Volume 0 (1981), Number 0 pp. 1–12 COMPUTER GRAPHICS *forum*

Mesh Simplification For Unfolding

M. Bhargava[1] C. Schreck [2] M. Freire [2] P. A. Hugron [2] S. Lefebvre [2] S. Sellán [*3,4] B. Bickel [*1,5]

[1]ISTA (Institute of Science and Technology Austria), Austria [2]Université de Lorraine, CNRS, Inria, France
[3]Massachusetts Institute of Technology, USA [4]University of Toronto, Canada [5]ETH Zürich, Switzerland
([*]Joint Last Authors)

{manas.bhargava@ist.ac.at camille.schreck@inria.fr marco.freire@inria.fr }
{pierre-alexandre.hugron@inria.fr sylvain.lefebvre@inria.fr sgsellan@mit.edu bickelb@ethz.ch}

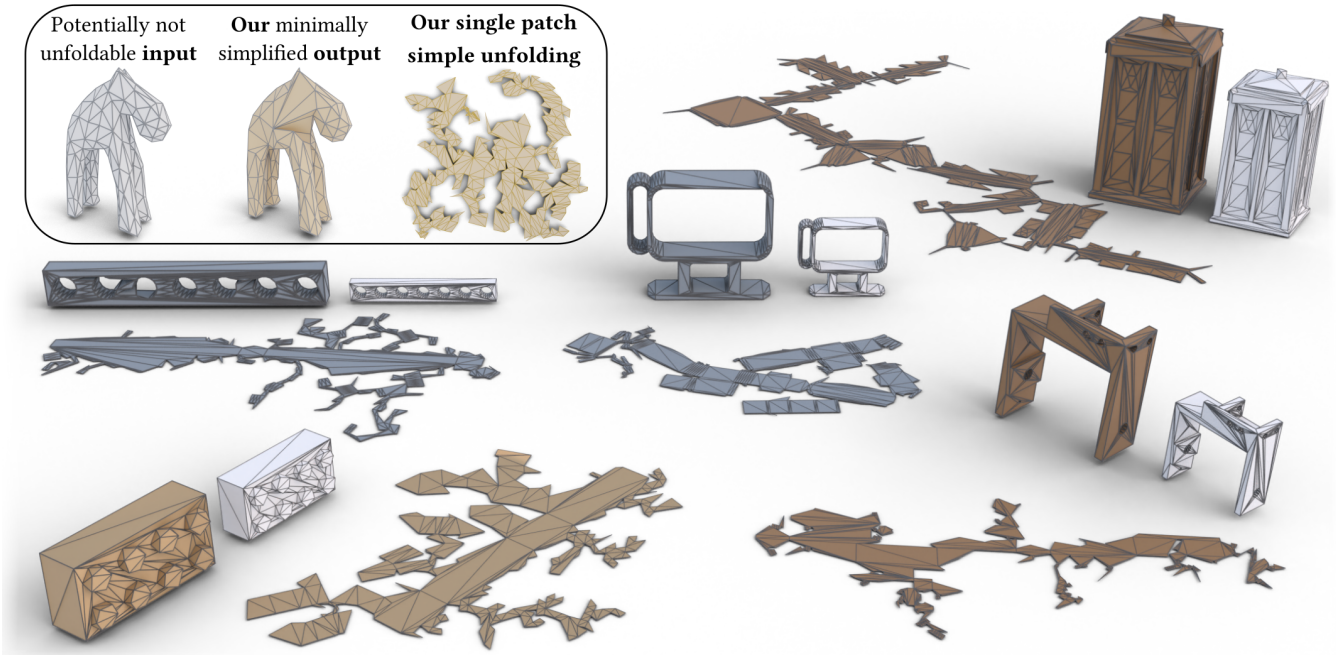


Figure 1: *Computing a single-patch unfolding for an arbitrary input (corner window, left) can be hard or outright impossible. Instead, we propose relaxing the problem and minimally modifying the input geometry (corner window, center) to make it easily unfoldable (corner window, right). Our algorithm is robust and returns single-patch unfoldings for shapes of arbitrary genus and triangulation quality (see collage).* ***See accompanying video for more results.***

Abstract

We present a computational approach for unfolding 3D shapes isometrically into the plane as a single patch without overlapping triangles. This is a hard, sometimes impossible, problem, which existing methods are forced to soften by allowing for map distortions or multiple patches. Instead, we propose a geometric relaxation of the problem: we modify the input shape until it admits an overlap-free unfolding. We achieve this by locally displacing vertices and collapsing edges, guided by the unfolding process. We validate our algorithm quantitatively and qualitatively on a large dataset of complex shapes and show its proficiency by fabricating real shapes from paper.

CCS Concepts
• ***Computing methodologies*** → ***Shape modeling;***

Keywords: Fabrication, Single patch unfolding, Mesh simplification

1. Introduction

An *unfolding* is a map from a surface to a single connected region of the two-dimensional plane that is isometric except for a finite set of discontinuities or *cuts*. If it is bijective (and therefore invertible), it is also called *simple* or *overlap-free* (see examples in Fig. 1). Computing simple unfoldings of arbitrary polygonal meshes is a well-known research problem in the geometric fabrication community, as its inverse map provides the necessary instructions to fabricate the shape efficiently by bending pieces of non-stretchable materials like paper, cardboard, wood or Printed Circuit Boards (PCB).

Despite the appeal of this application, finding an overlap-free unfolding of a given input is computationally challenging. Indeed, it is known that many shapes do not even admit such an unfolding [DDE20], and even finding if it is possible at all for an arbitrary mesh remains an open problem.

Because of this, existing approaches opt for *relaxing* the problem: for example, by allowing for a limited amount of distortion (i.e., removing the isometric constraint) or by permitting the unfolding to result in several, disconnected patches. Unfortunately, these relaxations severely hamper the methods' applicability in manufacturing, as many materials cannot stretch without tearing or losing their physical properties (e.g., paper, wood or PCB). Furthermore, assembling an object from multiple unfolded pieces is often impractical and impossible in certain fabrication processes that strictly require single patch unfoldings. For instance, [NYNM23] relies on single patch unfoldings to create 3D shapes using pull-up nets. Likewise, [FBS*23] require single patch unfoldings to ensure electrical connectivity in their 3D circuits.

Instead, we propose a *geometric* relaxation of the problem, modifying the input until it can be unfolded in a single patch. Ironically, our reframing takes advantage of the same complexity of the space of foldable shapes that has thwarted previous efforts: indeed, given an input mesh without a bijective unfolding, it is likely that there exists a *very similar* shape which *is* easily unfoldable in a single patch. Through a combination of local vertex displacements and guided edge collapses, our algorithm is a robust mesh processing strategy that returns a simple unfolding without making any assumption about the topological properties or triangulation quality of the input (see Fig. 1).

We evaluate our algorithm on a vast array of qualitatively representative examples for which previous works are unable to find overlap-free unfoldings (see Fig. 2). We quantitatively confirm these findings through large-scale experiments, which we also use to validate our parameter choices and ablate different algorithmic steps. Our method can be readily applied to common manufacturing frameworks, as we showcase with our fabricated results. Finally, we prototype the incorporation of user-defined constraints and end by discussing potential avenues for future work.

2. Related Work: Simple Unfoldings

Finding a simple unfolding of a surface is a core challenge faced by methods fabricating from planar sheets, whether to create origami by folding paper [DD02, CZ18, Pol09], to laser-cut objects that can be pulled up with strings [NYNM23], bend shape memory composites [FTS*13], to bend PCBs to manufacture surface electronics [FBS*23] or to make robots using robogami [SSS*17, YBCP19, Bel20]. Thus, it has been extensively studied.

Beyond basic shapes like spheres [VW08] and other convex surfaces [AAOS97], computing an overlap-free unfolding of a general input surface (if it exists at all [DDE20]) remains an unsolved open problem. Thus, existing approaches resort to *relaxing* the problem, computing mappings from surfaces to the plane that satisfy only most of the properties of a simple unfolding.

