DOI: 10.1111/cgf.15031
EUROGRAPHICS 2024 / A. Bermano and E. Kalogerakis
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 43 (2024), Number 2

Unfolding via Mesh Approximation using Surface Flows

Lars Zawallich
University of Zurich, Department of Informatics,
Switzerland
LarsZawallich@gmail.com

Renato Pajarola
University of Zurich, Department of Informatics,
Switzerland
pajarola@ifi.uzh.ch

Abstract

Manufacturing a 3D object by folding from a 2D material is typically done in four steps: 3D surface approximation, unfolding the surface into a plane, printing and cutting the outline of the unfolded shape, and refolding it to a 3D object. Usually, these steps are treated separately from each other. In this work we jointly address the first two pipeline steps by allowing the 3D representation to smoothly change while unfolding. This way, we increase the chances to overcome possible unfoldability issues. To join the two pipeline steps, our work proposes and combines different surface flows with a Tabu Unfolder. We empirically investigate the effects that different surface flows have on the performance as well as on the quality of the unfoldings. Additionally, we demonstrate the ability to solve cases by approximation which comparable algorithms either have to segment or can not solve at all.

CCS Concepts

• ***Computing methodologies*** → ***Mesh geometry models;*** • ***Applied computing*** → *Computer-aided manufacturing;*

1. Introduction

Unfolding and folding polyhedra is an alternative manufacturing process to create a physical object from piecewise planar elements. A typical manufacturing pipeline based on this principle can be described by the following steps [GBKK98]:

1. Approximate a 3D object with a discrete surface.
2. Unfold the approximation.
3. Cut out the unfolding.
4. Refold it (possibly automated).

It is important to note, that the planar unfolded object needs to be overlap-free to cut it from a 2D material, and it is of great efficiency advantage, if it is also a single-patched unfolding [DO07, Section 22.1.1].

Typically, the steps of the above-mentioned pipeline have been treated independently. To the best of our knowledge, there exists no holistic approach yet, which takes into account the effects of one step onto the other. Our approach provides key insights and a novel method towards treating these steps in an integrated way.

A commonly used technique to unfold polyhedra is edge unfolding (see Section 3). Even though some non-convex polyhedra can not be unfolded using edge unfolding, see also Table 1, the technique still works for many such polyhedra. In fact, there is yet no known general way to check if a polyhedron is edge-unfoldable or not. The only known measures determine local unfoldability (e.g. around a vertex), but cannot be generalized to a global unfoldability statement. Therefore, it is impossible to approximate a given 3D shape with unfoldability as a constraint. Thus, there is no guarantee, that the approximation of the first pipeline step is unfoldable. Approaches treating each pipeline step individually, either ignore this issue, or try to solve it solely within the unfolding step. A common solution for the issue is to segment the unfolding into parts, which are individually unfoldable. As mentioned above, however, a single-patched unfolding is highly desirable.

The main contribution of our work is to explore one possibility of joining the first two steps of the above mentioned pipeline. To prevent confusion, we will call the result of the approximation in Step 1 *representation* and approximations of this representation created by our method *approximation*.

In this paper we want to take a different approach than previous works did. Instead of fixing the representation and then trying to solve the unfoldability issue only within the unfolding step, we allow changes to the representation while unfolding. This is done by transforming the input mesh into an “easier to unfold” mesh, then unfolding this mesh and transforming it back into the original shape, while maintaining the unfolding overlap-free. If the initial representation can not be unfolded, our approach will stop the back-transformation at some stage and return the last intermediate transformed mesh which it found an unfolding for with its corresponding unfolding. This intermediate mesh can then be seen as an unfoldable approximation of the initial input mesh. We argue that since the initial representation is already an approximation, a slightly different approximation can be acceptable within most applications.

2. Related Work

Within the papercraft community, there are more approaches than just (un)folding polyhedra, to create a 3D shape. Classical origami only allows creasing and forbids any sort of cuts (e.g. [HHL19]). Results typically do not have an interior by design. In contrast, our approach aims to create 2D surfaces in 3D space, which have a hollow interior.

A variation of classical origami are the so-called tucking molecules [Tac09,DT17]. They also work with creases and without cuts, but aim to recreate a given discrete surface with hollow interior space. The faces of the surface are laid out in a 2D space and then filled with tucking molecules. These molecules are designed such that they disappear into the interior of the shape when folded. This approach needs a lot of extra material and the folding is a very challenging task. Compared to tucking molecules, our work does not need any filling material and the folds are less complex.

A different way of creating papercraft models are developable surfaces (e.g. [MS04, STL06, SGC18]). The core idea is to create patches which have a Gaussian curvature of zero at every point, but still approximate the input surface. This technique can be considered approximative, while having the "unfolding" in mind. Here, the unfolding is a set of disconnected planar patches, which each needs to be bent in a certain way. The bent patches are then connected together, to approximate the original surface. In contrast to developable surfaces, we are aiming to unfold a given shape using creases instead of bends and without segmentation.

The foundation for the field of parameterization has been laid by Fáry [F́48], with his proof that if a finite graph can be represented in a plane, it can also be represented with straight lines. In later works, different aspects of parameterization have been studied. E.g., the approach by Hormann and Greiner [HG00] allowed for a non-fixed boundary. Sheffer et al. [SLMB05] work in angle space and only in the very end produce a 2D representation of a given mesh. Kharevych et al. [KSS06] use cone singularities to allow parameterizing meshes of arbitrary topology. An overview of the field of parameterization has been given by Hormann et. al [HLS07]. While parameterization approaches try to minimize distortion, unfoldings do not allow any. Thus, the unfolding of a mesh can also be seen as a distortionless parameterization with non-convex boundary.

All approaches described below are using edge unfolding as an underlying technique, similar to our approach.

Straub et al. [SP11] explored different heuristics to create cut-trees. These heuristics aim to calculate a weight for each edge in the mesh to then apply classical minimal-spanning-tree algorithms. Similar in spirit, Haenselmann et al. [HE12] explored different heuristics to create unfold-trees. Here, the heuristics aim to calculate a weight for each dual-edge. However, both approaches use segmentation to overcome remaining overlaps in their unfoldings.

Xi et al. [XKKL16] proposed a way to use overlaps to segment their mesh. By repeatedly unfolding a mesh, they are able to segment different parts, which internally rarely overlap. On top of that, they also considered continuous foldability – the question if the unfolding self-intersects while folding it.

Kim et al. [KXL17] approximate a given mesh by so-called disjoint convex shells. These shells are designed to be easy-to-unfold, while still approximating the initial mesh very well. Especially, since these shells are only approximating a part of the mesh each, they are a better approximation, than a convex hull.

In another work, Xi et al. [XL17] developed an approach to make polyhedra easier to unfold. The core idea is to reduce local concavities by inflating the mesh. Afterwards, the inflated mesh gets segmented, to reduce global concavities. From all the related work, this approach can be considered closest to ours, since our approach works with a deformation to make the unfolding easier as well. A major difference is the transformation technique and our goal to return a single-patched unfolding.

An approach aiming to create a net was presented by Takahashi et al. [TWS*11]. Their idea was to lay out a highly segmented mesh into the 2D plane and to iteratively merge them based on a genetic algorithm, to eventually obtain a single-patched unfolding. If their approach fails to find such an unfolding, it will return a segmented result, which our method avoids.

Another approach aiming to create a net was designed by Korpitsch et al. [KTGW20]. They applied simulated annealing to find an overlap-free unfolding, while also considering glue tabs, which makes reassembly easier. However, their approach is unable to handle not-unfoldable polyhedra and scales poorly.

A third approach aiming to create a net is the Tabu Unfolding method, which applies tabu search to the topic of unfolding [Zaw23] (see Section 3.2). The resulting algorithm performs fast and is reliable, however, still can not handle not-unfoldable polyhedra.

Our presented approach tries to overcome the ununfoldability problem, by transforming the input first, then unfolding the transformed version and to reverse the transformation afterwards, while keeping the unfolding overlap-free. In contrast to all other previous work, our approach is able to work with not-unfoldable input, while keeping the unfolding single-patched.

3. Background and Definitions

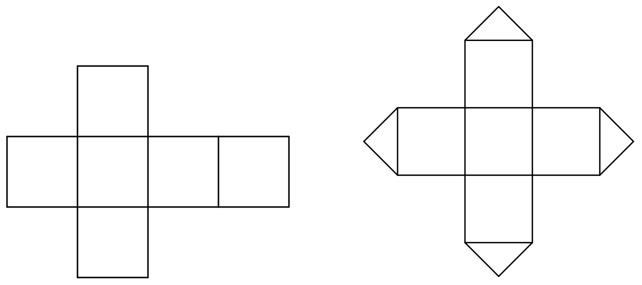
In this section we review the most important definitions and concepts. Many definitions and explanations can also be found in the book *Geometric Folding Algorithms: Linkages, Origami, Polyhedra* [DO07].

