BoolSurf: Boolean Operations on Surfaces

MARZIA RISO, Sapienza University of Rome, Italy
GIACOMO NAZZARO, Sapienza University of Rome, Italy
ENRICO PUPPO, University of Genoa, Italy
ALEC JACOBSON, University of Toronto, Adobe Research, Canada
QINGNAN ZHOU, Adobe Research, USA
FABIO PELLACINI, Sapienza University of Rome, Italy

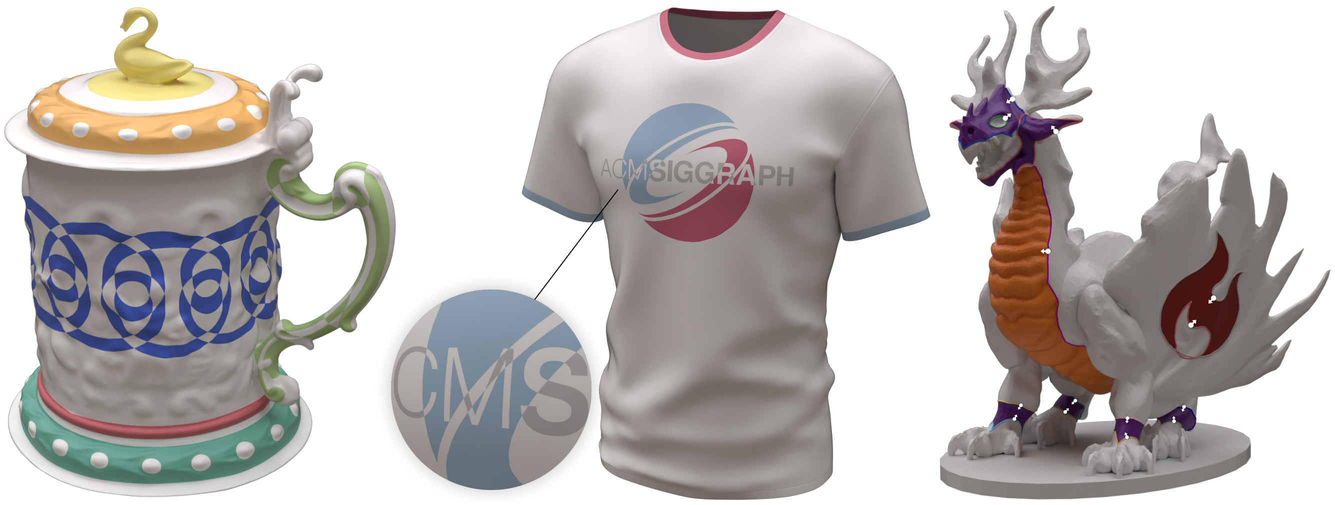

Fig. 1. We compute Boolean operations between shapes bounded by closed curves on surfaces, consisting of geodesic polygons and Bézier splines. We support, intricate and self-intersecting shapes on surfaces of any genus and optionally open boundaries (left and center). As an alternative to Booleans, we also define Boundary Sampled Curves, which extend the work of Du et al. [2021] to the manifold domain (right).

We port Boolean set operations between 2D shapes to surfaces of any genus, with any number of open boundaries. We combine shapes bounded by sets of freely intersecting loops, consisting of geodesic lines and cubic Bézier splines lying on a surface. We compute the arrangement of shapes directly on the surface and assign integer labels to the cells of such arrangement. Differently from the Euclidean case, some arrangements on a manifold may be inconsistent. We detect inconsistent arrangements and help the user to resolve them. Also, we extend to the manifold setting recent work on Boundary-Sampled Halfspaces, thus supporting operations more general than standard Booleans, which are well defined on inconsistent arrangements, too. Our implementation discretizes the input shapes into polylines at an arbitrary resolution, independent of the level of resolution of the underlying mesh. We resolve the arrangement inside each triangle of the mesh independently and combine the results to reconstruct both the boundaries and the interior of each cell in the arrangement. We reconstruct the control points of curves bounding cells, in order to free the result from discretization and provide an output in vector format. We support interactive usage, editing shapes consisting up to 100k line segments on meshes of up to 1M triangles.

CCS Concepts: • **Computing methodologies → Graphics systems and interfaces**; **Shape modeling**.

Additional Key Words and Phrases: user interfaces, geometry processing

ACM Reference Format:
Marzia Riso, Giacomo Nazzaro, Enrico Puppo, Alec Jacobson, Qingnan Zhou, and Fabio Pellacini. 2022. *BoolSurf*: Boolean Operations on Surfaces. *ACM Trans. Graph.* 41, 6, Article 247 (December 2022), 13 pages. https://doi.org/10.1145/3550454.3555466

Authors' addresses: Marzia Riso, Sapienza University of Rome, Italy, riso@di.uniroma1.it; Giacomo Nazzaro, Sapienza University of Rome, Italy, nazzaro@di.uniroma1.it; Enrico Puppo, University of Genoa, Italy, enrico.puppo@unige.it; Alec Jacobson, University of Toronto, Adobe Research, Canada, alecjacobson@adobe.com; Qingnan Zhou, Adobe Research, USA, qzhou@adobe.com; Fabio Pellacini, Sapienza University of Rome, Italy, pellacini@di.uniroma1.it.

0730-0301/2022/12-ART247 $15.00
https://doi.org/10.1145/3550454.3555466

1 INTRODUCTION

Vector graphics plays a key role in 2D design. Standard editors provide tools for drawing polygons and splines. Sets of closed curves define solid shapes, which may be combined by Boolean set operations, as done, e.g., in the PATHFINDER tools in Adobe Illustrator [Adobe 2021]. Recent research has shown the possibility of editing vector graphics directly on surfaces, by tracing geodesic polygons and splines robustly and interactively on triangle meshes [Mancinelli et al. 2022; Mancinelli and Puppo 2022; Nazzaro et al. 2021; Sharp and Crane 2020]. To the best of our knowledge, though, Boolean operations between shapes defined by curves on surfaces have not been demonstrated yet.

Boolean Operations on Surfaces. Boolean operations require resolving the arrangement of the shape boundaries, and discriminating the inside and outside regions with respect to them. In this work, we follow an approach similar to Zhou et al. [2016], labeling the cells in the arrangement with tuples of integers that express their insideness with respect to the shapes. Compared to the Euclidean setting, tackling this problem on a manifold involves two fundamental challenges: (I) an arbitrary arrangement containing non-contractible curves may yield an inconsistent labeling, and (II) the computations necessary to intersecting curves and discriminating the inside/outside of regions are significantly more involved.

BoolSurf. We propose *BoolSurf*, a fast and sound method for computing Boolean operations between shapes bounded by freely intersecting curves on surfaces of any genus, possibly with open boundaries. Some results from our work are shown in Fig. 1.

We address issue (I) above by providing necessary and sufficient conditions for an arrangement to be consistent. We automatically detect consistent arrangements of curves and provide a labeling of their cells. In the case of an inconsistent arrangement, we provide automatic tools to fix it, guided by user choices. Alternatively, we also support *Boundary Sampled Curves* (BSC), a manifold extension of the work presented by Du et al. [2021], which does not require cell labeling and works on inconsistent arrangements, too.

We address issue (II) by resolving the problem in a discrete setting. We represent a surface with a mesh of triangles, tessellating curves as polylines over such mesh, and we design efficient algorithms to support interactive usage on large meshes. We recover the control points of the curves in output, freeing them from the discretization. In this way, we both encode the result in vector format, and achieve a smooth appearance by adjusting the level of resolution.

Validation. In this work, we focus on performance and rely on standard floating point arithmetic, striving to exploit its expressiveness at its best, and taking advantage of topological constraints whenever possible. This approach proved to work well in all our experiments. The integration of robust methods, which is orthogonal to this work, is out of our scope and left as future work.

We tested our method on a variety of conditions, by changing the mesh complexity, as well as the number, resolution and distribution of curves on the surface. Our arrangements form intricate decorations, involving up to hundreds of shapes, defined with thousands of control points and discretized to polylines consisting of up to 100k vertices over meshes consisting of up to 1M triangles. We further validated our algorithm on shapes placed randomly on a collection of objects from the Thingi10k meshes [Zhou and Jacobson 2016]. In all cases, our method works reliably and efficiently enough to be executed interactively, as also shown in the supplemental video.

2 RELATED WORK

Boolean operations. Being a fundamental building block in geometric modeling, Boolean operations have been investigated for longtime. The basics for Booleans in 2D were introduced by Knuth [1986] in *Metafont*, including the treatment of self-intersecting shapes. The *Clipper* package [Johnson 2014] provides a robust and efficient implementation of the algorithm for polygon clipping by Vatti [1992], by using integer arithmetic.

Most recent contributions address robust solutions for polyhedral shapes in Euclidean space. CGAL's exact-arithmetic implementation sets a standard in robustness for arrangements in 2D [Wein et al. 2021] and in 3D [Granados et al. 2003]. Several recent works base their computation on this approach, achieving results that are provably correct [Hu et al. 2019, 2018; Zhou et al. 2016]. Some approaches combine exact arithmetic with Binary Space Partitions (BSP) to improve efficiency [Bernstein and Fussell 2009; Campen and Kobbelt 2010; Naylor et al. 1990]. Other approaches improve efficiency by using predicates, as proposed by Attene [2020], Fortune and Wyk [1993] and Levy [2016], which recur to exact arithmetic only when strictly necessary [Cherchi et al. 2020; Wang et al. 2021].