2.1. Relaxations: Distorted and Multi-Patch Unfoldings

A common relaxation of the problem is no longer forcing the map to preserve the metric of the surface, instead allowing it to introduce a minimal amount of distortion. This more general class of maps are known as *parametrizations* of the surface, and their computation has long been a commonly considered task in the Computer Graphics community due to its application to texture mapping. While a comprehensive review of the study of surface parametrizations is beyond the scope of this paper (see surveys by [WPF10] and [FSZ*21]), it suffices to say that great progress has been made: e.g., in obtaining conformal [LPRM23, SC17], as rigid as possible [LZX*08] or low distortion parametrizations [FW22, SYLF20, SS15, JSP17] or in efficiently optimizing for both cut placement and distortion [LKK*18, PTH*17].

However, such low distortion mappings are not sufficient for manufacturing applications like those listed at the beginning of this section. Materials like wood, paper or metal are truly non-stretchable, thus requiring *strictly isometric* unfoldings. Thus, a more relevant relaxation of the problem allows the unfolding to result in more than one patch or disconnected regions.

A popular approach is to make the shape piecewise developable [SGC18, SAJ20], where each segment can be mapped isometrically to 2D [Law11, YCS23]. [Tac09] showed how to fold a piece of paper into a 3D shape using tuck folds, an approach turned by [DT17] into a more practical algorithm. Although this works on many complex meshes, fabricating objects with tuck folds requires a lot of extra paper as well as significant folding time.

To simplify manufacturing, an alternative strategy assumes that the input shape is represented via a triangle mesh (which is trivially piecewise-developable) and that the cuts in the unfolding are restricted to coincide with mesh edges [DO07, O'R19]. This class of mappings, commonly known as *edge unfoldings*, is the one we concern ourselves with.

2.2. Edge Unfoldings of Triangle Meshes

For a general shape represented as a 3D mesh, finding a single-patch edge unfolding boils down to generating the dual graph and finding a spanning tree over it. When a mesh has a few hundred triangles a random spanning tree usually results in a single-patch unfolding with multiple overlaps [Sch89] (see Fig. 4).

Finding an edge unfolding consisting of a minimal number of disconnected patches is computationally hard [DO07]. Thus to obtain a single-patch edge unfolding many methods rely on heuristics and split the final unfolding into several patches to get rid of any remaining overlaps [Sch97].

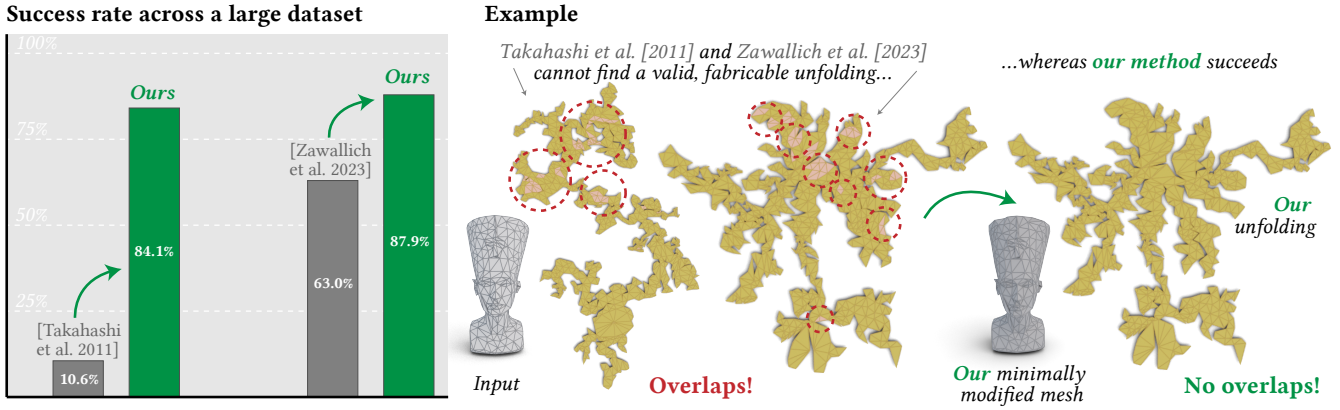


Figure 2: *Existing methods fail at consistently producing simple, non-overlapping unfoldings of complex shapes (middle). Instead, our method finds a close approximation to the input shape that admits a simple unfolding (right). In practice, this massively increases the number of shapes that can be fabricated from a single sheet of material (see quantitative bar chart on the left).*

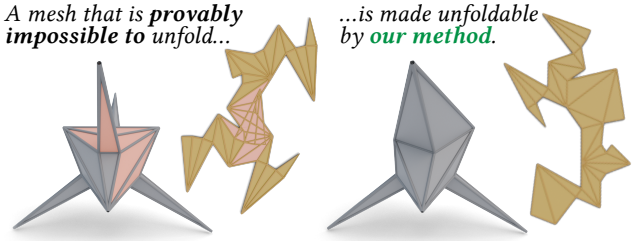


Figure 3: *Not only are simple unfoldings hard to compute; in some cases, they can be theoretically shown not to exist [DDE20]. This motivates us to find a close approximation to the input shape that admits an unfolding.*

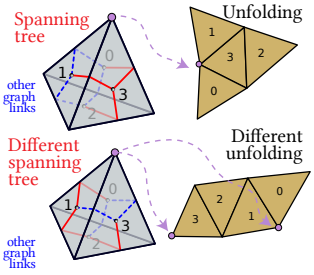
Agarwal et al. [AAOS97] suggest star unfoldings to obtain a single-patch unfolding, which works for general unfoldings of convex shapes but fails to find a solution for edge unfoldings. The minimum perimeter heuristic [SP11] minimizes the number of cuts used to obtain an initial unfolding and then resolves all overlaps by adding further cuts to separate them into multiple patches by solving a minimum set cover problem. Takahashi et al. [TWS*11] introduce a topological surgery approach where the input mesh is divided into smaller patches that are later merged using a genetic algorithm to obtain an unfolding. If it fails to generate a single-patch unfolding, the method outputs multiple overlap-free patches. Korpitsch at al. [KTGW20] use simulated annealing on the different spanning trees to obtain an unfolding with fewer overlaps.

Addressing a similar goal, Zawallich [Zaw24] demonstrated empirically that Tabu search can give a significant improvement in finding an unfolding with real-time performance for moderate mesh complexities. In a concurrent work, Zawallich [ZP24] used their unfolder in combination with different existing surface flow methods to showcase with great success, the unfolding of genus-zero meshes. Unfortunately, none of these algorithms can generate simple unfoldings on general input shapes with arbitrary genus, nor do they suggest a method to approximate the mesh to obtain one (see Fig. 2).

Interestingly, Demaine et al. [DDE20] noted that an unfolding being overlap-free is not a topological property but is connected to the geometry of the shape (see Fig. 3). This matches our observation that even small changes to the geometry can greatly reduce overlaps, and turn a challenging case for which no solution is found into a feasible one. We propose taking advantage of this through a *geometric* relaxation of the problem: starting from a potentially overlapping initial isometry, we progressively modify the input geometry to achieve a single-patch overlap-free unfolding.

3. Motivation: *Not-so-simple* unfoldings

Let us briefly consider the problem of finding an exact, single-patch overlap-free (or *simple*) unfolding Γ of a given input mesh Ω with vertices $\mathcal{V} = \{v_1, \ldots, v_n\}$, triangular faces $\mathcal{T} = \{t_1, \ldots, t_m\}$ and (undirected) edges $\mathcal{E} = \{e_1, \ldots, e_k\}$. The space of all single-patch unfoldings of Ω is explored through the mesh's *dual graph*, i.e. the graph whose nodes correspond to mesh faces and in which any two faces are connected with a link if they share a mesh edge. For consistency, we will use *edge* and *vertex* to refer to the triangle mesh's elements and *node* / *link* for the dual graph's elements, to avoid confusion.



Critically, each *spanning tree* (i.e. a connected sub-graph that includes all nodes without links creating a cycle) of the dual graph corresponds to a possible unfolding of the mesh: one need to only choose an arbitrary start face (node) and rigidly transform it onto the plane, then use the shared mesh edges as *hinges* to unwrap the faces that are linked to it in the tree (see inset) until all faces are exhausted. Every mesh edge that is not used as a hinge is commonly referred to as a *cut* edge since these are the edges that should be physically cut to fabricate the object.

Up to an arbitrary rigid global transformation, this process ends with a rigid mapping associated to every mesh face $R_1, \ldots, R_m$ that unfolds it onto the plane. By extension, this also creates a correspondence between each mesh vertex v_i and its potentially multiple 2D images $u_{i,1}, \ldots, u_{i,n_i}$. (see the inset above, where the purple vertex has one copy in 2D in the first unfolding but two in the second).

