Immersion of Self-Intersecting Solids and Surfaces

YIJING LI and JERNEJ BARBIČ, University of Southern California, USA

Self-intersecting, or nearly self-intersecting, meshes are commonly found in 2D and 3D computer graphics practice. Self-intersections occur, for example, in the process of artist manual work, as a by-product of procedural methods for mesh generation, or due to modeling errors introduced by scanning equipment. If the space bounded by such inputs is meshed naively, the resulting mesh joins ("glues") self-overlapping parts, precluding efficient further modeling and animation of the underlying geometry. Similarly, near self-intersections force the simulation algorithm to employ an unnecessarily detailed mesh to separate the nearly self-intersecting regions. Our work addresses both of these challenges, by giving an algorithm to generate an "un-glued" simulation mesh, of arbitrary user-chosen resolution, that properly accounts for self-intersections and near self-intersections. In order to achieve this result, we study the mathematical concept of immersion, and give a deterministic and constructive algorithm to determine if the input self-intersecting triangle mesh is the boundary of an immersion. For near self-intersections, we give a robust algorithm to properly duplicate mesh elements and correctly embed the underlying geometry into the mesh element copies. Both the self-intersections and near self-intersections are combined into one algorithm that permits successful meshing at arbitrary resolution. Applications of our work include volumetric shape editing, physically based simulation and animation, and volumetric weight and geodesic distance computation on self-intersecting inputs.

CCS Concepts: • **Computing methodologies → Volumetric models**; *Physical simulation*;

Additional Key Words and Phrases: self-intersections, meshing, immersion, physically based modeling, animation, computational topology

ACM Reference Format:
Yijing Li and Jernej Barbič. 2018. Immersion of Self-Intersecting Solids and Surfaces. *ACM Trans. Graph.* 37, 4, Article 45 (August 2018), 14 pages. https://doi.org/10.1145/3197517.3201327

1 INTRODUCTION

Polygonal meshes are commonly used in computer graphics due to their versatility and ease of rendering. Many interesting shapes, however, are modeled with geometry that nearly self-intersects, or even self-intersects (Figure 1). Self-intersection may occur simply due to errors in scanning equipment, inadequate modeling by artists; and are sometimes unavoidable. A common example is meshing the mouth of a human face, whereby the upper and lower lip (Figure 2), and/or the mesh of the mouth cavity (teeth and tongue) typically self-intersect in the neutral pose. Another example is meshing the eyes and the eye socket. Self-intersections pose a significant problem for simulation-based modeling and animation pipelines because they preclude an easy generation of a quality simulation mesh. Naive algorithms glue the self-intersecting parts, and/or require extremely small elements to be employed to resolve near self-intersections in the input geometry, resulting in long simulation times.

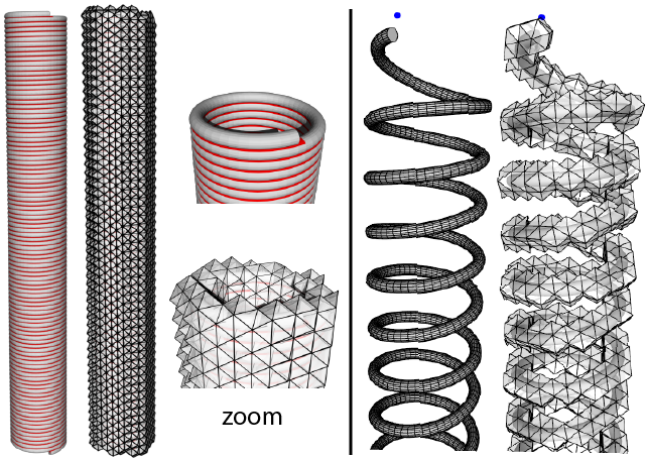

Fig. 1. **Self-intersection-aware tet meshing:** Our method properly duplicated and connected the tets to account for helix triangle mesh self-intersections. Left: input triangle and tetrahedral mesh. Intersecting triangles are shown red. Right: the output tet mesh has been successfully pulled apart using volumetric ARAP [Sorkine and Alexa 2007].

In our work, we give an algorithm to generate simulation meshes that are properly "un-glued" both for self-intersecting inputs and nearly self-intersecting inputs. Our algorithm is applicable both in 2D and 3D. This is achieved by first generating a world-space simulation mesh (tetrahedral mesh in 3D; triangle mesh in 2D) against the input geometry, using any method and at any user-chosen resolution. Our method then properly duplicates the simulation elements (tets in 3D; triangles in 2D), correctly connects the element copies, and embeds the input geometry into these duplicated elements for correct subsequent modeling and/or animation. Our method works in a unified way both for self-intersecting inputs and nearly self-intersecting inputs. Often, in animation practice, the *only* reason for using detailed meshes is to resolve nearly self-intersecting inputs; *even if the required deformation resolution does not necessitate such detailed meshing*. For nearly self-colliding inputs, inspired by the work of Sifakis [2007], we propose a novel method to unglue tetrahedra containing nearly self-intersecting geometry. We do so using exact arithmetic Sutherland-Hodgman clipping and pseudo-normal tests; which accelerates meshing 30× compared to prior work.

In order to address these intuitive practical goals, one requires a proper mathematical language to express concepts such as "unglued" meshes, "meshing" and "embedding" of overlapping spaces, and state

Authors' address: Yijing Li, yijingl@usc.edu; Jernej Barbič, jnb@usc.edu, University of Southern California, Computer Science, 941 Bloom Walk SAL 104, Los Angeles, CA, 90089, USA.

0730-0301/2018/8-ART45 $15.00
https://doi.org/10.1145/3197517.3201327

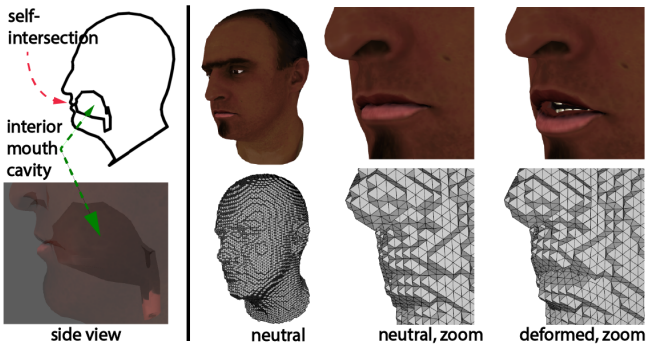

Fig. 2. **Unglued meshing of self-intersecting lips and mouth cavity:** Column 1: side view of the head and the self-intersecting mouth connected to an interior cavity. Such self-intersecting modeling of the mouth cavity is very common in visual effects and games, to add a plausible mouth appearance to characters. The self-intersecting mouth is a common technology blocker for head simulation algorithms. Column 2: head triangle mesh (top) and our tetrahedral mesh (bottom). Column 3: zoom to the self-intersecting lips. Column 4: our tetrahedral mesh "unglued" the input self-intersection, enabling the simulation to successfully open the mouth.

well-defined algorithms and prove properties about them. This journey very quickly led us to geometric and algebraic topology, and the mathematical concept of immersion, which is the natural language for expressing self-intersections in $\mathbb{R}^d$, for $d = 2, 3$. Immersions are continuous maps between geometric shapes (more precisely, topological spaces) and $\mathbb{R}^d$ that are locally (but not necessarily globally) injective (precise definition is in Supplementary Material 1). We show that the problem of creating the "un-glued" mesh in $\mathbb{R}^d$ is equivalent to discovering an immersion from a compact d-manifold onto $\mathbb{R}^d$, such that the restriction of this "volume-immersion" onto the boundary of the d-manifold is an immersion onto the input mesh. Such immersions do not always exist (e.g., due to input inversions); and we give an algorithm that both determines if an immersion exists, and if yes, constructs the immersion and the underlying "unglued" simulation mesh. For both nearly self-intersecting inputs, and self-intersecting inputs, we also give a method to clone and properly connect elements containing multiple connected components of the embedded mesh, and embed them into the cloned copies, to properly separate those parts during simulation.

Our method improves the robustness of geometry processing tools, enabling direct tetrahedralization and embedding of 3D self-intersecting surfaces. Different to prior work on self-intersecting meshing, our method is general, works both in 2D and 3D, and on meshes of arbitrary genus (number of handles). We analyze the input surface directly without needing to deform the mesh. This makes our approach fast, and allows any complex shape as input, as long as it has manifold connectivity (no T-junctions or hanging faces), is orientable (no Klein bottles) and consistently oriented, and without inversions (no self-intersections poking through the body). For example, in Figure 3, we generate an un-glued tetrahedral mesh for a self-intersecting tree triangle mesh. This mesh was generated using procedural modeling without any regard for intersections and self-intersections. On our output simulation meshes, one can compute volumetric weights for skinning and other techniques

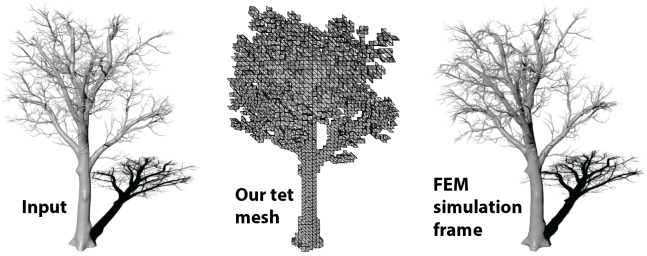

Fig. 3. **Tetrahedral meshing of self-intersecting procedurally generated geometry.** The triangle mesh for this tree (328,152 triangles; 2,634 branches) was generated using a procedural modeling method without regard for branch intersections and self-intersections. Our method successfully generated a simulation tet mesh that "unglues" all the branches. Because our tets can be overlapping, our mesh can be relatively coarse (32,457) for a model of this complexity, permitting faster FEM simulation. For comparison, naive meshing glues branches and results in visibly suboptimal tree motion when simulated using FEM (supplemental video).

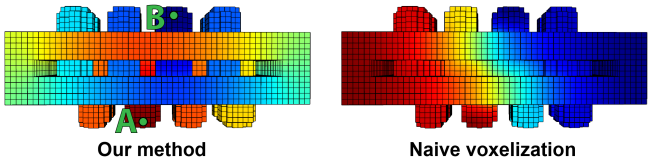

Fig. 4. **Our method produces topology-aware weights.** We computed bounded biharmonic weights[Jacobson et al. 2011], using the two handles "A" and "B" indicated; the figure (left) shows the BBW weight function for handle "A". BBWs computed on our volumetric mesh (left) reflects the correct topology of the embedded surface (we re-use the Sacht's test model [Sacht et al. 2013] from Figure 24). In contrast, naive voxelization (right) glues the cylinders together, resulting in unsatisfactory weights.

(Figure 4), calculate geodesic distances, edit shapes with volume conservation, and apply physically based simulation. The user can also use our mesh to resolve the self-intersections, for example by applying contact-resolving physically based simulation [Baraff et al. 2003; Heidelberger et al. 2004] to our output mesh (Figure 22); which is better than resolving the self-intersection directly on the surface mesh due to better volume conservation. Our method can also help automatic segmentation of 2D shapes for hand-drawn animations [Huang et al. 2014; Noris et al. 2012] and sketch-based shape retrieval and placement [Xu et al. 2013].

