). An image processing approach to surface matching. In *Symposium on Geometry Processing*, volume 255.

Liu, S., Ferguson, Z., Jacobson, A., and Gingold, Y. I. (2017). Seamless: seam erasure and seam-aware decoupling of shape from mesh resolution. *ACM Trans. Graph.*, 36(6).

Liu, Y.-J., Xu, C.-X., Fan, D., and He, Y. (2015). Efficient construction and simplification of delaunay meshes. *ACM Transactions on Graphics (TOG)*, 34(6).

Loriot, S., Rouxel-Labbé, M., Tournois, J., and Yaz, I. O. (2020). Polygon mesh processing. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.0.3 edition.

Mandad, M., Cohen-Steiner, D., and Alliez, P. (2015). Isotopic approximation within a tolerance volume. *ACM Transactions on Graphics (TOG)*, 34(4):64.

Maron, H., Galun, M., Aigerman, N., Trope, M., Dym, N., Yumer, E., Kim, V. G., and Lipman, Y. (2017). Convolutional neural networks on surfaces via seamless toric covers. *ACM Trans. Graph.*, 36(4):71–1.

Mitchell, J. S., Mount, D. M., and Papadimitriou, C. H. (1987). The discrete geodesic problem. *SIAM Journal on Computing*, 16(4):647–668.

Müller, M., Chentanez, N., Kim, T.-Y., and Macklin, M. (2015). Air meshes for robust collision handling. *ACM Trans. Graph.*, 34(4).

Nguyen, H. (2007). *Gpu gems 3*. Addison-Wesley Professional.

Ovsjanikov, M., Ben-Chen, M., Solomon, J., Butscher, A., and Guibas, L. (2012). Functional maps: A flexible representation of maps between shapes. *ACM Trans. Graph.*, 31(4).

Ovsjanikov, M., Corman, E., Bronstein, M., Rodolà, E., Ben-Chen, M., Guibas, L., Chazal, F., and Bronstein, A. (2017). Computing and processing correspondences with functional maps. In *ACM SIGGRAPH 2017 Courses*, SIGGRAPH '17.

Panozzo, D., Baran, I., Diamanti, O., and Sorkine-Hornung, O. (2013). Weighted averages on surfaces. *ACM Transactions on Graphics (TOG)*, 32(4):60.

Peng, J., Kristjansson, D., and Zorin, D. (2004). Interactive modeling of topologically complex geometric detail. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 635–643. ACM.

Phong, B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317.

Pietroni, N., Tarini, M., and Cignoni, P. (2010). Almost isometric mesh parameterization through abstract domains. *IEEE Transactions on Visualization and Computer Graphics*, 16(4):621–635.

Porumbescu, S. D., Budge, B., Feng, L., and Joy, K. I. (2005). Shell maps. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 626–633. ACM.

Praun, E., Sweldens, W., and Schröder, P. (2001). Consistent mesh parameterizations. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 179–184.

Rabinovich, M., Poranne, R., Panozzo, D., and Sorkine-Hornung, O. (2017). Scalable locally injective mappings. *ACM Trans. Graph.*, 36(2):16:1–16:16.

Sacht, L., Vouga, E., and Jacobson, A. (2015). Nested cages. *ACM Transactions on Graphics (TOG)*, 34(6):170.

Schmidt, P., Born, J., Campen, M., and Kobbelt, L. (2019). Distortion-minimizing injective maps between surfaces. *ACM Transactions on Graphics (TOG)*, 38.

Schreiner, J., Asirvatham, A., Praun, E., and Hoppe, H. (2004). Inter-surface mapping. *ACM Trans. Graph.*, 23(3):870–877.

Schüller, C., Kavan, L., Panozzo, D., and Sorkine-Hornung, O. (2013). Locally injective mappings. In *Symposium on Geometry Processing*, pages 125–135.

Sharp, N. and Crane, K. (2020). A Laplacian for Nonmanifold Triangle Meshes. *Computer Graphics Forum (SGP)*, 39(5).

Sharp, N., Soliman, Y., and Crane, K. (2019). Navigating intrinsic triangulations. *ACM Transactions on Graphics (TOG)*, 38(4):55.

Sheffer, A., Praun, E., and Rose, K. (2006). Mesh parameterization methods and their applications. *Found. Trends. Comput. Graph. Vis.*, 2(2):105–171.

Shewchuk, J. R. (1997). Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3).

Shoham, M., Vaxman, A., and Ben-Chen, M. (2019). Hierarchical functional maps between subdivision surfaces. *Computer Graphics Forum*, 38(5):55–73.

Smith, J. and Schaefer, S. (2015). Bijective parameterization with free boundaries. *ACM Trans. Graph.*, 34(4):70:1–70:9.

Springborn, B., Schröder, P., and Pinkall, U. (2008). Conformal equivalence of triangle meshes. *ACM Trans. Graph.*, 27(3):1–11.

Stellato, B., Banjac, G., Goulart, P., Bemporad, A., and Boyd, S. (2017). OSQP: An operator splitting solver for quadratic programs. *ArXiv e-prints*.

Stephenson, K. (2005). *Introduction to circle packing: The theory of discrete analytic functions*. Cambridge University Press.

Surazhsky, V. and Gotsman, C. (2001). Morphing stick figures using optimized compatible triangulations. In *Computer Graphics and Applications, 2001. Proceedings. Ninth Pacific Conference on*, pages 40–49. IEEE.

The CGAL Project (2020). *CGAL User and Reference Manual*. CGAL Editorial Board, 5.0.3 edition.

Thiery, J.-M., Tierny, J., and Boubekeur, T. (2012). Cager: Cage-based reverse engineering of animated 3d shapes. *Comput. Graph. Forum*, 31(8):2303–2316.

Tutte, W. T. (1963). How to draw a graph. *Proceedings of the London Mathematical Society*, 13(3):743–768.

Wang, L., Wang, X., Tong, X., Lin, S., Hu, S., Guo, B., and Shum, H.-Y. (2003). View-dependent displacement mapping. In *ACM Transactions on graphics (TOG)*, volume 22, pages 334–339. ACM.

Wang, X., Tong, X., Lin, S., Hu, S., Guo, B., and Shum, H.-Y. (2004). Generalized displacement maps. In *Proceedings of the Fifteenth Eurographics conference on Rendering Techniques*, pages 227–233. Eurographics Association.

Weber, O. and Zorin, D. (2014). Locally injective parametrization with arbitrary fixed boundaries. *ACM Trans. Graph.*, 33(4):75:1–75:12.

Zhang, E., Mischaikow, K., and Turk, G. (2005). Feature-based surface parameterization and texture mapping. *ACM Trans. Graph.*, 24(1):1–27.

Zhou, Q., Grinspun, E., Zorin, D., and Jacobson, A. (2016). Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)*, 35(4):1–15.

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

Zomorodian, A. and Edelsbrunner, H. (2000). Fast software for box intersections. In *Proceedings of the sixteenth annual symposium on Computational geometry*, pages 129–138.