Compatible Intrinsic Triangulations

KENSHI TAKAYAMA, National Institute of Informatics / CyberAgent, Japan

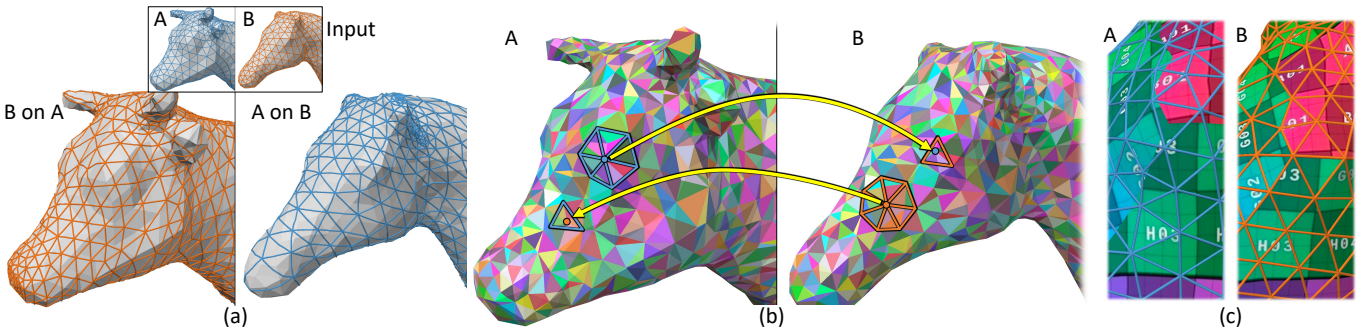

Fig. 1. Given two triangle meshes A and B, we find a continuous, bijective and low-distortion map between them (their edge images shown in (a)). Internally, we build a Compatible Intrinsic Triangulation (CIT), a pair of intrinsic triangulations on both models sharing the same connectivity; every vertex of A is inserted to B's intrinsic mesh, and vice versa (b). Our algorithm generates a piecewise-linear map in the form of overlay polygons; here, B's texture is being transferred to A with all the seams preserved (c).

Finding distortion-minimizing homeomorphisms between surfaces of arbitrary genus is a fundamental task in computer graphics and geometry processing. We propose a simple method utilizing intrinsic triangulations, operating directly on the original surfaces without going through any intermediate domains such as a plane or a sphere. Given two models A and B as triangle meshes, our algorithm constructs a *Compatible Intrinsic Triangulation* (CIT), a pair of intrinsic triangulations over A and B with full correspondences in their vertices, edges and faces. Such a tessellation allows us to establish consistent images of edges and faces of A's input mesh over B (and vice versa) by tracing piecewise-geodesic paths over A and B. Our algorithm for constructing CITs, primarily consisting of carefully designed edge flipping schemes, is empirical in nature without any guarantee of success, but turns out to be robust enough to be used within a similar second-order optimization framework as was used previously in the literature. The utility of our method is demonstrated through comparisons and evaluation on a standard benchmark dataset.

CCS Concepts: • **Computing methodologies** → **Mesh models**; **Mesh geometry models**.

Additional Key Words and Phrases: cross-parameterization, inter-surface mapping, bijection, texture transfer, intrinsic triangulation, compatible triangulation

ACM Reference Format:
Kenshi Takayama. 2022. Compatible Intrinsic Triangulations. *ACM Trans. Graph.* 41, 4, Article 57 (July 2022), 12 pages. https://doi.org/10.1145/3528223.3530175

1 INTRODUCTION

Computing maps between surfaces is needed in many contexts, and has been a classical topic of great importance in computer graphics and geometry processing. In particular, maps that are continuous and bijective, called *homeomorphisms*, are required for many applications such as texture transfer and template fitting in order to avoid various kinds of artifacts, but at the same time they are generally more difficult to compute than other relaxed versions of maps.

Given two surfaces A and B as triangle meshes, a naive and intuitive idea one might think of is as follows:

(1) map A's vertices onto B somehow, e.g., by using some form of projection, then
(2) extend the above vertex-based map to A's edges and faces somehow, e.g., by using geodesics on B.

If one pursued the above idea while working directly on the original surfaces, however, they would quickly realize that things get complicated. As such, all previous works introduce an *intermediate domain* such as a plane or a sphere and establish homeomorphisms by going through that intermediate domain. The method proposed by Schmidt et al. [2020] is the current state of the art in this area, where the surfaces are locally mapped to an intermediate domain in a globally consistent manner by using constant-curvature metrics. While their theory is elegant and results are impressive, their approach is conceptually not so intuitive, potentially making its implementation difficult for non-expert practitioners.

In this work, we present a simpler alternative that pursues the above intuitive idea using *intrinsic triangulations* [Sharp et al. 2019b] as our key ingredient. We propose *Compatible Intrinsic Triangulation* (CIT), a pair of intrinsic triangulations defined on A and B where the vertices, edges and faces are fully in correspondence (Fig. 1). Given vertex images as input, our algorithm constructs a CIT by employing carefully designed edge flipping schemes and other local operations (Sec. 4). CITs define strict homeomorphisms, and allow one to easily compute mapping distortions and their derivatives as well as to optimize the vertex images by using the similar second-order global optimization scheme as was used in the previous work [Schmidt et al. 2020] (Sec. 6).

We do not claim, however, any practical advantages over Schmidt et al. [2020]'s method. In fact, our method is less robust than theirs

Author's address: Kenshi Takayama, kenshi84@gmail.com, National Institute of Informatics / CyberAgent, Japan.

in the sense that it relies on the input vertex images being sufficiently high-quality; otherwise, our algorithm can fail, as discussed in Sec. 7. Nevertheless, to the best of our knowledge, our work is the first in the literature on computing surface homeomorphisms without going through any intermediate domains. In exchange for the conceptual simplicity and intuitiveness of our approach, we had to deal with a number of combinatorial problems which make our algorithm inherently empirical and heuristic, providing no guarantee for success. Yet, our algorithm is overall straightforward to implement, and we experimentally demonstrate practical level of robustness of our method using a standard benchmark dataset (Sec. 7.3).

We believe it is natural to utilize intrinsic triangulations for the purely intrinsic problem of inter-surface mappings. The two research domains have received increased attentions in the community recently, but only in separate, unrelated contexts. This work aims at bridging the two, thereby inspiring other researchers to develop new solutions to this difficult problem. In this light, we release our reference implementation at https://github.com/kenshi84/compatible-intrinsic-triangulations.

2 RELATED WORK

There are a large body of work on computing maps between surfaces, and they can be roughly divided into two groups based on whether strict homeomorphism is sought after or not. Those which do not seek for homeomorphism seem more popular, presumably because it is easier to represent and optimize maps in this setting. A variety of methods have been proposed including:

- projection-based methods [Ezuz et al. 2019a,b; Panozzo et al. 2013],
- distribution-based methods [Mandad et al. 2017; Ovsjanikov et al. 2012; Solomon et al. 2012], and
- methods developed in the context of surface registration [Bouaziz et al. 2013; Huang et al. 2008; Sharf et al. 2006; Tam et al. 2013; Wu et al. 2007; Yang et al. 2019; Yang et al. 2020; Zhang et al. 2006].

While there are many use cases for these relaxed maps, turning them into actual homeomorphisms is a non-trivial, open problem.

Computing homeomorphisms between surfaces is generally more difficult, and usually an intermediate domain is used in order to define the final map as a composition of individual maps between surfaces and the intermediate domain. Examples of intermediate domains include:

- a plane [Aigerman and Lipman 2015; Aigerman et al. 2014, 2015; Kanai et al. 1997; Kim et al. 2011; Lipman and Funkhouser 2009; Litke et al. 2005; Schmidt et al. 2019; Tierny et al. 2011],
- a sphere [Aigerman et al. 2017; Alexa 2000; Asirvatham et al. 2005; Baden et al. 2018],
- a hyperbolic plane [Aigerman and Lipman 2016; Shi et al. 2017], and
- base complexes [Kraevoy and Sheffer 2004; Praun et al. 2001; Schreiner et al. 2004].

Among the above, only a very few address the problem of minimizing the map distortion in an end-to-end manner. Methods by Schreiner et al. [2004] and Kraevoy et al. [2004] only allow local optimization per each vertex's 1-ring neighborhood, and are prone to converging to undesirable local minima. Methods by Litke et al. [2005] and Schmidt et al. [2019] do offer global optimization, but they are applicable only to surfaces of disk topology.

Distortion minimization of homeomorphisms between closed surfaces of arbitrary genus in a global and end-to-end manner was addressed only recently by Schmidt et al [2020]. Our problem setting is exactly the same, and our results are similar with theirs. Unlike their method where surfaces are locally mapped to a respective intermediate domain (either a plane, a sphere, or a hyperbolic plane, depending on the surface genus) by using constant-curvature metrics, our method constructs consistent mappings between surfaces directly on the original surfaces without going through any intermediate domains by using intrinsic triangulations [Sharp et al. 2019b], which we believe leads to a simpler (albeit heuristic) algorithm.

Intrinsic triangulations. Our work is directly inspired by the powerful concept of intrinsic triangulations based on the signpost data structure [Sharp et al. 2019b], which has already found a number of extensions and use cases in the literature. Sharp and Crane [2020a] utilized it for defining a high-quality Laplacian operator on non-manifold meshes as well as point clouds. Sharp and Crane [2020b] also demonstrated that one can find polyhedral geodesics on surfaces by just flipping edges intrinsically. The intrinsic Delaunay triangulation algorithm has already been used for various geometry processing tasks [El Ouafdi et al. 2021; Fumero et al. 2020; Tao et al. 2021]. Our work widens the application domains of intrinsic triangulations to the problem of inter-surface mappings.

Compatible triangulations for 2D animation. Our algorithm for constructing CITs is mainly about deciding which edges to flip based on the configuration of nearby vertices in a local 2D coordinate system, and thus is quite related to the existing literature on compatible triangulations in the context of 2D animation [Alexa et al. 2000; Baxter III et al. 2009; Liu et al. 2018; Surazhsky and Gotsman 2004]. Revisiting these algorithms in our context may lead to substantial improvement of our algorithm, which is left for future work.