2 RELATED WORK

Research on detecting and decomposing self-intersecting shapes dates back to Shor and Van Wyk [1989], who gave a polynomial-time algorithm to find all possible immersions of a 2D shape which comes from stretching and bending a disk, by finding triangulations of the input curve. Eppstein [2009] analyzed the complexity of various immersion and embedding problems and found out that it is NP-complete to determine several immersion and embedding problems. Frisch [2010] studied extending immersions of 2D circles into 2D disks and analyzed existence and uniqueness of the problem. Our specific problem in 3D was not studied by these prior works. In computer graphics, Mukherjee [2011] described another method to solve the problem of 2D disk immersion. As acknowledged in the author's

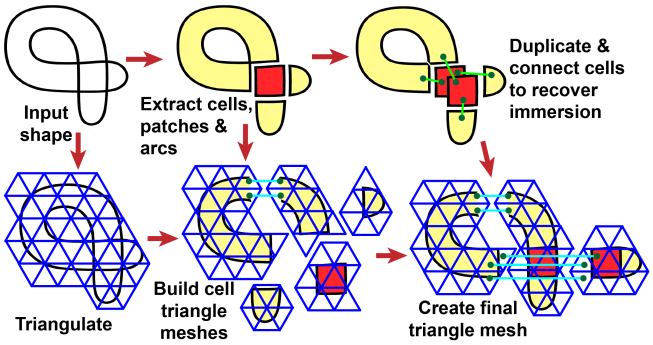

Fig. 5. **Basic steps of our method** are illustrated in this 2D example. Given the input shape (in this 2D example, a closed polygonal line / curve in $\mathbb{R}^2$), our method extracts the cells, patches and arcs (Section 4), then duplicates and connects cells to recover the immersion of a two-dimensional disk such that the disk boundary immerses onto the input shape (Section 5 and 6). Note that the largest cell self-touches, which we address in Section 4.2.1. To produce the immersion of the disk, we first triangulate the entire space covered by the input shape, then create a submesh for each cell (Section 7). Some triangle vertices are duplicated to avoid gluing parts of the cell together. For visualization purposes here, a cyan line indicates that the two triangles joined by it are actually the same triangle. Finally, submeshes for the cells are merged together according to the cell connectivity recovered by our algorithm (Section 7).

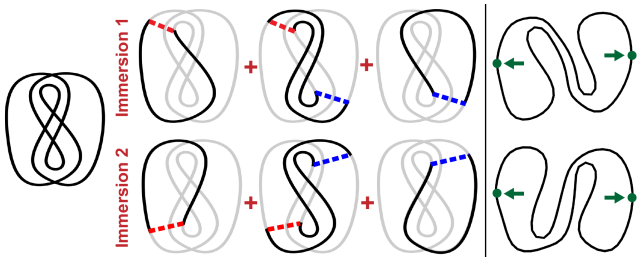

Fig. 6. **A self-overlapping curve with two distinct immersions.** Left: input curve. Middle: decompositions of the two different immersions. Right: our method is able to run on this 2D example and find both two immersions. Green dots are handles used to pull out the mesh with ARAP energy.

follow-up work [Mukherjee 2014], the original paper [Mukherjee et al. 2011] did not provide details. Later, Mukherjee [2012] presented a way to interpolate boundary curves of disk immersions based on Shor and Van Wyk's triangulation. Mukherjee [2014] gave a new method to solve the same disk immersion problem by cutting the curve at "crest points". These papers were limited to 2D. As noted by Shor [1989], a self-overlapping 2D curve can lead to more than one distinct immersion (Figure 6). Both methods of Shor and Mukherjee [2014] search for all possible immersions of a given curve. When applied to immersions of 2D disks in 2D, our method is able to find all possible immersions, paralleling these previous results. Different from previous approaches, our algorithm is applicable both to 2D and 3D immersions, and finds all possible immersions on non-disks both in 2D and 3D (e.g., tori), and on multiple-component inputs (e.g., with a hole in the shape, Figure 7).

There has been further research on 3D immersions in computer graphics. Li [2011] used Gauss diagrams from knot theory to detect

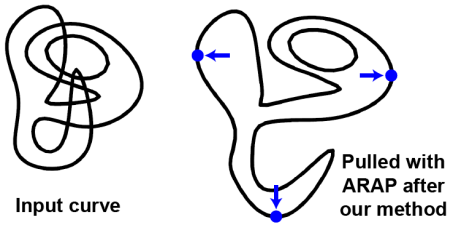

Fig. 7. **Input with multiple connected components in 2D.** The input model consists of two components, forming an egg-shaped hole. Our method is able to find the correct immersion and construct an "unglued" simulation triangle mesh, which enables ARAP to pull the model apart.

all distinct embeddings of a circle in 3D sharing the same projection as the given self-overlapping curve. Weber and Zorin [2014] introduced a method to map a triangle mesh of disk topology to arbitrary domain with potentially self-intersecting boundaries. Although operating in a 3D space, these methods have been limited to immersions onto 1D and 2D manifolds, whereas we study volume immersions onto 3D manifolds. Several mesh repair methods [Attene 2010, 2014; Campen and Kobbelt 2010; Hétroy et al. 2011] process 3D self-intersecting meshes, by gluing the mesh using a union-like operation at the self-intersections, or forming the outer hull of the model. In contrast, we generate meshes that correctly separate the self-intersecting parts. Sacht's method [2013] utilized conformalized mean-curvature flow to resolve self-intersections of a 3D surface mesh. While their method can be used for both 2D and 3D immersions, the theory behind it is limited to disk or sphere topologies (no tori of any number of handles), whereas our method is designed to process meshes of any genus; we give a comparison to Sacht's method in Section 8. Sacht's work aimed at "unwrapping" the input self-intersecting surface and did not directly aim to generate a simulation tet mesh. Although an extension whereby a tet mesh can be generated in the unwrapped space and then deformed forward to the world-space is mentioned, such a tet mesh may suffer from poor quality in case the "unwrapped" mesh is flattened (see Section 8). In comparison to Sacht, our method generates the tet mesh in the world space and as such the size and shape of tetrahedra is better controllable and suitable for subsequent simulation. We are not aware of any work, in computer graphics or computational topology or any other field, that has studied our 3D immersion problem onto volumes bounded by surfaces of arbitrary genus. Recently, Mitchell [2015] provided a method to create non-manifold level sets for nearly self-intersecting and self-intersecting meshes. Their work addresses implicit functions and focuses on self-collision resolution, whereas we operate on triangle meshes, generating an embedding into arbitrary user-provided tet meshes.

We note that several methods have utilized a self-intersecting volumetric mesh to embed an *intersection-free* surface mesh with narrow features [Molino et al. 2004; Nesme et al. 2009; Sifakis 2007; Sifakis et al. 2007; Teran et al. 2005], or for mesh cutting [Sifakis et al. 2007; Wang et al. 2014]. Our method can also operate on nearly self-intersecting inputs, producing "un-glued" self-intersecting volumetric meshes for subsequent embedded simulation. To do so, we follow the general spirit of Sifakis's method [Sifakis et al. 2007]; but greatly accelerate it by better managing the usage of exact arithmetic.

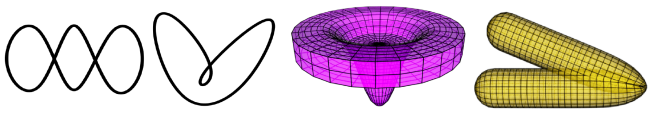

Fig. 8. Examples of self-intersecting curves in $\mathbb{R}^2$ and surfaces in $\mathbb{R}^3$ which are **not boundaries of immersions**.

In such applications, it is critical to employ exact arithmetic for triangle vs tet intersections. Sifakis's method uses constructive solid geometry. We demonstrate how to largely avoid the exact queries of constructive solid geometry, by demonstrating an exact arithmetic variant of the Sutherland-Hodgman tet vs triangle clipping algorithm [Sutherland and Hodgman 1974]. Tet vs triangle clipping has previously been used to re-embed triangle meshes into dynamically subdivided tetrahedral meshes [Wojtan et al. 2009; Wojtan and Turk 2008]. We observe a 30× overall meshing and embedding speedup compared to the exact Constructive Solid Geometry (CSG) method of Sifakis in our examples. Teran [2005] explained how to assign and duplicate the tet vertices to connect the duplicated tets, which we re-use in our work.

3 TOPOLOGY OF SELF-INTERSECTING MESHES

The input to our method is a self-intersecting orientable triangle mesh M without boundary (in $\mathbb{R}^3$), or a self-intersecting polygonal line M without boundary (i.e., closed) (in $\mathbb{R}^2$). By a triangle mesh, we hereby mean a collection of vertices with positions, and a list of integer triples specifying the vertex indices for each triangle. While such a data-structure is common in computer graphics and may even be taken for granted, it is important for our proofs to view M merely as a set-theoretic ordered pair (V, T) of vertices V and triangles T; M is **not** seen as a geometric object and should not be equated with the set of positions of all points on M. Our inputs M must have *manifold connectivity* in the usual sense, i.e., every edge must be shared by exactly two triangles, and the triangles touching a vertex must form a continuous fan. Informally, our method asks how to determine and volume-mesh the "interior volume" bounded by M, in a way that respects self-intersections and does not "glue" self-intersecting regions. In our work, we properly mathematically formulate this intuitive question using topology. While it is common in computer graphics to use the informal term "manifold mesh" to refer to any triangle mesh with manifold connectivity, self-intersecting meshes require more precise mathematical treatment; otherwise, it is not possible to soundly formulate the immersion algorithm or prove well-defined statements about it. In topology, a manifold without boundary of dimension s is a topological space where every point has a neighborhood that is homeomorphic to $\mathbb{R}^s$ (see Supplementary Material 1 for precise definitions). Self-intersecting meshes with manifold connectivity (inputs to our method) are *not* manifolds because the intersection points have no such neighborhood.

Given a triangle mesh M with manifold connectivity, we can form a 2-manifold $\hat{M}$ which intuitively corresponds to the concept of seeing M as a "topological mesh", i.e., forming one continuous surface whereby any self-intersections are properly ignored. Intuitively, $\hat{M}$ is formed by "gluing" the triangles along the shared edges. Formally, one can form it by taking a disjoint union (see Supplementary Material 1) of all triangles, and then applying the equivalence relation whereby points of the same edge shared by two triangles are made equivalent. It is easy to verify that $\hat{M}$ is a manifold (proof is in Supplementary Material 1). This manifold is an "abstract" manifold, and, when M has self-intersections, *cannot* be directly equated with some manifold in $\mathbb{R}^d$. For any point x on any triangle of M, denote by $\rho(x)$ the location of x in $\mathbb{R}^d$. It can be easily seen that the mapping ρ can be augmented to an immersion $\hat{\rho}$ of $\hat{M}$ onto $\rho(M)$ (Supplementary Material 1). An immersion is a continuous map between two topological spaces X and Y that is locally injective, i.e., for any $x \in X$, there exists a neighborhood of x such that the restriction of the map onto this neighborhood is injective (i.e., one-to-one).