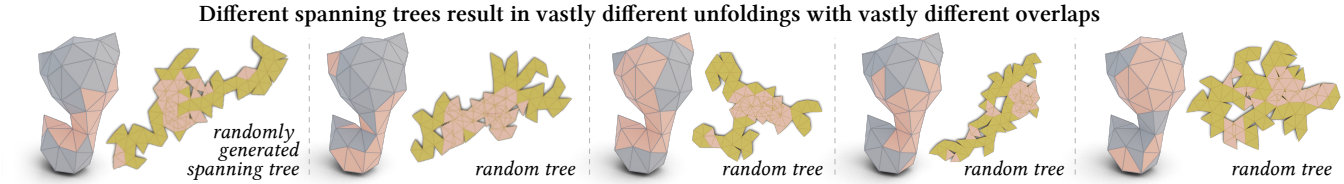


Figure 4: *The potential single-patch unfoldings of a given mesh can be represented by the spanning trees of its dual graph. The choice of spanning tree (here, random) can greatly affect the shape of the unfolding and the number of overlaps in it.*

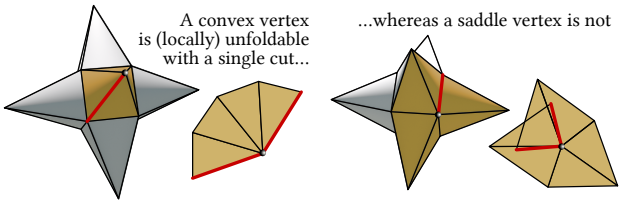
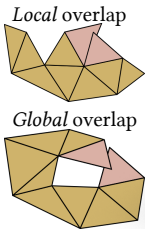


Figure 5: *A convex vertex unfolds in 2D without overlaps with one cut, but a saddle vertex's single cut always causes overlapping triangles.*

Unfortunately, for a general input mesh and a general spanning tree, the unfolding will contain a large number of overlaps (i.e. intersections among the unwrapped faces, highlighted in Fig. 4).

Sometimes, these overlaps can be explained *locally* by simple geometric reasons: for example, as explored by [TWS*11], any vertex neighborhood with non-zero angle deficit cannot be unfolded onto the plane without cuts (as shown in Fig. 5, locally convex or concave vertices will require at least one cut, whereas saddle-like ones will require at least two). Apart from local overlaps, an unfolding may also have *global* overlaps (see inset), which are significantly more challenging to identify as they require explicit checking for triangle overlaps.



Exploring the entire space of spanning trees (and thus of mesh unfoldings) to find an overlap-free one explodes in combinatorial complexity and rapidly becomes computationally intractable. Interestingly (for a mathematician) but disappointingly (for a manufacturer), an overlap-free unfolding may not exist even for relatively simple geometries (see Fig. 3). Instead of following the approach of previous works and softening the problem statement by allowing minimal overlaps or more than a single disconnected patch, we propose a paradigm shift and instead consider simplifying the *mesh geometry* to ensure a single-patch unfolding.

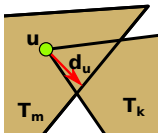
However, standard simplification methods (e.g., [GH97]) are not guaranteed to resolve or even reduce the unfolding's overlaps. Thus, we propose an overlap-aware strategy to simplify a mesh while ensuring that each step reduces the overlap.

4. Method: Mesh Simplification for Unfolding

In lieu of the difficulties stated in Sec. 3, we now update our problem statement to that of finding a *new* mesh $\Omega^\star$ similar to the input Ω^0 but having a single-patch non-overlapping unfolding $\Gamma^\star$. We start with an initial, potentially overlapping, unfolding Γ^0 (we will discuss initialization strategies in 4.4). We then iteratively alternate between reducing the overlaps, first *geometrically*, by manipulating the vertex positions of the mesh (Sec. 4.1) and then, *topologically*, through a purpose-built unfolding-aware mesh decimation strategy (Sec. 4.2). Once we achieve an overlap-free unfolding, we post-process the obtained mesh to ensure it is as close as possible to the input without introducing any new overlaps (Sec. 4.3). The effect of each individual step is showcased in Fig. 6, and the overall approach is summarized in Algorithm 1 in the Supplemental.

4.1. Unfolding-aware vertex manipulation

At a given iteration t, we call the current mesh Ω^t and its unfolding Γ^t. We first seek to remove small overlaps that can be handled by only displacing vertices without changing the topology of the mesh. Experimentally, we distinguish between three classes of overlaps in the unfolding Γ^t for which overlaps may be resolved geometrically by slightly repositioning its mesh vertices.



Case 1: In the unfolding Γ^t, if a vertex $u_{i,j}$ of a triangle T_k falls inside exactly one other, topologically non-adjacent triangle T_m (see inset) we find p, the closest point to $u_{i,j}$ just outside of the boundary of T_m and compute the vector d_u that would remove the overlap between triangles T_k and T_m in Γ^t. We then lift the vector d_u from the unfolding to the mesh Ω^t with the help of barycentric coordinates of u in triangle T_m to obtain d_v. We displace the three-dimensional vertex v_i with d_v thus moving its unfolding vertex u_i to p). While this process will resolve the given overlap, the effect of displacing v_i is not limited to $u_{i,j}$, but results in a displacement in every other planar image of v_i, $u_{i,s}$, $s \neq j$ which may result in new overlaps. If this were the case, we skip this operation.

Case 2: If we identify an overlap between two topologically adjacent triangles that share a common non-saddle vertex, we resolve this overlap by moving the corresponding 3D non-saddle vertex in the direction opposite to the vertex normal. This operation flattens the vertex locally in its 1-vertex neighborhood, thus reducing the curvature around that vertex. Since all triangles attached to this vertex contract, no new triangle overlaps are created. We contract the vertex just enough to resolve the overlap via adaptive stepping.

Case 3: Finally, we attempt to reduce the overlaps between all pairs of overlapping triangles that are only overlapping with one another, by following the gradient flow of a geometric energy

$$\Phi(v_1, \ldots, v_n) = w_c \Phi_c + w_s \Phi_s \tag{1}$$

where w_c and w_s are weights, Φ_c is a standard collision energy and

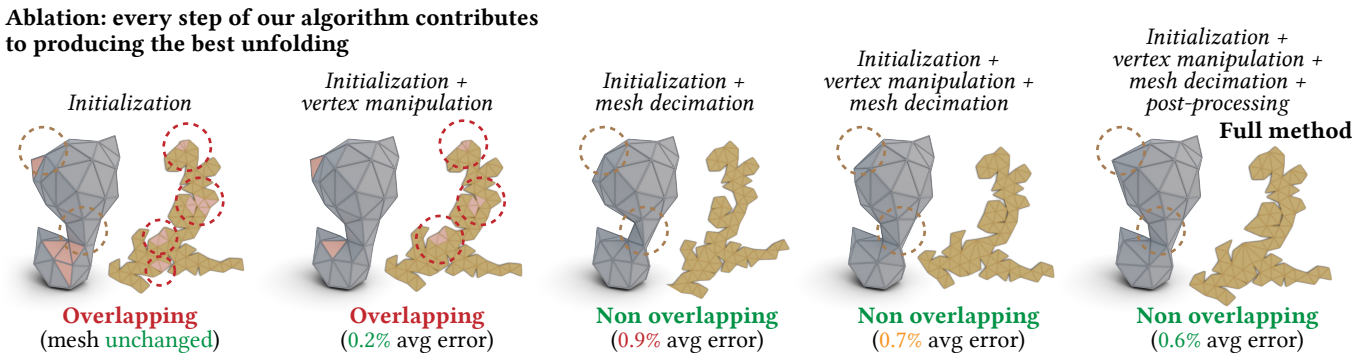


Figure 6: *Our algorithm is composed of several steps, all of which are crucial to obtaining a single patch unfolding that is non-overlapping (see first two subfigures) and differs as little as possible from the original mesh (see regions circled in brown).*

Φ_s is an elastic regularizer. Details on the computation of Φ_c and Φ_s can be found in Appendix B in the Supplemental.