3.1. Unfolding

Unfolding a polyhedron means to cut it open and then unfold it along its given edges, such that all faces end up in a common plane. When unfolding, all faces must keep their shape and size, that is, no distortions are allowed. Oftentimes, the term *unfoldable* is used synonymous to "unfoldable without self-overlap".

There are two main ways to unfold a polyhedron, one is called *edge unfolding*, and the other one *general unfolding*. Edge unfolding only allows cuts along existing edges of the polyhedron, while general unfolding allows arbitrary cuts through the polyhedron's faces. Exemplary cases of both methods for unfolding a cube are given in Figure 1. An overlap-free single-patched unfolding created via edge unfolding is called a *net*. If a polyhedron does not have a net, it is called *not-unfoldable*, or *unfoldable*. In Table 1, an overview of the capabilities of the two unfolding-techniques applied to (non-)convex polyhedra is given.

In contrast to edge unfolding, general unfolding has no theoretical upper limit of how many folds have to be done. Also, general unfolding can result in arbitrarily small faces, which may be too small to fold in the real world. Therefore, we chose to base our work specifically on edge unfolding.



(a) *An unfolding of a cube, generated via edge unfolding.* **(b)** *An unfolding of a cube, generated via general unfolding.*

Figure 1: *Two different unfoldings of a cube.*

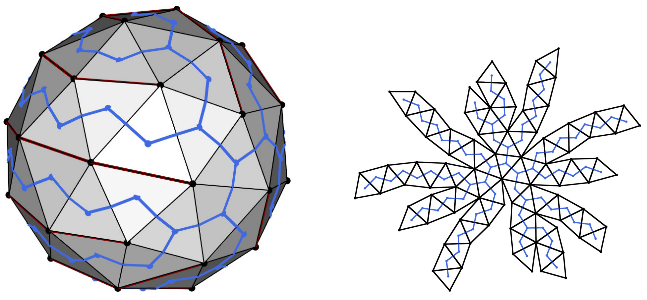
	Edge unfolding	General unfolding
Convex	open	always possible
Non-convex	not always possible	open

Table 1: *Status of main questions concerning non-overlapping unfoldings. [DO07, Table 22.1]*

An (edge) unfolding can be defined via the cuts done along the edges of the mesh. These cuts form a spanning-tree over the edges of the mesh if the mesh is of genus zero. For higher genii, the cut-structure needs to be a graph with as many loops as the genus of the mesh. This cut-structure is referred to as a *cut-tree* for genus zero meshes, and for higher genii the structure is called a *cut-graph*. Unfortunately, the literature is not consistent and sometimes uses the term cut-tree also for non-genus zero objects. An example unfolding including the cut-tree highlighted on the mesh is shown in Figure 2a.

Alternatively, an unfolding can be defined as a spanning-tree over the dual-graph of the mesh. We call such a spanning-tree an *unfold-tree*. Such an unfold-tree is a tree, regardless of the genus of the mesh. In theory, flat regions of a mesh do not need to be cut open and thus would allow for a loop in the tree. We chose to still do the cut, to increase flexibility while solving overlaps. Within the unfold-tree each node represents a face, and also an entire subtree. Each node only needs to hold a transformation rotating the face it represents into the plane of the parent face along their shared edge. Obtaining this transformation for a given unfold-tree is straightforward. Unfolding the whole mesh can then be done by traversing the tree once, while accumulating transformations. In Figure 2 the unfold-tree is shown on the original as well as the unfolded mesh.

We call the interior angles of a face the *face angles*. Given a vertex v in a mesh, we call the sum of all face angles incident to v



(a) *A folded icosphere.* **(b)** *The corresponding unfolding.*

Figure 2: *A folded and overlap-free single-patched unfolded icosphere with 80 faces. The unfold-tree is indicated in blue. The cut-tree over the edges of the mesh is visualized in dark red. In the unfolding, the cut-tree is the boundary and is not colored.*

the *total face angle* of v. While all face angles of a planar polygon add up to a constant (e.g. π for triangles), the total face angle of a vertex is a positive floating number. In most practical cases the total face angle is smaller than 4π, but in theory there is no upper limit.

The *angle deficit* of a vertex is equal to 2π minus the total face angle of the vertex. If the angle deficit of a vertex is positive, only one cut is needed to unfold the faces around it into a plane. Only if the angle deficit of a vertex is zero, no cut is needed to unfold the faces around it into a plane.

3.2. Tabu Unfolding

A central building block of our approach is the *Tabu Unfolding* [Zaw23] algorithm, which we want to recap here. The algorithm uses tabu search [Glo86] to iteratively resolve overlaps in an unfolding, represented as an unfold-tree, until none are left. An initial unfolding is generated using the *Steepest Edge* unfolder [Sch97]. In each iteration one node – possibly representing a whole subtree – in the unfold-tree is re-attached to a new parent. This action is called a *move*.

In each iteration, a random overlapping node is selected as a candidate to be moved. If the node can be moved in a way that reduces the number of overlaps the move is performed. Else, the algorithm recursively selects the parent node and does the same test, until a move is found which reduces the number of overlaps, or until the root node is reached. In the latter case the algorithm performs the best move it found while climbing the tree. This move may worsen the number of overlaps, but is still the best move available for the sequence of nodes from the selected one to the root node.

To overcome local minima, the algorithm remembers the last m moves and prevents these from being undone. If the algorithm detects all possible moves are on the tabu list, the tabu list is cleared. As in the original work, we use $m = val \cdot \log_{val}(|F|)$, with val being the average valence of the dual graph of the mesh and $|F|$ being the number of faces in the mesh.

While providing a lot of benefits, a tree-structure poses some problems when it comes to resolving overlaps in an unfolding [Zaw23, Section 4.5]. These problems include inefficiency, but

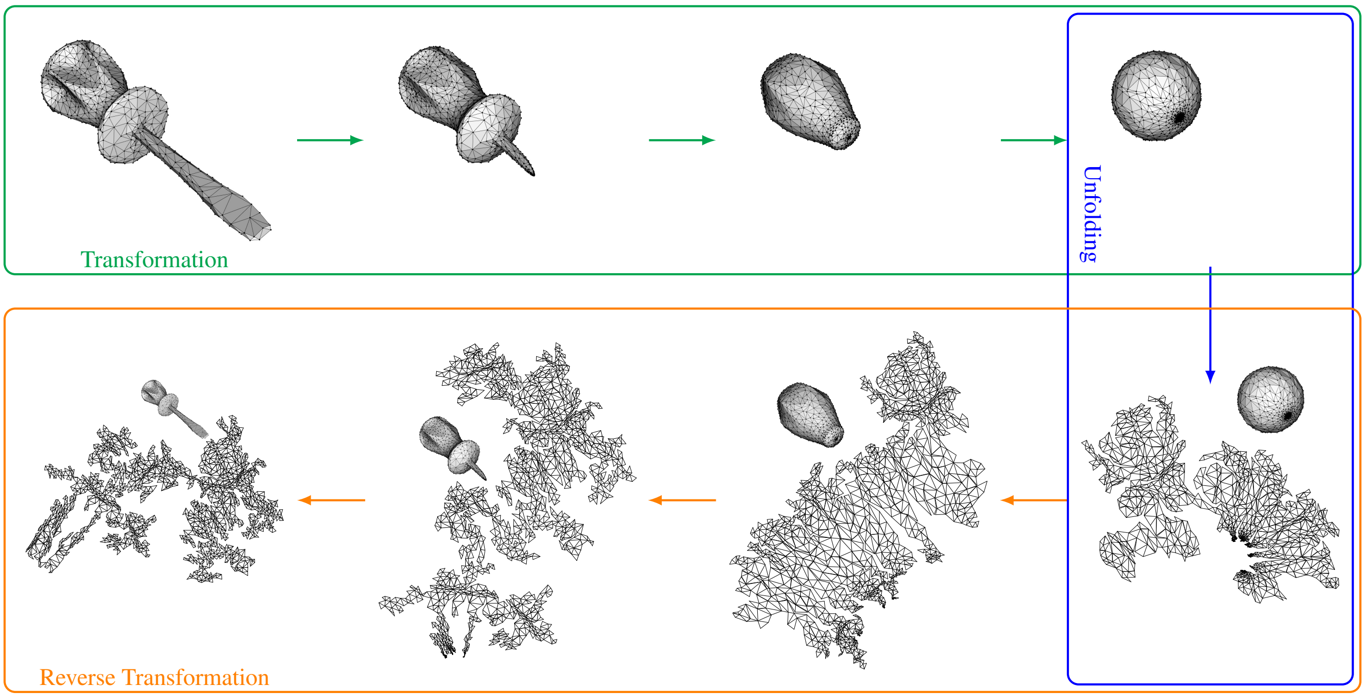


Figure 3: *A visualization of our pipeline applied to a screwdriver mesh with 2000 faces. In the first phase, the transformation (top row, green arrows) is applied. Then, the transformed mesh is initially unfolded (right column, blue arrow). In the third phase, the transformation is reversed while the unfolding is kept overlap-free (bottom row backwards, orange arrows).*

also deadlocks, where the algorithm can not perform any more moves. To prevent and resolve such situations Tabu Unfolding uses a rerooting method. This method selects a new node to be the root node of the unfold-tree. Please note that the unfold-pattern does not change by rerooting. By rerooting in every iteration, the algorithm can mimic the behavior of a flexible unfold-pattern, while maintaining the benefits of a tree-structure. Within this work, we will refer to this method as *TU*.