We further assume that M intersects itself in a general way, i.e., the intersections are not degenerate. We formally define non-degeneracy as follows. Define the *collision set* of $x \in \hat{M}$ as

$$\gamma(x) = \{y \in \hat{M};\ \hat{\rho}(x) = \hat{\rho}(y)\} \subset \hat{M}, \tag{1}$$

i.e., points that share the same location in $\mathbb{R}^d$ with x. The "self-intersecting set" of M assembles all points x where there exists at least one more point with the same location,

$$\Gamma = \{x \in \hat{M};\ |\gamma(x)| \geq 2\} \subset \hat{M}. \tag{2}$$

We also define the *triple set*

$$\Gamma_{\geq 3} = \{x \in \hat{M};\ |\gamma(x)| \geq 3\} \subset \hat{M}. \tag{3}$$

For $d = 3$, we define M to be non-degenerate if Γ is a union of zero or finite number of 1-dimensional loops and $\Gamma_{\geq 3}$ consists of zero or more disjoint points. For $d = 2$, M is defined to be non-degenerate if Γ consists of zero or more disjoint points and $\Gamma_{\geq 3}$ is empty. For both $d = 2, 3$, we also prohibit degenerate inputs where a surface touches itself at a single point, or along a loop, but does not penetrate (exact definition is in Supplementary Material 1). For $d = 2$, an example of such a degeneracy is a circle touching a square, and for $d = 3$, an example is a solid bowl placed upside down on a solid box. Non-degenerate inputs correspond to the general way in which surfaces intersect. The set Γ typically consists of pairs of loops on $\hat{M}$. Note that any degenerate input can be perturbed by an infinitesimal amount to remove the degeneracy. To summarize the requirements on our input, we define *valid input* to be a non-degenerate orientable triangle mesh M (for $d = 3$; and a polygonal line for $d = 2$) without boundary and with manifold connectivity.

Given a valid input M, our algorithm determines if the surface immersion $\hat{\rho}$ can be extended to a volume immersion for $d = 3$, and if the curve immersion $\hat{\rho}$ can be extended to a surface immersion for $d = 2$. Formally, it answers the question of whether there exists a compact d-manifold $\hat{S}$ and an immersion $\hat{\sigma}$ from $\hat{S}$ into $\mathbb{R}^d$ such that $\hat{M}$ is the boundary of $\hat{S}$, and the restriction of $\hat{\sigma}$ onto $\hat{M}$ is $\hat{\rho}$. Compactness, intuitively, means that $\hat{S}$ contains all of its limits (exact definition is in Supplementary Material 1), and is needed to avoid immersions from unbounded manifolds, or manifolds $\hat{S}$ where one has artificially removed points from $\hat{S}$; e.g., to rule out immersions from punctured disks or similar. We call valid inputs M for which such a pair $(\hat{S}, \hat{\sigma})$ exists *volume-immersible*. The algorithm also explicitly constructs the immersion if it exists. Practically, in 3D, this means that we can construct a tetrahedral mesh that meshes the

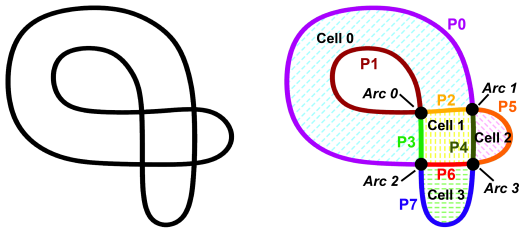

Fig. 9. **Cells, patches and arcs in 2D.** Left: input 2D curve. Right: 4 cells (open disks), 8 patches (open curves; Pi with different colors), four arcs (isolated points). Cells and their B-patches are as follows: Cell 0: P0, P1, P2*, P3*; Cell 1: P2, P3, P4, P6; Cell 2: P4*, P5; Cell 3: P6*, P7. Here, * denotes that the B-patch orientation disagrees with the cell. Arcs and the topological neighboring patches at those arcs are as follows: Arc 0: (P1,P2), (P1,P3); Arc 1: (P0,P4), (P2,P5); Arc 2: (P0,P6), (P3,P7); Arc 3: (P4,P7), (P5,P6).

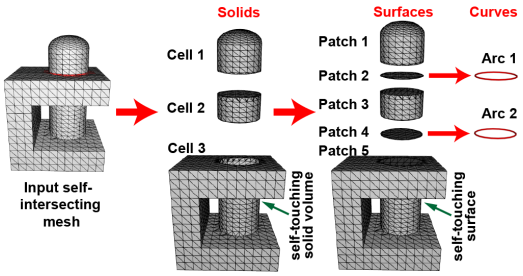

Fig. 10. **Cells, patches and arcs in 3D.**

space "occupied" by M and does not "glue" any self-intersections. A simple illustration of our method is shown in Figure 5. Note that not all closed, orientable and non-degenerate triangle meshes are boundaries of valid immersions. For example, triangle meshes shown in Figure 8 are not; intuitively, this is because these shapes require an inversion of the space, which violates local injectivity. Our algorithm is able to reject such inputs.

4 CELL COMPLEX

A self-intersecting mesh cuts $\mathbb{R}^d$ into multiple components, forming cells, patches and arcs. These components are glued together "nicely" along their boundaries and form what is known in topology as a *cell complex* [Erickson 2009]. We now introduce these concepts. They play a key role in our immersion algorithm.

4.1 Cells, Patches and Arcs

Cells are the connected components of $\mathbb{R}^d \setminus \rho(M)$ in $\mathbb{R}^d$, except the infinite component and the components with winding number 0. The latter are "interior voids" (cavities) inside objects, and should not be meshed. We will further discuss winding numbers later in the paper. Because $\rho(M)$ is a closed set, cells are open subsets of $\mathbb{R}^d$. Topologically, cells are d-manifolds without boundary (due to being open). For connected inputs M for $d = 3$, they are the interior of either a topological solid 3-ball, or a topological solid torus with $k \geq 1$ handles. We note that the torus case can occur even if the input mesh M is the boundary of a 3-ball (e.g., Cell 3 in Figure 10 is a torus with $k = 1$ hole). For connected inputs M for $d = 2$, cells are 2-manifolds that are the interior of a topological 2-disk. Justifications

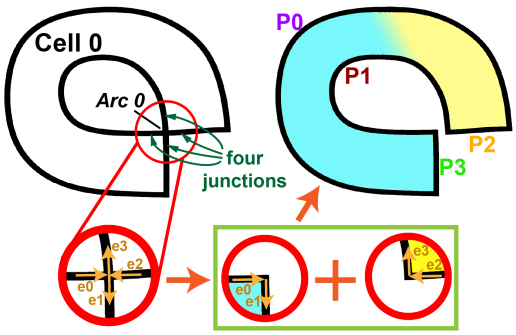

Fig. 11. **Finding patch geometric neighbors on self-touching cells.** Cell 0 in Figure 9 self-touches at Arc 0. We first identify self-touching arcs (there is only one in this example; bottom-left), then find the two components that should be disconnected (bottom-right). We use them to identify the geometric neighbor pairs at Arc 0 as (P2, P1) and (P1, P3).

of these statements are in Supplementary Material 1. The closure of a cell in $\mathbb{R}^d$ may be "self-touching" (e.g., Cell 0 in Figure 9), which we will address later. *Patches* are the connected components of $\hat{\rho}(\hat{M} \setminus \Gamma)$ in $\mathbb{R}^d$. Topologically, patches are 2-manifolds without boundary (i.e., open disks with $k \geq 0$ holes) when $d = 3$, and 1-manifolds without boundary (i.e., non-intersecting loops, or open non-intersecting curves) when $d = 2$. Patches form the boundaries between cells. *Arcs* are the connected components of $\hat{\rho}(\Gamma \setminus \Gamma_{\geq 3})$. Arcs are open subsets of $\mathbb{R}^d$. They form the boundaries of the closures of the patches, and are 1-manifolds without boundary (i.e., open non-intersecting curves) for $d = 3$, and isolated points for $d = 2$. The cell-patch-arc complex is illustrated in Figures 9 and 10.

We call two patches P_1 and P_2 *topological neighbors* if they are adjacent as per the mesh connectivity of M; formally, if $\overline{\hat{\rho}^{-1}(P_1)} \cap \overline{\hat{\rho}^{-1}(P_2)}$ is a 1-manifold ($d = 3$), or a point ($d = 2$). Here, $\overline{X}$ denotes the topological closure operation (Supplementary Material 1). This means that topological neighboring patches neighbor each other in $\hat{M}$ by a shared arc. For $d = 3$, this excludes cases where patches neighbor only in a corner point in $\hat{M}$. Figure 9 shows examples of topological neighbors.

4.2 B-patches

Each cell is surrounded by one or more patches. In our discussions, we found it convenient to call the patches surrounding a cell the *B-patches* (boundary patches) of that cell. We define two cells to be neighbors if they share a B-patch. Each B-patch has an orientation induced by the orientation of M. For $d = 3$, we define the orientation of a B-patch to *agree* with its cell if its orientation is outward from the cell, and *disagree* if inward (Figure 9). The condition for $d = 2$ is that the B-patch is oriented so that the cell interior is to the right.

For each cell, we define two B-patches to be *geometric neighbors* with respect to that cell, as follows. We first remove any singularities of self-touching cells (such as Cell 0 in Figure 9 whose B-patches P1, P2, P3 meet at Arc 0), as described in Section 4.2.1 and illustrated in Figure 11. Then, two patches are geometric neighbors, by definition, if they are spatially adjacent in $\mathbb{R}^d$, i.e., there exists an arc A such that $\overline{P_1} \cap \overline{P_2} = \overline{A}$. The purpose of this definition is that if, for a cell,

one stitches all pairs of geometrically neighboring B-patches along the common arc, one obtains the complete and *manifold* boundary of the cell. Note that without self-touching singularity removal, a non-manifold boundary would be produced. As seen in Figure 15, there can be cells with only one B-patch. Our immersion algorithm has to treat such a B-patch as a geometric neighbor of itself (by definition).

4.2.1 Geometrically Neighboring B-Patches for Self-Touching Cells. Self-touching cells are detected as follows. For each arc, observe how the patches connect to the arc, in some sufficiently small neighborhood of the arc. For valid inputs M, due to non-degeneracy, there are locally four separate patch connections to the arc; we call them the "patch junctions" (see Figure 11, top-left). Formally, we define *patch junctions* as the connected components of the intersection of a sufficiently small d-dimensional ball centered at any one arc point, with the union of all patches. Note that patches and arcs are open sets by definition; they do not contain their boundaries. Each of the four patch junctions is a subset of some patch. Some of the patch junctions may be subsets of the same patch (e.g., two patch junctions are subsets of P1 in Figure 11). A cell is self-touching at an arc if all four patch junctions are subsets of B-patches of this cell. We first describe our algorithm for handling self-touching cells for $d = 2$. To define geometric neighbors for a self-touching cell, we observe that the cell around the arc locally consists of two manifold regions (cyan and yellow in Figure 11, top-right). For each of the four patch junctions, we identify the last line segment on it, i.e., the one that touches the arc. Denote these segments by $\mathbf{e}_i$, for $i = 0, 1, 2, 3$ (see Figure 11, bottom-left). Let us orient the line segments so that the interior of the cell is on the right. Pick $\mathbf{e}_0$ to be one of the two line segments whose direction points into the arc. We then find the line segment $\mathbf{e}_1$ among the other three line segments that has the smallest counterclockwise angle with $\mathbf{e}_0$. Then, $\mathbf{e}_0$ and $\mathbf{e}_1$ form the boundary of one local manifold region of the cell, and the remaining two line segments $\mathbf{e}_2$ and $\mathbf{e}_3$ form the boundary of the other manifold region (see Figure 11, bottom-right). The B-patches containing $\mathbf{e}_0$ and $\mathbf{e}_1$ are thus assigned to be geometric neighbors at Arc 0; and same for B-patches containing $\mathbf{e}_2$ and $\mathbf{e}_3$. We note that our choice of $\mathbf{e}_0$ is deterministic, but arbitrary; an alternative algorithm would be to pick $\mathbf{e}_0$ to point away from the arc and search clockwise. In 3D, an arc is an open curve, so we arbitrarily pick one line segment on the arc, and find the triangle on each patch junction that shares the line segment. We then proceed in the same way as with $d = 2$, except we use dihedral angles between triangles as opposed to angles between line segments.