Winding number. Several recent approaches to Boolean operators use winding numbers of boundaries to define the inside and outside of shapes in an arrangement [Barill et al. 2018; Jacobson et al. 2013; Zhou et al. 2016]. The theory of winding numbers for curves in the plane is a classical subject, for which Grünbaum and Shephard [1990] provides a comprehensive account. Among the several equivalent ways of assigning labels to regions with respect to a closed curve, we follow the same approach of [Zhou et al. 2016]. We assume that boundaries are consistently oriented, and we count the signed number of intersections with the oriented boundaries while visiting the arrangement across adjacent regions.

The theory in the Euclidean case cannot be extended to the manifold setting in a straightforward way, though. Reinhart [1960] introduces the winding numbers for curves on manifold surfaces, and Mcintyre and Cairns [1993] provides a geometric method for computing them. Both such works show that winding numbers in this setting are defined only for a chosen set of smooth generators of the first homology group of the surface. We elaborate on results by Mcintyre and Cairns [1993] to provide necessary and sufficient conditions for the decidability of an arrangement of curves.

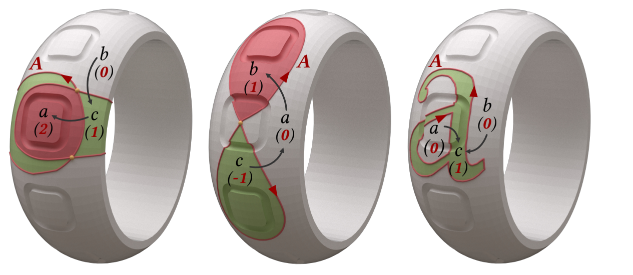

Fig. 2. Examples of shapes together with their patch and cell graphs and related cell labeling: all three arrangements consist of three regions, but have different configurations. Left: a self-overlapping shape winding about the hole. Center: a shape defined with a self-intersecting curve. Right: a shape defined with two simple curves.

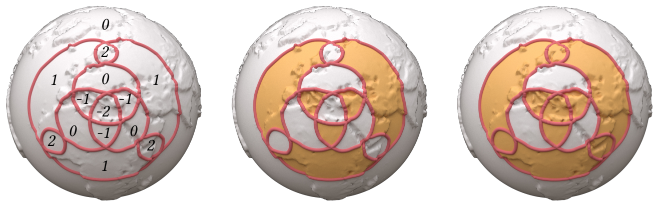

Fig. 3. Left: Labeling of a shape defined with a single self-intersecting curve. Center: inside with the even-odd rule. Right: inside with the non-zero rule.

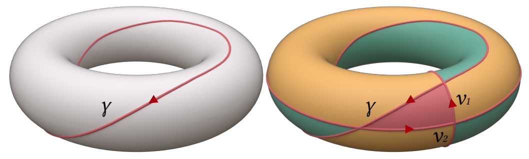

Fig. 4. Left: a curve γ defines an inconsistent shape. Right: curves v_1 and v_2 are generators of the homology; the shape defined with γ, v_1, v_2 has a consistent labeling, while no subset of such curves has a consistent labeling.

3 BOOLEAN OPERATIONS ON SURFACES

We begin with some basic definitions, followed by an overview of our algorithm, including our main result about non-trivial shapes. Implementation details are deferred to Sec. 4

3.1 Shapes through Cell Labeling

Shapes on surfaces. Let $\mathcal{S}$ be an orientable surface, possibly with open boundaries. A *shape* A on $\mathcal{S}$ is defined with a set of *closed oriented curves* $\gamma_1, \ldots, \gamma_h$, which may mutually and self intersect, as shown Fig. 2. We use multiple curves per shape to naturally represent holes, such as the glyph for the letter 'a', assuming that they are oriented consistently. We split curves at all intersection points. A *patch* is a portion of a given curve between two consecutive intersection points, which inherits the orientation of the curve defining it. The set of patches forms a planar graph, partitioning $\mathcal{S}$ into connected *cells*.

Cell Graph. By construction, cells form a complete partition of $\mathcal{S}$. Two cells are mutually adjacent if they share a common patch at their boundaries. The *cell graph*, which is dual to the graph of patches defined above, has one node per cell and one arc per pair of adjacent cells. We orient each arc in the cell graph to cross its corresponding patch from the right to the left; and we label it with the shape containing the patch it crosses. See Fig. 2.

Cell Labeling. In the Euclidean setting, it is always possible to label each cell of the cell graph in a unique way with an integer number, such that the outer unbounded cell is labeled zero, and the label increases by one when crossing a patch from the right to the left [Zhou et al. 2016]. The label of a cell counts how many times the curves wind about it, with a positive/negative sign for the cycles in counterclockwise/clockwise orientation, respectively. This mechanism extends directly to arrangements of *contractible* curves on a surface, provided that one arbitrary cell is selected to be the *outer cell* and labeled with zero. This case is shown in Fig. 2.

Given a labeling, the inside and outside of shape A is defined with either the *even-odd rule*, where only the parity of the label is considered and even is identified with the outside, or the *non-zero rule*, where only cells with a zero label are classified outside. See Fig. 3 for an example.

Inconsistent Labelings. On a surface of genus non-zero, not all arrangements of curves define a valid shape. For example, a shape defined with a single non-contractible curve, as in Fig. 4(left), has the same region on both sides, and the corresponding cell graph consists of a single cell and a single self-loop crossing the curve. We say that the labeling of such a shape is *inconsistent.* Note that, the shape can be turned into a consistent configuration by adding more non-contractible curves, as in Fig. 4(right). However, such curves must be selected in a suitable way, which we will clarify later on.

We will present first the algorithm for the consistent case and next the method to resolve inconsistent shapes guided by user input.

3.2 Algorithm Overview

A collection $\{A_1, \ldots, A_k\}$ of shapes on $\mathcal{S}$ forms an arrangement, as shown in Fig. 5 (A). The patch and cell graphs naturally extend to multiple shapes, as shown in Fig. 5 (B) and Fig. 5 (C), respectively.

The cells of the arrangement are the building blocks for computing Boolean operations. We relate each cell to each input shape, by assigning labels that encode the insideness relationship between the cell and each of the shapes. We follow the approach of Zhou et al. [2016], labeling each cell with a k-tuple of integer values, such that the i-th entry in the tuple characterizes the cell with respect to shape A_i, as shown in Fig. 5 (D).

Labeling algorithm. For the sake of clarity, we describe how to obtain a labeling for a generic shape A_i, noting that this can be done for all shapes together in a single pass. Since in the manifold setting there is no natural notion of "outer region", we pick a cell c_i in the arrangement and we arbitrarily assign it a label zero with respect to A_i. See Sec. 4.3 for details about the choice of c_i. Starting at c_i, we set the labels at all other cells by performing a single visit of the cell graph and propagating the labels throughout the visit. During the visit, if an arc is labeled with a cell different from A_i, the label is propagated unchanged through it. Otherwise, the label is either incremented or decremented by one, depending on whether the arc is traversed according to its orientation or in reverse orientation.

Inconsistency is easily detected during the visit since it manifests as two paths assigning different labels to the same cell. In that case, we activate the user-assisted procedure described in the next subsection. We proceed with Boolean operations once all shapes are consistent and all cells are correctly labeled.

Boolean operations. We apply a set of rules on the cell labeling to select which cells to include in the result, as shown in Fig. 5 (E). The definition of union, intersection, difference, and xor operations through bit vectors is straightforward. We also support *variadic* operations, i.e., operations that combine several shapes at once, as defined in [Zhou et al. 2016]. Variadic queries are resolved in a single pass, by performing a check on the tuple of labels of each cell. This is more performant than trees of pairwise Booleans. Furthermore,

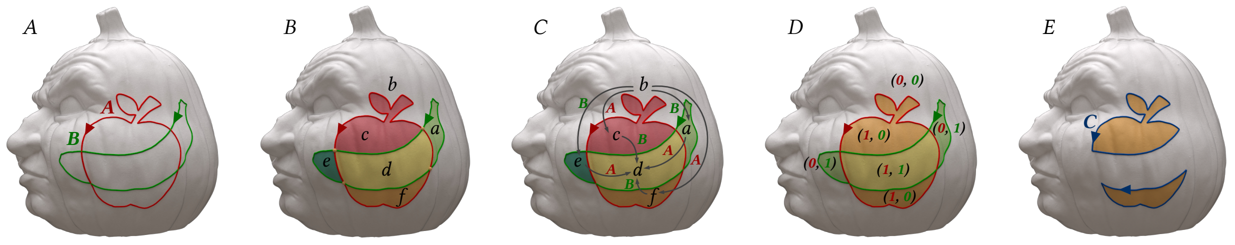

Fig. 5. Steps of our algorithm. From the left: (A) The input is a set of oriented curves defining two shapes A and B. (B) The arrangement is built by intersecting such curves, splitting them into patches, and finding the cells in which they tessellate the surface. (C) The cell graph is built that encodes the adjacency between cells and is labeled with the shapes crossed with arcs. (D) The adjacency graph is visited to assign to each cell an integer code per input shape. (E) Boolean operations consist of selecting cells based on their codes. In this example, the difference between A and B is computed.

variadic operations may be more expressive, too. Fig. 18 shows an arrangement of multiple shapes, which are combined at once.