4. Methods

A common technique to overcome a difficult problem is to transform it, solve the transformed problem and then reverse-transform the result. Our approach follows this approach and consists of the following three main parts, which are further explained in the following subsections:

1. Transform the mesh into an easier to unfold version
2. Unfold the transformed mesh
3. Reverse the transformation while keeping the unfolding overlap-free

Our pipeline is illustrated in Figure 3. Since there is no clear definition of what “easier to unfold” is, we investigate different methods transforming a mesh. These methods are outlined in Section 4.1. Please note that we are not just interested in the final result of such a transformation, but also in every intermediate step.

As an input, we restrict our meshes to be orientable, triangular manifold meshes of genus zero. The orientability and manifold constraints are necessary conditions for a mesh to be unfoldable. In theory, it would be possible to use a transformation which can process arbitrary faces. However, such a transformation would need to yield intermediate and final results with planar faces, which is a necessary constraint to be unfoldable. Thus in this work, we restrict the input to be triangular. Since we aim to compare the effect of different transformation methods, all methods need to be applied to the same input. Some methods presented below in Section 4.1 can only process an input of genus zero. Therefore, we restrict our input to be of genus zero as well.

4.1. Different Transformation Methods

In this work, we want to investigate the effect of different transformation methods on the aforementioned unfolding pipeline. When reverse transforming the mesh (Step 3), introducing as little new overlaps as possible during the process is desirable. Due to the transformation (Step 1), faces will change their shape and size and “grow” (or shrink) back to their original size during reverse transformation. Besides growing/shrinking uniformly and rotating, each face may also distort, e.g. by shearing or growing non-uniformly, during the (reverse) transformation.

Looking at faces in the unfolding as nodes within the unfold-tree, a deformation of a face will have an effect on the whole subtree it is the root of. The effect which causes most problems is a change in the folding axis, i.e. the direction of an edge between two neighboring faces in the unfold-tree. Such a change in the folding direction will cause all faces in the unfolding, represented by the subtree, to rotate. See Figure 4 for an example. Any deformation which changes the quasi-conformal error, defined as the ratio of maximum (Γ) and minimum singular value (γ) of the unique affine mapping S [SSGH01, Section 3], of a triangle, will also change at least one axis of the triangle. In contrast, isometric transformations will not change the axis at all. Unfortunately, isometric transformations are too rigid, to change the shape of a mesh into a more convex form.

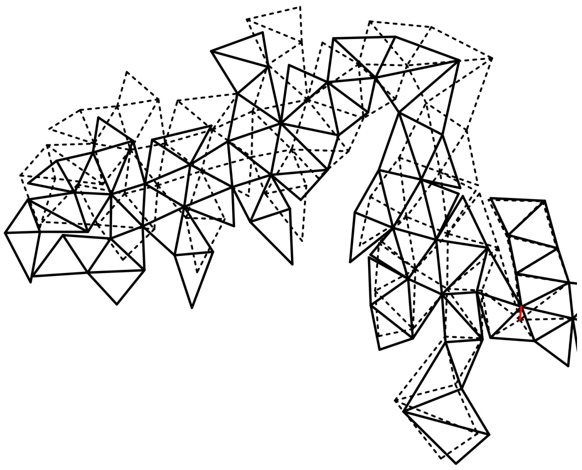


Figure 4: *The effect of a small change in a vertex on an entire unfold-tree. The solid lines represent the initial unfolding. The dashed lines represent the unfolding after a vertex has been shifted by a bit, with the red line showing the shift of the vertex. This vertex shift causes a deformation effect in some of the surrounding faces. Please note the enormous difference on the far left of the unfolding, caused by the deformation that a small vertex move introduces.*

Thus, in our work, we chose mostly conformal transformation methods, which minimize the quasi-conformal error. While three methods are conformal in every iteration, two are only conformal in the result, but have conformal distortions in intermediate iterations and two are not conformal at all. The different methods are explained below. Figure 5 illustrates the effect of the transformation methods on an example mesh.

Conformal Curvature Flow

Presented by Crane et al. [CPS13], this method works directly on curvature and restores vertex positions from the modified curvature. To mimic the behavior of different curvature flows, this method allows to filter curvature, before working with it [CPS13, Section 4]:

$$v \leftarrow v - (I - \sigma\Delta^k)^{-1} v,$$

with v being the flow direction (in our case the negative curvature of the mesh), I the identity matrix, σ some step size and Δ the Laplace-Beltrami operator. The value k determines the applied type of flow. For $k = 0$ no filter is applied at all, $k = 1$ resembles mean curvature flow and $k = 2$ corresponds to Willmore flow. This flow is by nature conformal in every step (up to a discretization error), which makes the differences between this method with $k = 1$ and the cMCF method interesting. Our implementation follows the original method without modifications. We will refer to these methods as *CCFk* with $k \in \{0, 1, 2\}$.

Conformalized Mean Curvature Flow

The deformation method was presented by Kazhdan et al. [KSBC12] and aimed to remove singularities from ordinary mean curvature flow. While yielding conformal results, intermediate steps of this method may have conformal distortions. Moreover, this method can only be applied to meshes of genus zero, which is the reason for restricting our input to this class of meshes as indicated above. The core concept is to repeatedly solve the following equation:

$$M_n V_n = (M_n - tL) V_{n+1},$$

with V being the vertex positions of the mesh, M the mass matrix, L the stiffness matrix, and t some step factor. The subscript denotes the iteration the respective variable belongs to. Our implementation follows the original method without modifications. We will refer to this method as *cMCF*.

Inflating-Inspired Flow

Inspired by the idea to treat the input mesh like a balloon, we implemented a flow based on a mass-spring model. This flow repeatedly solves the following equation:

$$V_{n+1} = V_n - L^{-1}(t M_n N_n)$$

with V being vertex positions, L the cotan-Laplace matrix, t some step size, M the mass matrix, and N the vertex normals of the mesh. The subscript denotes the iteration the respective variable belongs to. This flow resembles Hooke's law, with $(t M_n N_n)$ serving as the outward force and the cotan weights of L as the spring-constants. Just like the cMCF method, this flow also can only process meshes of genus zero. We will refer to this method as *IIF*.

Edge Normal Alignment Flow

Unfolding a triangle fan around a vertex is trivial as long as the angle deficit around it is greater or equal to zero, i.e. the sum of angles around that vertex is $\sum_i \theta_i \leq 2\pi$. In such cases, one cut along any edge of the vertex is sufficient to flatten the triangle-fan into a plane. Even though there is no guarantee a mesh fulfilling this criterion in every vertex is unfoldable, the angle deficit is still a measure worth investigating in terms of unfolding. Since the sum over all angle deficits is a topological constant, we decided to minimize the sum of squared angle deficits, i.e. the distribution of angle deficits over all vertices:

$$\arg\min_V \sum_V (2\pi - \sum_i \theta_i)^2$$

Instead of minimizing this sum directly, we chose a different route. We align two different edge-normals. One is calculated as the area-weighted mean of adjacent face normals and the other one is calculated as the mass-normalized mean of adjacent vertex normals. We will refer to these two variants as *face-based edge normal* and *vertex-based edge normal* respectively. These two types of normals define a rotation per edge, which we then use to recover the mesh in an As-Rigid-As-Possible [SA07] sense. To incorporate the concept of step sizes to this method, we use spherical linear interpolation between the identity rotation and the one given by the two edge-normals. In our work, we always rotate the face-based edge normal into the vertex-based edge normal: By doing so, the face-based