4.3 Computing the Cell Complex

We compute the cells and patches robustly using exact arithmetic [Zhou et al. 2016] available inside libigl [Jacobson et al. 2013b]. This procedure produces a "cut" version of M, where edges are added to M along its self-intersections. It utilizes the exact arithmetic kernel of the CGAL library [CGAL 2018]. We also identify all the arcs, the B-patches of all cells, the four patch junctions at each arc, the topological and geometric neighbors of each patch, and whether their orientation agrees with the cells. Patches are stored as triangle meshes, given by the indices of the triangles of the "cut" version of M. Arcs are stored as edges of the "cut" mesh. We also compute the winding number [Jacobson et al. 2013a] of each cell using libigl.

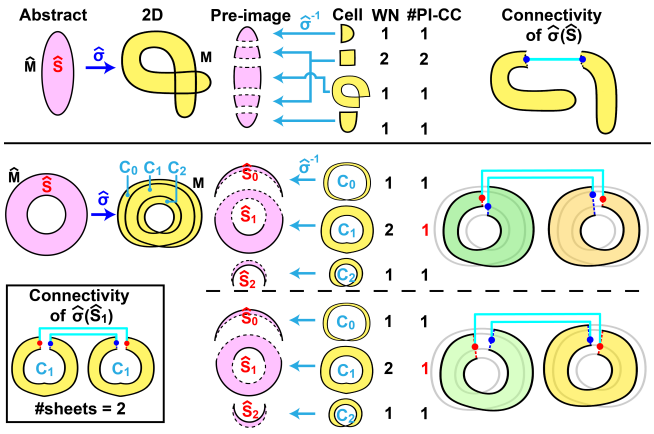

Fig. 12. **2D examples of simple and non-simple immersions.** Top: a disk-topology abstract manifold $\hat{S}$ is immersed onto 2D. Its boundary $\hat{M}$ is immersed onto M. For cell C_i with winding number (WN) w, its pre-image, $\hat{\sigma}^{-1}(C_i)$ has w connected components (PI-CC). The restriction of $\hat{\sigma}$ to each connected component in $\hat{\sigma}^{-1}(C_i)$ is a surjective immersion onto C_i, and the number of sheets for the surjective immersion is 1. Therefore, this is a simple immersion. Bottom: a ring-topology abstract manifold $\hat{S}$ is immersed onto 2D. Its boundary $\hat{M}$ is immersed onto a 2D multi-component curve, which creates three cells. There are two possible immersions (each row, bottom-right). Both immersions wind the 2D ring twice and connect it back to itself. Among the three cells, the interesting one is the ring-shaped cell C_1. In both immersions, C_1 has a winding number of 2, but the number of connected components of its pre-image is only 1 (highlighted in red in the PI-CC column). The restriction of $\hat{\sigma}$ to $\hat{\sigma}^{-1}(C_1)$ (which has a sole connected component) is a surjective immersion where the number of sheets is 2 (bottom-left). Intuitively, this means that $\hat{\sigma}$ "wraps" $\hat{\sigma}^{-1}(C_1)$ onto C_1 twice. Therefore, $\hat{\sigma}$ is not a simple immersion.

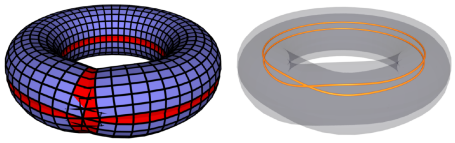

Fig. 13. **3D example of a non-simple immersion.** Left: a torus mesh which winds around the center axis twice, forming a 3D self-connecting cell. Intersecting mesh faces are colored red. Right: the orange curve is the skeleton of the torus mesh.

5 IMMERSION GRAPH

Intuitively, immersions can be formed by "unwrapping" the volume "occupied" by M. During this process, some cells of our cell complex (Section 4) will require multiple copies to form the "unwrapped" d-dimensional manifold $\hat{S}$. The core idea of our algorithm is to construct the manifold $\hat{S}$ by properly gluing together replicated copies of cells from the cell complex. The information about which cell copy should be glued to which other cell copy and where (across which patch) can be naturally encoded into a graph. Our algorithm will therefore operate on graphs whose nodes are duplicated cells

of the cell complex of M, and edges denote "gluing" of cell copies across shared B-patches (see Figure 5, top-right).

We note that our graph is in general a *multigraph*, i.e., two nodes may be joined by more than one edge. This is because a cell may have more than one B-patch shared by the same neighboring cell (e.g., Cells 0 and 1 share Patches 2 and 3 in Figure 9). Therefore, each graph edge needs to record which B-patch it crosses. The requirements in this paragraph define the set of *eligible graphs*. Additionally, due to the requirement that the input surface $\rho(M)$ must be the boundary of the immersed volume $\hat{\sigma}(\hat{S})$ (i.e., the restriction of $\hat{\sigma}$ onto $\hat{M}$ is $\hat{\rho}$), each patch must be on the external geometric boundary of exactly one node. We say that this node "owns" this patch, and all other nodes "declined" it. It is the task of our algorithm to form the graph and find out which node owns each patch.

5.1 Simple immersions

The next question to address is then how many copies of each cell do we need to make in our immersion graph? This simple question turned out more difficult to answer than expected. Mukherjee [2011] hypothesized, without proof and in two dimensions only, that the number of copies of a cell in $\hat{S}$ has to equal its winding number with respect to M. We observe and prove that this is true for immersed disks (both for $d = 2$ and $d = 3$), but for general inputs (both for $d = 2$ and $d = 3$), it is not true. The issue are *self-connecting shapes* (Figure 12,13) that contain a cell whose two (or more) node copies connect *to each other*. In the presence of such cells, the winding number hypothesis no longer holds.

We address self-connecting cells as follows. First, observe that, if there exists a volume immersion $\hat{\sigma}$ from some $\hat{S}$ onto $\mathbb{R}^d$ whose boundary is $\hat{M}$, then the pre-image of each cell $\hat{\sigma}^{-1}(C) \subset \hat{S}$ consists of a finite number of components (proof in Supplementary Material 1). The restriction of $\hat{\sigma}$ to any component $\mathcal{S}$ in $\hat{S}$ is a surjective immersion from $\mathcal{S}$ onto C, but is not necessarily globally injective. It can be easily shown (Supplementary Material 1) that that this surjective immersion is what is known in algebraic topology as a *covering projection* of $\mathcal{S}$ onto C. It follows that the pre-image of each individual point $x \in C$ has the same finite cardinality for all $x \in C$, called the "number of sheets" (Supplementary Material 1). Visually, this means that the immersion wraps $\mathcal{S}$ onto C number of sheet times, e.g., 2× in Figure 12. Guided by our application of generating unglued tet meshes, we define that the immersion $\hat{\sigma}$ is *simple* if the number of sheets for each cell C is 1, i.e., all cell immersions are globally injective and therefore bijective. We define a valid input to be *simply volume-immersible* if it is volume-immersible with a simple immersion. The geometric interpretation of simple immersions is that they disallow self-connecting cells, by definition. Because modeling self-connecting cells is rarely the intent in computer graphics, we think it makes sense to develop our theory and algorithms for simple immersions. We now state our first two theorems, establishing that the winding number intuition holds for simple immersions, and that immersions from disks are automatically simple, both for $d = 2$ and $d = 3$.

Theorem 1: If a valid input M is simply volume-immersible, then for each cell C of the cell complex of M, the number of connected components of $\hat{\sigma}^{-1}(C)$ equals the winding number $\text{wind}(x, M)$ of any point $x \in C$ with respect to M. The winding number $\text{wind}(x, M)$ is the same for all points $x \in C$. This holds both for $d = 2$ and $d = 3$.

Theorem 2: If a valid input M is the boundary of an immersion from a disk in $\mathbb{R}^d$, then the immersion is simple. This holds both for $d = 2$ and $d = 3$.

We prove these theorems, using algebraic topology, in Supplementary Material 1. The following Corollary proves Mukherjee's intuition for immersions of disks, both for $d = 2$ and $d = 3$.

Corollary 3: For a valid input M that is the boundary of an immersion from a disk in $\mathbb{R}^d$, the number of connected components of $\hat{\sigma}^{-1}(C)$ equals the winding number of C with respect to M. This holds both for $d = 2$ and $d = 3$.

Proof: This follows directly from Theorems 1 and 2. ■

5.2 Rules for Graph Edges

We now proceed to defining a set of rules that further restrict the edges of eligible graphs, and the specific B-patch ownership assignment, such that the graphs and B-patch ownerships that satisfy them correspond to volume immersions of M. The rules are also shown in Figure 14. We start by the assumption that M is volume-immersible, and derive the rules as a necessary consequence of this assumption. We will later prove the converse, namely that any eligible graph that satisfies all the rules gives a simple volume-immersion.

Theorem 4: Suppose M is simply volume-immersible. Then, the following conditions must be satisfied for the cell graph and B-patch ownership assignment:

(1) A node has at most one edge across each B-patch.
(2) A node cannot have an edge across a B-patch it owns, and must have an edge across one it declines.
(3) A node from cell C must decline a B-patch whose orientation does not agree with C.
(4) If nodes c_1, c_2 from cells C_1, C_2 are connected across patch q, and p_i is a B-patch of C_i, and geometric neighbor of q, for $i = 1, 2$, and p_1, p_2 are topological neighbors, then either both c_i own p_i, or both decline p_i. Conversely, if c_i owns p_i, for both $i = 1, 2$, and patches p_1, p_2 are topological neighbors, then c_1 and c_2 must be connected across q.
(5) If a node c from cell C owns a patch p_1, and p_1 is a geometric neighbor of p_2 on C, then c must decline p_2.
(6) If four nodes c, d, e, f surround the same arc A, and x connects to y across a patch that neighbors A, for (x, y) equaling (c, d) and (d, e) and (e, f), then c must connect to f across a patch that neighbors A. This also applies to the case where c and e are the same node, and/or d and f are the same node. More details are in Figure 15.
(7) Each patch is owned by exactly one node.

Proof: If Rule 1 was violated, then a B-patch would be shared by at least three nodes. For any point x on the shared B-patch, its neighborhood includes a part of interior of all three nodes, which contradicts local injectivity of $\hat{\sigma}$. Note that Rule 1 implies that a node from a cell that has k B-patches has at most k edges. Rules 2, 3 and 4 follow from the requirement that $\hat{M}$ is the boundary of $\hat{S}$, and the restriction of $\hat{\sigma}$ onto $\hat{M}$ is $\hat{\rho}$. Note that Rule 2 establishes an equivalence between B-patch ownership and the absence of an edge connection. Rule 5 can be proven similarly to Rule 1. Suppose

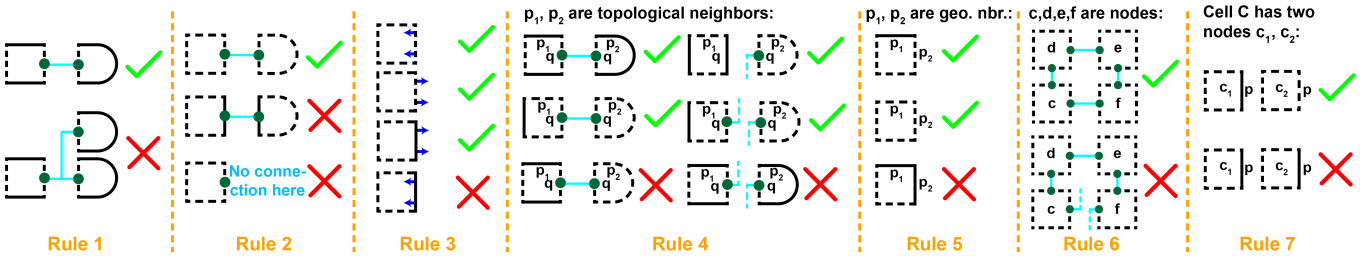

Fig. 14. **Immersion graph connectivity rules.** Solid black curves denote "owned" patches, dashed black curves denote "declined" patches, cyan lines are connections between nodes, and blue arrows specify patch orientations. Checkmarks and red crosses denote that a rule is followed or violated, respectively.