Boundary-Sampled Curves. Du et al. [2021] introduced Boundary-Sampled Halfspaces (BSH) in the Euclidean setting, which allow users to directly control how cells are combined into final results via user-provided boundary samples, and can represent configurations that cannot be expressed with Booleans operations. We extend this framework to the manifold domain, with a method called *Boundary-Sampled Curves (BSC)*. In BSC, the user specifies a set of oriented samples on patches of the arrangement, indicating a set of desired boundary patches to the output shape, as well as the shape insideness with respect to them. This approach bypasses the issue of inconsistent labelings, since it allows users to directly control the final result via boundary samples, denoted by bullets with arrows in Fig. 6, which effectively disambiguate inconsistent configurations in a user-friendly manner. More complex examples are shown in Figures 1 and 19.

In order to apply BSC, all the first part of our pipeline is left unchanged. Cell labeling is no longer necessary and BSC computes the simplest shape, measured by boundary length, which fulfills the constraints given by the samples. This operation is performed on the cell graph with the iterative graph cut algorithm presented in [Du et al. 2021]. The notable difference is that, in our manifold setting, the cell graph may contain self-loops generated by non-contractible curves. Since no cut in the graph can include a self-loop, the graph cut algorithm is oblivious of curves that cause inconsistency. Thus, our BSC algorithm inherits the same existence, uniqueness, and describability properties of BSH.

3.3 Inconsistent Labeling

In this section, we set the theoretical results about shape consistency and we provide a practical method to resolve inconsistent cases. Whenever no ambiguity arises, we will refer indifferently to a shape A, the set of curves $\gamma_1, \ldots, \gamma_h$ defining it, and the set of cells in which they partition $\mathcal{S}$.

Homology. The *first homology group* $H_1\mathcal{S}$ provides an algebraic characterization of the topology of a surface $\mathcal{S}$. We just give an intuitive definition, referring to books in algebraic topology for a rigorous treatment [Fulton 1995; Hatcher 2002]. Let g be the genus of $\mathcal{S}$. A set of non-contractible closed curves $(\nu_1, \ldots, \nu_{2g})$ can be

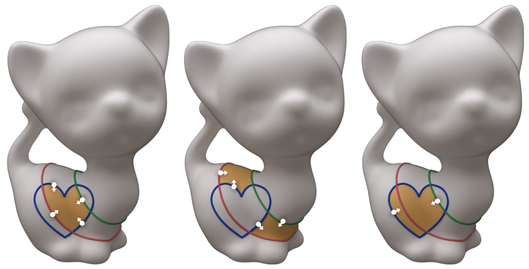

Fig. 6. Three results of applying BSC to a set of curves, controlled by different user-selected samples, shown in arrowed bullets. BSC is an alternative to Boolean operations that gives control over cell selection. The first two results can also be obtained with Boolean operations, but require joining the red and green curves with opposite orientations into the same shape; while the third result cannot be obtained with Boolean operations.

found, whose combination defines all possible classes of closed curves that can be traced on $\mathcal{S}$. The ν_i are called the *generators of the homology* and can be combined into *chains* through operator +. Given a generic curve γ on $\mathcal{S}$, its homology class in $H_1\mathcal{S}$ is

$$[\gamma] = n_1[\nu_1] + \ldots + n_{2g}[\nu_{2g}]$$

where $[\cdot]$ denotes the homology class of a curve and the n_i's are integer numbers, which intuitively express how many times curve γ winds about ν_i. Note that the orientation of a curve with respect to the generators is relevant to set the sign of its coefficients. Homology applies to sets of curves, too, by just summing their coefficients. A curve, or set of curves, is said to be *null-homologic* if all the n_i in its expression are zeros. Contractible curves are null-homologic, but also combinations of non-contractible curves that disconnect $\mathcal{S}$ may be null-homologic. For example, the combination of the three curves in Fig. 4(right) is null-homologic, while each of them is not.

Consistency. The following result provides necessary and sufficient conditions for a shape to be consistent.

Theorem 1. *Let A be a shape on $\mathcal{S}$, defined with curves $\gamma_1, \ldots, \gamma_h$. A has a consistent labeling if and only if $[\gamma_1 + \ldots + \gamma_h]$ is null-homologic.*

The proof is provided in Appendix A. The following corollary is straightforward from Theorem 1 and is stated without a proof.

Corollary 2. *An inconsistent shape A can be turned into a consistent shape A' in one of the following ways (see Fig. 7):*

(1) by adding $\sum_{j=1}^{h} n_{i,j}$ copies of each generator v_i (which has a non-zero coefficient in the homology of $[\gamma_1 + \ldots + \gamma_h]$) with a reverse orientation (i.e., $-v_i$);
(2) by duplicating, with a reverse orientation, all curves of A, whose homology contains generators that have non-zero coefficients in the homology of $[\gamma_1 + \ldots + \gamma_h]$;
(3) by removing from A a set of curves whose combination has the same homology of $[\gamma_1 + \ldots + \gamma_h]$.

Resolution. Upon loading the input dataset, we pre-compute a set of generators of the homology of $\mathcal{S}$. Once an inconsistent shape A is detected, we compute the homology of all the curves in A. Now let $(n_{1,j}, \ldots, n_{2g,j})$ be the tuple of integers encoding the homology of γ_j and $(\sum_{j=1}^{h} n_{1,j}, \ldots, \sum_{j=1}^{h} n_{2g,j})$ be the corresponding tuple for shape A. The non-zero coefficients in the latter give the "generators in excess" in the homology of A.

In the GUI of our system, we show the inconsistent shape to the user, by highlighting all the curves in it that contain generators in excess. By hovering each such curve, we show its corresponding generators in excess, with their orientation and multiplicity. The user can then choose how to resolve each curve. The homology of A is re-computed on the fly after each modification, and the operation is repeated until A becomes consistent.

We provide three automated methods to resolve a shape, which follow from Corollary 2 and are illustrated in Fig. 7 and Fig. 8: (1) add copies of the generators in excess to A, with opposite orientation, as they appear in the homology basis, adding a geometric offset in case of multiple copies; (2) duplicate curve γ_j in reverse orientation with an offset in the direction normal to its control points, to obtain a solid strip; and (3) remove γ_j from A, or flip its orientation, which is equivalent to remove it twice, in terms of homology. While any combination of these methods can be used on the different curves of A, we also allow the user to alternatively apply manual editing, e.g., adding more curves and/or changing the shape and homology of some curves defining A, or even combining A with some other inconsistent shape in the arrangement to form a single shape. All our curves are editable, thus if some added curve does not have the desired geometric appearance, it can be edited by dragging its control points across the surface, without altering the homology.

3.4 Open boundaries

Our method works seamlessly for a surface $\mathcal{S}$ with open boundaries. Let $\bar{\mathcal{S}}$ be a watertight surface where all holes have been plugged with discs at the boundaries of $\mathcal{S}$, and let A_0 be a shape on $\bar{\mathcal{S}}$ defined by the boundary curves of $\mathcal{S}$. Shape A_0 is consistent by construction and the curves are oriented so that the inside of A_0 is $\mathcal{S}$. Resolving a Boolean operation on $\mathcal{S}$ for shapes $A_1, \ldots, A_k$ is equivalent to resolve the same operation on $\bar{\mathcal{S}}$ for shapes $A_0, \ldots, A_k$, where the bit for A_0 is set to 1. Note that both $\bar{\mathcal{S}}$ and A_0 are virtual and this does not require any change in the implementation.

3.5 Discretization

Input shapes. We represent the surface with a triangle mesh M. Our input curves are either geodesic polygons, defined with their vertices on M, or geodesic Bézier splines, defined with their control polygons directly on M, as in [Mancinelli et al. 2022]. Fig. 9 shows

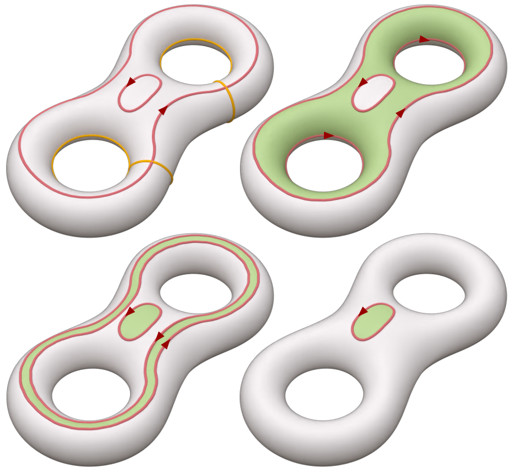
Fig. 7. An inconsistent shape defined with two curves, shown in red, together with the generators of the homology, shown in yellow. Decidability may be resolved in three different ways, with different results: by adding generators of the non-contractible curve; by duplicating the non-contractible curve; or by removing the non-contractible curve.

an example with splines. We discretize such curves with polylines, which may be arbitrarily refined in the interior of triangles. The discretization of input curves is independent from the discretization of the surface, as shown Figures 10 and 11. See Sec. 4.1 for details.

Computing the arrangement. We first determine which triangles may contain an intersection, and then compute the intersections between line segments in each triangle. See Sec. 4.2 for details. The curve patches form the boundaries of the cells in the arrangement. We determine both the cell interiors and the cell graph, by a flood-fill over the surface. We perform all such operations in a non-destructive way, on a dual graph that represents a refinement of M embedding the input shapes. See Sec. 4.3 for details.

Extracting the output. The Boolean operations are performed on the cell graph, which is independent from the underlying discretization. The output consists of grouped sets of cells, which are bounded by chains of curve patches. Since the input polylines were obtained by discretizing geodesic lines and splines, we recover the control points of the resulting boundaries, as shown in Fig. 9, freeing them from the discretization. All control points are encoded with barycentric coordinates with respect to the triangles of the ambient mesh M, thus in a inherently robust manner.