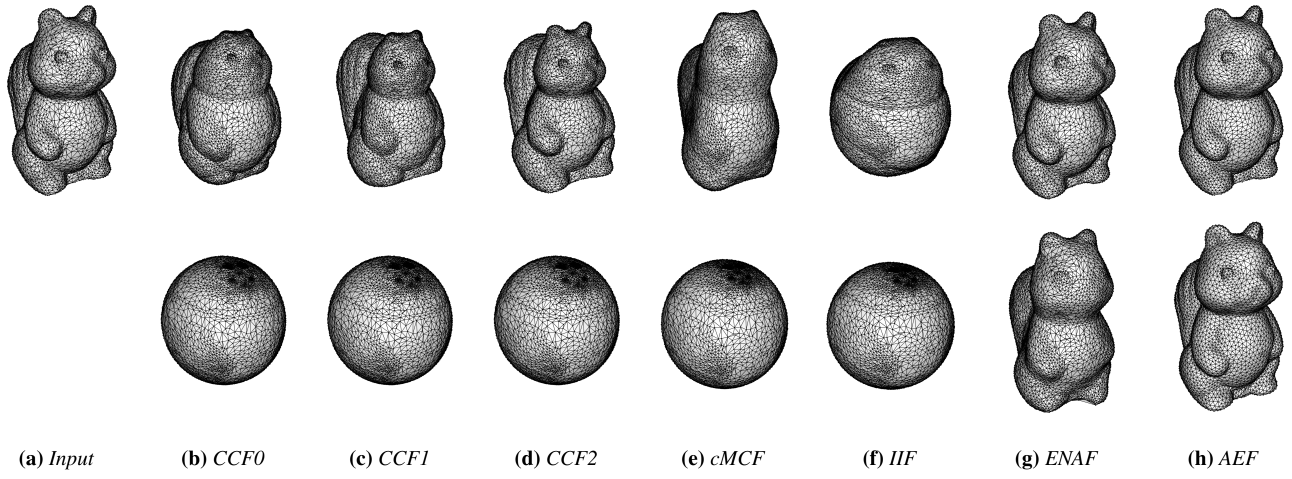

(a) *Input* **(b)** *CCF0* **(c)** *CCF1* **(d)** *CCF2* **(e)** *cMCF* **(f)** *IIF* **(g)** *ENAF* **(h)** *AEF*

Figure 5: *Results of the flows used in this paper applied to a Squirrel model with 10.000 faces. Top row: After one iteration with maximal step size. Bottom row: Final results.*

edge normals are smoothly aligned between the vertex-based edge normals. This results in the face-based edge normals around a vertex to be as aligned as possible to the respective vertex normal. By aligning the face-based edge normals (as much as possible) of a triangle fan, the angle deficit of the center vertex is minimized. Since this flow aligns the face-based edge normals towards the neighboring vertex normals, the angle deficit of all vertices is distributed as evenly as possible. We will refer to this method as *ENAF*.

Angle-Equalizing Flow

The original flow was presented by Zhou et al. [ZS00] and aims to equalize angles in a 2D domain. The core idea is to measure two neighboring angles at a vertex and then to rotate the enclosed edge such that the angles are considered optimal. In case of a homogenous domain (e.g. triangles), the optimum would be equal angles. The final vertex positions are the vertex-wise means of all rotated results. In its original form, this flow only works in a purely 2D domain. Our implementation extends the approach to also work on a 2D surface in 3D. Instead of rotating edges in a fixed plane, we rotate edges around the vertex normal of its end-vertex:

$$v_j = v_i + R(t\theta_{i,j})e_{i,j},$$

with v_j being vertex at index j, $e_{i,j}$ the edge from v_i to v_j, $\theta_{i,j}$ the angle the edge $e_{i,j}$ needs to be rotated, $R(\theta_{i,j})$ the 3D rotation around the vertex normal of v_j by the angle $\theta_{i,j}$, and t some step size. In contrast to all other flows discussed in this section, this one aims to preserve the appearance of the mesh, while only changing the quality of its triangles. We will refer to this method as *AEF*.

Step Sizes

Each of the aforementioned methods has a step size t as a parameter, which only makes sense if it falls within a certain range. The CCFk variants are using an explicit euler scheme, which limits the step size by $t < 1$ from above [CPS13, Section 4]. For the cMCF we follow the von Neumann stability analysis [MG80] and set the maximal step size to $t < \frac{1}{2\lambda_{max}}$, with λ_{max} being the largest eigenvalue of the respective stiffness matrix. The IIF uses an explicit euler scheme as well, which limits the step size to $t < 1$. Finally, the AEF and ENAF both rotate edges by a certain angle and use the step size to scale the rotation angle. Therefore, for these two flows, $t \leq 1$ is the limit. All step sizes are bound by $t > 0$ from below.

4.2. Applying a Transformation Method

In the first stage of our algorithm, the transformation method is applied to the mesh. The method also defines the approximation in case our pipeline stops before finishing the reverse-transformation back to the input mesh. Therefore, it is advisable to use different transformation methods depending on the application. In this work, we investigate seven different methods, but our pipeline is designed to work with any iterative transformation method.

Our pipeline aims to produce very few new overlaps in each reverse-transformation step. This can be achieved by only changing the shape of the object gradually, respectively performing small steps while transforming. In particular, we pick an initial step size and increase it by $x\%$ per iteration, up to the maximal step size of the respective method. The increasing step size counterweights the well-known decreasing geometric change with constant step size yielded by most surface flows. The smaller x, the finer the change in each transformation step, resulting in less newly introduced overlaps per reverse-step. Choosing x too small will result in a lot of calculation overhead, though. In our work, we chose $x = 100\%$, i.e. we double the step size in every transformation iteration.

Similarly, the initial step size also has a direct effect on the approximation quality, as well as the amount of newly introduced overlaps per reverse-transformation iteration. While there exist hard upper bounds for the initial step size, the lower bound is $\epsilon > 0$. Again, a small initial step size will result in finer transformation steps, which will also result in more computational work. In our work we chose the initial step size depending on the transformation method slightly larger than zero (between 10^{-7} and 10^{-1}).

4.3. Unfolding

In the second stage, the transformed mesh is unfolded. When unfolding the transformed mesh, it is important to keep the reverse transformation in mind. During the transformation, some faces change more than others. The faces which have changed more during the forward transformation will also change a lot back during the back transformation. If those faces are high up in the unfold tree, the branches attached to them will move a lot, as illustrated in Figure 4, which can cause a lot of avoidable overlap resolution work. Therefore, we aim to place the faces which changed the most as far down in the initial unfold tree as possible.

The initial unfolding of the transformed mesh is created, using Tabu-Unfolding [Zaw23], but instead of using the steepest edge unfolder, we use a minimal spanning tree heuristic. For this heuristic, we assign a value to each face which represents how much it changed during the transformation, i.e. the change in size of the face. Then, the weights for directed edges of dual-graph are the ratios of weights of neighboring faces:

$$w_{i,j} = \frac{w_j}{w_i}$$

with $w_{i,j}$ being the weight of the edge from face i to face j and w_i being the weight of face i. Afterwards, we grow a spanning-tree in the fashion of Prim's algorithm, with the face having the smallest weight as a root node. With these weights, Prim's algorithm will always add the node next, which is the locally relative best regarding the node-weights, i.e. faces which have changed the least bad relative to their neighbor will be added first. This way, within each branch of the tree, node-weights will increase as gradually as possible. Please note that this tree is not a minimal spanning tree, since Prim's algorithm is designed for undirected graphs.

4.4. Reverse Transformation

In the last stage, the transformation of the first stage is undone in reverse order while keeping the unfolding overlap-free. While reversing the transformations, we again use Tabu-Unfolding [Zaw23] as a basis. In each step, we use the unfolding computed in the previous step as a starting point.

If eventually no overlap-free unfolding can be found, the process stops and returns the last intermediate overlap-free unfolding as the best possible approximation.

5. Results

Example unfoldings of a triangular mesh as well as a refolded example are shown in Figures 6 and 7.

To determine the effect of the different transformers introduced above in Section 4.1, we evaluated six different metrics, which are all relevant to unfolding:

- **Coverages:** The coverage of an unfolding is defined as its summed triangle areas divided by the area of its oriented bounding box.
- **Aspect ratios:** The aspect ratio of an unfolding is defined as the ratio of the oriented bounding box sides, such that the result is greater or equal to one.

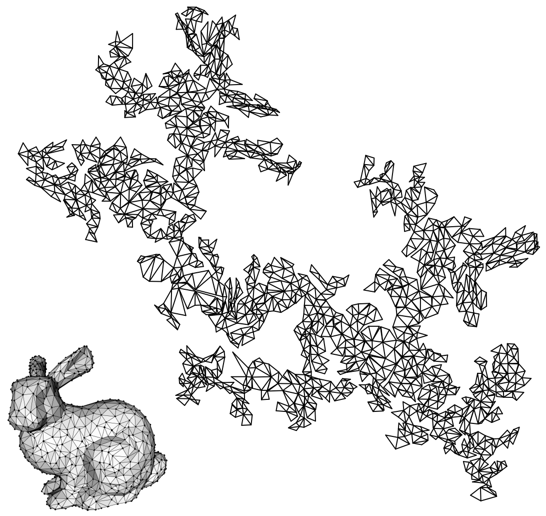


Figure 6: *The unfolding of a Bunny with 2000 faces.*

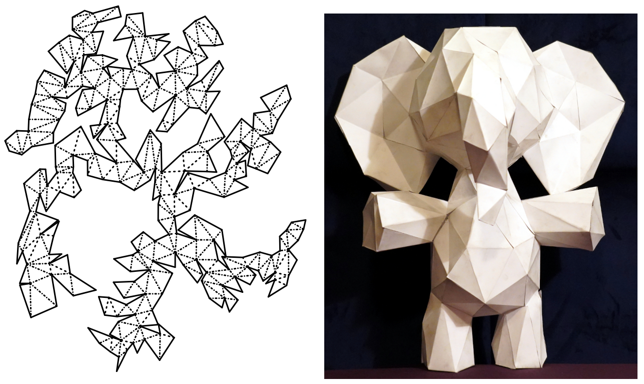

(a) *The unfolded Elephant.* **(b)** *The refolded Elephant.*

Figure 7: *An Elephant model with 300 faces unfolded and folded.*

- **Branching:** We define the average branching factor of a tree as the sum of children of all internal nodes divided by the number of internal nodes.
- **Success rates:** A success was given, when a method was able to unfold the original model without approximation. In our pipeline, this is equivalent to finishing the reverse transformation completely.
- **Success rates approx:** Success rate including approximative results.
- **Timings:** Timings of successful unfoldings.