4.2. Unfolding-aware mesh decimation

Likely, the vertex manipulations described in Sec. 4.1 do not remove all existing overlaps, forcing us to resort to more invasive mesh surgery. Inspired by the classic work of [GH97], we collapse mesh edges sequentially by drawing them from a priority queue. In our case, our queue is populated by all edges that participate in any overlaps, and their priority value is chosen heuristically as the integer number of other triangles each edge is overlapping.

Given that an edge collapse is a change in the dual graph (specifically, it eliminates two nodes and a maximum of five links, creating two new links), it may cause the spanning tree to cease being a tree (by no longer being connected) or cease being a spanning one (by no longer including every node), thus no longer corresponding to a valid mesh unfolding. Thus, a naive mesh decimation strategy would require recomputing a spanning tree (i.e. a new unfolding) after every collapse. This would be computationally prohibitive and, given the diversity in possible unfoldings, it would make our simplification algorithm highly unstable (see Fig. 8).

To avoid this, we instead propose preserving the spanning tree during the collapse. Fortunately (see Fig. 7), this can be done in constant complexity by looping over every node in the edge's one-ring, and adding as many of the links in it to the spanning tree as possible without introducing a cycle. At the end of this loop, we are guaranteed to still be in possession of a single-component spanning tree. If the number of overlaps caused by this tree's unfolding is equal or higher than before the collapse, we revert the collapse and proceed to the next element in the queue. As is common in remeshing algorithms, we similarly check if the collapse caused non-manifold vertices, self-intersections or flipped triangles, in which case we also revert it. Notably, this construction is independent of the placement of the vertex resulting from the collapse. In practice, we consider the midpoint as well as at the original edge's vertices, and choose whichever one results in the lowest number of overlaps.

We repeat this decimation process until the queue is exhausted or every edge in the queue is rejected, in which case we begin a new iteration of our algorithm, starting from the unfolding-aware vertex manipulation of Sec. 4.1. Pseudocode for our full unfolding-aware decimation strategy is provided in Algorithm 2 in the Supplemental.

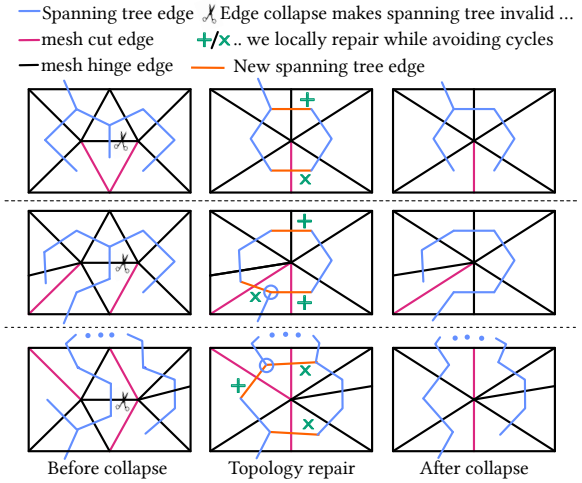


Figure 7: *Illustration of the spanning tree-preserving edge collapse operation on three examples, one per row.*

Exit strategy: The iterative approach described above can get *stuck* if every edge in the initial collapse queue is rejected (i.e. because its collapse results in degenerate features, which could cause us to skip over the decimation step altogether. Experimentally, we find our algorithm getting stuck only in later iterations after the vast majority of overlaps have been resolved but a small minority remain. In these cases, we rely on two heuristic exit strategies that ensure progress in the algorithm: *broadening our search* (by reinitializing the priority queue to include the neighbors of overlapping edges) and *lowering our expectations* (by choosing the collapse which causes the lowest number of new overlaps).

We repeat this outer iteration loop until we obtain a single-patch, non-overlapping unfolding (a *success* by our algorithm) or we exhaust a maximum number of iterations (in practice, we use 100) without finding one (a *failure*).

4.3. Post-processing

If successful, the iterative approach described in Sections 4.1 and 4.2 terminates with a mesh Ω and its single-patch, non-overlapping

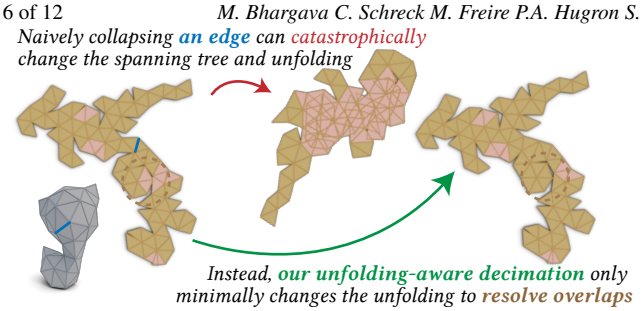


Figure 8: *Through the strategy described in the text, we repair the spanning tree after each edge collapse. This removes the need to recompute the unfolding after each collapse and ensures that the new unfolding is only minimally different from the pre-collapse one.*

unfolding Γ. However, in achieving this unfolding, we may have deviated unnecessarily from the input Ω^0. As a final step, we search to optimize the position of the vertices of Ω in order to reduce the distance between Ω and Ω^0 while avoiding creating new overlaps.

Experimentally, we find that optimizing the vertices of Ω while strictly avoiding overlaps is unnecessarily restrictive. Instead, we observe that more significant improvements in accuracy can be achieved by allowing for temporary, small overlaps in Γ to appear during the optimization and subsequently removing them. Because of this, we advocate for a post-processing strategy that *decouples* the mesh from its unfolding and instead links the two through a coupling energy term, simultaneously penalizing overlaps and deviation from Ω^0. We do this by considering v_i and u_i as separate variables, and iteratively optimizing the energy

$$E_p(\Omega,\Gamma) = \delta_d E_d(\Omega,\Omega^0) + \delta_d E_o(\Gamma) + \delta_c E_c(\Omega,\Gamma) \qquad (2)$$

where E_d is a distance energy between Ω and Ω^0, E_o measures the overlap in Γ, and E_c couples the two (see Appendix C in Supplemental for details on the construction of these energies). Note that this optimization step relies on an initial overlap-free unfolding. We optimize using a gradient descent. δ_d, δ_o, and δ_c are boolean terms that allow for scheduling the energies: we first optimize only E_d to bring Ω closer to Ω^0, and then alternate between optimizing E_o to remove overlaps in Γ, and E_c to ensure that the output remains a viable unfolding. If this is not achieved, we make the method output the last valid single-patch, non-overlapping unfolding (e.g., the output of Sec. 4.2).

4.4. Implementation details

We implemented our method in C++, using Libigl [JP*18] and CGAL [FP09] for common geometric processing subroutines. We report runtimes carried out on a Linux machine with 64GB memory and a i9-9900K 3.60GHz processor.

Initialization. Like any other optimization method, our approach is also sensitive to the initial (overlapping) unfolding: broadly, the fewer overlaps present in our initialization, the more likely our algorithm is to successfully terminate within a small distance of the input mesh. Thus, instead of opting for a random or heuristic based initial unfolding, we use two different search based methods: the unfolder proposed by [TWS*11] and the one proposed by [Zaw24].

On complex examples, Takahashi et. al.'s genetic algorithm may fail to find a single-patch unfolding, in which case it generates multiple patches of overlap-free unfolding. We merge these separate patches into a single (overlapping) patch by sorting them by size in decreasing order, experimenting with all the possible edges through which each patch can be merged and selecting the one that causes the lowest number of overlaps. On the other hand, Zawallich et al.'s Tabu-search based method uses an efficient data structure to find the least overlapping unfolding, making its output readily available for our method.

Our method remains agnostic to any specific initialization method and can seamlessly incorporate new initialization methods should future algorithms present improvements.

5. Results

Our algorithm is motivated by a very practical problem: that of unfolding a given 3D shape so that it can be fabricated from a planar material. Thus, beyond the theoretical observations made in Sec. 4, we first and foremost evaluate our algorithm practically, by justifying every algorithmic and parametric choice as well as its superior performance over previous works on a vast array of inputs.

In order to quantitatively evaluate these aspects of our method and others, we produce two datasets of meshes obtained by decimating the Thingi10K dataset [ZJ16] to 500 and 1000 faces, respectively. After pruning the dataset to remove non-manifold, self-intersecting, and multi-component meshes, we are left with *3049* meshes with 500 faces (which we refer to as our *coarse* dataset) and *2440* meshes with 1000 faces (our *fine* dataset).