4 IMPLEMENTATION

We aim at providing a practical algorithm that scales well with mesh complexity and is fast enough for interactive use. In this section, we present the details of the algorithm and its implementation. The description is split into the steps discussed in the previous sections.

4.1 Data Representation

Mesh Representation. We encode a mesh M with an indexed data structure, consisting of an array of positions for vertices and an

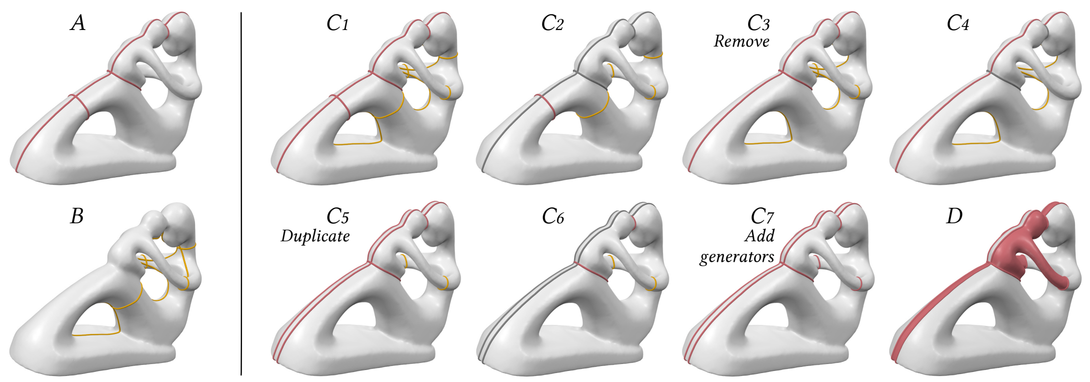

Fig. 8. A more complex example of inconsistent shape defined with four curves (A), the generators of homotopy (B), and a sequence of edits (C) needed to produce a consistent shape (D). The edit sequence shows the steps as implemented in out prototype. Upon detecting an invalid shape we display it to the user together with the generators in excess (C1); the user can then select any set of curves, for which we display the generators in excess (C2); the user resolves one curve by removing it from the shape, after which we recompute and display the generators in excess for the updated shape (C3); the user proceeds by selecting a new curve (C4) and duplicating it (C5); in the final edit, two curves are selected (C6) and two generators are added (C7) to fully resolve the shape.

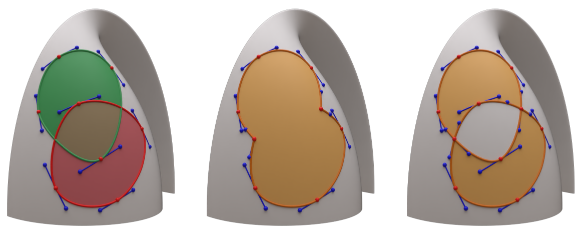

Fig. 9. Left: Two shapes defined with Bézier splines are given as input. Center: Union of the shapes. Right: Symmetric difference of the shapes. Our algorithm splits the splines at intersection points and outputs the control points that define the boundary curves resulting from Boolean operations.

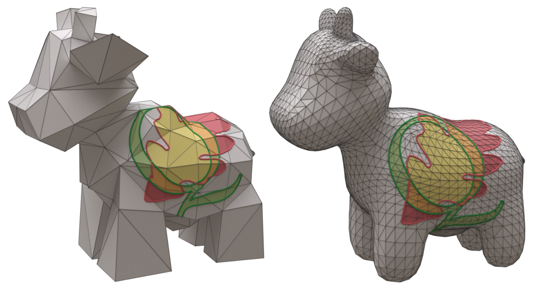

Fig. 10. The same shapes are drawn onto low-poly and a high-poly version of the same surface. The discretization of curves is the same in both cases. Overall, the input curves will cross triangles of the low-poly version with more complex configurations.

array of indices for triangles. While we focus on interactivity on meshes with millions of triangles, we also support low-resolution meshes, as shown in Fig. 10.

The *Dual Graph of Adjacencies (DGA)* of the mesh is an undirected graph whose nodes correspond to the mesh triangles, and whose arcs correspond to triangle adjacencies. The DGA is used to visit the surface and find the cells in the arrangement, but it cannot represent the cells as it is. To this aim, the DGA is refined during the first stage of the algorithm, expanding every node crossed by input curves into a suitable subgraph, as discussed in Sec. 4.3. We compute the DGA of the input mesh M once in the beginning. Upon editing, we only pay the cost of its refinement, while its original state is restored by just discarding additional entries in the data structure.

Curve Representation. We discretize the input curves over M as polylines using the method of Mancinelli et al. [2022]. This method is inherently robust, guaranteeing that a discretized curve traverses a strip of triangles on M. A polyline has a vertex each time it crosses a mesh edge, and possibly other vertices inside triangles, depending on its level of discretization, as shown in Fig. 11. Each curve vertex is represented with a tuple consisting of the index of a triangle t of M containing it, together with its barycentric coordinates in t.

All subsequent computations are done in barycentric coordinates in 2D and all our results are encoded in the same way. The advantage of this approach is threefold. First, all computations and results refer to the intrinsic geometry of the ambient surface, not to its embedding in 3D. Second, we avoid as much as possible conversions of coordinates between 2D and 3D, thus reducing the sources of error. Third, we exploit floating point arithmetic at its best, by constraining all our computations to the interval $[0, 1]$.

4.2 Intersecting Curves

The first step of our algorithm consists of splitting the input curves at their intersections, as shown in Fig. 11. We first locate the triangles in which intersections may occur, then we bound geometric computations inside single 2D triangles.

Curve Hashmap. We build a hashmap whose keys are the triangle indices and whose values are the curve segments contained in a triangle, as shown in Fig. 11. Vertices lying on mesh edges are encoded with respect to both incident triangles, and possible intersections occurring at them are de-duplicated in post-processing. Hashmaps are efficient, since they are built with amortized complexity of $O(n)$

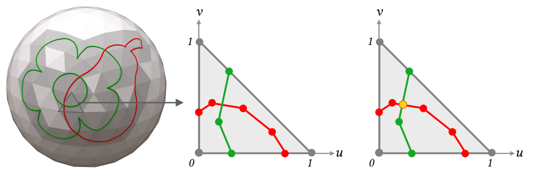

Fig. 11. Hashmap representation of the input shapes. Left: The triangles crossed by at least a shape are colored in light grey. The id's of such triangles provide the keys of a hashmap. Each entry in the hashmap will contain a 2D representation of the portion of arrangement that covers that triangle. Middle: The local 2D polylines are expressed in barycentric coordinates. The highlighted triangle contains two polylines belonging to two different input shapes. Right: The intersections between the shapes are computed per triangle, by testing pairs of 2D segments in barycentric space.

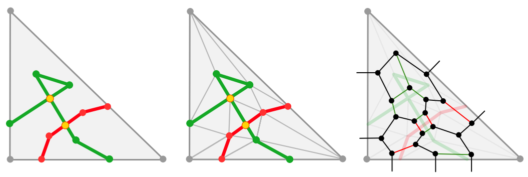

Fig. 12. Refinement of the Dual Graph of Adjacencies inside a triangle. Left: The set of input polylines that cross the triangle and their intersections, expressed in barycentric coordinates. Middle: a Constrained Delaunay Triangulation is computed to embed the polylines as triangle edges. Right: The corresponding graphlet of the DGA.

in the number of curve segments, and the data structure is sparse with respect to the number of triangles in the mesh.

Curve Intersections. We find intersections between curves segments by iterating over the triangles in the hashmap. For each triangle, we find intersections, including self-intersections, by testing pairs of curve segments that cross it. Intersections are computed in 2D with barycentric coordinates, which correspond to Cartesian coordinates in the standard simplex, $(0, 0)(1, 0)(0, 1)$, while guaranteeing topological consistency upon affine deformations. We split the curves at each intersection point by performing a point insertion operation, and update the hashmap accordingly.

4.3 Cells and the Cell Graph

Cells in our arrangements are regions of surface bounded by chains of curve patches. For the purpose of Boolean operations, we just need to relate regions to their boundary patches. On the other hand, the region corresponding to each cell in output may be relevant for subsequent operations (e.g., coloring). Since the input curves cross triangles, such regions cannot be represented as sets of triangles of M. We thus refine the Dual Graph of Adjacencies of M to both represent the interior of cells explicitly, and build the cell graph.

DGA Refinement. Fig. 12 shows the refinement of a node corresponding to a triangle in the DGA. Each node t, which is crossed by input curves, is substituted with a subgraph, which corresponds to a triangulation of t that embeds the curves crossing it. In this refined DGA, cells consist of subsets of nodes, and partitioning the mesh into cells is equivalent to partitioning the graph into sub-graphs. We compute a 2D Constrained Delaunay Triangulation (CDT) of t with an off-the-shelf robust software [Amirkhanov 2022]. We use the resulting meshlet to define the sub-graph that substitutes t. Note that the original mesh M is unchanged, while we only add nodes to the temporary DGA. We do keep the 2D meshlets, in the hashmap, since they are useful for rendering the final shapes.

Computing the CDT is the slowest part of our algorithm, which we speed up by (1) fetching from the hashmap just the triangles that need refinement, (2) parallelizing the execution over all of them, and (3) skipping the CDT step when a triangle is crossed by just a single segment, which happens most of the times on hi-res meshes.