3 OVERVIEW

Our input consists of a pair of triangle meshes $M_\mathsf{A} = (V_\mathsf{A}, E_\mathsf{A}, F_\mathsf{A})$ and $M_\mathsf{B} = (V_\mathsf{B}, E_\mathsf{B}, F_\mathsf{B})$ for the two models, along with A's vertex image $\phi_{\mathsf{A}\to\mathsf{B}} : V_\mathsf{A} \mapsto F_\mathsf{B} \times \Lambda$, with $\Lambda = \{(\lambda_1, \lambda_2, \lambda_3) | \sum_i \lambda_i = 1\} \subset \mathbb{R}^3$ being the space of barycentric coordinates, which maps A's vertex to a point inside B's face, and the other one in the opposite direction $\phi_{\mathsf{B}\to\mathsf{A}}$. Any method of choice can be used to obtain these vertex images, such as the Hyperbolic Orbifold Tutte (HOT) method [Aigerman and Lipman 2016]; the only requirement is that they need to be reasonably consistent in both directions in order for our algorithm to succeed. Given such input data, our algorithm generates a Compatible Intrinsic Triangulation (CIT), as explained in Sec. 4.

In Sec. 5, we explain how to process the generated CIT further in order to obtain images of E_A on M_B and vice versa as piecewise-geodesic polylines, as well as the piecewise-linear mapping between A and B as an overlay mesh.

From the generated CIT, we compute the map's distortion energy and its derivatives using automatic differentiation, just like the previous method [Schmidt et al. 2020]. We displace the current

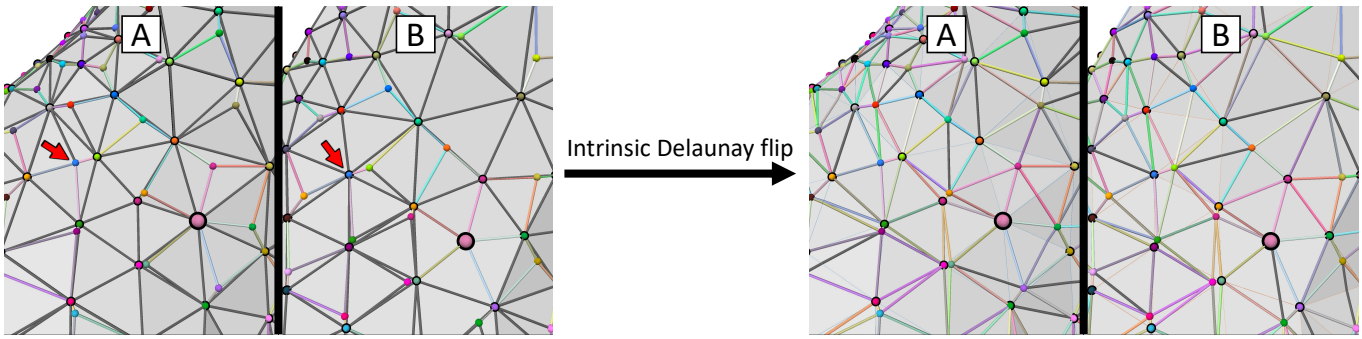

Fig. 2. The vertex insertion step (left) followed by the intrinsic Delaunay flipping step (right). Each vertex in correspondence is drawn in its unique color, and each original (i.e., non-inserted) vertex is drawn with a black silhouette. A merged vertex is drawn as a larger ball. The vertex highlighted by the red arrow originates in B and is inserted to A's input edge, splitting the two adjacent intrinsic faces. Each compatible edge (i.e., connecting the same pair of corresponding vertices) is drawn in its unique color, while each incompatible edge is drawn in dark gray. Notice how the intrinsic Delaunay flipping step reduces the number of incompatible edges significantly.

vertex images by the computed descent direction multiplied by a certain step size, and plug them into our CIT generation algorithm again to arrive at a new configuration. We explain this optimization process in Sec. 6.

Notations. We use a bold font style to distinguish the intrinsic mesh and its elements from the input mesh and its elements on either model. For example, $v_\mathsf{A} \in V_\mathsf{A}$ refers to a vertex of A's input mesh, while $\mathbf{e}_\mathsf{B} \in \mathbf{E}_\mathsf{B}$ refers to an edge of B's intrinsic mesh.

4 CIT GENERATION

4.1 Vertex insertion

After initializing intrinsic meshes $\mathbf{M}_\mathsf{A}$ and $\mathbf{M}_\mathsf{B}$ as copies of input meshes M_A and M_B, we first insert A's input vertices V_A into $\mathbf{M}_\mathsf{B}$ according to the vertex image $\phi_{\mathsf{A}\to\mathsf{B}}$. For each input vertex $v_\mathsf{A} \in V_\mathsf{A}$, we check if its image $\phi_{\mathsf{A}\to\mathsf{B}}(v_\mathsf{A}) = (f_\mathsf{B}, \boldsymbol{\lambda})$ is strictly inside f_B, i.e., if $\lambda_i > 0, \forall i$ (we call such a mapped point a *face point*). If so, we insert a new vertex into an intrinsic face $\mathbf{f}_\mathsf{B} \in \mathbf{F}_\mathsf{B}$ corresponding to that face point, splitting $\mathbf{f}_\mathsf{B}$ into three.

If one component of $\{\lambda_i\}$ is zero, the mapped point is exactly on B's input edge $e_\mathsf{B} \in E_\mathsf{B}$ (we call such a mapped point an *edge point*). In this case, we insert a new vertex into an intrinsic edge $\mathbf{e}_\mathsf{B} \in \mathbf{E}_\mathsf{B}$ corresponding to that edge point, splitting $\mathbf{e}_\mathsf{B}$ into two.

If two components of $\{\lambda_i\}$ are zero, this means A's vertex v_A is mapped exactly onto B's vertex v_B, i.e., $\phi_{\mathsf{A}\to\mathsf{B}}(v_\mathsf{A}) = v_\mathsf{B}$. In this case, we ensure that the vertex image in the opposite direction is consistent, i.e., $\phi_{\mathsf{B}\to\mathsf{A}}(v_\mathsf{B}) = v_\mathsf{A}$. We conceptually interpret such a case as the original vertex and the inserted vertex being *merged* in the intrinsic mesh, and we do not insert a new vertex in this case. In our method, we achieve hard constraints on a set of fixed corresponding vertex pairs by keeping them as merged (we call such fixed vertices *anchors*). If not using this hard constraints mode, we will later split each merged vertex into two (as explained in Sec. 4.7) in order to treat the two vertices as variables in the optimization.

Additionally, we introduce the following adjustment procedure for stability reasons: when all of $\{\lambda_i\}$ are positive but one component is extremely small, the mapped face point is almost on one of the edges of the mapped face. Having an intrinsic vertex at a face point extremely close to an input edge causes numerical instability for our overlay mesh extraction algorithm. As such, we clamp the smallest value below a threshold to zero and increase the other two components to sum to one, and treat it as an edge point.

We perform the same vertex insertion process in the opposite direction (B → A) as well.

4.2 Compatible edges & faces

After the vertex insertion step, the intrinsic meshes $\mathbf{M}_\mathsf{A}$ and $\mathbf{M}_\mathsf{B}$ have the exact same numbers of vertices, edges and faces. Also, the intrinsic vertices of both models $\mathbf{V}_\mathsf{A}$ and $\mathbf{V}_\mathsf{B}$ are fully in correspondence. If the endpoints of A's intrinsic edge $\mathbf{e}_\mathsf{A} \in \mathbf{E}_\mathsf{A}$ correspond to the endpoints of B's intrinsic edge $\mathbf{e}_\mathsf{B} \in \mathbf{E}_\mathsf{B}$, we say $\mathbf{e}_\mathsf{A}$ and $\mathbf{e}_\mathsf{B}$ are *compatible*. If no such intrinsic edge in $\mathbf{E}_\mathsf{B}$ exists, we say $\mathbf{e}_\mathsf{A}$ is *incompatible*. We can further define the notion of face compatibility: if all the three edges of A's intrinsic face $\mathbf{f}_\mathsf{A} \in \mathbf{F}_\mathsf{A}$ are compatible, and if there exists B's intrinsic face $\mathbf{f}_\mathsf{B} \in \mathbf{F}_\mathsf{B}$ having the same set of compatible edges in the consistent order, then we say $\mathbf{f}_\mathsf{A}$ and $\mathbf{f}_\mathsf{B}$ are compatible. If $\mathbf{f}_\mathsf{B}$ has the same set of compatible edges but in the reversed order, we say $\mathbf{f}_\mathsf{A}$ and $\mathbf{f}_\mathsf{B}$ are *reversed*. If an intrinsic face is neither compatible nor reversed, we say it is incompatible. If all the intrinsic edges are compatible, all the intrinsic faces will be compatible by necessity. Our goal is to reach this state by mutating $\mathbf{M}_\mathsf{A}$ and $\mathbf{M}_\mathsf{B}$ in certain ways.

Fig. 2 left shows an example state after the vertex insertion step. We draw each vertex in correspondence as well as each compatible edge in their unique color, while we draw each incompatible edge in dark gray. Notice that only some fraction of the edges are compatible at this point.

There are some rules about intrinsic edges that must be adhered to in our algorithm: first, we do not allow self-edges, i.e., edges starting from and ending at the same vertex. We also do not allow multiple intrinsic edges connecting the same pair of intrinsic vertices, because they prevent the definition of one-to-one correspondence between A's intrinsic edge and B's intrinsic edge. These rules imply that every intrinsic vertex always has degree greater than two. In

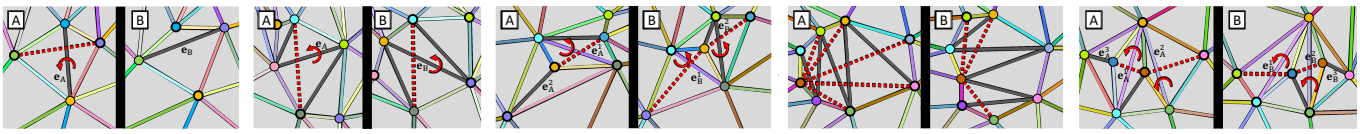

Fig. 3. Five subroutines constituting our FlipToCompatible algorithm.

the following where we flip intrinsic edges, in addition to the geometric feasibility check in the original signpost method [Sharp et al. 2019b], we perform this uniqueness check in order to determine flippability of intrinsic edges.