The main metric we use to evaluate our method and others if it is *successful* or not is defined as whether the algorithm finds a single-patch, non-overlapping unfolding. On every success of our method, we also measure and report the time taken to find a solution, the number of faces in the output, as well as the quality of the final mesh approximation, given by the Hausdorff and Chamfer distances between the input and output meshes.

5.1. Experiments

Our algorithm is composed of several sequential steps. As we show in Fig. 6, in which we highlight overlapping triangles and use Chamfer distance as a proxy for the average approximation error, the combination of all of these steps balances the lack of overlap with the quality of the output's approximation. Like the unfoldable tetrahedron in Fig. 3, Fig. 6 is a didactic example. However, we also qualitatively showcase the proficiency of our method over several complex examples in Fig. 1 (using [TWS*11] as initializer) and Fig. 12 (with **accompanying video**), for which quantitative results are also reported in Table 2.

In particular, there are two algorithmic choices that we choose to evaluate in more detail: namely, the vertex manipulation in Sec. 4.1 and the heuristic placement of the collapsed vertex in Sec. 4.2 (see Table 3 in Supplemental), where we report the metrics of different variants of our method for our two datasets). As expected, we find that the vertex manipulation step consistently leads to better results, while a more careful choice of the collapsed vertex generally leads to a lower approximation error.

Table 1: *When ran on large-scale datasets generated from Thingi10K [ZJ16], our algorithm significantly increases the success rate of the methods by [TWS*11] (Ta.) and [Zaw24] (Za.) at finding a simple, single-patch unfolding. Additionally, we use Haussdorf and Chamfer distance as error metrics to report maximum and average shape differences before and after our algorithm is applied and report the median value of these two metrics over the entire dataset.*

Dataset	Success (Ta.)	Runtime (Ta.)	Success (Ta. + **Ours**)	Runtime (**Ours**)	Dist. Metric	Success (Za.)	Runtime (Za.)	Success (Za. + **Ours**)	Runtime (**Ours**)	Dist. Metric
coarse	40.24%	24.5 s	**92.42%**	2.77 s	0.01,0.001	85.30%	9.8 s	**91.96%**	11.69 s	0.02,0.002
fine	10.55%	111.7 s	**84.10%**	18.64 s	0.03,0.005	62.99%	30.7 s	**87.93%**	14.52 s	0.01,0.001

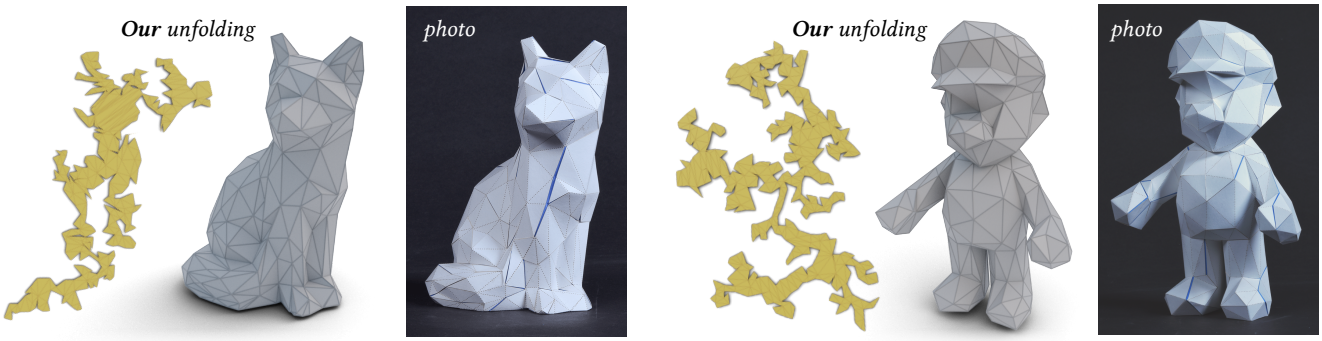


Figure 9: *We carry out the clearest validation of our algorithm by utilizing it to generate physical models of a fox and an Italian plumber from pieces of cut paper. Both shapes were originally non-unfoldable by [TWS*11].*

5.2. Comparisons

By relaxing the problem and allowing for minimal changes in the input geometry, we dramatically increase the success rate of previous works, as we show quantitatively in Table 1 and qualitatively in Fig. 2 (the bar chart corresponding to our *fine* dataset).

As shown in Table 1, on our *coarse* dataset, the algorithm by [TWS*11] successfully unfolds 40.24% of the meshes; when using their attempts as initializers for our method, we more than double that success rate into 92.42%. The effect is even more pronounced on the *fine* dataset, where the success rate increases eight-fold, from 10.55% to 84.10%. Our algorithm even outperforms the concurrent work by [ZP24] where they use their unfolder [Zaw24], which achieves a success rate of 85.30% on the *coarse* dataset (compared to our 91.96%) and 62.99% on the *fine* one (compared to our 87.93%).

Our unfolding-aware decimation strategy is crucial. To validate this, we performed a further test on the subset of meshes which are unfolded by our method but not by any of the related works. If the meshes are decimated using standard techniques (e.g., *qslim* [GH97]) to the same refinement level, they are most often still non-unfoldable. For roughly 45% of meshes in our *coarse* subset dataset and 55% of the *fine* subset dataset both [TWS*11] and [Zaw24] failed.

5.3. Applications & Generalizations

Our method can be readily applied to common manufacturing frameworks, as we showcase with our fabricated results in Fig. 9. For this figure, initially unfoldable input meshes of a fox and an Italian plumber are simplified by our algorithm and then fabricated from paper, with the aid of a 3D printed guide.

The mesh simplification carried out by our method may be undesirable in some fabrication settings: for example, when one wishes to preserve certain features because of their functionality. Our algorithm can be easily modified to incorporate these fixed points (by simply not allowing them to be collapsed or displaced), as we showcase by simulating a fabricated keychain in Fig. 10.

6. Conclusions & Future Work

We have introduced a computational approach for finding an approximate mesh of a given 3D shape that admits a single-patch, non-overlapping unfolding. We propose doing this via a custom mesh processing algorithm that combines unfolding-aware vertex manipulation and mesh decimation to iteratively modify a mesh with an overlapping unfolding into a mesh with a non-overlapping one. As we have shown, our approach can be used to dramatically increase the range of shapes that can be successfully unfolded in a single patch, making it particularly promising for applications in fabrication and design.

Despite its improvement on previous approaches, our method still occasionally fails to find a valid unfolding for certain inputs, instead getting 'stuck' at a stage without valid edge collapses or vertex displacements. In these cases, one may consider discarding the current spanning tree and starting over with a newly computed one. In Table 4 (Supplemental), we experiment with a reinitialization strategy that does just that and shows that it can improve the success rate of our algorithm. Nonetheless, this comes at a significant cost, as re-computing a high-quality spanning tree is computationally expensive. Instead, future work may consider less aggressive, more efficient reinitialization strategies, or even ways of avoiding getting stuck in the first place.

The most promising avenues for future work relate to making our method even more directly applicable to fabrication settings. For example: our algorithm is aimed at producing single-patch unfoldings. However, for large numbers of triangles, this can be impractical, and future work may consider allowing a user to specify a particular number of patches. Similarly, fabricating objects with

Table 2: *Quality metrics of our method on a gallery of shapes from the Stanford 3D Scanning Repository with different initialisation methods.*

Mesh name	Input vertex and face count	Init. method and overlap count	Output vertex and face count	Hausdorff distance	Chamfer distance	Runtime
Armadillo	602, 1200	[TWS*11], 56	584, 1164	0.040	0.0013	2.38 s
Armadillo	602, 1200	[Zaw24], 5	598, 1192	0.023	0.0004	2.53 s
Bunny	502, 1000	[TWS*11], 34	480, 956	0.0098	0.00103	6.78 s
Bunny	502, 1000	[Zaw24], 7	501, 998	0.0016	0.00008	0.66 s
Bishop	250, 496	[TWS*11], 4	246, 488	0.011	0.00078	0.64 s
Bishop	250, 496	[Zaw24], 15	247, 490	0.011	0.00073	1.32 s
Nefertiti	502, 1000	[TWS*11], 58	467, 930	0.017	0.0011	11.85 s
Nefertiti	502, 1000	[Zaw24], 44	477, 950	0.011	0.0008	10.20 s
Plane	602, 1200	[TWS*11], 347	495, 986	0.017	0.0016	27.53 s
Plane	602, 1200	[Zaw24], 16	537, 1070	0.008	0.0011	55.57 s

paper can lead to gluing a lot of edges even if it is a single component. To ease this task, one can integrate the glue tabs proposed by [TWS*11] and [KTGW20] on our output.