As shown in Fig. 12, processing a single triangle leaves some dangling arcs in the corresponding subgraph of the DGA. Each dangling arc has a mate in the refinement of the adjacent triangle. Dangling arcs are paired after all entries of the hashmap have been processed, exploiting the adjacency between triangles of M.

For later usage, we tag each arc of the DGA, which crosses a curve patch, with the id of the corresponding shape. For the sake of clarity, we call *tagged arcs* end *tagged nodes* those arcs that cross curve segments and their adjacent nodes, respectively.

DGA Partition. After refinement, each node in the DGA belongs to exactly one cell, and each cell corresponds to a maximally connected subgraph that contains just non-tagged arcs. We perform a visit of the DGA, with a flood-fill algorithm that visits each node just once. Starting at any node of the DGA, we initialize one new cell and we perform the visit in breadth-first order, by expanding it only across non-tagged arcs. Meanwhile, we collect all arcs that connect the boundary of the cell to the rest of the graph. We restart the visit of a new cell from any other node, which does not belong to a visited cell and can be reached by crossing its boundary. We proceed in this way until all nodes have been visited. See Fig. 13.

Extracting the Cell Graph. While visiting the DGA for partition, we also build the cell graph, as described in Sec. 3.2. See Fig. 13. Once a new cell has been flooded, the corresponding node in the cell graph is created, and this cell is connected with arcs to all the already existing cells that are reached by crossing its borders. We label each arc in the cell graph with the id of the shape crossed to reach the neighboring cell, and set its orientation so that it crosses the shape boundary from the right to the left.

Cell labeling. We compute all labels with a visit of the cell graph and store them as a tuple of labels for each cell. In the remainder of this section, we describe how we do the labeling for a single shape, noting that it can be done for all shapes together, by just addressing the proper entries in the tuple. Inconsistent shapes are also detected during this process, while they will be resolved in a later stage.

We visit the cell graph in a breadth-first manner, as shown in Fig. 14. At the beginning of the process, each cell is labeled with a "null" value. In the absence of an ambient cell, which is guaranteed to be outside all shapes, the visit starts from an arbitrary cell, which receives a zero label. Such initial cell can be either user selected, or chosen randomly. Given a cell a with a known label, the label of its neighboring cell b is computed as $l(b) = l(a) + 1$, or $l(b) = l(a) - 1$ depending on whether the arc that connects a to b is traversed according to its orientation, or in reverse orientation, respectively.

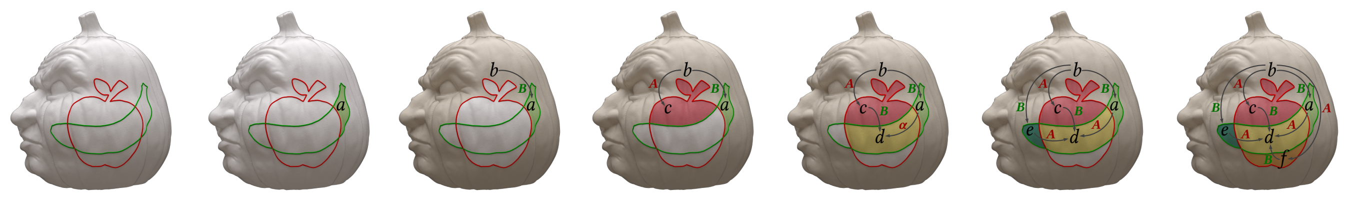

Fig. 13. The surface is visited by flooding the Dual Graph of Adjacencies, starting at a random node/triangle. Cells are filled one at a time by visiting the DGA. The cells graph is also built during the visit when the border of an already visited cell is reached.

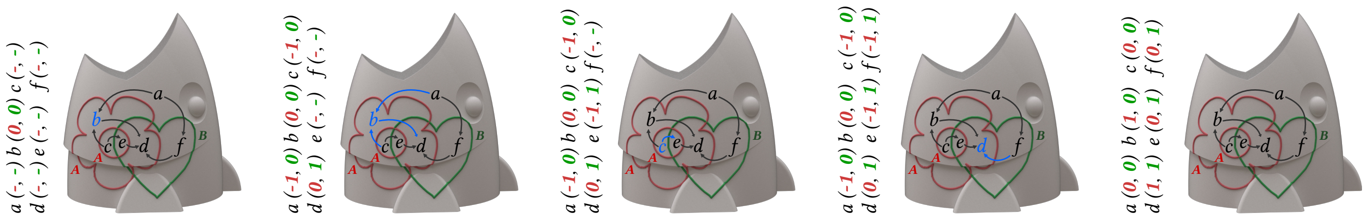
a (-, -) b (0, 0) c (-, -)
d (-, -) e (-, -) f (-, -)
a (-1, 0) b (0, 0) c (-1, 0)
d (0, 1) e (-, -) f (-, -)
a (-1, 0) b (0, 0) c (-1, 0)
d (0, 1) e (-1, 1) f (-, -)
a (-1, 0) b (0, 0) c (-1, 0)
d (0, 1) e (-1, 1) f (-1, 1)
a (0, 0) b (1, 0) c (0, 0)
d (1, 1) e (0, 1) f (0, 1)
A
B

Fig. 14. Visit of the cell graph to compute labels: visit starts at cell b, which is set to zero with respect to both shapes; the label is propagated to cells c, d, a adjacent to b, and such cells are enqueued for further propagation; cell c is popped from the queue and the label is propagated to cell e; cell d is popped from the queue and the label is propagated to cell f; the remaining cells in the queue do not effect propagation (not shown); after completion, the labels for shape A are updated with the heuristics described in the text.

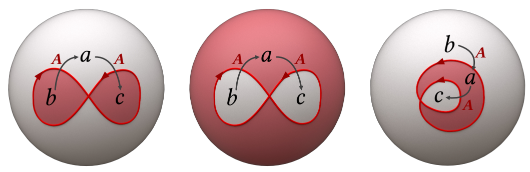

Fig. 15. Setting the "outer" region for a shape may be not intuitive. Examples of labeling a self-loop, colored with self-union and the even-odd rule. Left-Center: the result depends on the choice of the cell set to zero; in this case, with a to the left and b at the center, the heuristics computes the labeling at the center. Center-Right: the two configurations come from a homotopic deformation of the same curve and have the same labeling.

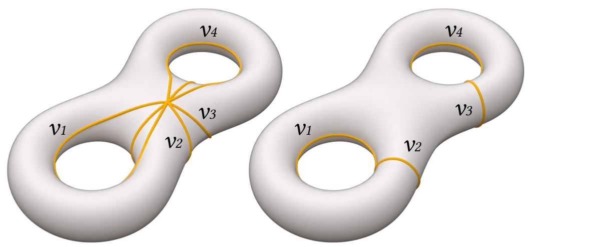

Fig. 16. Left: A set of generators for the homotopy and homology that all pass through a common point x_0 is used in our computations. Right: an equivalent set for the homology, consisting of shortest geodesic loops represented with splines, is used for the GUI.

If the connecting edge is associated to a different shape, the label is unchanged, i.e. $l(b) = l(a)$.

When the zero cell has been chosen randomly, we finally update the labels as follows: we find the cell with the smallest value and set it to zero, and we update all other labels consistently by an arithmetic shift. Although this produces a valid result in most cases, sometimes the zero cell needs to be user-selected. Fig. 15 shows typical examples that may need manual disambiguation.

Inconsistent labelings are easily detected during propagation. When a node receives a label, we check that the already-visited neighbors are labeled consistently. If an inconsistency is found, then the corresponding shape is declared inconsistent.

4.4 Resolving inconsistent shapes

Computing the homology. Upon loading the input mesh M, we compute a set of generators $v_1, \ldots, v_{2g}$ for its homotopy and homology. The generators are found by the method of Erickson and Whittlesey [2005], using an off-the-shelf implementation. All generators pass through a common point x_0, which is selected manually.

The location of x_0 affects the geometry of the generators, hence their appearance, but it is irrelevant to the subsequent algorithms. We also extract a better looking set for the homology, by relaxing the generators from passing through x_0 and shrinking them to shortest geodesic loops with the method by Xin et al. [2012]. The two sets are depicted in Fig. 16 for a double torus and they are equivalent: we use the first one for algorithmic purposes and the second for the GUI. Since the loops in the second set are geodesics, we turn each of them into a Bézier spline, whose control polygon is obtained by sampling regularly a set of points along it. This provides suitable curves for interactive editing.