4.3 Delaunay flipping

The first step in improving our meshes' compatibility is to perform the intrinsic Delaunay flipping algorithm [Sharp et al. 2019b] on both intrinsic meshes $\mathbf{M}_\mathsf{A}$ and $\mathbf{M}_\mathsf{B}$ independently. When the input vertex images are reasonably consistent in both directions, the intrinsic vertices $\mathbf{V}_\mathsf{A}$ and $\mathbf{V}_\mathsf{B}$ tend to have vertex neighborhoods in similar geometric configurations, thus the Delaunay flipping procedures on both models tend to result in similar connectivities. Often, this step already makes a significant portion of the intrinsic edges compatible (Fig. 2 right).

4.4 Merging nearby vertices

An extremely short intrinsic edge in a CIT can be a source of numerical problems both for the computation of the energy derivatives and for the generation of the overlay mesh. The Delaunay flipping algorithm tends to have connected such nearby vertex pairs as compatible edges. Such extremely short intrinsic edges also tend to be *collapsible*, i.e., its endpoints consist of an original vertex and an inserted vertex, as they are usually caused by the input vertex images mapping A's vertex very close to B's vertex and vice versa.

As such, before proceeding to our main algorithm for improving compatibility as described below, we collapse each of such collapsible compatible edges whose length is below a threshold and turn them into a merged vertex. This edge collapse operation is realized by deleting the inserted vertex and reconnecting all of its adjacent edges to the original vertex.

4.5 FlipToCompatible algorithm

In this section, we describe our FlipToCompatible algorithm which tries to make $\mathbf{M}_\mathsf{A}$ and $\mathbf{M}_\mathsf{B}$ as compatible as possible by just flipping edges. Note that it is rare to obtain a valid CIT with this algorithm alone, especially when the input vertex images start to deviate from being mutually consistent (which happens when we examine large step sizes during optimization), and we will need to resort to some additional procedures that involve relocation of inserted vertices, as we explain later. We do prefer, however, to utilize edge flips maximally to improve compatibility, because we do not want to modify the input vertex images unnecessarily.

Our algorithm consists of five subroutines (Fig. 3), and it executes each of them in order. If a certain subroutine was able to increase the number of compatible edges, it goes back to the beginning and repeats until no more changes can be made. In the following, we explain each of the subroutines in the order of the more frequently applicable to the less frequent. The less frequent ones take effect only occasionally, but they do increase the chance of success of our CIT generation algorithm and are thus necessary.

Below, when deciding whether to flip an edge or not, we assume it is flippable (i.e., we ignore edges that are not flippable).

Subroutine 1: SimpleFlip. For A's incompatible edge $\mathbf{e}_\mathsf{A} \in \mathbf{E}_\mathsf{A}$, we check its opposite vertices, and if there is an incompatible edge $\mathbf{e}_\mathsf{B} \in \mathbf{E}_\mathsf{B}$ connecting these vertices, we flip $\mathbf{e}_\mathsf{A}$. We perform the same procedure in the opposite direction as well.

Subroutine 2: SimpleCoFlip. If there is a pair of incompatible edges $\mathbf{e}_\mathsf{A}$ and $\mathbf{e}_\mathsf{B}$ that have the same opposite vertices, we flip both of them.

Subroutine 3: FlipCompatible. Given a compatible edge $\mathbf{e}^1_\mathsf{A}$ and its counterpart $\mathbf{e}^1_\mathsf{B}$, we check if there is an incompatible edge $\mathbf{e}^2_\mathsf{B}$ that is either connecting $\mathbf{e}_\mathsf{A}$'s opposite vertices, or is flippable and has the same opposite vertices as $\mathbf{e}^1_\mathsf{A}$. We perform the same check in the opposite direction for $\mathbf{e}^1_\mathsf{B}$, and if the two checks both pass, we flip $\mathbf{e}^1_\mathsf{A}$ and $\mathbf{e}^1_\mathsf{B}$ (as well as $\mathbf{e}^2_\mathsf{B}$ (resp. $\mathbf{e}^2_\mathsf{A}$) if it has the same opposite vertices as $\mathbf{e}^1_\mathsf{A}$ (resp. $\mathbf{e}^1_\mathsf{B}$)).

Subroutine 4: FlipFlatPolygon. Sometimes, incompatible edges are clustered and form a patch of connected faces with no interior vertex. In such a case, all the faces in the patch can be flattened to 2D without any distortions, resulting in a pair of compatible 2D polygon boundaries for A and B. Our algorithm tries to triangulate these two polygons in a compatible way. We use a simple heuristic to achieve this: for each vertex, we check if it is "visible" from each of all the other vertices of the polygon (i.e., the line segment between them is contained within the polygon) for both A and B. If such a vertex is found, we connect all the other vertices to that found vertex by repeatedly flipping edges.

Subroutine 5: DoubleFlipCompatible. For a pair of nearby compatible edges $(\mathbf{e}^1_\mathsf{A}, \mathbf{e}^2_\mathsf{A})$ and its counterpart $(\mathbf{e}^1_\mathsf{B}, \mathbf{e}^2_\mathsf{B})$, we flip all these four edges if the following conditions hold:

- $\mathbf{e}^1_\mathsf{A}$ and $\mathbf{e}^2_\mathsf{B}$ have the same opposite vertices.
- There is an incompatible edge $\mathbf{e}^3_\mathsf{B}$ that is connecting the opposite vertices of $\mathbf{e}^2_\mathsf{A}$.
- There is an incompatible edge $\mathbf{e}^3_\mathsf{A}$ that is connecting the opposite vertices of $\mathbf{e}^1_\mathsf{B}$.

4.6 Resolving incompatible patches

We define a connected set of non-compatible (i.e., reversed or incompatible) faces as an *incompatible patch*. At this point, assuming

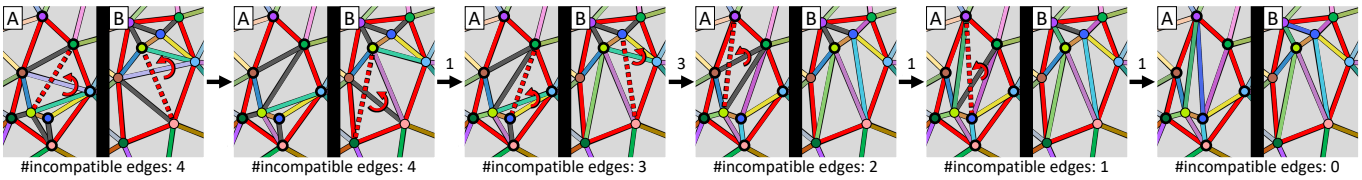

Fig. 4. An example where FlipToCompatible is stuck (leftmost) but flipping one compatible edge pair within an incompatible patch (its boundary edges shown in red) leads to a different configuration (middle left) which can be handled by FlipToCompatible. The number above each arrow refers to the used subroutine of FlipToCompatible, while the number of incompatible edges in the patch is shown at the bottom of each figure.

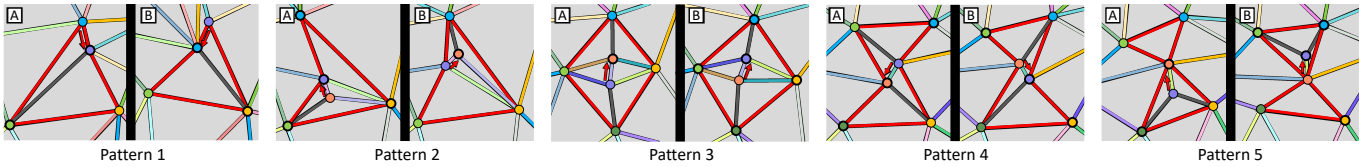

Fig. 5. Five patterns for merging inconsistently positioned vertices. Original and inserted vertices are drawn with thick and thin black silhouettes, respectively. The inserted vertex to be removed after the merge are highlighted by red arrows in each pattern.

the input vertex images are reasonably consistent, we expect that the majority of edges have been made compatible and that there are small and sparsely distributed incompatible patches left. Our strategy then is to examine each of these incompatible patches and try to make them compatible by various means as explained below.

Note that a pair of incompatible patches in A and B can be made correspondent by checking if they consist of the same set of corresponding vertices. Thus, we extract all the corresponding pairs of incompatible patches and process each of them in order.

4.6.1 Exploring variations by flipping edges within patches. Consider an example incompatible patch shown in Fig. 4 at the far left where our FlipToCompatible algorithm cannot make any progress. It is, however, still possible to make FlipToCompatible process this patch if we flip the compatible edge pair highlighted by the red arrows in the figure. Note that flipping this edge pair does not increase the number of incompatible edges.

Based on this observation, after the termination of FlipToCompatible, for each corresponding pair of incompatible patches, and for each corresponding pair of their compatible edges, we check if flipping the edge pair preserves the number of incompatible edges. If so, we flip the edge pair and run FlipToCompatible again and see if it makes any progress, and if it does, we go back to the extraction of incompatible patches and repeat.

4.6.2 Merging inconsistently positioned vertices. After running our edge flipping algorithms as described above, there can still remain a few incompatible patches that cannot be resolved by just flipping edges. Such incompatible patches are often caused by the input vertex images mapping some vertex pairs close to each other on both A and B, but in slightly inconsistent positions. Our next strategy then is to merge these vertices so that incompatible edges arising from them will disappear. We use a pattern-based approach for determining which vertex pair to merge, and we empirically identified five common patterns (Fig. 5). Note that for pattern 5 we relaxed the notion of face connectedness when extracting incompatible patches so that faces sharing only one vertex can be included in the same patch. See Appendix A for details about how we handle each pattern.

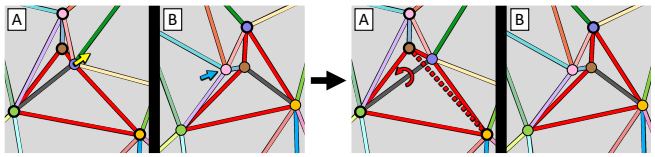

Fig. 6. An example where our vertex merging algorithm cannot merge the purple and brown vertices due to the presence of the pink vertex on B (as highlighted by the blue arrow), since the line segment between the green and purple vertices crosses the cyan and orange edges adjacent to the pink vertex. In such a case, we move the purple vertex on A outwards a bit (as highlighted by the yellow arrow) to make the boundary shape of this incompatible patch convex, making its incompatible edge flippable (as highlighted by the red arrow).