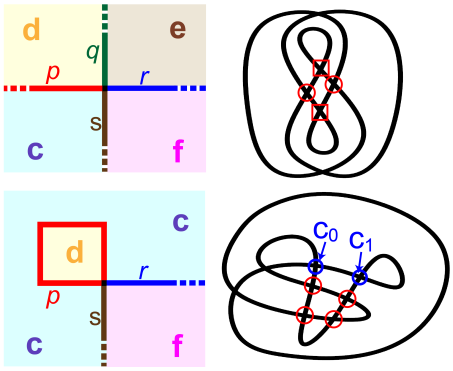

Fig. 15. **Examples of Rule 6:** Top-left: one common case of Rule 6, where four patches p, q, r, s and four nodes c, d, e, f from four cells join at the same arc. Bottom-left: a rare case of Rule 6, where c and e are the same node and p and q are the same patch. Right: examples of such situations. Red circles denote the four-patch cases. Red rectangles denote the three-patch cases. Note that the rule does not apply to arcs c_0 and c_1 highlighted with blue circles in the bottom-right figure, because they both have a surrounding empty cell that does not belong to the cell complex of M.

p_1 and p_2 are both owned by c, then pick a point x at the arc shared by p_1 and p_2. Any neighborhood of x includes a part of interior of p_1 and p_2. However, as p_1 and p_2 are B-patches of the same cell and are geometric neighbors, they cannot be topological neighbors. This neighborhood of x contradicts local injectivity of $\hat{\sigma}$. Rule 6 is proved as follows. In Figure 15 (top-left), node c connects to d across patch p, and d connects to e across q, and e connects to f across r. If c does not connect to f across patch s, then by Rule 2, c must be connected to another node f_2 at the same cell as f, and the same for f. Then, since c has declined p, by Rule 4 f_2 must decline patch r and connect to another node e_2 from the same cell as e. By Rule 4 again e_2 should connect to another node d_2 and d_2 to c_2. Repeating this process either requires an infinite number of nodes, which is a contradiction, or requires c_k to connect back to f. This last case implies that we constructed an immersion of a disk into $\mathbb{R}^2$, such that the disk boundary winds around the immersed disk more than once. This contradicts the total turning number Lemma given in Supplementary Material 1. Therefore, f must connect to c across s. Similar reasoning can be applied to the other situations of Rule 6 (e.g., Figure 15, bottom-left). ■

5.3 Valid Graphs Give Immersions

As suggested in Figure 5, finding $\hat{\sigma}$ involves replicating cells and connecting them so that the $\hat{M}$ is the immersion boundary. We now prove a theorem that makes it possible to construct an immersion if the graph rules from Section 5.2 are satisfied.

Theorem 5: Let M be a valid input. Assume that there is a cell graph and a B-patch ownership assignment that satisfy Rules 1-7 given in Section 5.2. Then, M is simply volume-immersible, and the graph can be used to construct the immersion.

Our proof of this theorem is very technical. We give the complete proof in Supplementary Material 1. The key intuition is to employ disjoint unions of the cells belonging to each graph node, and glue the cells according to the edges of the immersion graph. One then uses Rules 1-7 of Section 5.2 to prove that that there is a manifold neighborhood of every point x. In order to do so, one has to analyze the different cases for the position of x (interior of cell, on an owned boundary, on a declined boundary, etc.). We can now state and prove our final immersion result:

Corollary 6: For a valid input M, M is simply volume-immersible if and only if there exists a cell graph and a B-patch ownership assignment that satisfy Rules 1-7 of Section 5.2.

Proof: This follows directly from Theorems 4 and 5. ■

6 ALGORITHM TO DISCOVER IMMERSIONS

We say that a graph G and patch ownership information are *compliant* if they satisfy Rules 1-7. Our algorithm (Algorithm 1) searches the space of eligible graphs and ownership assignments to discover compliant ones. Figure 16 shows an example of algorithm execution. Although in principle one could enumerate all eligible graphs and ownership assignments and test each one, we give an algorithm that explores the space of all eligible graphs much faster. Our algorithm builds the graph one edge at a time. At each step, it examines all cells for possible connections that could be added, and adds it to a cell where the Rules 1-7 force a single choice. If there is more than a single choice at each cell, it then makes an arbitrary choice and pushes the other choices onto a stack for future traversal. In many cases in practice, Rules 1-7 are powerful enough that there is always a forced choice to be made. However, in more challenging cases such as those in Figure 6, there are two or more distinct immersions, necessitating stack use to discover all immersions.

In addition to the "owned" and "declined" ownership information, we use a third state during the course of the algorithm, "undecided". Initially, all relations are initialized to "undecided"; at the end of the

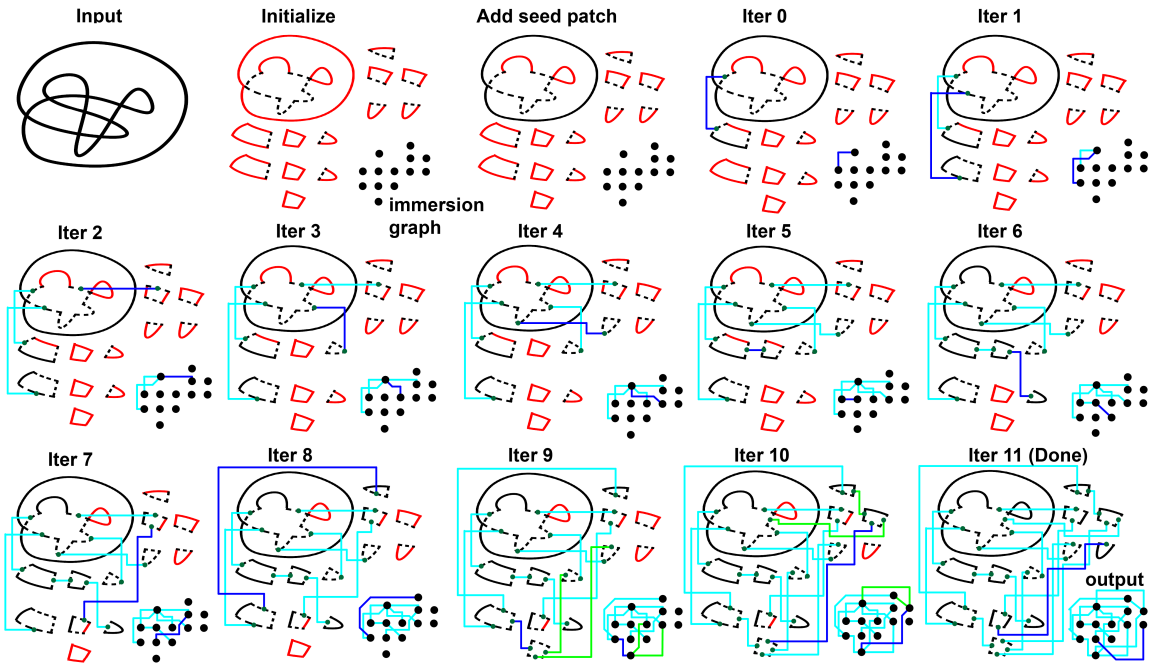

Fig. 16. **An example of the execution of our algorithm to construct an immersion.** The input is the 2D curve shown in the bottom-right of Figure 15. Red, solid black and dashed black curves denote "undecided", "owned" and "declined" patches, respectively. At each iteration, a blue line represents the first valid connection between two nodes, green lines are the subsequent connections made by Rule 6 after the first connection, and cyan lines are previous connections. This is a complete example showing all 11 iterations of our algorithm to produce the output immersion graph (bottom-right).

algorithm they must all be either "owned" or "declined". In addition, a declined B-patch p on node c is tagged as *incomplete* if there is no edge from c across p. Note that by Rule 2, there cannot be incomplete patches; hence these states are transient and must disappear by the end of the algorithm.

We first pick any cell that has at least one B-patch whose orientation agrees with the cell and let the node own that B-patch. At each iteration, we first check whether the graph is compliant by checking whether there are incomplete B-patches. If not compliant, we try to find a node to connect across an incomplete B-patch. For any candidate, we use the function connectNodes to check whether the connection satisfies Rules 1-7. This function first updates the patch ownership of the two nodes, then recursively propagates the change to other nodes as needed. Whenever any of the Rules becomes un-satisfiable during the process, the function returns failure. If connectNodes returns success on connecting node a to b, and there is no ambiguity where there is another node that can also be connected to a successfully to create a different graph, then we connect a to b. We continue building the graph until encountering an ambiguity that cannot be avoided. Then, we push the current graph and patch ownership on a stack, pick one possible connection, and continue. When done with the algorithm, we pop the graph and patch ownership from stack, make another choice, and continue.

Theorem 7: For a valid input M, M is simply volume-immersible if and only if our algorithm terminates successfully, i.e., discovers a graph and patch ownership assignment that satisfies all rules. If it terminates successfully, it also explicitly constructs the immersion.

Proof: Suppose the volume immersion exists. Then, by Corollary 6, there exists a cell graph and patch ownership assignment with the stated properties. Because at each step, our algorithm considers all possible steps that do not violate the rules, our algorithm tests all possible graphs. Therefore, it will discover the compliant cell graph and patch ownership assignment, i.e., terminate successfully. By Theorem 5, the graph then gives the immersion. Suppose the algorithm reports that there is no graph and ownership assignment satisfying the rules. Then, by Corollary 6, M is not simply volume-immersible. ■

6.1 Running time

For $d = 2$, it has been proven by Eppstein and Mumford [2009] that the problem of determining if the given planar curve is a boundary of an immersed surface, is NP-complete. The size of the problem is measured as a number of intersections (arcs in 2D), or equivalently, patches or cells. In our Supplementary Material 1, we prove that the problem is NP-complete also for $d = 3$, and therefore the running time of our algorithm has to be theoretically non-polynomial (unless $P = NP$). In practice, however, the running time is very small even for very complex examples with high winding numbers and genus. It is always dominated by the meshing time (Table 1).

6.2 Extensions

Our algorithm can be extended to non-connected inputs M consisting of multiple connected components. This can be achieved by running the algorithm as usual. When the algorithm runs out of

ALGORITHM 1: Test if M is simply volume-immersible; if yes, construct the immersion graph and patch ownerships.