Furthermore, our method could be extended to enforce user-defined constraints. We showcase one such use case in Fig. 10 where the keyhole of the keychain is kept unchanged during the algorithmic process. We envision that other user-defined constraints like symmetry can be incorporated likewise.

From a more general perspective, we hope that the lessons in this work can serve to develop new unfolding algorithms that are combined with mesh processing techniques: for example, by ensuring that overlaps occur in regions of smooth curvature where local decimation would not affect the overall mesh quality too much (See Fig. 11), or in convex regions on which overlaps are easier to resolve. Similarly, we believe that more sophisticated edge collapse strategies (for example, those that consider the consequences of several sequential collapses at a time instead of a single one) may improve the robustness of unfolding-aware decimation algorithms like ours.

Finally, our algorithm sits in a more general class of methods that *relax* known-to-be-hard problems into easier, approximate ones that are equally valid for the needs of a given application. We believe fabrication and computational design are particularly promising application realms for this kind of approach, and hope our work can inspire others to explore similar strategies.

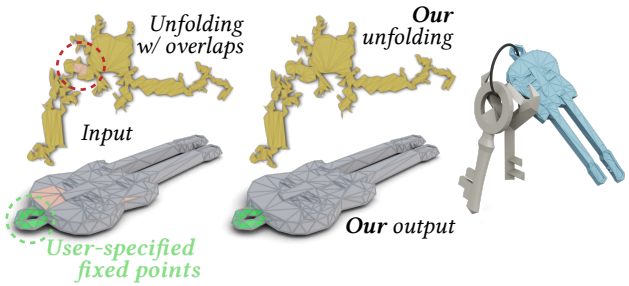


Figure 10: *Fixed-point constraints can be easily incorporated into our algorithm; for example, to ensure that this guitar-shaped keychain is functional.*

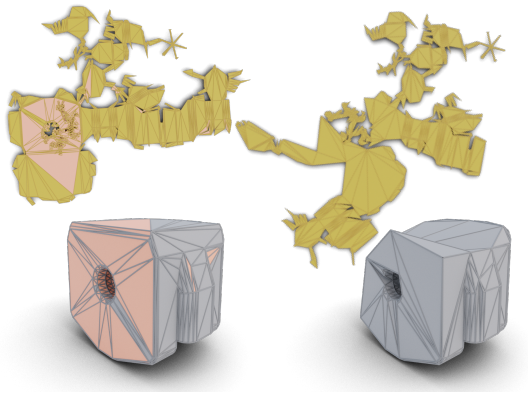


Figure 11: *Example of the worst modified mesh by our algorithm. Initialized using [TWS*11], the algorithm originally had 361 overlapping faces (in red), which were fully removed at the cost of Hausdorff and Chamfer distances of 0.18 and 0.035, respectively.*

7. Acknowledgements

Researchers from INRIA received support from the DORNELL Inria Challenge. Silvia Sellán acknowledges support from NSERC Vanier Doctoral Scholarship and an MIT SoE Postdoctoral Fellowship for Engineering Excellence.

8. Conflict of Interest and Data Availability

The Authors declare no conflict of interest. The code and data for this paper is available at git.ista.ac.at/mbhargav/mesh-simplification-for-unfolding.

References

[AAOS97] AGARWAL P., ARONOV B., O'ROURKE J., SCHEVON C.: Star unfolding of a polytope with applications. *SIAM Journal on Computing 26*, 6 (Dec. 1997), 1689–1713. doi:10.1137/S0097539793253371. 2, 3

[Bel20] BELKE C. H.: From modular origami robots to polygon-based modular systems: a new paradigm in reconfigurable robotics. 129. URL: http://infoscience.epfl.ch/record/276481, doi:https://doi.org/10.5075/epfl-thesis-7300. 2

[CZ18] CALLENS S. J., ZADPOOR A. A.: From flat sheets to curved geometries: Origami and kirigami approaches. *Materials Today 21*, 3 (2018), 241–264. 2

[DD02] DEMAINE E. D., DEMAINE M. L.: Recent results in computational origami. In *Origami3: Third International Meeting of Origami Science, Mathematics and Education* (2002), pp. 3–16. 2

[DDE20] DEMAINE E. D., DEMAINE M. L., EPPSTEIN D.: Acutely triangulated, stacked, and very ununfoldable polyhedra. *arXiv preprint arXiv:2007.14525* (2020). 2, 3

[DO07] DEMAINE E. D., O'ROURKE J.: *Geometric Folding Algorithms: Linkages, Origami, Polyhedra*. Cambridge University Press, Cambridge ; New York, 2007. 2

[DT17] DEMAINE E., TACHI T.: Origamizer: A practical algorithm for folding any polyhedron. 2

[FBS*23] FREIRE M., BHARGAVA M., SCHRECK C., HUGRON P.-A., BICKEL B., LEFEBVRE S.: Pcbend: Light up your 3d shapes with foldable circuit boards. *ACM Trans. Graph. 42*, 4 (jul 2023). URL: https://doi.org/10.1145/3592411, doi:10.1145/3592411. 2

[FP09] FABRI A., PION S.: Cgal: The computational geometry algorithms library. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems* (2009), pp. 538–539. 6

[FSZ*21] FU X.-M., SU J.-P., ZHAO Z.-Y., FANG Q., YE C., LIU L.: Inversion-free geometric mapping construction: A survey. *Computational Visual Media 7* (2021), 289–318. 2

[FTS*13] FELTON S. M., TOLLEY M. T., SHIN B., ONAL C. D., DEMAINE E. D., RUS D., WOOD R. J.: Self-folding with shape memory composites. *Soft Matter 9*, 32 (2013), 7688–7694. 2

[FW22] FARGION G., WEBER O.: Globally injective flattening via a reduced harmonic subspace. *ACM Trans. Graph. 41*, 6 (2022), 1–17. 2

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., p. 209–216. URL: https://doi.org/10.1145/258734.258849, doi:10.1145/258734.258849. 4, 5, 7

[JP*18] JACOBSON A., PANOZZO D., ET AL.: libigl: A simple C++ geometry processing library, 2018. https://libigl.github.io/. 6

[JSP17] JIANG Z., SCHAEFER S., PANOZZO D.: Simplicial complex augmentation framework for bijective maps. *ACM Transactions on Graphics 36*, 6 (2017). 2

[KTGW20] KORPITSCH T., TAKAHASHI S., GRÖLLER E., WU H.-Y.: Simulated annealing to unfold 3d meshes and assign glue tabs. 3, 8

[Law11] LAWRENCE S.: Developable surfaces: their history and application. *Nexus Network Journal 13* (2011), 701–714. 2

[LKK*18] LI M., KAUFMAN D. M., KIM V. G., SOLOMON J., SHEFFER A.: Optcuts: Joint optimization of surface cuts and parameterization. *ACM Transactions on Graphics (TOG) 37*, 6 (2018), 1–13. 2

[LPRM23] LÉVY B., PETITJEAN S., RAY N., MAILLOT J.: Least squares conformal maps for automatic texture atlas generation. In *Sem. Graph. Papers: Pushing the Boundaries, Vol 2*. 2023, pp. 193–202. 2

[LZX*08] LIU L., ZHANG L., XU Y., GOTSMAN C., GORTLER S. J.: A local/global approach to mesh parameterization. In *Computer graphics forum* (2008), vol. 27, Wiley Online Library, pp. 1495–1504. 2

[NYNM23] NIU L., YANG X., NISSER M., MUELLER S.: Pullupstructs: Digital fabrication for folding structures via pull-up nets. In *Proceedings of the Seventeenth International Conference on Tangible, Embedded, and Embodied Interaction* (2023), pp. 1–6. 2

[O'R19] O'ROURKE J.: Unfolding polyhedra. *arXiv preprint arXiv:1908.07152* (2019). 2

[Pol09] POLTHIER K.: Imaging maths - unfolding polyhedra. 2

[PTH*17] PORANNE R., TARINI M., HUBER S., PANOZZO D., SORKINE-HORNUNG O.: Autocuts: simultaneous distortion and cut optimization for uv mapping. *ACM Transactions on Graphics (TOG) 36*, 6 (2017), 1–11. 2