4.6.3 Deforming patch boundary. Our vertex merging algorithm is sometimes unable to merge the intended vertices due to various configurations of vertices neighboring the incompatible patch, as illustrated in Fig. 6. In such cases, as a last resort, we try to deform the boundary shape of each incompatible patch so that it becomes convex, which will likely make some non-flippable incompatible edges flippable. Note that this operation can also fail due to various nearby vertex configurations. We run FlipToCompatible again if any of the patches could be deformed.

4.7 Splitting merged vertices

If there are any incompatible edges left at this point, our CIT generation algorithm has failed. Otherwise, we move to the next step of splitting merged (non-anchor) vertices (Fig. 7). To split a merged vertex, we first need to determine along which pair of edges the split should occur. We choose such a pair of edges (consistently across A and B) in a way that makes the angle between the two edges as close to π as possible. The newly created vertex is positioned at a

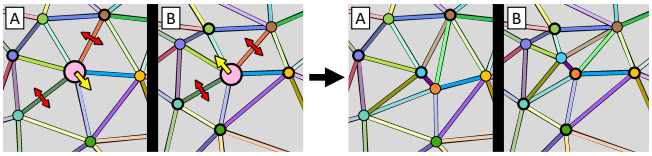

Fig. 7. The split of a merged vertex. The pair of edges to be split is indicated by the red arrows, while the displacement vectors for the newly created vertex are indicated by the yellow arrows.

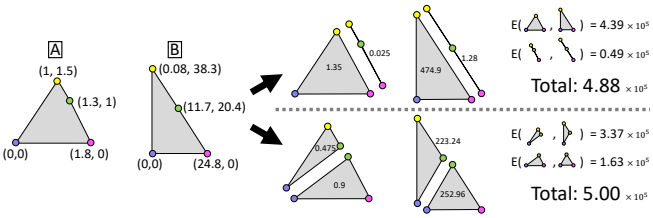

Fig. 8. An example where flipping edges to minimize the distortion energy can give rise to almost degenerate faces. A number inside or next to each triangle denotes its area.

point displaced from the original vertex in the average direction of the two edges and by a small distance (10% of the smallest height of the adjacent faces with the opposite edges being their bases). Note that the displacements on A and B should be opposite in order to maintain consistency.

4.8 Optimizing connectivity

Having succeeded in generating a valid CIT, we have a freedom to explore the space of CITs with various connectivities by flipping pairs of corresponding edges. A tempting strategy then, which we tried initially, is to repeatedly flip each edge pair if doing so results in lower energy, as per the Delaunay flipping algorithm [Sharp et al. 2019b], with an expectation that the final converged configuration will give the smoothest inter-surface map for a given vertex image. We realized, however, that this strategy has a serious issue: it does not care at all about the generation of almost degenerate faces (Fig. 8) which will cause numerical problems both in the computation of energy derivatives and the extraction of the overlay mesh.

As such, the strategy we adopted is to flip edges such that the smallest angle of corners of triangles on both A and B is maximized. While leading to a slightly higher energy, this approach generates much better quality triangles which makes our overall pipeline robust, and it tends to make the final edge and face images appear smoother, to our surprise. An in-depth analysis on the space of CITs by flipping edges is left for future work.

5 OBTAINING PIECEWISE-LINEAR MAP

5.1 Obtaining edge images

To obtain the piecewise-linear map induced by a CIT in the form of an overlay mesh, we first need to obtain images of E_A on M_B and vice versa, which tell us about where edges of A and B intersect. The process consists of three steps: first, we trace A's intrinsic edges $\mathbf{E}_\mathsf{A}$ over A's input mesh M_A to detect all their intersections with A's original edges E_A, as was done in the original signpost method for visualizing intrinsic edges [Sharp et al. 2019b] (Fig. 9a). Next, we reinterpret the obtained intersections (edge points on M_A) as edge points on $\mathbf{M}_\mathsf{A}$. This is easily done by calculating the relative arc-length parameter value of each intersection point with respect to the relevant intrinsic edge (Fig. 9b). Note that, by definition, a path corresponding to $e_\mathsf{A} \in E_\mathsf{A}$ on $\mathbf{M}_\mathsf{A}$ is always a geodesic. Finally, we linearly map such a path to B's intrinsic mesh $\mathbf{M}_\mathsf{B}$, obtaining a (generally) non-geodesic path on $\mathbf{M}_\mathsf{B}$, and connect its corresponding points on M_B by geodesics, obtaining a piecewise-geodesic path on M_B (Fig. 9c). The directions and distances for this final geodesic tracing are calculated based on the geometry of $\mathbf{M}_\mathsf{B}$. Intersections of this piecewise-geodesic path and E_B are recorded as intersections between e_A and E_B.

We run the same process in the opposite direction to obtain images of B's original edges on M_A. Note that this second run generates the same set of intersections between E_A and E_B up to small numerical errors.

5.2 Overlay mesh generation

Having detected all the intersections between all possible pairs of edges, we are now ready to generate an overlay mesh for each of A and B. As the two intrinsic meshes $\mathbf{M}_\mathsf{A}$ and $\mathbf{M}_\mathsf{B}$ are compatible, we no longer need to distinguish between them, so we drop the subscript in the following description when referring to the intrinsic mesh and its elements. Our goal is to enumerate all the overlay polygons in each intrinsic face $\mathbf{f} \in \mathbf{F}$ which can be intersected by any number of A's original edges and B's original edges (Fig. 10 left). We achieve this using a data structure called an *overlay wedge* which is generated for each corner of an overlay polygon and encodes information about which halfedge to switch to when going around the overlay polygon (Fig. 10 right), as explained below.

We call each vertex in an overlay mesh an *overlay vertex* which can come from either

(1) an intersection between edges (which can come from either E_A, E_B, or $\mathbf{E}$),
(2) A's original vertex V_A, or
(3) B's original vertex V_B.

For the first case, we generate a quadruplet of overlay wedges for each intersection (Fig. 11a). An overlay wedge w stores a reference to the halfedge h_IN which comes into the corner when viewed from inside its associated overlay polygon (assuming the counterclockwise ordering), along with the relative arc-length parameter value t_IN representing the position of the intersection on h_IN. Note that h_IN can refer to either the intrinsic mesh's halfedge $\mathbf{h}$, A's original halfedge h_A, or B's original halfedge h_B. Likewise, w also stores a reference to the outgoing halfedge h_OUT and its associated parameter value t_OUT.

For the other two cases, we generate a number of overlay wedges according to the number of edges incident to the overlay vertex (Fig. 11b-d). To do so, we first need to determine the correct ordering of halfedges incident to the overlay vertex originating from different sources (M_A, M_B, or $\mathbf{M}$), which is possible thanks to the signpost data structure storing directions of intrinsic edges relative to the

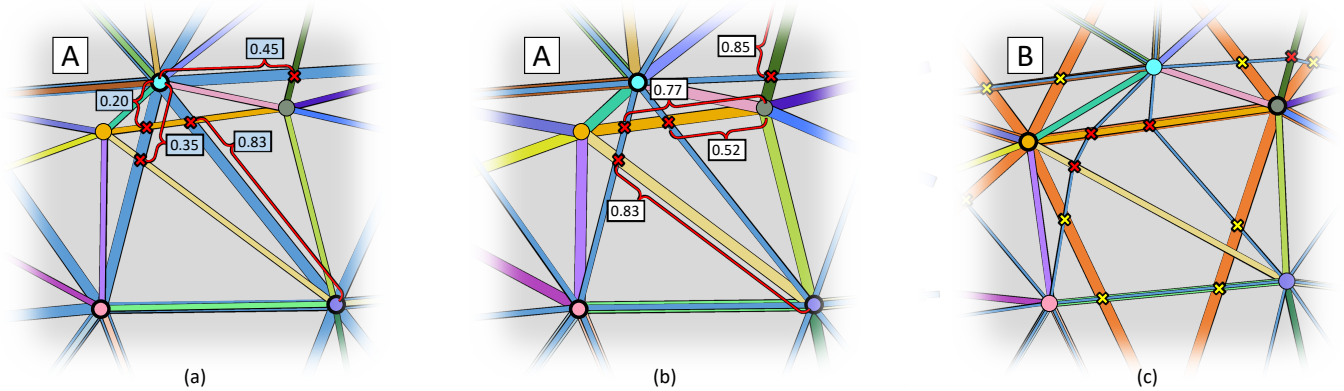

Fig. 9. The generation of images of A's original edges on B's input mesh. We first detect all the intersections of A's intrinsic edges against A's original edges, depicted as red crosses (a). The numbers in the blue boxes denote the relative arc-length parameter values of the intersections with respect to A's original edges. We then reinterpret these intersections as points on A's intrinsic edges, with their relative arc-length parameter values denoted by the numbers in the white boxes (b). Finally, we linearly map these points to B's intrinsic mesh, obtaining non-geodesic paths, and connect each pair of consecutive path points by a geodesic, obtaining piecewise-geodesic paths on B's input mesh (c). The yellow crosses denote intersections of these geodesic pieces and B's original edges, representing the intersections between A's edges and B's edges. We use line thickness to visualize different kinds of edges (A's original edges, B's original edges, and intrinsic edges), and we use the largest thickness to imply the underlying domain on which the geodesic paths are computed or defined.

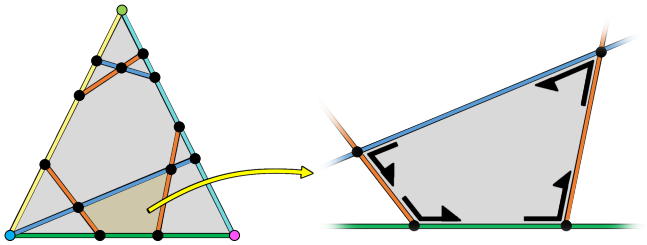

Fig. 10. (Left) Each intrinsic face can be intersected by any number of edges of A and B, resulting in overlay polygons. (Right) We enumerate each overlay polygon by using a data structure called *overlay wedge* associated with each corner of the overlay polygon.

canonical tangent spaces defined on vertices, edges, and faces of the input mesh [Sharp et al. 2019b].