Given an inconsistent shape A, for each curve γ_j defining it, we compute the homology of γ_j by the method of Dey and Schipper [1995]. In short, we first map the generators to the *polygonal schema*, which is a regular polygon with $4g$ edges: each generator is mapped to two edges of the polygon, with opposite orientations, in an order consistent with the radial order in which generators appear about x_0 (see Fig. 17). Next, we collect all intersections of γ_j with the

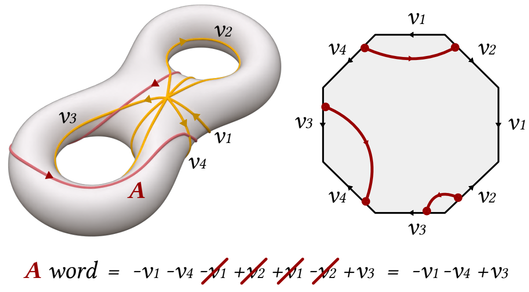
v2
v3
v1
v4
A
v1
v4
v2
v3
v1
v4
v2
v3
A word $= -v_1 -v_4 \cancel{-v_1} \cancel{+v_2} \cancel{+v_1} \cancel{-v_2} +v_3 = -v_1 -v_4 +v_3$

Fig. 17. The homology of a curve γ is computed by intersecting it with the generators $v_1, \ldots, v_{2g}$ and mapping it to the polygonal schema to the right. The homology is obtained by collecting the labels of the edges of the polygon subtended by the patches of curve and canceling equal symbols with opposite orientation [Dey and Schipper 1995].

generators, then we map the patches of γ_j between consecutive intersections to the polygonal schema. Finally, we collect the labels of the corresponding edges of the polygonal schema. The result is a word of symbols corresponding to the v_i's, which is simplified by deleting all symbols that occur with different signs, and by counting the multiplicity of all remaining symbols. This provides the collection $n_{1,j}, \ldots, n_{2g,j}$ of integers encoding the homology of γ_j. See [Dey and Schipper 1995] for further details.

Curve Offset. The automated resolution operations described before depend on either offsetting curves or deleting them. The latter is trivial, while the former requires parallel transport. All our curves are either geodesic polygons, or splines, hence each curve is defined by a small set of control points. Mancinelli et al. [2022] provide the basic tools to translate control points and parallel transport control tangents of curves over the surface. Leveraging such tools, we offset each vertex of a polygon, and each anchor point of a spline for a fixed distance in the normal direction of the curve in tangent space; handle points, which control the tangents of splines, are computed by casting geodesic lines from the anchor points in the direction of the parallel transported tangents.

4.5 Boolean Operations

A Boolean operation is expressed with a bit vector encoding the desired inside/outside relation with respect to the various shapes in the arrangement. Once the labels have been assigned, we compute Boolean operations by selecting a subset of the cells, according to their labels. The output of our algorithm consists of the boundaries of the selected regions, represented as curves.

Recovering control points. We process the output boundaries to recover the control points that define the new curves, which are sub-curves of those ones in the input, thus freeing the output from discretization. This is actually done right after computing the intersections, as shown in Fig. 9. For geodesic polygons, each segment is encoded just by its endpoints, which correspond to intersections of the arrangements, and vertices from the corresponding polygon in input. For geodesic splines, curves are split at all intersection points, by computing the control polygons of the split segments with the point insertion algorithm described in [Mancinelli et al. 2022]. In this way, each Bézier curve is substituted in the refined arrangement with a spline interpolating all intersection points.

Note that the output can now be encoded just as a collection of control points, in the style of vector graphics, which refer to the intrinsic structure of M by means of barycentric coordinates. This description is independent of both the discretization used during computation, and the embedding of M in 3D space.

The boundaries of the output regions can be reproduced later on by generating the corresponding curves at the desired level of resolution. Note also that the result can be directly transferred to an arbitrarily refined representation of the ambient surface, obtained through subdivision, because barycentric coordinates are easily propagated to the subdivided mesh.

4.6 Boundary-Sampled Curves

Similar to [Du et al. 2021], the input to our BSC algorithm is an oriented cell graph and a set of user-provided boundary samples. Each edge of the cell graph is oriented consistently with respect to the input shape, and its weight is the length of the corresponding patches, which can be easily computed in our pipeline. Given the set of samples provided by the user, we employ the BSH iterative graph cut algorithm to compute the collection of cells that are considered as inside. The output cells are compatible with the user samples, and semantically describe a minimal region bounded by as many samples as possible, while satisfying a set of constraints such as orientation-preserving and sample-connectedness as described in [Du et al. 2021]. Note that BSC only involves combinatorial processing of the cell graph, hence it is as cheap as computing Boolean operations.

The main distinctions with respect to BSH are that, in the manifold domain, self-loops are present for non-contractible shapes, and that we support different orientations for patches of the same shape. iIn this sense, our method is a strict superset of the cases demonstrated in prior work, which uses the same optimization method for a different graph and input samples.

4.7 Rendering

Once converted to absolute 3D coordinates in the embedding space, we can render the output shapes, both by stroking their boundaries and by filling their interiors. We distinguish two cases, coloring using GPU shaders and coloring in a path tracer.

Prior work has shown how to rasterize vector textures on GPUs by building a data structure that encodes the per-triangle curve boundary, which is used in the fragment shader to draw the inside or outside of each shape [Nehab and Hoppe 2008; Ramanarayanan et al. 2004]. These methods are hard to implement, though, they require to build and update a GPU data structure interactively, and are restricted in the complexity of the per-triangle curve segments. We rather extract fine surface patches using the meshlets computed during DGA refinement, and draw them with face colors. We use this GPU implementation in the supplemental video.

In a CPU path tracer, we implemented a simpler method. When evaluating material properties at an intersection point, we look up the meshlet in the barycentric space of the intersected triangle. We check the *uv* coordinates of the intersection point and perform 2D point-in-triangle operations to find the intersected cell, assigning a

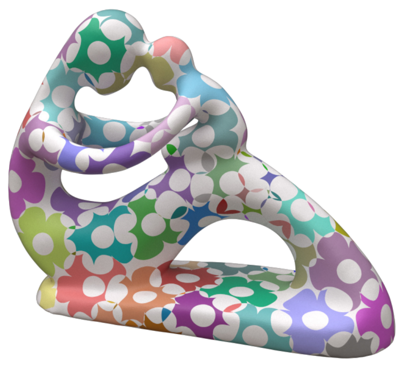

Fig. 18. Boolsurf supports *variadic* Booleans, as defined in [Zhou et al. 2016], executing them in a single pass. Here, 112 shapes, which span a whole surface of genus four, are combined with an all-vs-all Xor operation.

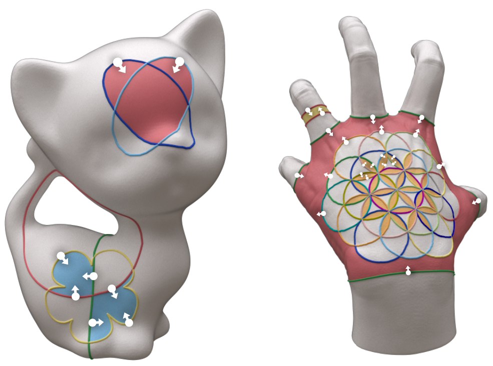

Fig. 19. Examples of using Boundary-Sampled Curves to compute cell selections that are either not representable, or hard to specify with Boolean operations. Left: The decoration on the head cannot be obtained with Boolean operations, and the arrangement on the body is inconsistent because of non-contractible loops. Right: While the flower decoration can be represented as a complex CSG tree, it is more easily achieved by placing two samples per petal (some such samples are omitted in the figure for better readability).

color accordingly. This method does not require extracting patches and is based purely on the data we already use for the arrangement. We used this method for all images in this paper.

5 RESULTS

We validated our work by computing a variety of cell arrangements on many meshes representing surfaces with different characteristics in terms of genus, boundaries, meshing, and roughness. We use hand-drawn shapes as well as SVG icons consisting of geodesic Bezier splines. The hand-drawn shapes are traced directly on the surface with our prototype GUI. For SVG icons, we first map the control points to the surface via normal coordinates, and then trace the geodesic splines directly on the surface. Once mapped, icons become editable in the GUI just like the hand-drawn shapes. In both cases, we used the method presented in [Mancinelli et al. 2022].

Experimental Setting. We tested our prototype on a variety of meshes, both hand-modeled and 3D scanned, ranging in complexity from a few thousands to one million triangles. We execute Boolean operations on arrangements from a few up to a few hundreds shapes. All curves are finely tessellated on the mesh, using typically a few hundreds segments per curve. We chose the shapes in the examples to span various conditions, which generate between a few to a few hundreds cells in the arrangement, Fig. 1 shows detailed splines on scanned meshes, Fig. 5 shows smooth icons drawn on a corrugated mesh, Fig. 18 shows an arrangement of numerous shapes, and Fig. 20 show shapes with many control points. Figures 1, 6 and 19 shows examples obtained with Boundary-Sampled Curves.

Execution Speed. Tab. 1 summarizes the statistics for all results in this paper, corresponding to Figures 1, 5, 6, 10, 13, 18, 19, and 20.

On a 3.8 GHz desktop with 16 cores, execution times are in the order of tens of milliseconds to perform all operations. The only outlier took a couple of hundred milliseconds for hundreds of shapes on a highly detailed mesh. This makes our method fast enough for interactive use, as we also demonstrate in the accompanying video. Execution time increases mostly with the complexity of the input shape and not with the complexity of the mesh. This comes from using the hashmap during the computation of the arrangement.

The breakup of times for each stage of the algorithm shows bottlenecks in the refinement and partitioning of the Dual Graph of Adjacency. For the refinement, execution time grows with the number of nodes that need refinement, and with the resolution of curve tessellation. This stage was parallelized, since each node is independent of all others and data structures are read-only. On the contrary, the partitioning time only depends on graph size, which is just slightly higher than the number of triangles in the mesh. Our algorithm ensures that during graph partition we visit each node only once, independently of the number of input shapes, making the algorithm efficient even with hundreds of shapes.

The costs of the last two stages of the algorithm, namely cell labeling and Boolean or BSC operations, are negligible, accounting for 1%-2% of the total execution time. Cell labeling works on the cell graph, which is trivially small compared to the DGA, and performs just a simple visit. A Boolean operation and extraction of the resulting shape boundaries just consists of a selection of cells, and a visit of the cell graph. The alternative cost of computing shapes with BSC is negligible as well, as it also works on the cell graph. If the interior of the selected cells is also needed, this is easily obtained from a single visit of the portion of DGA spanned by them.