Coverages determine how much material is wasted when cutting, and aspect ratios are important to fit standard paper sizes. In our evaluation, we focused on the success rate including approximative results. The success rates without approximation are shown in Figure 23 in Appendix B. The differences within all aspects are evaluated in the following subsections. Besides these metrics, we also investigate the approximative effect of our algorithm on specific and general input. Lastly, we also discuss limitations.

We chose a subset of the Thingi10k dataset [ZJ16] for our evaluation. To match our constraints, we only kept meshes which were manifold, consisted of a single component, and were of genus zero. Our subset consisted of 1,339 meshes, which we represented in ten different resolutions (100, 200, 400, 600, 800, 1000, 1250, 1500, 1750, and 2000 faces).

5.1. Coverage, Aspect Ratios, Branching, and Success Rates

Besides a few exceptions, the coverages (see Figure 8), aspect ratios (see Figure 9), and success rates (see Figures 11 and 23), of our algorithm in all variants and the Tabu Unfolder are very similar. In contrast, the branching factors (see Figure 10) of the unfold-trees show significant differences.

In the coverage curve (see Figure 8) all variants and the Tabu Unfolder look very similar. Except for the AEF one, all variants stay above 20% coverage. Since our method as well as the Tabu Unfolder aim for single-patched unfoldings, coverages are naturally lower compared to segmenting approaches.

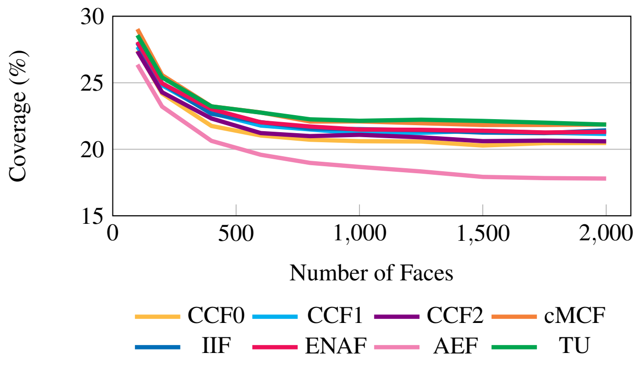


Figure 8: *The mean coverages of our algorithm in all variants compared to Tabu Unfolding.*

Furthermore, the aspect ratios (see Figure 9) of all variants are within a narrow range between 1.5 and 1.8. In the current evaluation, our algorithm seems to result in slightly higher aspect ratios than the basic Tabu Unfolder.

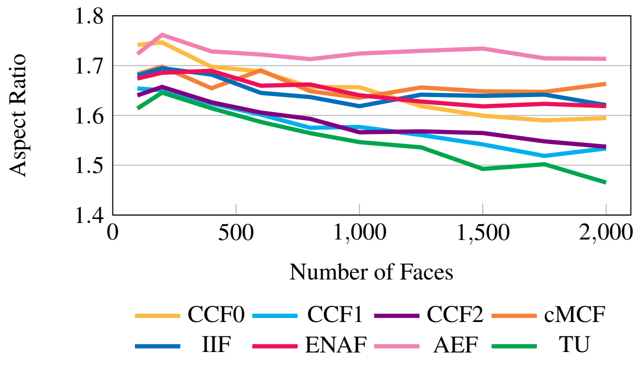


Figure 9: *The mean aspect ratios of our algorithm in all variants compared to Tabu Unfolding.*

We also investigated the mean average branching factors (see Figure 10) of the underlying trees. All variants of our algorithm show similar behavior. In contrast, the Tabu Unfolder has a clear downward trend. Please note that this is a topological measure and does not allow any inference about the geometric layout of the unfolding. We also evaluated other topological measures, like tree heights, but did not find any difference.

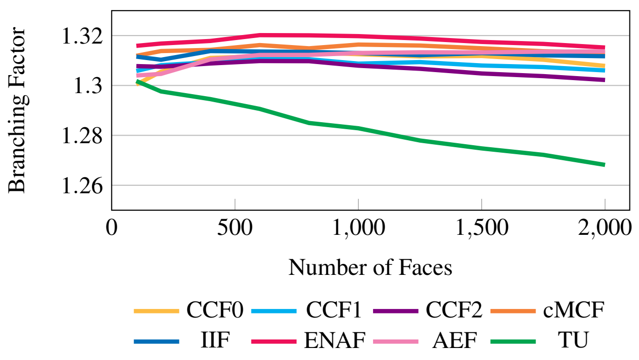


Figure 10: *The mean average branching factors of our algorithm in all variants compared to Tabu Unfolding.*

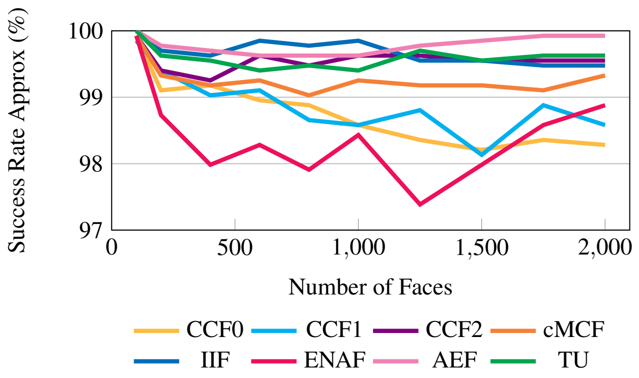


Figure 11: *The success rates of our algorithm in all variants compared to Tabu Unfolding including approximative results.*

Finally, within the success rates including approximative results (see Figure 11), the worst reliabilities were all above 97%. Moreover, the best variants of our algorithm show even better reliability scores than the Tabu Unfolder. Please note that our variants and the Tabu Unfolder succeed on different inputs. E.g., even though their reliability values are very close, the IIF method yielded an approximation or a solution for 50% of the cases, where the Tabu Unfolder failed. Unfortunately, our algorithm still fails to produce approximative results for some inputs, see Section 5.4 for a discussion about limitations. The success rates of our algorithm excluding approximative results are shown in Figure 23 in Appendix B. Even without considering approximative results, some variants of our algorithm show similar reliability values as the Tabu Unfolder does.

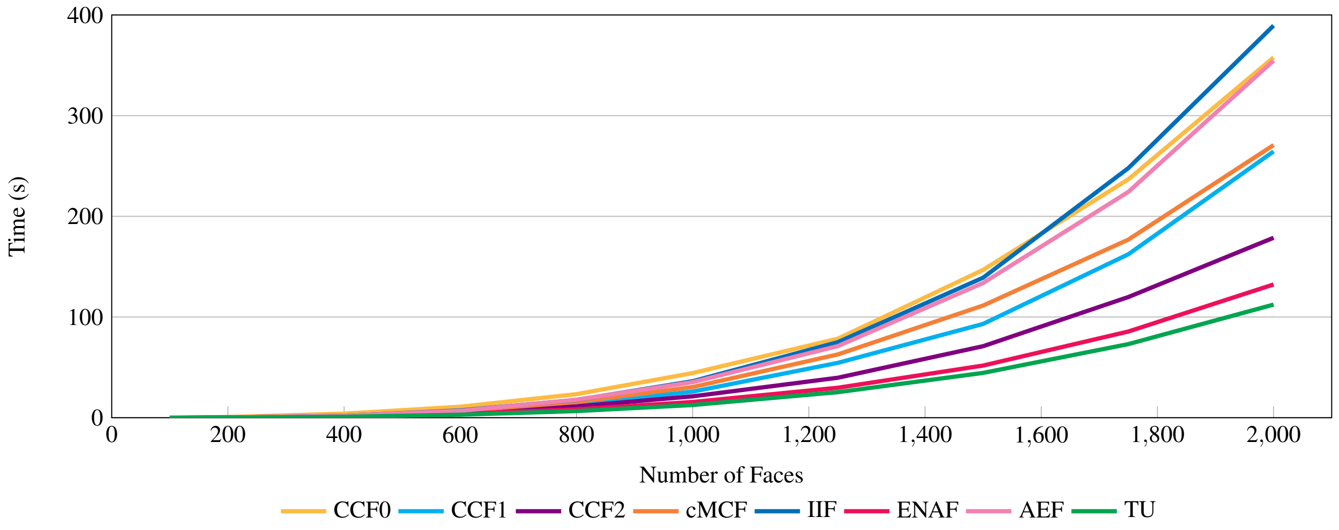


Figure 12: *Median unfold timings of our method in all variants (see Section 4.1) compared to Tabu Unfolding.*

5.2. Timings

All timings have been measured on a machine with an i7-10700K CPU (3.8GHz) and 128GB RAM running Linux. All implementations were written in C++ and were executed without parallelization. We executed each variant of our algorithm, as well as Tabu Unfolding, on each mesh in the test set in each resolution once. Even though the test set is large, each model has only been unfolded once and thus average times are still too much affected by outliers. Therefore, we chose to analyze the timings via the median, which is much less sensitive to outliers. The median timings of all runs are shown in Figure 12. In Figure 22 a side-by-side comparison of the timings including distribution-indicators are shown. Detailed results are shown in Tables 2 in Appendix C.