[SAJ20] SELLÁN S., AIGERMAN N., JACOBSON A.: Developability of heightfields via rank minimization. *ACM Trans. Graph. 39*, 4 (2020), 109. 2

[SB12] SIFAKIS E., BARBIC J.: Fem simulation of 3d deformable solids: a practitioner's guide to theory, discretization and model reduction. In *ACM SIGGRAPH 2012 Courses* (New York, NY, USA, 2012), SIGGRAPH '12, Association for Computing Machinery. URL: https://doi.org/10.1145/2343483.2343501, doi:10.1145/2343483.2343501. 11

[SC17] SAWHNEY R., CRANE K.: Boundary first flattening. *ACM Transactions on Graphics (ToG) 37*, 1 (2017), 1–14. 2

[Sch89] SCHEVON C. A.: Algorithms for geodesics on convex polytopes, 1989. 2

[Sch97] SCHLICKENRIEDER W.: Nets of polyhedra. *Unpublished. Technische Universitat Berlin* (1997). 2

[SGC18] STEIN O., GRINSPUN E., CRANE K.: Developability of triangle meshes. *ACM Trans. Graph. (TOG) 37*, 4 (2018), 1–14. 2

[SP11] STRAUB R., PRAUTZSCH H.: *Creating optimized cut-out sheets for paper models from meshes*. KIT, Fakultät für Informatik, 2011. 3

[SS15] SMITH J., SCHAEFER S.: Bijective parameterization with free boundaries. *ACM Transactions on Graphics (TOG) 34*, 4 (2015), 1–9. 2

[SSS*17] SCHULZ A., SUNG C., SPIELBERG A., ZHAO W., CHENG R., GRINSPUN E., RUS D., MATUSIK W.: Interactive robogami: An end-to-end system for design of robots with ground locomotion. *The International Journal of Robotics Research 36*, 10 (2017), 1131–1147. 2

[SYLF20] SU J.-P., YE C., LIU L., FU X.-M.: Efficient bijective parameterizations. *ACM Trans. Graph. (TOG) 39*, 4 (2020), 111–1. 2

[Tac09] TACHI T.: Origamizing polyhedral surfaces. *IEEE transactions on visualization and computer graphics 16*, 2 (2009), 298–311. 2

[TWS*11] TAKAHASHI S., WU H.-Y., SAW S. H., LIN C.-C., YEN H.-C.: Optimized topological surgery for unfolding 3d meshes. In *Computer graphics forum* (2011), vol. 30, Wiley Online Library, pp. 2077–2086. 3, 4, 6, 7, 8, 10

[VW08] VAN WIJK J. J.: Unfolding the earth: myriahedral projections. *The Cartographic Journal 45*, 1 (2008), 32–42. 2

[WPF10] WEI M.-Q., PANG M.-Y., FAN C.-L.: Survey on planar parameterization of triangular meshes. In *2010 International Conference on Measuring Technology and Mechatronics Automation* (2010), vol. 3, IEEE, pp. 702–705. 2

[YBCP19] YAO M., BELKE C. H., CUI H., PAIK J.: A reconfiguration strategy for modular robots using origami folding. *The International Journal of Robotics Research 38*, 1 (2019), 73–89. 2

[YCS23] YUAN C., CAO N., SHI Y.: A survey of developable surfaces: From shape modeling to manufacturing. *arXiv preprint arXiv:2304.09587* (2023). 2

[Zaw24] ZAWALLICH L.: Unfolding polyhedra via tabu search. *The Visual Computer* (2024), 1–14. 3, 6, 7, 8

[ZJ16] ZHOU Q., JACOBSON A.: Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797* (2016). 6, 7

[ZP24] ZAWALLICH L., PAJAROLA R.: Unfolding via mesh approximation using surface flows. In *COMPUTER GRAPHICS forum* (2024), vol. 43. 3, 7, 10

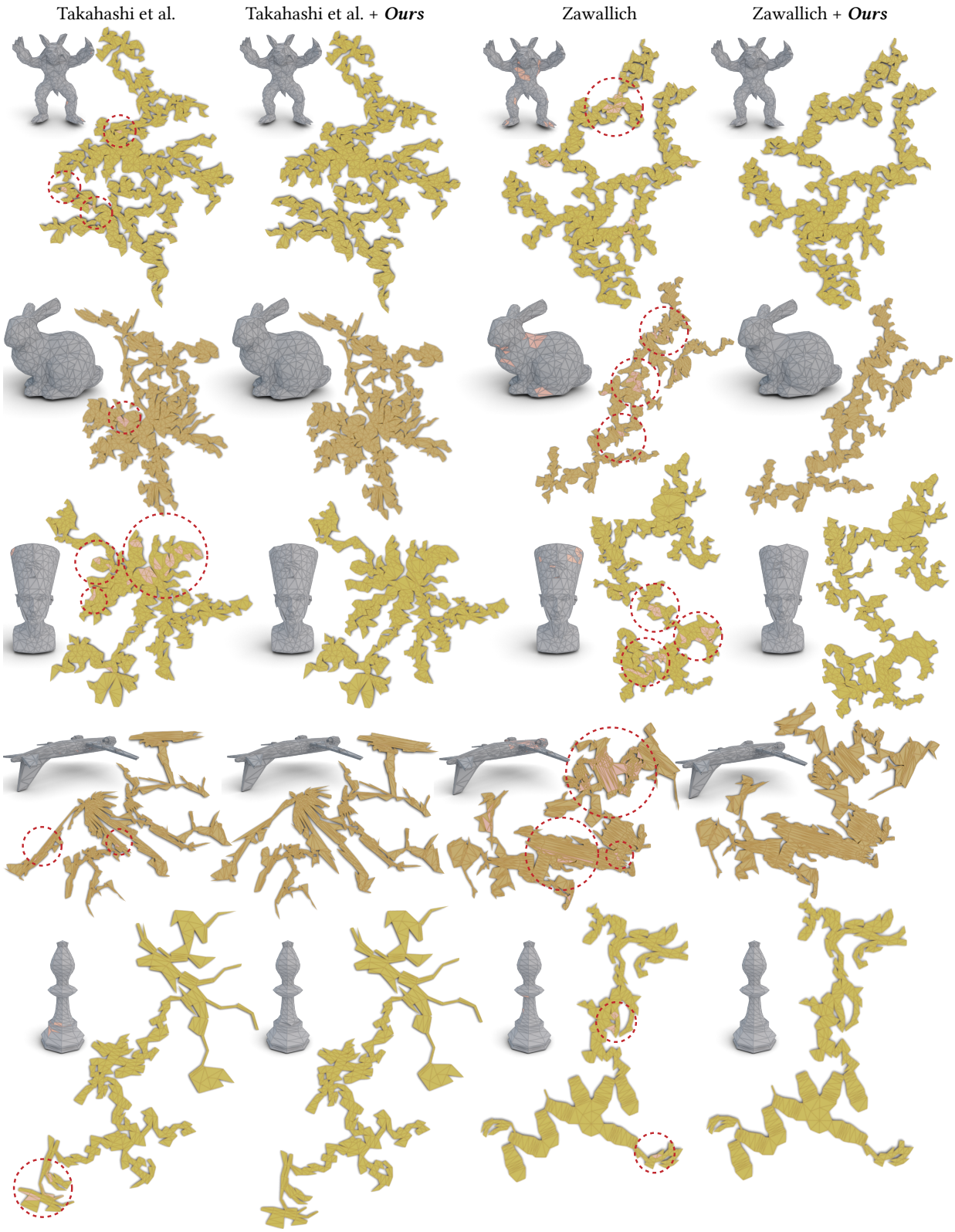


Figure 12: *For all relatively simple meshes in this gallery, the methods by [TWS*11] and [ZP24] fail to find a single-patch, non-overlapping unfolding. Our algorithm minimally modifies the mesh to accomplish this in every case.* ***See accompanying video*** *and quantitative results in Table 2*

Appendix A: Pseudocode

This appendix provides pseudocode for the main steps of our algorithm, to aid in their implementation. 1 shows the main outer loop of our method, while 2 covers the unfolding-aware decimation strategy in Sec. 4.2 step by step.