Clipping. In 2D vector graphics, Boolean operations are also used to clip drawings to desired regions. Drawings may consist of solid shapes and open lines. Our method can be applied seamlessly to this purpose, too. The arrangement in input consists of all objects in the drawing, plus the clipping window, which can have any shape. The corresponding Boolean operation gathers all portions of objects in the drawing that fall inside the clipping region. The only difference in the algorithm is that we do not label the arcs in the DGA that cross open curves of the drawing, which do not bound cells in the arrangement. Fig. 21 shows an example of clipping operation.

Timings on Mesh Collection. We further validated the speed of our algorithm on a subset of the Thingi10k dataset [Zhou and Jacobson

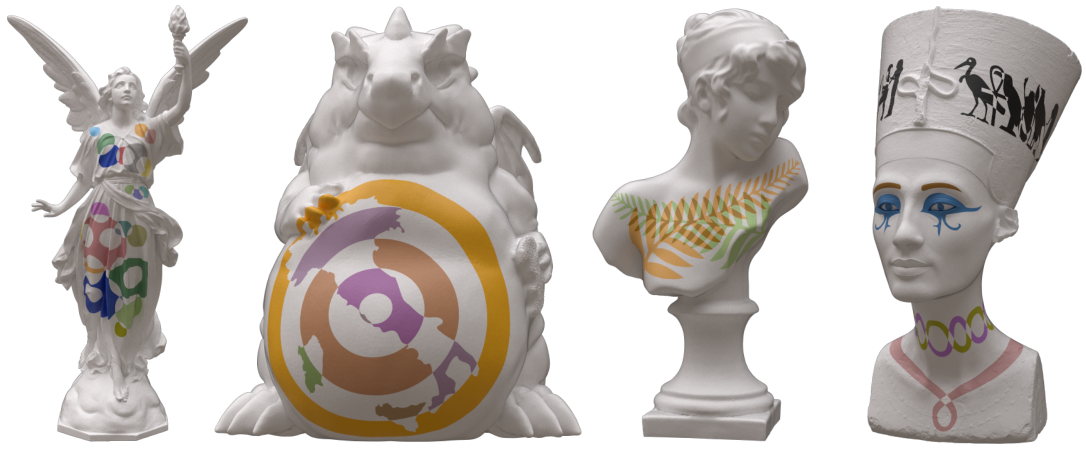

Fig. 20. A gallery of models decorated by executing Boolean operations on geodesic shape arrangements. Models show different scenarios that our algorithm supports. From left to right: (A) an arrangement of many simple shapes (geodesic circles) traced over a highly rough and complex mesh; (B) a very detailed shape (the map of Italy contains 3849 vertices) is drawn onto a dense mesh (1M triangles); (C) many disjoint cells with the same label are generated by overlapping of two complex shapes; (D) an intricate arrangement of hand-drawn shapes and SVG icons on a scanned mesh.

Table 1. Result statistics. We report sizes of the meshes and shapes in input, as well as statistics computed during the algorithm execution. Total execution time is dominated by the DGA refinement stage, which is influenced by the size of the input shapes and by the number of triangles they cross. While the time for partitioning the DGA into cells increases with the size of the mesh. The detailed execution times fo the other stages are not reported as they are negligible.

model		arrangement			DGA refinement		time (ms)		
name	triangles	shapes	cells	curve vertices	refined nodes	added nodes	total	DGA refinement	DGA partitioning
cow-lowpoly	372	2	20	901	47	2085	1	1	–
cow-highpoly	5856	2	20	901	310	2904	2	2	–
t-shirt	206k	5	50	2266	2534	12594	12	9	3
lady	282k	2	161	2281	4895	20955	23	16	7
pumpkin	395k	2	6	636	1693	6399	16	7	9
nefertiti	497k	24	122	22094	10648	74576	64	50	14
lucy	500k	25	46	6080	5340	25404	34	21	13
fertility	500k	112	644	56250	55270	263186	210	192	19
cup	539k	48	205	15808	20236	85798	68	58	10
fat-dragon	970k	6	26	4765	3383	20001	30	11	19
kitten-BSC	250k	5	10	3410	2889	14131	15	11	4
spiked-dragon-BSC	144k	14	16	2165	2460	10966	10	7	3
hand-BSC	143k	22	82	2916	6509	26333	21	17	4

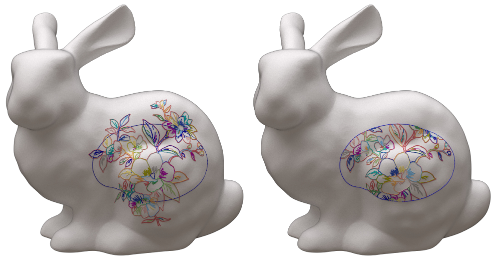

Fig. 21. Clipping a composite drawing with closed and open curves.

2016]. We collected all closed, manifold, single-component meshes with more than 100k triangles and genus less than 5, discarding duplicate meshes. We stick to large meshes to obtain reliable statistics on execution times, while the Thingi10k dataset is skewed on small meshes. In the filtered dataset, mesh size varies between 102k triangles and 3.1M triangles.

Each mesh was decorated with two overlapping shapes, each consisting of either a single, or multiple geodesic curves, all described with Bézier splines, randomly selected from a small collection of 22 SVG drawings. In order to find a large enough region for each drawing, we manually selected a center point on each mesh. Then we map the SVG control points around this center with random orientation, by means of exponential mapping. And we finally trace the splines to curves, as in [Mancinelli et al. 2022].

We tested our algorithm on a 3.8 GHz desktop with 16 cores. A gallery of results is shown in figure 22. Execution time ranges from a minimum of 3 ms to a maximum of 108 ms, although the majority of tests executed in a mean execution time of 14 ms. This experiment confirms that the most expensive stages are due to DGA refinement and partitioning, while just 1% - 2% of total execution time is spent in propagating labels and evaluating Booleans.

6 LIMITATIONS

Discretization. Our method is intrinsically approximated, since we discretize both the ambient surface with a mesh, and the input curves as polylines lying on the said mesh. Thus all our geometries are piecewise-linear. The accuracy of the approximation of a smooth solution depends on the resolution of the mesh and of the resolution of the discretized lines, which are independent from each other.

While this is a limitation, to the best of our knowledge, there exists no method at the state of the art to address such computations on smooth geometries. Methods to compute the distance function in the smooth setting exist, such as on subdivision surfaces [de Goes et al. 2016]. However, it is not clear how to extend such methods to extract smooth geodesic lines, which are the integral lines of the distance field, or, worse, geodesic Bézier curves, let alone computing the intersections of such objects.

Numerical robustness. As already outlined, in this work we focus on performance, relying on floating point computations, to achieve interactivity. This means we cannot provide theoretical guarantees on robustness. To be fair, floating point arithmetic works remarkably well in our testing since it proved correct on all our experiments, mostly due to the fact that we work always in the barycentric coordinate of the unit triangle. Furthermore, our current discretization of input lines over the mesh is guaranteed to be robust.

If desired, numerical robustness can be addressed simply by substituting our floating point computations for segment intersection with robust methods at the state of the art, such as [Attene 2020; Wein et al. 2021], at the price of slower computations and increased complexity. In particular, the package available in CGAL [Wein et al. 2021] may seamlessly substitute all the steps concerning the computations of intersections and the CDT inside a single triangle. This would guarantee the topological consistency of the result, although geometric inconsistencies, such as triangle flipping and self intersections, might still arise after snap rounding.

7 CONCLUSIONS

In this work, we have introduced *BoolSurf*, a fast and reliable algorithm for computing Boolean operations between geodesic shapes, providing a second milestone for the porting of 2D vector graphics to the manifold settings. Our method is capable of correctly handling arrangements on a manifold surface, which admit a consistent labeling. We detect inconsistent shapes and support both automatic and manual tools to fix them. Alternatively, we support Boundary-Sampled Curves, a manifold extension of BSH, which works for any possible arrangement. We tested our algorithm by manually designing complex arrangements on large meshes, and by running randomized tests on a subset of the Thingi10K dataset. Results show that in all cases a correct cell arrangement was found and a valid labeling was assigned to each cell, thus allowing for the correct execution of Boolean operations.

In the future, we plan to integrate this algorithm in a more comprehensive framework for vector graphics on surfaces, by including the more advanced features present in state-of-the-art 2D editors.

ACKNOWLEDGMENTS

The Sapienza group was partially supported by MIUR under grant *Dipartimenti di Eccellenza.* Alec Jacobson was partially supported by a Canada Research Chair and Sloan Fellowship. We wish to thank Tamal K. Dey and Stefano Vigni for useful discussions. Most meshes come from the Thingi10k repository [Zhou and Jacobson 2016]. *T-shirt* courtesy of Turbosquid (by Mladelfa); *Nefertiti* courtesy of Scan the World; *Mug* courtesy of Malopolska‘s Virtual Museums; *Lucy* courtesy of the Stanford 3D Scanning Repository; *Hand* courtesy of Caterina Zamai.