Function `isVolumeImmersible(`M`)`
- generate cell complex for M ;
- create nodes of graph G at each cell, #nodes = winding number of cell ;
- edges(G) $\leftarrow \emptyset$; pick any node a, make it own any B-patch p;
- $S \leftarrow$ geometric neighbors of p on a + available connections;
- stack $\leftarrow$ empty ; push G, patch ownerships P, and S onto stack;
- **while** stack *not empty* **do**
  - pop G, P, S from stack;
  - **while** *exist incomplete B-patches in* S **do**
    - **for** *each incompl. B-patch at node* n*, cell* C*, B-patch* p *in* S **do**
      - connectableNodes $\leftarrow$ 0 ; $D \leftarrow$ cell that neighbors C at p ;
      - **if** *#isolated nodes of* D *in* $S \geq 1$ **then**
        - connectableNodes $\leftarrow$ 1 // always connectable;
        - nodeToConnectTo $\leftarrow$ any isolated node of D in S ;
      - **for** *each non-isolated node* m *of* D *in* S **do**
        - status = `connectNodes(`n, m`)` ;
        - **if** *status* == success **then**
          - reset the effect of `connectNodes(`n, m`)` ;
          - connectableNodes ++ ; nodeToConnectTo $\leftarrow$ m ;
      - **end**
      - ambiguous $\leftarrow$ (connectableNodes ≥ 2) ;
      - **if** ambiguous **then continue** // next incomplete B-patch;
      - **else break** // connect non-ambiguously;
    - **end**
    - **if** *ambiguous* **then** push G, P, dropLastEntry(S) onto stack;
    - status = `connectNodes(`n, nodeToConnectTo`)` ;
    - $S \leftarrow$ current incomplete B-patches + available connections ;
    - **if** *status* == failure **then return** failure;
  - **end**
- **end**
- **return** G and P // success ;

end

Function `connectNodes(`a, b, p`)` *// connect nodes* a *and* b *across patch* p
- set a and b to decline p ; `updateNode(`A`)` ; `updateNode(`B`)` ;
- **for** *each non-connected node pair* (c, d) *in* G **do** `// apply Rule 4`
  - **if** *their owned patches* q, r *are topological neighbors* **then**
    - find patch s between c and d that shares arc with q and r ;
    - `connectNodes(`c, d, s`)` ;
- **end**
- **for** *each node set* c, d, e, f *around an arc where Rule 6 applies* **do**
  - connect c and f by calling `connectNodes()` ;
- **if** *any function called above failed* **then return** failure;
- **return** success;

end

Function `updateNode(`a`)` *// update patch ownership for* a *and its neighbors*
- **for** *each node* b *connecting to* a **do**
  - apply Rule 4 to update a's patch ownership based on b ;
- apply Rule 5 to decline patches of a ;
- **for** *each owned patch* p *of* a **do**
  - **for** *each node* b *having an undecided patch* p **do**
    - set b to decline p ; `updateNode(`b`)` // apply Rule 7 ;
- **end**
- **if** *a's patch ownership has been updated in this function* **then**
  - **for** *each node* b *connecting to* a **do** `updateNode(`b`)` // apply Rule 4;
- **if** *Rule 3, 4, 5 broken at* a*, or a function above failed* **then return** failure;
- **return** success;

end

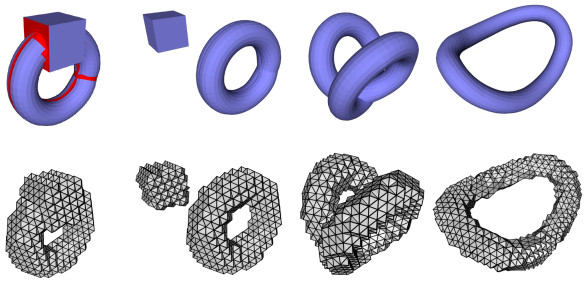

Fig. 17. **Resolving the double-loop torus.** A cube is added to cut the self-connecting cell. Left to right column-wise: the model (collided faces in red) and our tetrahedral mesh, the helper cube pulled away to show the torus mesh, volumetric ARAP deformation exposing the topology, the double-loop torus further deformed into a single-loop.

valid connections to make, but there are still undecided patches at nodes, we then pick one undecided patch and assign it to a node, and continue the algorithm.

We can also extend our algorithm to handle non-simple-immersions that have self-connecting cells (Figure 12). Such immersions require connecting nodes of the graph to nodes that are copies *of the same cell*. They have limited applicability in computer graphics practice, but can introduce an arbitrary complex subgraphs on nodes from the same cell. Designing an algorithm to connect those nodes is quite daunting. Instead, we adopt a simple workaround: we cut the self-connecting cells. We do this by forming a box whereby one of the six faces intersects the self-connecting cell. We modify the input to be M plus the box. Note that we did not perform any CSG here; the input consists of two separate meshes, namely M and the box. We then run our algorithm. During the cell complex creation, the original self-connecting cell will disappear because it was cut by a box face. The output tet mesh will have two components: one for M which we keep, and the other for the box mesh which we discard (see Figure 17). If there are still self-connecting cells remaining, we can add more boxes for more cuts. We always place the box in such a way that it intersects the self-connecting cell along its longest axis. Because our inputs are meshes with a finite number of line segments or triangles, we will, in a finite number of cuts, reach a situation where the resulting cells are not self-connecting any more. Note that by Corollary 3, inputs M that are topologically spheres are guaranteed not to produce self-connecting cells and do not require this extension. Even for inputs that are topologically tori with $k \geq 1$ handles, we did not encounter self-connecting cells, unless we created such an example on purpose.

7 CELL TETRAHEDRALIZATION

The boundary of our volume immersion matches the input triangle mesh. So far, no tet mesh was needed in our work; building the cell complex and the immersion algorithm only require the input triangle mesh. We now address the situation where a given input tet mesh embeds the input triangle mesh. Arbitrary tet meshes can be used, as long the volume enclosed by the input triangle mesh is a subset of the volume covered by the tet mesh. This makes our method versatile, as it permits to easily adjust the tet mesh resolution. For example, such a tet mesh can be obtained by voxelizing, or by

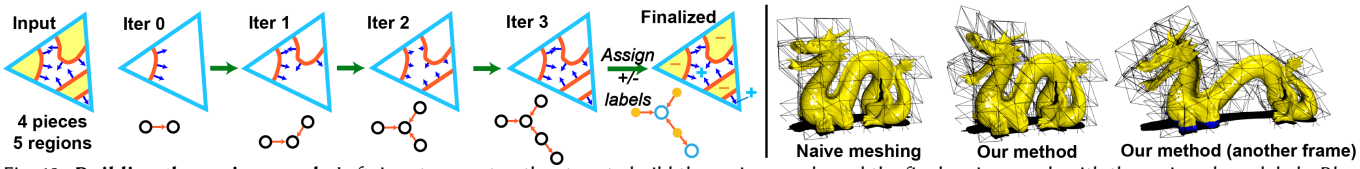

Fig. 18. **Building the region graph.** Left: input geometry, the steps to build the region graph, and the final region graph with the assigned +,− labels. Blue arrows are the triangle normals. Black empty circles are region graph nodes. Right: nearly self-intersecting dragon. Naive meshing glues the mouth and back and produces almost no motion in animation; whereas our method produces good dynamics with a coarse mesh.

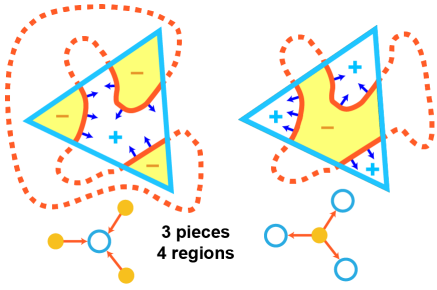

Fig. 19. **Examples of geometry inside a tetrahedron.** Top row: 3 pieces partition the tet into 4 regions. The embedded triangle mesh is shown dashed. Blue arrows represent triangle normals. Regions labeled + are outside of the embedded triangle mesh. Regions labeled − are inside of the triangle mesh, and are a part of Ω. Note that the left and right image have identical piece geometry, but opposite +, − labelings, due to the opposite piece orientation. Bottom row: corresponding region graphs. The nodes of the + and − regions are denoted by empty blue and solid yellow circles, respectively. Each orange arrow represents a directed edge corresponding to a piece.

computing a BCC lattice [Molino et al. 2003b] of the input triangle mesh, and then flood-filling the interior with tets.

We now show how to "unglue" the given tet mesh to respect the immersion. For each cell c, we generate the submesh T_c of T containing the tets of T that are intersecting c. To avoid gluing proximity features on the surface of c, we apply our "virtual tets" algorithm to T_c (Section 7.1). Next, we assign to each immersion graph node that is a copy of c a unique copy of T_c. Finally, we connect the tet meshes based on the edges of the immersion graph, as follows. Suppose graph nodes a and b are connected by an edge across a B-patch p. Then, we fuse two identical tets from tet meshes of a and b if they both embed the same triangle of p. Note that if the tet mesh covers a larger space than the input triangle mesh, the immersion will not follow the boundary of the tet mesh.

7.1 Virtual tets

The input to our virtual tets algorithm is a closed manifold self-intersection-free triangle mesh, and a tet mesh that covers the volume Ω enclosed by the triangle mesh. We give a novel fast algorithm to "virtualize" the tet mesh, i.e., duplicate tetrahedra that cover more than one "local region" of Ω. Each tet copy knows what "local region" of Ω it is embedding, and we then connect the copied tetrahedra into one consistent global mesh according to the connectivity of Ω. Our algorithm is similar in goal to [Sifakis et al. 2007], but is more than an order of magnitude faster. We note that this idea of "duplicate and connect" appears twice in our paper: once in finding the volume immersion, and the second time here. Sifakis used CSG to cut the triangle mesh with each tet to find out the local regions. However, the CSG operations are slow, and are the bottleneck of virtualization for complex examples. Our method uses the Sutherland-Hodgman tet-triangle clipping algorithm [Sutherland and Hodgman 1974] in exact arithmetic, and pseudo-normal tests to eliminate CSG, which boosts performance between $10-30\times$ in our examples.

7.1.1 Duplicating Tetrahedra. We apply the Sutherland-Hodgman algorithm [Sutherland and Hodgman 1974] to clip triangles against each tet. The clipped triangles form connected components (we call them "pieces") inside each tet. If there are no triangles inside the tet, then this is an interior tet, and the entire tet volume is one local region in Ω. Otherwise, the pieces partition the tet volume into disjoint regions. We label the regions with a + if they are outside of Ω, and − if inside. We now give an algorithm to label the regions, and identify the pieces that form the boundary of each region.

Similarly to our immersion graph, we form a *region graph*, where each region is a node and each piece corresponds to an edge (Figure 19) joining two regions that have this piece as their shared boundary. The region graph is connected and without cycles, i.e., a tree. The proof is simple: if we incrementally add pieces one by one into the tet, since each piece is a subset of a manifold triangle mesh, it always partitions the tet volume into two volumes. Since the mesh is self-intersection-free, a newly added piece will only subdivide one region. Therefore, if there are k pieces in a tet, we will form $k+1$ regions. Since the triangles in each piece are oriented consistently, we can direct edges from the inside to the outside region.