Algorithm 1 Main Algorithm

```
Input: Manifold mesh (Ω^0)
Output: Manifold mesh (Ω^⋆) and a simple unfolding (Γ^⋆)
Generate the initial unfolding (Γ^0) (Section 4.4)
iter ← 0
numOverlaps = computeOverlaps(Γ^0)
while numOverlaps > 0 or AND iter < maxIter do
    Unfolding-aware vertex manipulation. (Section 4.1)
    Unfolding-aware mesh decimation. (Section 4.2)
    iter ← iter + 1 (Section 4)
    numOverlaps = computeOverlaps(Γ^i)
Post-process the mesh. (Section 4.3)
```

Algorithm 2 Spanning tree-preserving edge collapse

```
Generate the priority queue for edges to collapse
while priority queue != empty do
    edge ← priorityQueue.pop()
    collapse(edge) as explained in section 4.2
    if edge collapse generates degenerate features then
        Revert
    compute unfolding from the spanning tree
    count overlapping triangles in the unfolding
    if overlap count not decreased. then
        Revert
    else
        confirm edge collapse.
    if (priority queue == empty) then
        if exit-strategy == broad.-our-search and iter > 1 then
            Add 1-vertex ring edges of overlapping edges as well.
        if exit-strategy == lowering-our-expectations then
            remove the Edge that causes least new overlaps in 2D.
end
```

Appendix B: Surface flow for resolving overlaps

For each pair of overlapping triangles, we compute the projected vertex u_{proj} (see Figure 13). The projection points u_{proj} are then lifted in 3D using barycentric coordinates to obtain v_{proj}. We define

$$\Phi_c = \frac{1}{2}\sum_{k=1}^{n} ||v_k - v_{k,proj}||^2.$$

Furthermore, to ensure that triangles do not flip or drastically change their shape, we add regularising energy Φ_s that helps maintain the shape of the triangles:

$$\Phi_s = \frac{1}{2}\sum_{h=1}^{m} A_h E_h : E_h$$

where A_h is the area of triangle h from Γ^t and E_h is the Green strain tensor of the deformation between the unfolding Γ^t and the current 3D mesh. The Green strain E is discretized by assuming a constant strain over each triangle. More details on the computation and differentiation of E can be found in the course by [SB12]. We perform a gradient descent to optimize the energy $\Phi = w_c\Phi_c + w_s\Phi_s$ over the vertices v_k of the 3D mesh. At each iteration of the surface flow, we displace each vertex v_k of $d_k = \delta t \frac{\partial \Phi(t)}{\partial v_k}$. We use 100 iterations and set $w_c = 1$ and $w_s = 10$ with a step $\delta t = 0.01$ for all our experiments.

Appendix C: Post-processing details

Our post-processing energy is composed of three terms. The *distance* term takes the form

$$E_d(\Omega,\Omega^0) = \frac{1}{2}\sum_{v_i\in\Omega} \|v_i - \text{Proj}(v_i,\Omega^0)\|^2 + \sum_{w_j\in\Omega^0} \|w_j - \text{Proj}(w_j,\Omega)\|^2, \tag{3}$$

which is linearized by making $p_i = \text{Proj}(v_i,\Omega^0)$ (ignoring the dependency with v_i) and $\text{Proj}(w_j,\Omega_t) = \mathbf{B}_j\mathbf{w}$, where $\mathbf{B}_j$ contains the barycentric coordinates of the projection on Ω_t and $\mathbf{w}$ is a matrix vertically concatenating the coordinates of every vertex in Ω_t. The *overlap* term is defined as

$$E_o(\Gamma) = E_1(\Gamma) + E_2(\Gamma), \tag{4}$$

where E_1 is the 2D collision energy

$$E_1(\Gamma) = \frac{1}{2}\sum_{i=1}^{n} ||u_i - u_{i,proj}||^2 \tag{5}$$

and E_2 is a 2D regularizer

$$E_2(\Gamma) = \frac{1}{2}\sum_{j=1}^{m} A_j E_j^{2d} : E_j^{2d}, \tag{6}$$

where the strain tensor E^{2d} is computed for the deformation between the reference state Γ and the current state during the gradient descent. Finally, the *coupling* term is given by

$$E_c(\Omega,\Gamma) = \frac{1}{2}\sum_{j=1}^{m} A_j E_j^{2d3d} : E_j^{2d3d}, \tag{7}$$

where E^{2d3d} is the Green strain of the deformation from Γ to Ω. These terms are combined into the post-processing energy $E_p(\Omega,\Gamma)$ as described in Sec. 4.3.

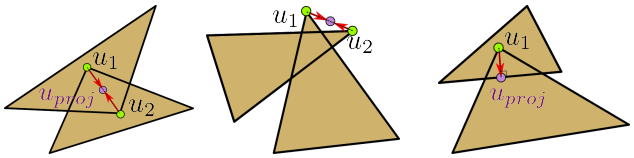


Figure 13: *We showcase the cases of overlapping triangles that we attempt to resolve by our vertex repositioning approach. The figure contains the target projected vertex that could potentially resolve the overlap.*

Table 3: *Our algorithm is most successful when incorporating the vertex repositioning strategy described in Sec. 4.1. On the other hand, the impact of the edge collapse heuristic described in Sec. 4.2 depends on the chosen initialization strategy. In all cases, our post-processing significantly improves mesh quality.*

Dataset	Initialization	'Smarter' edge collapse	Vertex Repositioning	Success	Runtime	Chmf error	Post-process runtime	Post-process improvement
coarse	Takahashi	No	No	92.06%	1.11 s	0.0017	0.37 s	6.91%
coarse	Takahashi	No	Yes	91.64%	3.14 s	0.0017	0.39 s	7.15%
coarse	Takahashi	Yes	No	91.70%	3.89 s	0.0015	0.36 s	7.05%
coarse	Takahashi	**Yes**	**Yes**	**92.42%**	**2.77 s**	**0.0015**	**0.36 s**	**4.11%**
fine	Takahashi	No	No	83.11%	13.80 s	0.0018	0.93 s	5.11%
fine	Takahashi	No	Yes	83.57%	25.35 s	0.0018	0.92 s	5.03%
fine	Takahashi	Yes	No	83.89%	12.96 s	0.0018	0.93 s	2.36%
fine	Takahashi	**Yes**	**Yes**	**84.10%**	**18.64 s**	**0.0015**	**0.76 s**	**3.31%**
coarse	Zawalich	No	No	91.87%	5.33 s	0.0050	0.30 s	1.57%
coarse	Zawalich	**No**	**Yes**	**91.96%**	**11.69 s**	**0.0050**	**0.28 s**	**3.26%**
coarse	Zawalich	Yes	No	91.18%	5.39 s	0.0048	0.29 s	3.26%
coarse	Zawalich	Yes	Yes	91.21%	8.15 s	0.0045	0.29 s	1.73%
fine	Zawalich	No	No	87.56%	7.52 s	0.0012	0.77 s	7.26%
fine	Zawalich	**No**	**Yes**	**87.93%**	**14.52 s**	**0.0011**	**0.76 s**	**0.87%**
fine	Zawalich	Yes	No	86.95%	5.85 s	0.0011	0.76 s	1.76%
fine	Zawalich	Yes	Yes	87.28%	9.47 s	0.0011	0.77 s	1.92%

Table 4: *Occassionally, our algorithm fails to find a solution. In such cases, we discard the current spanning tree and instead re-run our entire pipeline, treating the last iteration's mesh as the new input. This strategy results in a higher success rate, but at the cost of increased runtime.*

Dataset	Initialization	Success (without re-initialization)	Success (with re-initialization)	Time spent in our algorithm	Time spent in initializers
coarse	Takahashi	92.42%	**99.02%**	135.65 s	82.29 s
coarse	Zawalich	91.96%	**97.44%**	141.00 s	15.82 s
fine	Takahashi	84.10%	**95.33%**	660.68 s	367.58 s
fine	Zawalich	87.93%	**91.39%**	423.43s	114.34 s

Appendix D: Additional Results

We extensively evaluated our method on our two datasets with experiments beyond those contained in the main paper, which we provide in this section. Table 3 shows the effects of two specific algorithmic choices: namely, the use of the vertex repositioning strategy and the edge collapse heuristic (see discussion in Sec. 5.2). Table 4 demonstrates the impact of re-initializing our algorithm when it fails to find a solution (see discussion in Sec. 6).