REFERENCES

Adobe inc. 2021. *Adobe Illustrator.* Adobe inc. https://adobe.com/products/illustrator

A. Amirkhanov. 2022. CDT: Constrained Delaunay Triangulation. https://github.com/artem-ogre/CDT

M. Attene. 2020. Indirect Predicates for Geometric Constructions. *Computer-Aided Design* 126 (2020), 102856:1–102856:9.

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

G. Bernstein and D. Fussell. 2009. Fast, Exact, Linear Booleans. In *Proc. of the Symp. on Geom. Proc.* 1269–1278.

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

G. Cherchi, M. Livesu, R. Scateni, and M. Attene. 2020. Fast and Robust Mesh Arrangements using Floating-point Arithmetic. *ACM Trans. Graph.* 39, 6 (2020), 250:1–250:16.

F. de Goes, M. Desbrun, M. Meyer, and T. DeRose. 2016. Subdivision exterior calculus for geometry processing. *ACM Trans. Graph.* 35, 4 (2016), 133:1–133:11.

T.K. Dey and H. Schipper. 1995. A new technique to compute polygonal schema for 2-manifolds with application to null-homotopy detection. *Discrete Computational Geometry* 14 (1995), 93–110.

X. Du, Q. Zhou, N. Carr, and T. Ju. 2021. Boundary-Sampled Halfspaces: A New Representation for Constructive Solid Modeling. *ACM Trans. Graph.* 40, 4 (2021), 53:1–53:15.

J. Erickson and K. Whittlesey. 2005. Greedy optimal homotopy and homology generators. In *Proc. 16th ACM-SIAM Symp. Disc. Alg.*, Vol. 5. 1038–1046.

Steven Fortune and Christopher J. Van Wyk. 1993. Efficient exact arithmetic for computational geometry. In *SCG '93.*

W. Fulton. 1995. *Algebraic Topology: A First Course.* Springer, New York, USA.

M. Granados, P. Hachenberger, L. Hert, S. adn Kettner, K. Mehlhorn, and M. Seel. 2003. Boolean operations on 3d selective NEF complexes: Data structure, algorithms, and implementation. In *Proc. 11th Europ. Symp. on Alg.* LNCS, Vol. 2832. Springer, 174–186.

B. Grünbaum and G. C. Shephard. 1990. Rotation and winding numbers for planar polygons and curves. *Trans. AMS* 322, 1 (1990), 169–187.

A Hatcher. 2002. *Algebraic Topology.* Cambridge, UK.

Y. Hu, T. Schneider, X. Gao, Q. Zhou, A. Jacobson, D. Zorin, and D. Panozzo. 2019. TriWild: Robust Triangulation with Curve Constraints. *ACM Trans. Graph.* 38, 4 (2019), 52:1–52:15.

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

A. Jacobson, L. Kavan, and O. Sorkine-Hornung. 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Trans. Graph.* 32, 4 (2013), 33:1–33:12.

A. Johnson. 2014. Clipper - an open source freeware library for clipping and offsetting lines and polygons. http://www.angusj.com/delphi/clipper.php

D. Knuth. 1986. *Metafont: the Program.* Addison-Wesley.

B. Levy. 2016. Robustness and efficiency of geometric programs: The Predicate Construction Kit (PCK). *Computer-Aided Design* 72 (2016), 3–12.

C. Mancinelli, G. Nazzaro, F. Pellacini, and E. Puppo. 2022. B/Surf: Interactive Bézier Splines on Surfaces. *IEEE Trans. VIs. Comp. Graph* (2022). https://doi.org/10.1109/

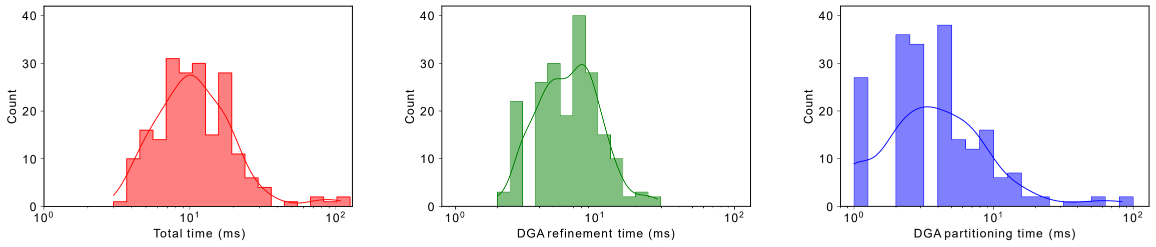

Fig. 22. Top: Renderings of the performance test on a subset of Thingi10K. Two overlapping SVG drawings are randomly projected over each mesh, generating a valid shape arrangement. A randomly chosen Boolean operation among union, intersection, difference and symmetric difference is computed between such shapes. Bottom: Timings density in milliseconds for, from left to right: total execution; DGA refinement stage; and cell graph extraction. Total execution time varies between 3 ms and 108 ms, with a mean execution time of 14 ms and less than 5 tests with execution time higher than 100 ms.

TVCG.2022.3171179 to appear.

Claudio Mancinelli and Enrico Puppo. 2022. Vector graphics on surfaces using straightedge and compass constructions. *Computers & Graphics* 105C (2022), 46–56. https://doi.org/10.1016/j.cag.2022.04.007

M. Mcintyre and G. Cairns. 1993. A new formula for winding number. *Geometriae Dedicata* 46, 2 (1993), 149–159.

B. Naylor, J. Amanatides, and W. Thibault. 1990. Merging BSP Trees Yields Polyhedral Set Operations. *SIGGRAPH Comput. Graph.* 24, 4 (1990), 115–124.

G. Nazzaro, E. Puppo, and F. Pellacini. 2021. geoTangle: Interactive Design of Geodesic Tangle Patterns on Surfaces. *ACM Trans. Graph.* 41, 2 (2021), 12:1–12:17.

Diego Nehab and Hugues Hoppe. 2008. Random-access rendering of general vector graphics. *ACM Transactions on Graphics (TOG)* 27, 5 (2008), 1–10.

Ganesh Ramanarayanan, Kavita Bala, and Bruce Walter. 2004. *Feature-based textures.* Technical Report. Cornell University.

B. L Reinhart. 1960. The winding number on two manifolds. *Annales de l'Institut Fourier* 10 (1960), 271–283.

N. Sharp and K. Crane. 2020. You Can Find Geodesic Paths in Triangle Meshes by Just Flipping Edges. *ACM Trans. Graph.* 39, 6 (2020), 249:1–15.

B. R. Vatti. 1992. A Generic Solution to Polygon Clipping. *Commun. ACM* 35, 7 (1992), 56–63.

B. Wang, Z. Ferguson, T. Schneider, X. Jiang, M. Attene, and D. Panozzo. 2021. A Large-Scale Benchmark and an Inclusion-Based Algorithm for Continuous Collision Detection. *ACM Trans. Graph.* 40, 5, Article 188 (2021), 16 pages.

R. Wein, E. Berberich, E. Fogel, D. Halperin, M. Hemmer, O. Salzman, and Zukerman B. 2021. CGAL 5.3.1 - 2D Arrangements. https://doc.cgal.org/latest/Arrangement_on_surface_2/index.html

S.-Q. Xin, Y. He, and C.-W. Fu. 2012. Efficiently Computing Exact Geodesic Loops within Finite Steps. *IEEE Trans. Vis. Comp. Graphi.* 18, 6 (2012), 879–889.

Q. Zhou, E. Grinspun, D. Zorin, and A. Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Trans. Graph.* 35, 4 (2016), 39:1–39:15.

Q. Zhou and A. Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. arXiv:1605.04797

A PROOF OF THEOREM 1

The proof stems from the following lemma.

Lemma 3 (by Mcintyre and Cairns [1993]). *Let us consider the partition of $\mathcal{S}$ induced by the arrangement of a curve γ and a set of generators $\nu_1, \dots, \nu_{2g}$, and let x_0 be a point of $\mathcal{S}$. One can associate integers to each of the regions such that at each segment of γ the number to the left of γ is 1 greater than the number to the right of γ, and for each $i = 1, \dots, 2g$, the number to the left of each segment of ν_i is n_i less than the number to the right of ν_i, and the region containing x_0 is numbered zero. Moreover, such a numbering is unique.*

To prove the theorem, we first observe that Lemma 3 can be extended to a set of curves $\gamma_1, \dots, \gamma_h$. In fact, the homology of their combination is

$$[\gamma_1 + \dots + \gamma_h] = \sum_{j=1}^{h} n_{1,j} [\nu_1] + \dots + \sum_{j=1}^{h} n_{2g,j} [\nu_{2g}]$$

where $n_{i,j}$ is the multiplicity of generator ν_i in the homology of curve γ_j. Now we build the labeling of Lemma 3 for each one of the γ_j separately; next, we overlay their arrangements and we label each resulting cell with the sum of the labels of the regions in the various arrangements containing it. Then the resulting labeling also fulfills the properties stated in Lemma 3.

Given the labeling above, let us assume that $[\gamma_1 + \dots + \gamma_h]$ is null-homologic. This means that $\sum_{j=1}^{h} n_{i,j} = 0$ for all i, hence the label does not change when crossing any of the ν_i. We obtain the cell arrangement induced by A by removing the generators and merging the regions adjacent through them without changing their labels. The resulting labeling differs by one when crossing any γ_j from right to left, hence is a consistent labeling for A. Conversely, if A has a consistent labeling, we offset all labels by the same amount to have a label zero in the region containing x_0. Then we overlay the generators by splitting regions with them and maintaining the same labeling. The result is a valid labeling for Lemma 3, where labels do not change when we cross generators, hence for all i we must have $\sum_{j=1}^{h} n_{i,j} = 0$.