The region graph is built incrementally as follows (see also Figure 18). We initialize it by placing only one piece into the tet, creating two region nodes and one piece edge. When adding a new piece P, we first identify which region will be subdivided by P. This is done by querying an arbitrary vertex v on P against the boundary surface of every region, until we discover the one that contains it. In Supplementary Material 2, we prove that we do not need to explicitly form the region boundary surface (which would require using CSG) to query this. It is sufficient to only perform inside-outside queries on the boundary pieces of the region. We first use our region graph to identify the pieces that bound this region, then we find the closest site on each piece to v. Next we perform the pseudo-normal test against the closest site [Bærentzen and Aanæs 2005] to identify if v is inside or outside each piece. After we find the region ν that contains v (and therefore P), we subdivide ν into ν_+ and ν_-, split its node in the region graph into two nodes, and connect them to each other with the piece edge of P. For a piece Q connecting ν before P is added, we move it to either ν_+ or ν_- by testing whether an arbitrary vertex w on Q is outside or inside P, using the same pseudo-normal test as done on v. Since pieces do not intersect, a

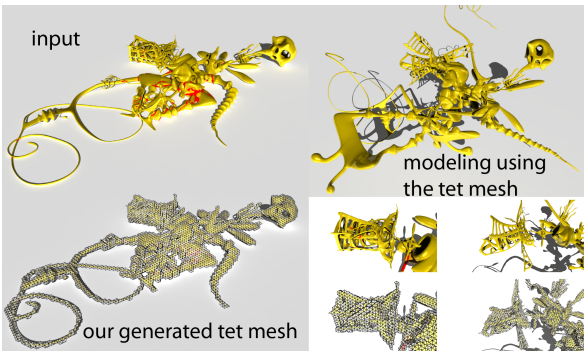

Fig. 20. **Yeahright-on-ground model.** Top left: Yeahright dropped on the ground. Collided faces are in red. Bottom left: generated tetrahedral mesh. Top right: we deformed the Yeahright-on-ground using volumetric ARAP on the tetrahedral mesh. Bottom right: closer look on how our method is able to untangle the intersecting features.

piece cannot be both inside and outside of P. After all the pieces are added to the region graph, we find all $-$ regions by checking whether they are inside their boundary pieces, i.e., they have only outward edges in the region graph. We then duplicate this tet by the number of $-$ regions and for each region we embed the triangles of its boundary pieces into the tet for this region.

7.1.2 Connecting Duplicated Tetrahedra. We connect a pair of neighboring tets if their embedded pieces share points on the boundary. The boundaries of each piece are polygonal lines lying on tet faces. If the shared tet face between the two tets does not have piece boundaries from either tet, and an interior point on the face is considered inside for the pieces in both tets (again by pseudo-normal test), we must also connect those tets. Points on faces of a tet with no pieces are considered inside in the above test.

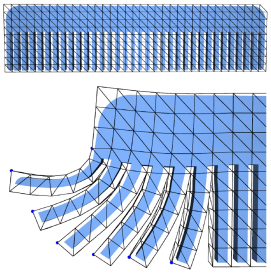

Fig. 21. **Duplication of tet vertices** to embed a comb.

Once we know which tets should be connected, the assignment and duplication of tet vertices is performed as explained in [Teran et al. 2005], Section 6. In short, each copied tet is initially assigned a totally unique set of 4 vertices. If two tets are connected, we merge the three pairs of vertices on the shared tet face. The merging is done using a disjoint-set data structure. Such tet vertex duplication produces correct topological connectivity of the tet mesh. This is illustrated in Figure 21 whereby none of the teeth of the comb are welded to adjacent teeth.

7.1.3 Details. For more detail and robustness considerations of our algorithm, please refer to our Supplementary Material 2. The tetrahedralization algorithm can be easily adapted to 2D to embed polygonal lines into triangles.

8 RESULTS

We give the statistics of our method in Table 1 (Intel Xeon 2.3GHz 2x6 cores, 32 GB memory). An illustration of the progress of our

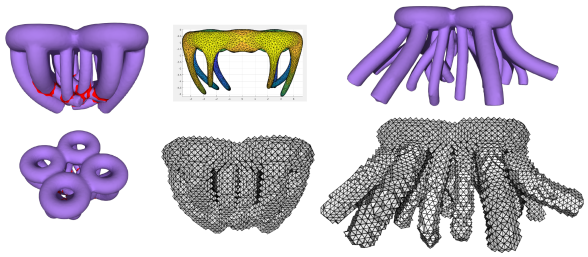

Fig. 22. **Quintuple Torus model.** Top-left: side view of the Quintuple Torus. Collided faces are in red. Bottom-left: top view with holes visible. Top-middle: Sacht's implementation became stuck in the reverse flow due to the complicated collisions, spending 8 hours without progress until we stopped it. Bottom-middle: our algorithm successfully generated a tetrahedral mesh to embed the shape. Top-right: we simulate the output mesh using FEM. Bottom-right: the deformed tetrahedral mesh.

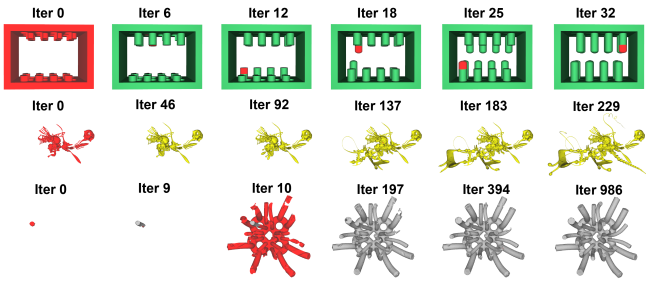

Fig. 23. **Intermediate stages of building immersion graphs.** The three rows correspond to the examples of Sacht (small), yeahright-on-ground and quintuple torus. Leftmost column: the first node added to the immersion graph. Rightmost column: the final node added.

immersion algorithm is shown in Figure 23. To help with implementation, we released the source code under a free BSD license as a part of Vega FEM 4.0 [Barbič et al. 2018]. The released code includes both self-intersecting meshing (Sections 4, 5, 6) and nearly self-intersecting meshing (Section 7). The input tet meshes in our examples are generated by red-green subdivision on tet grids [Molino et al. 2003a], or voxelization implemented in [Barbič et al. 2018]. We have tested our method on multiple meshes. The upper and lower lips in the *head* model in Figure 2 self-intersect. Our immersion method was able to generate an overlapping, but separated tetrahedral mesh despite the self-intersections, which makes it possible for FEM simulation to open the mouth. We can achieve this even with a very coarse mesh that only has 7 tets across the length of the mouth. We note that prior work [Li et al. 2016] achieved similar functionality by meshing the head in a deformed intersection-free configuration. This prior work, however, is tedious in practice because generating deformed intersection-free configurations cannot be easily automated, and requires substantial additional work and book-keeping in a computer animation pipeline. The *helix* model (Figure 1) consists of 100 tightly self-overlapping loops. Our method is able to mesh and animate it correctly. We generated the *yeahright-on-ground* model (Figure 20) by modifying the "yeahright" model from Keenan Crane's 3D Model Repository [Crane 2017]. Both the original model and our model have genus 131. The modification is that we dropped the original model under gravity using a FEM simulation, and let it come to rest on a plane without any self-collision

Table 1. **Immersion algorithm statistics.** t-cell, t-imm, t-mesh: times for generating the cell complex, immersion algorithm and meshing.

Model	vertices	triangles	genus	cells	patches	arcs	max winding #	output tets	t-cell	t-imm	t-mesh	t-total
head	3,020	6,036	0	3	3	2	2	566,515	0.34s	0.0090s	38.0s	38.4s
helix	100,020	200,036	0	3	3	2	2	36,182	56.4s	0.0003s	190s	246s
yeahright-on-ground	94,063	188,646	131	117	263	362	4	83,024	14.2s	0.12s	57.1s	71.4s
quintuple torus	15,211	30,438	5	314	886	1,700	7	84,905	11.6s	4.89s	35.9s	52.4s
tree	163,998	328,152	40	131	259	258	2	32,457	15.9s	0.19s	130s	146s
Sacht (small)	12,448	24,896	1	26	49	48	2	112,554	2.77s	0.0015s	19.7s	22.5s
Sacht (large)	24,160	48,320	1	41	79	78	2	225,338	6.01s	0.0045s	42.2s	48.2s

handling. This resulted in a massively self-intersecting shape of genus 131. We call this shape the "yeahright-on-ground", and we use it as input to our paper. Our method was able to generate an unglued tetrahedral mesh for this severely self-intersecting shape (Figure 20), which enabled us to correctly pull it apart using volumetric ARAP. The high genus of the model causes zero-area faces in the forward flow, which causes Sacht's method to fail, whereas our method succeeds. In the *quintuple torus* example, we attached 24 cylinders onto a quintuple torus (genus 5) and collided the cylinders (Figure 22), forming cells of a high winding number 7. Our method generated a quality tetrahedral mesh that is capable of separating all of the collided cylinders. In the tree example (Figure 3), we apply our method to procedurally generated geometry, and in the dragon example (Figure 18), we mesh nearly self-intersecting geometry. In addition to direct modeling and animation, our method also enables topology-aware computation of volumetric weights such as bounded biharmonic weights [Jacobson et al. 2011] (Figure 4), suitable for further use in shape deformation and simulation.

In Figure 24, we compare our work to Sacht's method. We tried all of the non-inverted inputs of [Sacht et al. 2013] and our method was successful on all of them. Our work can accommodate non-spherical inputs (genus ≥ 1), whereas Sacht's method often fails on such inputs; for example, it fails to produce any meaningful result on the yeahright-on-ground and quintuple torus models. We also tried the torus example that Sacht's work listed as problematic for their method ("Sacht large" example in Table 1), and confirmed that Sacht's method fails whereas ours produces a good result. We then decreased the difficulty of the example by removing some vertical columns ("Sacht small"), upon which Sacht's work succeeded, but, due to meshing in the unwrapped configuration, produces a very suboptimal tet mesh compared to our world-space meshing (Figure 24, B). Furthermore, because Sacht's method requires repeated self-collision detection at each iteration, whereas we only need it once in building the cell complex and once for tetrahedralization, our method is more than 100× faster than Sacht's method.

We compared our tetrahedralization method (Section 7) with the CSG method of [Sifakis et al. 2007] on the yeahright-on-ground model (Figure 20). We note that prior work [Sifakis et al. 2007] did not address self-intersecting inputs (only nearly self-intersecting), but we hereby use the prior work to replace our tetrahedralization algorithm. In contrast, the prior work [Sifakis et al. 2007] needed 2,545 seconds total, with more then 98% of the time spent in running CSG operations to duplicate tetrahedra. We use the same CSG routines ([Zhou et al. 2016]) when implementing both our method and the prior work. Our profiling shows that, in prior work, 97% of the total time is spent in exact-arithmetic mesh booleans. Our method does not need mesh booleans for tetrahedralization, and requires a lot fewer exact arithmetic operations (Table 2). We observed similar speedups also on smaller examples.

Table 2. **Timing comparison to Sifakis's algorithm.** We report the total time spent in four key CGAL exact arithmetic routines when using CSG operations as in [Sifakis et al. 2007], versus our novel pseudo-normal and exact arithmetic Sutherland-Hodgman algorithm. Yeah-right-on ground model. "Meshing total" is the time to generate the tet mesh after the immersion algorithm has generated the graph, including CGAL and other operations specific to each method. The "grand total" is the total time from loading mesh M to producing the output tet mesh; it equals the immersion algorithm time (16 sec; same for both methods), plus the "meshing total" time. The libigl::mesh_boolean times are for tetrahedralization in Sifakis's method; our method does not need it for tetrahedralization. We use it in the immersion algorithm; where it occupies 12.5 sec of the running time.