To analyze the performance difference, we also measured the accumulated number of overlaps that each method had to solve over the whole reverse-transformation phase. The medians of these sums are shown in Figure 13.

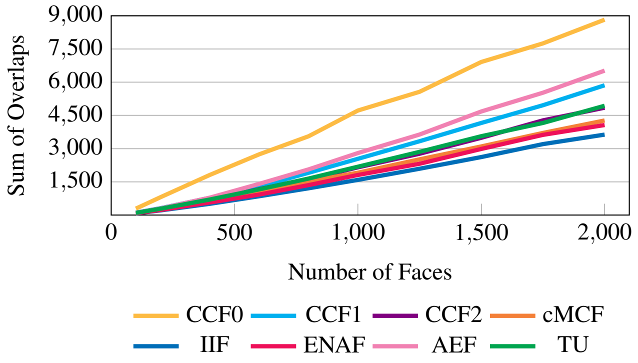


Figure 13: *Median accumulated overlaps each variant of our algorithm had to solve during the reverse-transformation.*

Interestingly, the number of overlaps does not always correlate with the timings. For example, the IIF variant had the least overlaps to solve, but performed among the slowest. Within the CCF variants, a clear correlation between *k* and the performance, in both the overlaps as well as the timings is visible. Looking only at the timings, the cMCF variant performed just a bit worse than the CCF1 one. When taking into account the number of overlaps as well, the CCF1 variant showed a way better time-per-overlap ratio than the cMCF one. Of all variants we tested our algorithm with, the ENAF one performed the best. Unfortunately, this variant showed the worst reliability values.

When taking a combination of success rate and runtime into account, the CCF2 method showed the best overall performance. Additionally, as shown in Figure 22 the timing distribution of the CCF2 variant is also the narrowest among all variants of our algorithm.

To investigate why some variants had to solve less overlaps, but took considerably longer to do so, a closer look at different cases where overlaps can occur during the reverse transformation is needed.

Angle Deficit And Overlaps

One crucial value to look at within an unfolding is the angle deficit at each vertex. In unfoldings, every vertex has an adjacent gap between two faces, which was created by cutting the mesh open. If the angle deficit at a vertex decreases, the angle of the gap also decreases. Such a decrease of angle will rotate the two branches the two adjacent faces are the root nodes of, which may result in the two branches to overlap with other branches. If the angle deficit becomes negative, the two faces adjacent to the gap will overlap. Typically, these two faces are also neighbors in the mesh. Within an unfolding, especially of convex (parts of) polyhedra, neighboring faces of the mesh tend to be close within the unfolding as well. E.g., in Figure 2b every neighbor of a face has no other face between them, except for the tips of the triangle-strips. Overlaps between neighboring faces oftentimes do not have a trivial solution, especially for triangles. If two neighboring triangles overlap, moving one overlapping triangle to the other will just result in the moved triangle to overlap with its current parent. Thus, there is at maximum only one triangle left to move to. This last triangle will oftentimes be located too close to the other two to resolve the overlap by moving a face to it. If both overlapping neighbors are inner nodes of the unfold-tree, there is no move left which will make both faces overlap-free. Such an example is illustrated in Figure 14b. Therefore, avoiding negative angle deficits within the reverse transformation will have a beneficial effect on the resulting performance.

The issue of overlapping neighbors becomes even more problematic if the overlapping faces are located on the inside of the unfolding. In such a situation, many faces need to be moved after another to resolve the overlap. This is a very tedious task. The problem of rotating branches, as described in Section 4.3, can be mitigated by constructing the unfold-tree such that faces which are expected to change most are placed close to the leaves. Unfortunately, there is no guarantee this construction also results in unfold-trees having their leaves on the outside of the unfolding. Therefore, overlaps on the inside of the unfolding regularly occur during the reverse-transformation. These overlaps are the main reason all variants of our algorithm perform slower than Tabu Unfolding, even though they have to solve comparably many overlaps.

Conformal Willmore Flow

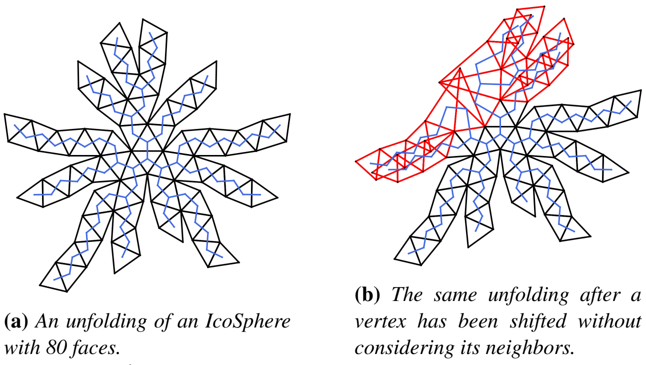
Most transformation methods used within this work converge towards a sphere for genus zero objects, as shown for an example in Figure 5. Thus, for these variants of our algorithm the reverse transformation starts with a sphere, which grows into the original object. To have a closer look at the effect, we investigated how this growing out affects the unfolding on an icosphere with 80 faces. The effects are shown in Figure 14.

Just shifting a vertex without considering its neighbors will result in a drastic decrease of angle deficit at the neighboring vertices. This is illustrated in Figure 14b. Please note, this example is just an illustration, none of the transforming methods used in this work follow such a strategy.

If the vertex is not just shifted, but the neighbors are considered as well, the angle deficit of a vertex may still decrease. E.g., the cMCF transformer does consider the neighboring faces as well, but introduces conformal distortions in intermediate steps. Transforming the neighbors according to the same vertex shift as before does not result in overlaps in the case of the IcoSphere. Still, angle deficits at some vertices decreased significantly (see Figure 14c).

Lastly, if the shift is done in a conformal way (up to discretization error), by using a CCFk transformer, the angle deficits decrease less. Many angle deficits are increasing, as e.g. the one at the center vertex, while the ones which decrease only do so by a bit and not as much as with the cMCF method. This is shown in Figure 14d.

Conformal transformations, which only apply local scale and rotation, change the shape of triangles as little as possible. If all triangles around a vertex keep their shape as much as possible, their face angles will also change very little, which results in the angle deficit to change proportionally little. This would result in no angle deficits of a vertex to change. In the IcoSphere with 80 faces example the three CCFk methods resulted in almost identical results, because of the low face count as well as the round shape and identical curvatures at each vertex.


(a) *An unfolding of an IcoSphere with 80 faces.* **(b)** *The same unfolding after a vertex has been shifted without considering its neighbors.*

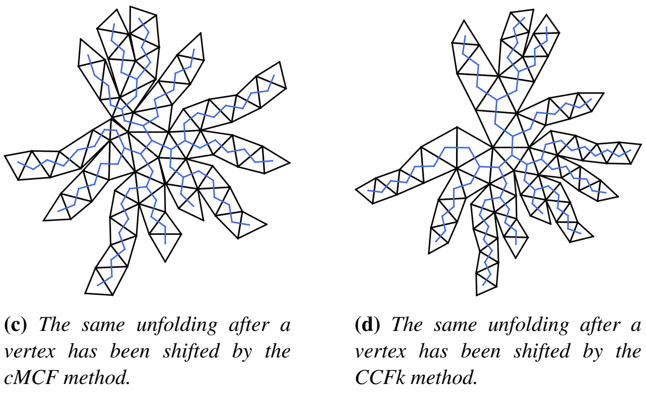

(c) *The same unfolding after a vertex has been shifted by the cMCF method.* **(d)** *The same unfolding after a vertex has been shifted by the CCFk method.*

Figure 14: *Transforming the surface of an IcoSphere with 80 faces has different effects on the unfolding. In all transformations the same vertex has been shifted in the same direction. Other vertices have been shifted as well according to the respective method. Depending on the transformation method, the angle deficit of neighboring vertices may decrease or increase. Subfigure 14b also illustrates the issue of overlaps close to the center of the unfolding as well as rotating branches (see Section 4.1).*

To investigate the differences between the CCF1, CCF2 and cMCF methods, we applied these three variants of our method to a teddy bear with 1200 faces and recorded every step of the reverse transformation. The effect each method has on the unfolding in the reverse transformation is exemplarily shown in Figures 15, 16, and 17.