Having processed all the overlay vertices and generated all the overlay wedges, we group them by their $\mathtt{h}_{\mathtt{IN}}$ fields, and for each group, we sort them by their $\mathtt{t}_{\mathtt{IN}}$ values. This allows us to find, given an overlay wedge $\mathtt{w}$, its "next" overlay wedge in the overlay polygon by the following procedure:

(1) obtain the sorted list of wedges associated with $\mathtt{w.h}_{\mathtt{OUT}}$, and
(2) scan the list in the ascending order and return the first item whose $\mathtt{t}_{\mathtt{IN}}$ value is greater than $\mathtt{w.t}_{\mathtt{OUT}}$.

This way, we can find cycles of overlay wedges and thus generate overlay polygons.

Also, because each overlay wedge stores references to halfedges of the input meshes, we can transfer the texture coordinate data present in the original meshes (which are typically stored per halfedge) to the overlay meshes. This allows us to transfer a texture image from one model to the other while preserving all the seams, as demonstrated in Fig. 1c.

6 OPTIMIZATION

6.1 Computing energy & derivatives

Given a CIT, computing its distortion energy (measured using the symmetric Dirichlet as was done in the previous work [Schmidt et al. 2019, 2020]) is simple if it were not for its derivative computation: the intrinsic edge lengths already define the 2D shape of each intrinsic face, so we can trivially compute the energy using these 2D shapes:

$$\mathcal{E} = \sum_{\mathbf{f}\in\mathbf{F}} \|\mathbf{J}(\mathbf{f})\|_{\mathrm{F}}^2 \, Area(\mathbf{f}_{\mathsf{B}}) + \|\mathbf{J}(\mathbf{f})^{-1}\|_{\mathrm{F}}^2 \, Area(\mathbf{f}_{\mathsf{A}}) \tag{1}$$

where $\mathbf{J}(\mathbf{f}) = \mathbf{D}(\mathbf{f}_{\mathsf{B}})\mathbf{D}(\mathbf{f}_{\mathsf{A}})^{-1} \in \mathbb{R}^{2\times 2}$ is a map Jacobian deforming $\mathbf{f}_{\mathsf{A}}$ into $\mathbf{f}_{\mathsf{B}}$ in an arbitrary 2D embedding, $\mathbf{D}(\mathbf{f}_{\mathsf{A}}) = [\mathbf{p}_2 - \mathbf{p}_1, \mathbf{p}_3 - \mathbf{p}_1] \in \mathbb{R}^{2\times 2}$, and $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3 \in \mathbb{R}^2$ are the embedded 2D coordinates of $\mathbf{f}_{\mathsf{A}}$'s three corners ($\mathbf{D}(\mathbf{f}_{\mathsf{B}}) \in \mathbb{R}^{2\times 2}$ is defined analogously). We use this lightweight method to compute the energy during the line search as described in Sec. 6.2.

To compute the derivative, however, we need to express the energy as a function of the input vertex images $\mathcal{E}(\phi_{\mathsf{A}\to\mathsf{B}}, \phi_{\mathsf{B}\to\mathsf{A}})$, which boils down to expressing those embedded 2D coordinates of intrinsic faces' corners (i.e., $\mathbf{p}_1, \mathbf{p}_2$ and $\mathbf{p}_3$ introduced above) as linear combinations of 2D coordinates of V_{A} or V_{B} locally embedded in 2D. For example, suppose one corner of $\mathbf{f} \in \mathbf{F}$ originates in $v_{\mathsf{B}} \in V_{\mathsf{B}}$ and is mapped to a face point on M_{A} as $\phi_{\mathsf{B}\to\mathsf{A}}(v_{\mathsf{B}}) = (f_{\mathsf{A}}, (\lambda_1, \lambda_2, \lambda_3))$ where $\lambda_1 > 0$, $\lambda_2 > 0$ and $\lambda_3 = 1 - \lambda_1 - \lambda_2 > 0$. Then, we need to express the 2D coordinate of the corresponding corner of $\mathbf{f}_{\mathsf{A}}$ as a function of λ_1 and λ_2:

$$\mathbf{p}(\lambda_1, \lambda_2) = \lambda_1\mathbf{q}_1 + \lambda_2\mathbf{q}_2 + (1 - \lambda_1 - \lambda_2)\mathbf{q}_3 \tag{2}$$

where $\mathbf{q}_1, \mathbf{q}_2$ and $\mathbf{q}_3$ are the 2D coordinates of f_{A} embedded in 2D locally. Note that this local embedding (or flattening) must be consistent (i.e., in a common coordinate frame) among all the original faces $\{f_{\mathsf{A}}\} \subset F_{\mathsf{A}}$ that support $\mathbf{f}_{\mathsf{A}}$ (we call this face set $\mathbf{f}_{\mathsf{A}}$'s *support patch*, see Fig. 12 left). This is always possible without introducing

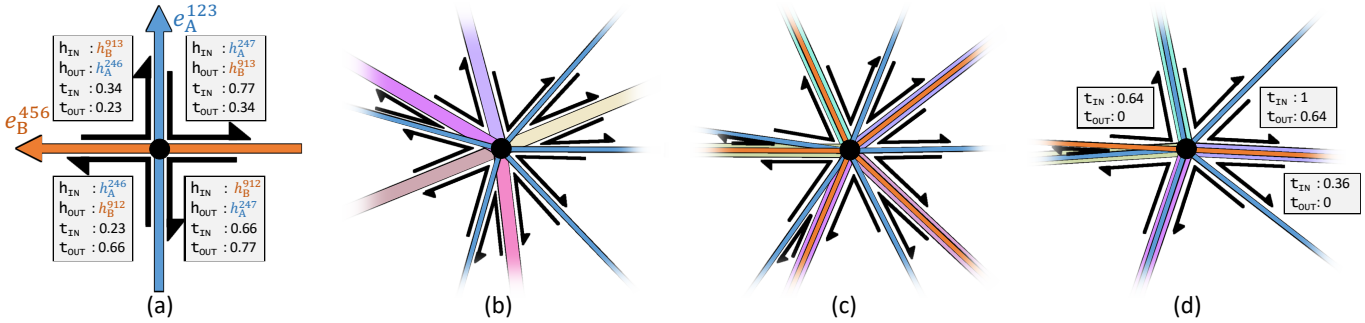

Fig. 11. Overlay wedges generated per overlay vertex (depicted as a black circle at the center in each figure). (a) For an overlay vertex due to the intersection of two edges, we generate four overlay wedges. Here, A's original edge at index 123 and B's original edge at index 456 are intersected (with their canonical directions depicted as arrows). We use a common convention that an edge at index n has its pair of halfedges at indices $2n$ and $2n+1$. (b,c,d) For an overlay vertex of the other types (A's vertex inserted into B's face, an anchor vertex where A's vertex and B's vertex overlap, and A's vertex inserted into B's edge, respectively), we generate overlay wedges according to the number of edges incident to the overlay vertex. Here, edges of the intrinsic mesh are depicted as thicker lines in various colors. Note that for (b) and (c), all the wedges have $t_{IN} = 1$ and $t_{OUT} = 0$.

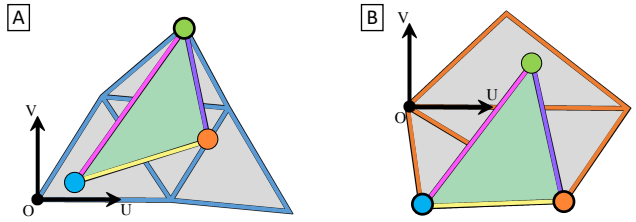

Fig. 12. Flattening of $\mathbf{f}_A$'s support patch (left) and $\mathbf{f}_B$'s support patch (right) for the computation of the energy derivative. Note that the orange vertex which originates in V_B is inserted into $e_A \in E_A$, and we need to include its adjacent face not overlapping with $\mathbf{f}_A$ in the patch because it is used as the tangent space for that edge point.

any distortions thanks to the construction of intrinsic triangulations guaranteeing nonexistence of original vertices inside any of intrinsic faces [Sharp et al. 2019b].

Note that when $\phi_{B\to A}(v_B)$ is an edge point, we use one of the edge's two adjacent faces (determined by the edge's canonical orientation) as the tangent space and reinterpret the edge point as a face point in that face. For this reason, $\mathbf{f}_A$'s support patch must include such an adjacent face even if it does not overlap with $\mathbf{f}_A$ (see the bottom right triangle of $\mathbf{f}_A$'s support patch in Fig. 12 left).

We perform the same procedure for $\mathbf{f}_B \in \mathbf{F}_B$ corresponding to $\mathbf{f}_A$ (Fig. 12 right), and derive the energy expression as a function of all the barycentric coordinates of the input vertex images involved in this intrinsic face. In the absence of anchor vertices, each corner of an intrinsic face corresponds to an inserted vertex in one model and to an original vertex in the other model, so each corner has two degrees of freedom, and each intrinsic face has six degrees of freedom. The existence of an anchor vertex decreases the degrees of freedom by two.

We use automatic differentiation to compute a local gradient vector $\mathbf{g}_\mathbf{f} \in \mathbb{R}^6$ and a Hessian matrix $\mathbf{H}_\mathbf{f} \in \mathbb{R}^{6\times 6}$ per intrinsic face $\mathbf{f} \in \mathbf{F}$. These local quantities are accumulated in a global gradient vector $\mathbf{g} \in \mathbb{R}^n$ and a Hessian matrix $\mathbf{H} \in \mathbb{R}^{n\times n}$ where $n = 2(|V_A| + |V_B|)$ in the absence of anchor vertices. As in the previous work [Schmidt et al. 2019, 2020], we make the global Hessian $\mathbf{H}$ positive definite by clamping negative eigenvalues of the local Hessian $\mathbf{H}_\mathbf{f}$ before accumulating.