Routine	Sifakis 2007 [sec]	Ours [sec]	Speedup
CGAL::Lazy_exact_Add	7.6	0.14	54×
CGAL::Lazy_exact_Sub	2.5	0.34	7.3×
CGAL::Lazy_exact_Mul	7.6	0.14	54×
CGAL::Lazy_rep_2	305	13.6	22.4×
libigl::mesh_boolean	2479	0	∞
Meshing total	2529	55	50×
Grand total	2545	71	36×

Although our immersion theorems apply to shapes that are free of self-connecting shapes, we can still accommodate such shapes (Figure 12) by cutting them with helper cubes (Figure 17). This increases the number of cells of M from 5 only to 11, and converts the input into one without self-connecting shapes. Our method is then able to produce a valid separating tetrahedral mesh. This tetrahedral mesh has two connected components: one for the cube (which we discard), and one for the torus. In Figure 17, right, we show that we can then use the produced torus tet mesh to deform the double-loop torus into a self-intersection-free configuration. Our method is able to find all possible immersions (Figure 6). Unlike prior work [Mukherjee 2014], our input is not restricted to a single connected component (see Figure 7), which means that our method accepts a broader input than the state of the art on 2D immersions.

9 CONCLUSION

We presented an algorithm to create a simple volume-immersion that matches the input triangle mesh if such an immersion exists, or reports that it does not exist. We demonstrated how to efficiently create a topologically-correct tetrahedral mesh for self-intersecting input triangle meshes in 3D, or polygonal lines in 2D. Different from previous methods, our method is robust both for 3D shapes that are topologically spheres and tori of high genus. Our method cannot handle surfaces that are not boundaries of volume-immersions, such as the elbow collision shape, or other inverted surfaces. Topologically-correct tetrahedralization is the most costly component in our pipeline, and we hope to accelerate it further in

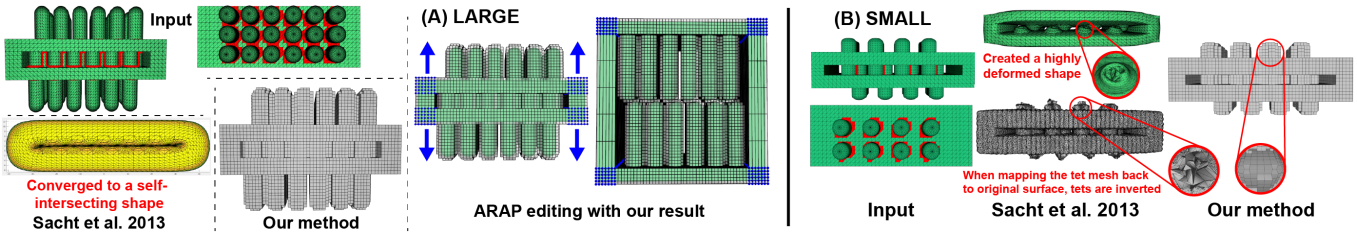

Fig. 24. **Comparison to Sacht's work:** Part (A): Left to right: side and top view of the model from [Sacht et al. 2013] (intersections are in red), Sacht's method result (failed in the middle of the process), the tetrahedral mesh generated successfully by our method, and the tetrahedral mesh and its embedded mesh deformed by volumetric ARAP, demonstrating our correct topology. Part (B): Left to right: side and top view of the "small" input model; Sacht's method produces a highly deformed shape that results in a low-quality tetrahedral mesh for the original surface; our method produces a good volumetric mesh.

the future. We described our volumetric meshing process for tetrahedral meshes, but the technique can be easily extended to other elements such as hexahedral meshing. There are several remaining open challenges; namely, how to detect and properly handle non-simple immersions such as multiple self-connecting tori, immersions where the compactness assumption does not hold, and non-immersions such as inversions. Challenges also include handling input meshes with boundaries, and constructing immersions in higher dimensions. The tet meshes we built for this work did not try to match the input triangle meshes. An interesting future work is to tetrahedralize a constrained boundary that self-intersects.

ACKNOWLEDGEMENTS

This research was sponsored in part by NSF (CAREER-1055035, IIS-1422869), and USC Annenberg Fellowship to Yijing Li.

REFERENCES

Marco Attene. 2010. A lightweight approach to repairing digitized polygon meshes. *The Visual Computer* 26, 11 (2010), 1393–1406.
Marco Attene. 2014. Direct repair of self-intersecting meshes. *Graphical Models* 76, 6 (2014), 658–668.
J. Bærentzen and H. Aanæs. 2005. Signed Distance Computation using the Angle Weighted Pseudo-normal. *IEEE Trans. on Visualization and Computer Graphics* 11, 3 (2005), 243–253.
David Baraff, Andrew Witkin, and Michael Kass. 2003. Untangling cloth. In *ACM Transactions on Graphics (TOG)*, Vol. 22. 862–870.
Jernej Barbič, Yijing Li, Bohan Wang, and Danyong Zhao. 2018. Vega FEM Library 4.0. (2018). http://www.jernejbarbic.com/vega.
Marcel Campen and Leif Kobbelt. 2010. Exact and Robust (Self-) Intersections for Polygonal Meshes. In *Computer Graphics Forum*, Vol. 29. 397–406.
CGAL. 2018. Computational Geometry Algorithms Library. (2018). http://www.cgal.org
Keenan Crane. 2017. 3D Model Repository. (2017). http://www.cs.cmu.edu/~kmcrane/Projects/ModelRepository
David Eppstein and Elena Mumford. 2009. Self-overlapping curves revisited. In *Proc. of ACM-SIAM Symp. on Discrete Algorithms*. 160–169.
Jeff Erickson. 2009. Computational Topology; Cell Complexes. In *Course Notes, University of Illinois at Urbana-Champaign*. http://jeffe.cs.illinois.edu/teaching/comptop/2009/notes/cell-complexes.pdf
Dennis Frisch. 2010. Extending Immersions into the Sphere. *arXiv preprint arXiv:1012.4923* (2010).
B. Heidelberger, M. Teschner, R. Keiser, M. Müller, and M. H. Gross. 2004. Consistent penetration depth estimation for deformable collision response. In *VMV'04*. 339–346.
F. Hétroy, S. Rey, C. Andújar, P. Brunet, and A. Vinacua. 2011. Mesh repair with user-friendly topology control. *Computer-Aided Design* 43, 1 (2011), 101–113.
Zhe Huang, Hongbo Fu, and Rynson WH Lau. 2014. Data-driven segmentation and labeling of freehand sketches. *ACM Trans. on Graphics (TOG)* 33, 6 (2014), 175.
Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. 2011. Bounded Biharmonic Weights for Real-time Deformation. *ACM Trans. Graph.* 30, 4 (2011), 78:1–78:8.
Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. 2013a. Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 33.
A Jacobson, D Panozzo, C Schüller, O Diamanti, Q Zhou, N Pietroni, et al. 2013b. libigl: A simple C++ geometry processing library. (2013).
Weishi Li. 2011. Detecting Ambiguities in 3D Polygons with Self-Intersecting Projections. In *Computer-Aided Design and Computer Graphics CAD/Graphics*. 11–16.
Yijing Li, Hongyi Xu, and Jernej Barbič. 2016. Enriching Triangle Mesh Animations With Physically Based Simulation. *IEEE Trans. on Visualization and Computer Graphics* (2016).
Nathan Mitchell, Mridul Aanjaneya, Rajsekhar Setaluri, and Eftychios Sifakis. 2015. Non-manifold level sets: A multivalued implicit surface representation with applications to self-collision processing. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 247.
N. Molino, Z. Bao, and R. Fedkiw. 2004. A virtual node algorithm for changing mesh topology during simulation. In *Proc. of ACM SIGGRAPH 2004*. 385–392.
Neil Molino, Robert Bridson, and Ronald Fedkiw. 2003a. Tetrahedral mesh generation for deformable bodies. In *Symposium on Computer Animation (SCA)*.
Neil Molino, Robert Bridson, Joseph Teran, and Ron Fedkiw. 2003b. A crystalline, red green strategy for meshing highly deformable objects with tetrahedra. In *12th Int. Meshing Roundtable*. 103–114.
Uddipan Mukherjee. 2014. Self-overlapping curves: Analysis and applications. *Computer-Aided Design* 46 (2014), 227–232.
Uddipan Mukherjee and M Gopi. 2012. Tweening boundary curves of non-simple immersions of a disk. In *Proc. of Indian Conference on Computer Vision, Graphics and Image Processing*. ACM, 36.
U. Mukherjee, M. Gopi, and J. Rossignac. 2011. Immersion and embedding of self-crossing loops. In *Proc. of the Eurographics Symposium on Sketch-Based Interfaces and Modeling*. 31–38.
Matthieu Nesme, Paul G. Kry, Lenka Jeřábková, and François Faure. 2009. Preserving Topology and Elasticity for Embedded Deformable Models. *ACM Trans. on Graphics (SIGGRAPH 2009)* 28, 3 (2009), 52:1–52:9.
G. Noris, D. Sỳkora, A. Shamir, S. Coros, B. Whited, M. Simmons, A. Hornung, M. Gross, and R. Sumner. 2012. Smart scribbles for sketch segmentation. In *Computer Graphics Forum*, Vol. 31. 2516–2527.
L. Sacht, A. Jacobson, D. Panozzo, C. Schüller, and O. Sorkine-Hornung. 2013. Consistent Volumetric Discretizations Inside Self-Intersecting Surfaces. In *Computer Graphics Forum*, Vol. 32. 147–156.
Peter W Shor and Christopher J Van Wyk. 1989. Detecting and decomposing self-overlapping curves. In *Proc. of ACM Symp. on Computational geometry*. 44–50.
E. Sifakis. 2007. *Algorithmic Aspects of the Simulation and Control of Computer Generated Human Anatomy Models*. Ph.D. Dissertation. Stanford University.
Eftychios Sifakis, Kevin Der, and Ronald Fedkiw. 2007. Arbitrary Cutting of Deformable Tetrahedralized Objects. In *Symp. on Computer Animation (SCA)*. 73–80.
Olga Sorkine and Marc Alexa. 2007. As-rigid-as-possible surface modeling. In *Symp. on Geometry processing*, Vol. 4. 109–116.
Ivan Sutherland and Gary Hodgman. 1974. Reentrant Polygon Clipping. *Commun. ACM* 17 (1974), 32–42.
J. Teran, E. Sifakis, S. Blemker, V. Ng-Thow-Hing, C. Lau, and R. Fedkiw. 2005. Creating and simulating skeletal muscle from the visible human data set. *IEEE Trans. on Visualization and Computer Graphics* 11, 3 (2005), 317–328.
Y. Wang, C. Jiang, C. Schroeder, and J. Teran. 2014. An adaptive virtual node algorithm with robust mesh cutting. In *Symposium on Computer Animation (SCA)*. 77–85.
Ofir Weber and Denis Zorin. 2014. Locally injective parametrization with arbitrary fixed boundaries. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 75.
Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. 2009. Deforming meshes that split and merge. In *ACM Transactions on Graphics (TOG)*, Vol. 28. ACM, 76.
Chris Wojtan and Greg Turk. 2008. Fast viscoelastic behavior with thin features. *ACM Trans. on Graphics (TOG)* 27, 3 (2008), 47.
K. Xu, K. Chen, H. Fu, W. Sun, and S. Hu. 2013. Sketch2Scene: sketch-based co-retrieval and co-placement of 3D models. *ACM Trans. on Graphics (TOG)* 32, 4 (2013), 123.
Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 39.