In accordance with the argument from the IcoSphere example, the cMCF method suffers from conformal distortions, which result in angle deficits to decrease. The overlaps shown in Figure 15b meet both criteria of difficult overlaps discussed above. Most overlaps are created between neighboring branches in the unfolding, some even additionally on the inside of the unfolding. Resolving such overlaps is very time-consuming and explains why the cMCF variant of our algorithm performed slower than other methods, which had to resolve more overlaps.

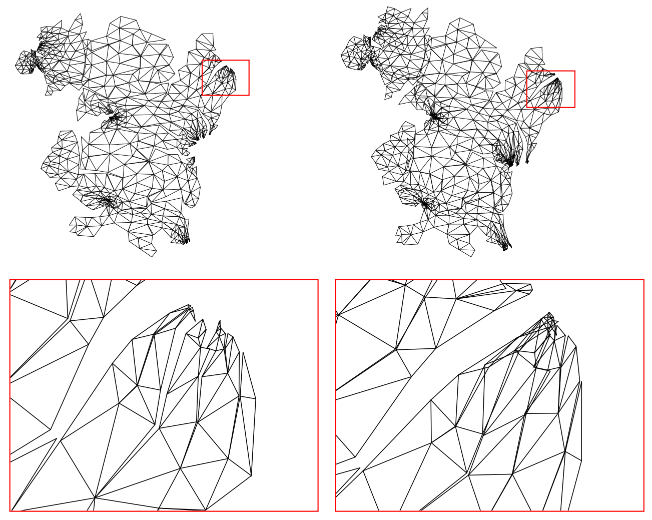

(a) *The unfolding before undoing a transformation step.* **(b)** *The unfolding after undoing a transformation step.*

Figure 15: *Intermediate steps of the reverse transformation while unfolding a Teddy with 1200 faces using our algorithm in the cMCF variant. Bottom: Zoom in on the marked area.*

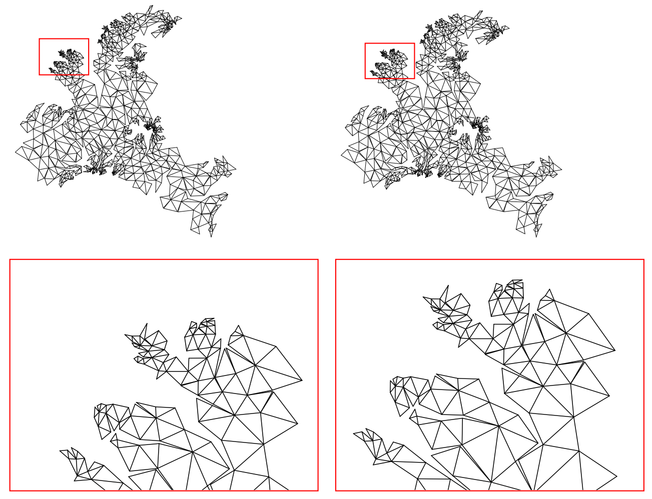

(a) *The unfolding before undoing a transformation step.* **(b)** *The unfolding after undoing a transformation step.*

Figure 16: *Intermediate steps of the reverse transformation while unfolding a Teddy with 1200 faces using our algorithm in the CCF1 variant. Bottom: Zoom in on the marked area.*

The CCF1 example meets the argument about conformality from above. Branches of the unfolding grow out without noticeably changing directions. While this is an improvement in comparison to the cMCF method, there are still problems left. While the observed behavior prevents neighboring branches from overlapping each other, it is still not suited well for unfolding. In the resulting sphere of conformally transformed meshes, there are dense areas with many small triangles in them (see e.g. Figure 5b bottom). By construction (see Section 4.3), these triangles will be very far down in the unfold-tree. Within the unfolding, they will at first be located at neighboring positions, or at the outside of the unfolding. See Figure 18 for an example. In this configuration, if two branches are growing towards each other, they will overlap at some point. To worsen the situation, after resolving the overlaps in one reverse iteration, the same two branches will overlap later on again. These overlaps are easier to resolve than neighboring overlapping branches, as the ones in Figure 15b. This explains why the CCF1 variant of our algorithm had to resolve many more overlaps than the cMCF variant, but still performed equally fast, i.e. had a better time per overlap ratio.

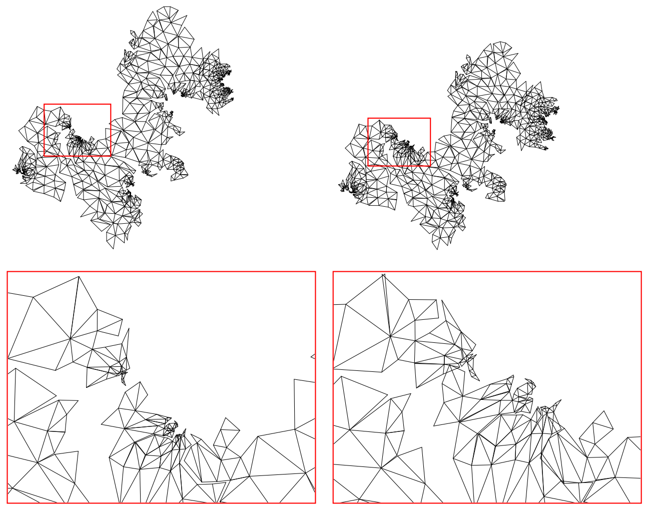

(a) *The unfolding before undoing a transformation step.* **(b)** *The unfolding after undoing a transformation step.*

Figure 17: *Intermediate steps of the reverse transformation while unfolding a Teddy with 1200 faces using our algorithm in the CCF2 variant. Bottom: Zoom in on the marked area.*

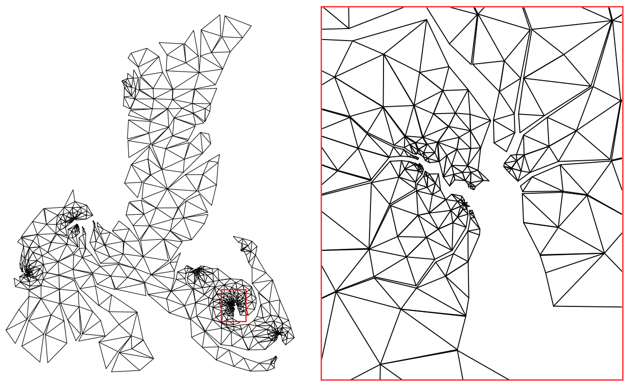


Figure 18: *The initial unfolding of a conformally transformed Teddy with 1200 faces.*

The CCF2 example shows the same "opening" behavior as shown in Figure 14d. Willmore flow is known to create rounder shapes [Cra13, Page 87, bottom]. The IcoSphere example with very few faces from above was not transformable in a conformal way without keeping the roundness within the transformation. It is exactly this roundness, which results in the "opening" behavior seen in Figures 14d and 17. This opening behavior avoids branches on opposing sides to grow into each other, avoiding the overlaps the CCF1 transformer encounters. With only "nice" overlaps left, the CCF2 variant had one of the best time per overlap ratios and the least overlaps among all variants of our algorithm.

Edge Normal Alignment Flow

The ENAF variant of our algorithm had an almost equal time per overlap ratio than the CCF2 method. Since it works on the angle deficits directly, a similar argumentation about the opening behavior in the unfolding holds as above. The main difference between the CCF2 and ENAF method is, the ENAF method does not converge to a sphere (see Figure 5). Therefore, in the reverse transformation, the ENAF variant needed to undo less geometric change than the CCF2 variant needed to. This naturally results in less overlaps to resolve as well (see Figure 13), but also in a higher failure rate (see Figure 11).

5.3. Approximative Results

One very well known polyhedron, which is not-unfoldable, is the spiked tetrahedron [DO07, Chapter 22.4]. To create it, we replaced all triangles of a tetrahedron with spiked triangle hats. The result is visualized in Figure 19a. A tetrahedron only has four faces (a spiked tetrahedron has 36 faces), which is too few to work on curvature directly. Therefore, we used the cMCF variant for this model.

Our method first transforms the spiked tetrahedron back into the shape of a tetrahedron with slightly curved faces. Then, while reverse-transforming, the spikes grow out again, until they reach a height, which is not unfoldable anymore. In Figure 19b the result is visualized. We measured the angle deficit of each vertex at the base of each spike and found that they were just slightly above zero. An angle deficit below zero makes the shape not-unfoldable.