6.2 Overall scheme

We basically adopt the same second-order optimization scheme as in the previous work [Schmidt et al. 2019, 2020]:

- We temporally smooth the gradient as $\bar{\mathbf{g}} = \sum_{i=0}^{5} 2^{-i}\mathbf{g}_i$ where $\mathbf{g}_i$ is the gradient of the configuration i steps previous to the current one. Temporally smoothed Hessian $\bar{\mathbf{H}}$ is obtained analogously.
- We compute the descent direction $\mathbf{d} \in \mathbb{R}^n$ by solving
$$(\bar{\mathbf{H}} + w_L(\mathbf{SL})^\top \mathbf{MSL})\mathbf{d} = -\bar{\mathbf{g}} \tag{3}$$
where w_L is a weight for the Laplacian preconditioning, $\mathbf{L}$ is the connection Laplacian, $\mathbf{S}$ is a block-diagonal matrix responsible for making the tangent vector magnitudes comparable, and $\mathbf{M}$ is the diagonal mass matrix.
- We perform a line search by evaluating the energy at $\mathbf{x} + s\mathbf{d}$ where $\mathbf{x} \in \mathbb{R}^n$ is the current configuration (i.e., barycentric coordinates in the vertex images) and $s > 0$ is the step size.

Because our variables are barycentric coordinates, all of the above are realized through the concept of tangent vector transport; please refer to Section 1 of the supplementary material of [Schmidt et al. 2020] for more details.

As opposed to the previous method which determined the feasibility of a given step size by the emergence of flipped triangles, we deem a given step size to be feasible if our CIT generation algorithm succeeds with it. When determining the maximum step size $s_{\max}$ for the line search, we initialize it based on the current descent direction vector (we set $s_{\max} = 0.01/d_{\max}$ where $d_{\max}$ is the largest length of the 2D descent direction vectors after being scaled from the barycentric coordinates to the comparable lengths). If this initial value turns out to be feasible, we enter the *upward* mode where we multiply $s_{\max}$ by 1.2 and see if the increased $s_{\max}$ is infeasible. Otherwise, we enter the *downward* mode where we divide $s_{\max}$ by

1.2 and see if the decreased $s_{\max}$ is feasible. With $s_{\max}$ determined, we look for the optimal step size satisfying the Armijo condition as was done in the previous method.

6.3 Dealing with local minima

Our method lacks one important feature that existed in the previous method [Schmidt et al. 2020]: the metric regularization. It was used in the initial few hundred steps to smooth out excess distortions in the input maps, and we believe this contributed greatly to the robustness of their optimization algorithm. In our case, unfortunately, we do not have anything that has the same effect as the metric regularization, and thus our optimization algorithm can easily get trapped by local minima, as discussed in Sec. 7.1.

We found empirically that when we get stuck in a local minimum, it is often possible to make (sometimes great) progress if we switch to the first-order optimization scheme. The reason we speculate is that the energy landscape can sometimes get very non-smooth with steep peaks and dents, making the quadratic approximation of the second-order optimization scheme a poor fit. As such, we devised the following simple workaround: we start with the second-order mode, and when we get stuck in a local minimum, we switch to the first-order mode and keep moving until we get stuck again in another local minimum, then we switch back to the second-order mode and repeat, until we get stuck in both schemes.

7 RESULTS

Implementation and setup. We implemented our algorithm using geometry-central [Sharp et al. 2019a] that includes the reference implementation of the signpost data structure [Sharp et al. 2019b]. We used an unsupported module in Eigen [Guennebaud et al. 2010] for automatic differentiation. We ran all the experiments on Ryzen 9 3900X with 32GB of RAM running Ubuntu 20.04.1 LTS, and we always used a single thread during the optimization. We normalized the given input triangle meshes such that their total surface areas equal to one, meaning that the best attainable energy is 4.0. As opposed to the previous work, we did not impose any limit on the number of the optimization steps and let the algorithm run until it gets stuck (we stopped the optimization when the energy decrease fell below 10^{-5}).

7.1 Initial map generation

To create the initial vertex images, we used the reference implementation of HOT [Aigerman and Lipman 2016] and our own implementation of Seamless Surface Mappings (SSM) [Aigerman et al. 2015] (which we will release at https://github.com/kenshi84/seamlesssurfmap) for genus 0 cases and high genus cases, respectively. Because the success of our CIT generation/optimization algorithm depends heavily on the quality of the initial map, we allowed ourselves to use as many landmarks as necessary to produce good quality maps.

While theoretical analysis of exact conditions for the success of our CIT generation/optimization algorithm is left for future work, empirically we found some tips on how to place landmarks in an effective way. First, common to HOT and SSM, the mapping distortion tends to concentrate near landmarks; in particular, a landmark, when mapped to the other surface, may appear as if it was "pinched" in a certain direction (Fig. 13a). For a vertex image where this pinching effect is very strong, our CIT generation tends to fail. In such cases, we slightly shift those problematic landmarks toward the opposite of the pinched direction (Fig. 13b).

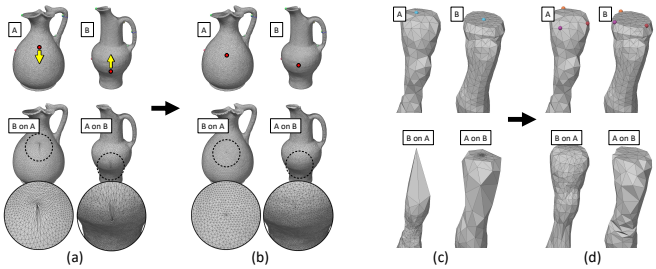

Fig. 13. Tips on placing landmarks. "B on A" visualizes the vertex image $\phi_{B\to A}$ as a mesh $(\phi_{B\to A}(V_B), E_B, F_B)$, and vice versa. Mapping distortion concentrated near a landmark may appear as if the landmark was "pinched" in a certain direction (a), and this may cause our CIT generation algorithm to fail. This can be remedied by shifting the landmark toward the opposite of the pinched direction (b). Placing just one landmark at the tip of an elongated part may lead to too uneven distribution of mapped vertices (c), which may cause our CIT optimization algorithm to get stuck in a local minimum. To avoid this, we place a few more landmarks (d).

Second, when mapping an elongated part, if we placed just one landmark at the tip of the part, its nearby vertices, when mapped to the other surface, may often get severely concentrated or dispersed (Fig. 13c). Even though our CIT generation algorithm often succeeds with such a vertex image, our CIT optimization algorithm often gets stuck in a local minimum, unable to distribute those vertices evenly. To avoid this, we add a few more landmarks near the tip of the part so that the vertex distribution becomes more even (Fig. 13d). See the supplemental material for our choices of landmarks.

7.2 Comparison to Schmidt et al. [2020]

We ran our method on the dataset released by Schmidt et al. [2020] and compared the final distortion energy (see Fig. 14 for the final optimized maps). As shown in Table 1, our optimized distortion energy is below that of the previous method [Schmidt et al. 2020] on all but the Ant-Octopus and Pig-Armadillo cases. We believe the lower values of our energy are due to the fact that Schmidt et al. terminated optimization after a fixed number of steps, whereas we let the optimizer run until it can make no more progress. The two unsuccessful cases are of extremely non-isometric shape pairs, and our algorithm quickly gets trapped by local minima.

7.3 Evaluation with Princeton Segmentation Dataset

Our method is inherently empirical, hence it is difficult to theoretically analyze how well it works in practice. To evaluate practical utility of our method, we ran our CIT optimization on models included in the Princeton 3D Mesh Segmentation Dataset [Chen et al. 2009]. The dataset consists of 19 categories, each containing 20 triangle meshes. We randomly chose 5 pairs in each category, totaling $19 \times 5 = 95$ pairs. To make our experiment more meaningful, we created some rules to filter out some clearly inappropriate pairs

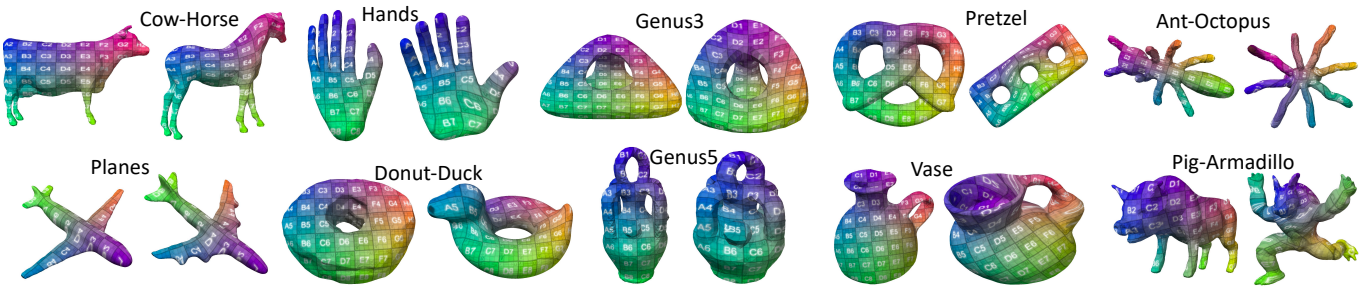

Fig. 14. Optimization results on the dataset of [Schmidt et al. 2020].

Table 1. The result statistics. **#F** refers to the total number of triangles of the input triangle mesh pair, **#N** refers to the number of optimization steps taken, **T** refers to the total time taken for the optimization, and **E** refers to the energy of the optimized map. The energy of the map generated by the previous method is shown in the rightmost column.

Case name	#F	#N	T	E	[Schmidt et al. 2020]
Planes	25k	231	15.0h	4.35	4.69
Cow-Horse	10k	225	3.8h	4.73	5.28
Hands	32k	366	37.6h	4.35	4.83
Genus3	3k	440	2.9h	4.21	4.36
Genus5	8k	767	15.0h	4.49	4.74
Pretzel	12k	532	11.2h	5.25	5.53
Donut-Duck	24k	585	78.9h	4.68	4.77
Vase	10k	542	9.1h	4.60	5.00
Ant-Octopus	8k	4	1m	98.3	16.2
Pig-Armadillo	22k	370	19.9h	8.10	7.78

as well as extremely poor quality geometries. In some categories, we also introduced some additional sub-categories within which we formed pairs randomly, so that the problem setting becomes more relevant. See the supplemental material for how we selected the pairs. Note also that we often applied moderate degree of mesh cleaning such as remeshing and reduction to eliminate erroneous configurations such as edges of almost zero dihedral angles.

Fig. 15 shows a sample of our optimized surface mapping from each category, indicating that our method indeed produces good quality maps. See the supplemental material for the detailed statistics as well as all the images of the mappings of all pairs.

Of the 95 cases, 7 cases (41-59, 50-46, 53-54, 55-51, 124-139, 146-156, 399-385), all genus 0, could not be handled by HOT (due to flipped triangles detected during its LBFGS optimization), no matter how we chose the landmarks. Out of the remaining 88 cases, we had 7 cases (62-72, 128-136, 129-126, 130-134, 152-149, 220-201, 386-397) where our CIT generation algorithm failed due to too extreme distortion in the mapping generated by HOT (Fig. 16). For the two cases 62-72 and 386-397, the shapes are widely different near the tail region, while for the two cases 152-149 and 220-201, the overall shapes are very different. For the three cases 128-136, 129-126 and 130-134, the very thin and long tentacles of the octopuses are arranged in some inconsistent manner.

Apart from these rather extreme cases, our CIT generation and optimization algorithm succeeded on 81 models out of 88, giving us success rate of 92%.

8 CONCLUSION AND FUTURE WORK

In this work, we showed that it is possible to consistently define, given vertex images as input, images of edges and faces of one model onto the other model using Compatible Intrinsic Triangulations instead of constant-curvature metrics [Schmidt et al. 2020]. We demonstrated that our CIT generation algorithm is robust enough to be used within a second-order global optimization framework.

There are, however, a number of issues in our current approach. First, there are no theoretical grounds supporting our CIT generation algorithm which we devised only through empirical observations. A thorough analysis of our problem setting is essential and can be a key to improving the robustness when handling extremely distorted vertex images.

One possible idea for improving our algorithm's success rate is to introduce additional vertices which are both inserted on $\mathbf{M}_A$ and $\mathbf{M}_B$, much like the Steiner vertices for 2D triangulation. We have not tried it yet as it seemed to complicate our optimization framework due to the number of variables changing at every step, especially when considering the treatment of the temporal smoothing operator.

Another possible way of improvement would be to devise a mechanism that has the same effect as the metric regularization feature in the previous work [Schmidt et al. 2019, 2020] which we believe will make our optimization algorithm significantly more robust.

Finally, an interesting future direction would be to combine our method with recent techniques for computing bijective maps between coarse and fine meshes such as [Jiang et al. 2020], in order to enable a multi-resolution framework for the inter-surface mapping problem. This will benefit from the simplicity of our approach, requiring only vertex images as input, in contrast to the previous method which requires valid constant-curvature metrics in addition to vertex images.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We also thank Ryoich Ando for generously offering us an access to his powerful computing environment which was essential for conducting our experiment.

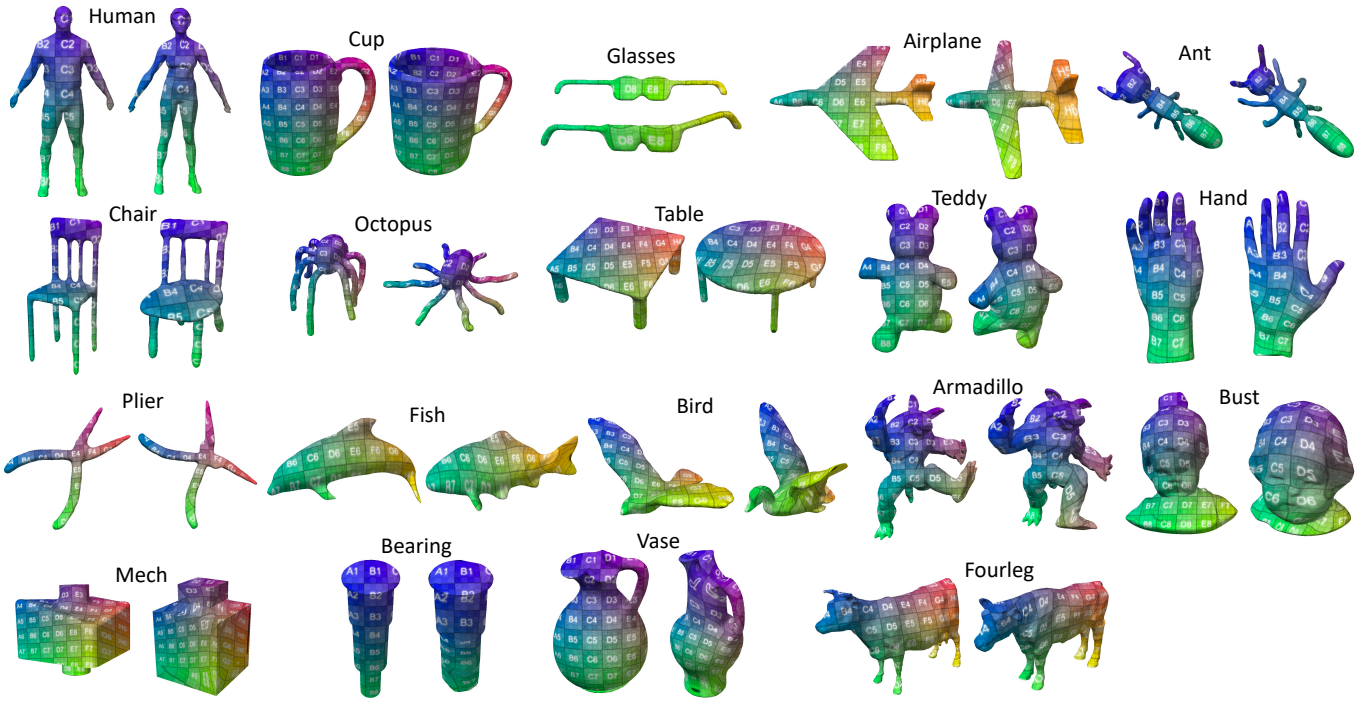

Fig. 15. Sample optimized maps from our experiment on the Princeton Segmentation Dataset.

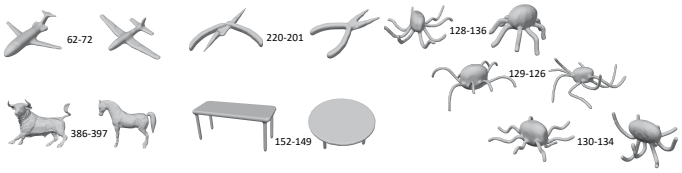

Fig. 16. 7 cases where our CIT generation algorithm failed..

REFERENCES

Noam Aigerman, Shahar Z. Kovalsky, and Yaron Lipman. 2017. Spherical Orbifold Tutte Embeddings. *ACM Trans. Graph.* 36, 4, Article 90 (2017).

Noam Aigerman and Yaron Lipman. 2015. Orbifold Tutte Embeddings. *ACM Trans. Graph.* 34, 6, Article 190 (2015).

Noam Aigerman and Yaron Lipman. 2016. Hyperbolic Orbifold Tutte Embeddings. *ACM Trans. Graph.* 35, 6, Article 217 (2016).

Noam Aigerman, Roi Poranne, and Yaron Lipman. 2014. Lifted Bijections for Low Distortion Surface Mappings. *ACM Trans. Graph.* 33, 4, Article 69 (2014).

Noam Aigerman, Roi Poranne, and Yaron Lipman. 2015. Seamless Surface Mappings. *ACM Trans. Graph.* 34, 4, Article 72 (2015).

Marc Alexa. 2000. Merging polyhedral shapes with scattered features. *The Visual Computer* 16, 1 (2000), 26–37.

Marc Alexa, Daniel Cohen-Or, and David Levin. 2000. As-Rigid-as-Possible Shape Interpolation. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. 157–164.

Arul Asirvatham, Emil Praun, and Hugues Hoppe. 2005. Consistent Spherical Parameterization. In *Computational Science – ICCS 2005*. 265–272.

Alex Baden, Keenan Crane, and Misha Kazhdan. 2018. Möbius Registration. *Comput. Graph. Forum* 37, 5 (2018), 211–220.

W. V. Baxter III, P. Barla, and K. Anjyo. 2009. Compatible Embedding for 2D Shape Animation. *IEEE Trans. Vis. Comput. Graph.* 15, 5 (2009), 867–879.

Sofien Bouaziz, Andrea Tagliasacchi, and Mark Pauly. 2013. Sparse Iterative Closest Point. *Comput. Graph. Forum* 32, 5 (2013), 113–123.

Xiaobai Chen, Aleksey Golovinskiy, and Thomas Funkhouser. 2009. A Benchmark for 3D Mesh Segmentation. *ACM Trans. Graph.* 28, 3 (2009).

A. F. El Ouafdi, H. El Houari, and D. Ziou. 2021. Adaptive estimation of Hodge star operator on simplicial surfaces. *The Visual Computer* 37, 6 (2021), 1433–1445.

D. Ezuz, B. Heeren, O. Azencot, M. Rumpf, and M. Ben-Chen. 2019a. Elastic Correspondence between Triangle Meshes. *Comput. Graph. Forum* 38, 2 (2019), 121–134.

Danielle Ezuz, Justin Solomon, and Mirela Ben-Chen. 2019b. Reversible Harmonic Maps between Discrete Surfaces. *ACM Trans. Graph.* 38, 2, Article 15 (2019).

Marco Fumero, Michael Möller, and Emanuele Rodolà. 2020. Nonlinear Spectral Geometry Processing via the TV Transform. *ACM Trans. Graph.* 39, 6, Article 199 (2020).

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. https://eigen.tuxfamily.org/dox/unsupported/group__AutoDiff__Module.html.

Qi-Xing Huang, Bart Adams, Martin Wicke, and Leonidas J. Guibas. 2008. Non-Rigid Registration Under Isometric Deformations. *Comput. Graph. Forum* 27, 5 (2008), 1449–1457.

Zhongshi Jiang, Teseo Schneider, Denis Zorin, and Daniele Panozzo. 2020. Bijective Projection in a Shell. *ACM Trans. Graph.* 39, 6, Article 247 (2020).

T. Kanai, H. Suzuki, and F. Kimura. 1997. 3D geometric metamorphosis based on harmonic map. In *Proceedings The Fifth Pacific Conference on Computer Graphics and Applications*. 97–104.

Vladimir G. Kim, Yaron Lipman, and Thomas Funkhouser. 2011. Blended Intrinsic Maps. *ACM Trans. Graph.* 30, 4, Article 79 (2011).

Vladislav Kraevoy and Alla Sheffer. 2004. Cross-Parameterization and Compatible Remeshing of 3D Models. *ACM Trans. Graph.* 23, 3 (2004), 861–869.

Yaron Lipman and Thomas Funkhouser. 2009. MöBius Voting for Surface Correspondence. *ACM Trans. Graph.* 28, 3, Article 72 (2009).

Nathan Litke, Marc Droske, Martin Rumpf, and Peter Schröder. 2005. An Image Processing Approach to Surface Matching. In *Eurographics Symposium on Geometry Processing 2005*.

Zhiguang Liu, Liuyang Zhou, Howard Leung, and Hubert P. H. Shum. 2018. High-quality compatible triangulations and their application in interactive animation. *Computers & Graphics* 76 (2018), 60–72.

Manish Mandad, David Cohen-Steiner, Leif Kobbelt, Pierre Alliez, and Mathieu Desbrun. 2017. Variance-Minimizing Transport Plans for Inter-Surface Mapping. *ACM Trans. Graph.* 36, 4, Article 39 (2017).

Maks Ovsjanikov, Mirela Ben-Chen, Justin Solomon, Adrian Butscher, and Leonidas Guibas. 2012. Functional Maps: A Flexible Representation of Maps between Shapes. *ACM Trans. Graph.* 31, 4, Article 30 (2012).

Daniele Panozzo, Ilya Baran, Olga Diamanti, and Olga Sorkine-Hornung. 2013. Weighted Averages on Surfaces. *ACM Trans. Graph.* 32, 4, Article 60 (2013).

Emil Praun, Wim Sweldens, and Peter Schröder. 2001. Consistent Mesh Parameterizations. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. 179–184.
Patrick Schmidt, Janis Born, Marcel Campen, and Leif Kobbelt. 2019. Distortion-Minimizing Injective Maps between Surfaces. *ACM Trans. Graph.* 38, 6, Article 156 (2019).
Patrick Schmidt, Marcel Campen, Janis Born, and Leif Kobbelt. 2020. Inter-Surface Maps via Constant-Curvature Metrics. *ACM Trans. Graph.* 39, 4, Article 119 (2020).
John Schreiner, Arul Asirvatham, Emil Praun, and Hugues Hoppe. 2004. Inter-Surface Mapping. *ACM Trans. Graph.* 23, 3 (2004), 870–877.
Andrei Sharf, Marina Blumenkrants, Ariel Shamir, and Daniel Cohen-Or. 2006. SnapPaste: an interactive technique for easy mesh composition. *The Visual Computer* 22, 9 (2006), 835–844.
Nicholas Sharp and Keenan Crane. 2020a. A Laplacian for Nonmanifold Triangle Meshes. *Comput. Graph. Forum* 39, 5 (2020), 69–80.
Nicholas Sharp and Keenan Crane. 2020b. You Can Find Geodesic Paths in Triangle Meshes by Just Flipping Edges. *ACM Trans. Graph.* 39, 6, Article 249 (2020).
Nicholas Sharp, Keenan Crane, et al. 2019a. geometry-central. https://www.geometry-central.net.
Nicholas Sharp, Yousuf Soliman, and Keenan Crane. 2019b. Navigating Intrinsic Triangulations. *ACM Trans. Graph.* 38, 4, Article 55 (2019).
R. Shi, W. Zeng, Z. Su, J. Jiang, H. Damasio, Z. Lu, Y. Wang, S. Yau, and X. Gu. 2017. Hyperbolic Harmonic Mapping for Surface Registration. *IEEE Trans. Pattern Anal. Mach. Intell.* 39, 05 (2017), 965–980.
Justin Solomon, Andy Nguyen, Adrian Butscher, Mirela Ben-Chen, and Leonidas Guibas. 2012. Soft Maps Between Surfaces. *Comput. Graph. Forum* 31, 5 (2012), 1617–1626.
V. Surazhsky and C. Gotsman. 2004. High Quality Compatible Triangulations. *Eng. with Comput.* 20, 2 (2004), 147–156.
Gary K.L. Tam, Zhi-Quan Cheng, Yu-Kun Lai, Frank C. Langbein, Yonghuai Liu, David Marshall, Ralph R. Martin, Xian-Fang Sun, and Paul L. Rosin. 2013. Registration of 3D Point Clouds and Meshes: A Survey from Rigid to Nonrigid. *IEEE Trans. Vis. Comput. Graph.* 19, 7 (2013), 1199–1217.
J. Tao, J. Zhang, B. Deng, Z. Fang, Y. Peng, and Y. He. 2021. Parallel and Scalable Heat Methods for Geodesic Distance Computation. *IEEE Trans. Pattern Anal. Mach. Intell.* 43, 2 (2021), 579–594.
Julien Tierny, Joel Daniels, Luis G. Nonato, Valerio Pascucci, and Claudio T. Silva. 2011. Inspired quadrangulation. *Computer-Aided Design* 43, 11 (2011), 1516–1526.
Huai-Yu Wu, Chunhong Pan, Qing Yang, and Songde Ma. 2007. Consistent Correspondence between Arbitrary Manifold Surfaces. In *International Conference on Computer Vision*.
Y. Yang, X. Fu, S. Chai, S. Xiao, and L. Liu. 2019. Volume-Enhanced Compatible Remeshing of 3D Models. *IEEE Trans. Vis. Comput. Graph.* 25, 10 (2019), 2999–3010.
Yang Yang, Wen-Xiang Zhang, Yuan Liu, Ligang Liu, and Xiao-Ming Fu. 2020. Error-Bounded Compatible Remeshing. *ACM Trans. Graph.* 39, 4, Article 113 (2020).
Lei Zhang, Ligang Liu, Zhongping Ji, and Guojin Wang. 2006. Manifold Parameterization. In *Advances in Computer Graphics*. 160–171.

A ALGORITHMS FOR MERGING INCONSISTENTLY POSITIONED VERTICES

We match the given incompatible patch pair against each of the five patterns shown in Fig. 5. Each pattern analyzes the given patch pair and returns an edge pair (one on $\mathbf{M}_\mathsf{A}$ and the other on $\mathbf{M}_\mathsf{B}$) that meets certain conditions defined for that pattern. We then collapse such a returned edge pair if it is collapsible. We give up on a patch pair if none of the patterns return an edge pair for collapse.

Below, we use $\boldsymbol{\phi}_{\mathsf{A}\to\mathsf{B}}$ and $\boldsymbol{\phi}_{\mathsf{B}\to\mathsf{A}}$ to refer to the available correspondences between $\mathbf{M}_\mathsf{A}$'s elements and $\mathbf{M}_\mathsf{B}$'s elements; e.g., $\mathbf{v}_\mathsf{B} = \boldsymbol{\phi}_{\mathsf{A}\to\mathsf{B}}(\mathbf{v}_\mathsf{A})$ refers to B's intrinsic vertex corresponding to A's intrinsic vertex (which is always available), and $\mathbf{e}_\mathsf{A} = \boldsymbol{\phi}_{\mathsf{B}\to\mathsf{A}}(\mathbf{e}_\mathsf{B})$ refers to A's intrinsic edge corresponding to B's intrinsic edge, assuming that the edge is compatible.

Pattern 1. We first check if the patch contains no interior vertices. We then look for vertices whose interior angle (i.e., the sum of corner angles of adjacent faces in the patch) is larger than π. If A's patch and B's patch contain one such vertex each (referred to as $\mathbf{v}_\mathsf{A}$ and $\mathbf{v}_\mathsf{B}$, respectively), and if there exists an edge at A's patch boundary connecting $\mathbf{v}_\mathsf{A}$ and $\boldsymbol{\phi}_{\mathsf{B}\to\mathsf{A}}(\mathbf{v}_\mathsf{B})$, we return that edge.

Pattern 2. We first check if the patch contains just one interior vertex $\mathbf{v}_\mathsf{A}$ and just one reversed face $\mathbf{f}_\mathsf{A}$. We also check if $\mathbf{v}_\mathsf{A}$ is adjacent to $\mathbf{f}_\mathsf{A}$. Then we examine the two edges $\mathbf{e}_\mathsf{A}^1$ and $\mathbf{e}_\mathsf{A}^2$ adjacent to both $\mathbf{v}_\mathsf{A}$ and $\mathbf{f}_\mathsf{A}$. We return $\mathbf{e}_\mathsf{A}^1$ if $|\mathbf{e}_\mathsf{A}^1| + |\boldsymbol{\phi}_{\mathsf{A}\to\mathsf{B}}(\mathbf{e}_\mathsf{A}^1)| < |\mathbf{e}_\mathsf{A}^2| + |\boldsymbol{\phi}_{\mathsf{A}\to\mathsf{B}}(\mathbf{e}_\mathsf{A}^2)|$, otherwise $\mathbf{e}_\mathsf{A}^2$, with $|\cdot|$ representing the edge's length.

Pattern 3. We first check if the patch contains even number of interior vertices. Then, we look for an interior compatible edge pair $\mathbf{e}_\mathsf{A}$ and $\mathbf{e}_\mathsf{B}$ whose adjacent two faces are both reversed. If the adjacent vertices of $\mathbf{e}_\mathsf{A}$ and $\mathbf{e}_\mathsf{B}$ are all interior, we return $\mathbf{e}_\mathsf{A}$. Otherwise, if $\mathbf{e}_\mathsf{A}$ and $\mathbf{e}_\mathsf{B}$ are both flippable and have corresponding opposite vertices which are both interior, we flip both of them and return $\mathbf{e}_\mathsf{A}$.

Pattern 4. We first check if the patch contains no interior vertices. We then look for an interior compatible edge pair $\mathbf{e}_\mathsf{A}$ and $\mathbf{e}_\mathsf{B}$ whose adjacent vertices all have interior angles larger than π. If there is just one such edge pair in the patch, we return that edge.

Pattern 5. We match the given patch with the fixed mesh topology shown in the figure; i.e., the patch must have four faces, three interior edges, six boundary edges, five boundary vertices, and one interior vertex. We then identify the "hourglass" vertex; i.e., one that is adjacent to four boundary edges. Then we return an edge between the interior vertex and the hourglass vertex.