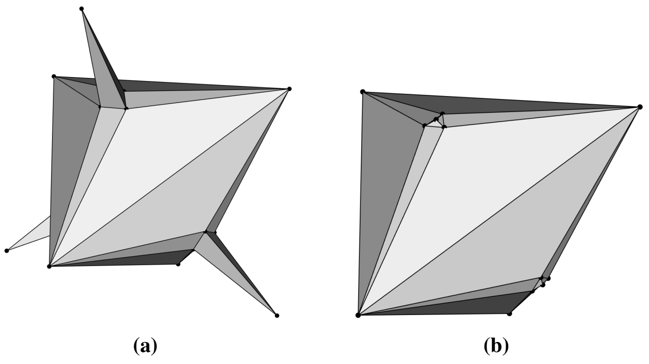
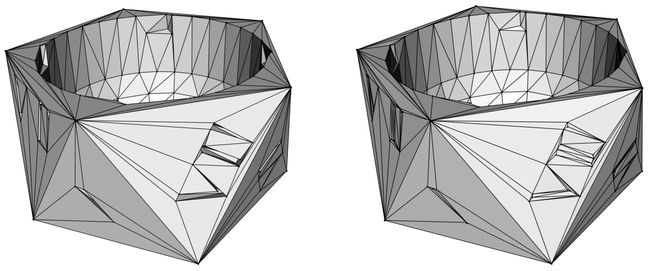


Figure 19: *A not-unfoldable spiked tetrahedron (left) and its closest unfoldable approximation (right) created by our algorithm in the cMCF variant.*

While the result visually does not match the initial input very well, it is still the closest unfoldable approximation. If the spikes are an important feature, which has to be preserved, an approximative method like ours may not be the correct choice in the first place. We explicitly chose an example resulting in a significantly different approximation, to show the limits of our approach.

Another example is half an icosahedron, which is filled with a half-sphere and has some carvings on the outside triangles. Especially the carvings pose a similar issue for unfolding as the spikes from the tetrahedron above. In total, the mesh has 600 faces. Please note, we do not know if this example has a net. The original as well as an unfoldable approximation are shown in Figure 20. While the carvings on the outside are only slightly visible anymore in our approximation, the half-sphere is fully recovered.



(a) *The original mesh.* **(b)** *Our approximation.*

Figure 20: *A mesh which we were not able to unfold and its closest unfoldable approximation.*

We investigated the approximation quality of our algorithm by measuring the Hausdorff distance between approximative results and their originals relative to the respective bounding box diagonal. In this evaluation, we scaled both bounding box diagonals to the same length and placed both bounding box centers at the same location. The results are shown in Figure 21. Please note that we only had very few approximative results. Interestingly, almost all approximations were just one or two steps away from finishing the reverse-transformation.

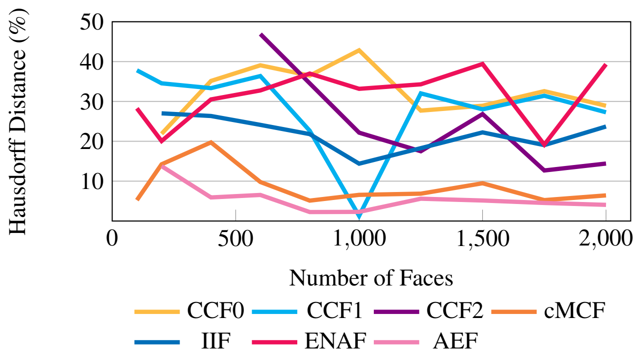


Figure 21: *Mean relative Hausdorff distances over all approximative results for each variant of our algorithm.*

5.4. Limitations

Naturally, our approach "inherits" all limitations and drawbacks of the transformation method used. Some methods are only applicable to genus zero or triangular input (see Section 4.1). Moreover, it is well known that numerical problems can occur when triangles shrink too much during the transformation. This problem typically arises with high resolution input and would prevent our method from scaling arbitrarily. Additionally, coarse triangulations of geometric features are difficult to transform reasonably, especially for conformal flows. E.g., the spikes shown in Figure 19a or the dents shown in Figure 20a prevent a reasonable conformal transformation. Generally, some input resulted in the transformers to not work at all, or in extreme deformations. The latter case is also the main reason for failures in our tests, where Tabu Unfolding succeeded.

Finally, there is no guarantee on the approximation quality to be optimal. This limitation applies in two different ways. On one hand, a different transformation method may yield a better approximation. On the other hand, there is also no guarantee that if a given approximation is not-unfoldable, a closer approximation of the same transformation method is also not-unfoldable.

6. Conclusion and Future Work

In this work, we presented an algorithm, which unfolds a given triangular mesh, by first transforming it into an "easier to unfold" shape, unfolds this shape and then undoes the transformation, while keeping the unfolding overlap-free. We showed that this approach performs reasonably well. In contrast to other methods, our approach keeps the unfolding in one part, rather than segmenting it into pieces. Instead, our method allows approximative results. Even though all flavors of our method performed slower than Tabu Unfolding, we are now able to process input which is not-unfoldable and yield an approximative result. From our experiments, we recommend using the CCF2 transformer, since it performs reasonably fast, while having a good reliability. If time is not an issue, but rather approximation quality, we recommend using our AEF variant. It also showed the best approximation quality among all variants (see Figure 21), while also having one of the highest success rates (see Figure 11).

Currently, we only investigated triangle meshes as well as genus zero meshes. In the future we would like to extend our research to higher genuses as well as higher order polygonal meshes. Also, exploring the effect and performance of different transformers, like other conformal Willmore transformers or even higher order flows (e.g., CCF3+), or combinations of different transformers (e.g. the ENAF and AEF one), is up for future work.

Moreover, in our current approach the reverse transformation is only rewinding the forward transformation. Investigating the effect of different reverse strategies, like using another transformer in combination with an energy minimizing term, is still open for future research.

Finally, our current approach is limited to triangular input. One major task for future works is to find a way to also process non-triangular input. In particular, it is up for research, how to maintain the properties of a transformation method, while keeping the faces flat in every transformation step.

References

[CPS13] Crane K., Pinkall U., Schröder P.: Robust fairing via conformal curvature flow. *ACM Transactions Graphics 32*, 4 (July 2013), 61:1–10. doi:10.1145/2461912.2461986. 5, 6

[Cra13] Crane K.: *Conformal Geometry Processing*. Phd thesis, Caltech, June 2013. 12

[DO07] Demaine E. D., O'Rourke J.: *Geometric Folding Algorithms: Linkages, Origami, Polyhedra*. Cambridge University Press, 2007. doi:10.1017/CBO9780511735172. 1, 2, 3, 12

[DT17] Demaine E. D., Tachi T.: Origamizer: A practical algorithm for folding any polyhedron. In *International Symposium on Computational Geometry* (2017), vol. 77, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 34:1–16. doi:10.4230/LIPIcs.SoCG.2017.34. 2

[F48] Fáry I.: On straight line representation of planar graphs. *Acta Universitatis Szegediensis Sectio Scientarium Mathematicarum 11*, 4 (1948), 229–233. 2

[GBKK98] Gupta S. K., Bourne D. A., Kim K. H., Krishnan S.: Automated process planning for sheet metal bending operations. *Journal of Manufacturing Systems 17*, 5 (1998), 338–360. doi:10.1016/S0278-6125(98)80002-2. 1

[Glo86] Glover F.: Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research 13*, 5 (1986), 533–549. doi:10.1016/0305-0548(86)90048-1. 3

[HE12] Haenselmann T., Effelsberg W.: Optimal strategies for creating paper models from 3d objects. *Multimedia Systems 18*, 6 (November 2012), 519–532. doi:10.1007/s00530-012-0273-1. 2

[HG00] Hormann K., Greiner G.: MIPS: An efficient global parametrization method. In *Curve and Surface Design: Saint-Malo 1999*, Innovations in Applied Mathematics. Vanderbilt University Press, 2000, pp. 153–162. 2

[HHL19] Hernandez E. A. P., Hartl D. J., Lagoudas D. C.: *Active Origami*. Springer, 2019. doi:10.1007/978-3-319-91866-2. 2

[HLS07] Hormann K., Lévy B., Sheffer A.: Mesh parameterization: Theory and practice. In *ACM SIGGRAPH 2007 Courses* (2007), Association for Computing Machinery, p. 1–es. doi:10.1145/1281500.1281510. 2

[KSBC12] Kazhdan M., Solomon J., Ben-Chen M.: Can mean-curvature flow be modified to be non-singular? *Computer Graphics Forum 31*, 5 (August 2012), 1745–1754. doi:10.1111/j.1467-8659.2012.03179.x. 5

[KSS06] Kharevych L., Springborn B., Schröder P.: Discrete conformal mappings via circle patterns. *ACM Transactions on Graphics 25*, 2 (April 2006), 412–438. doi:10.1145/1138450.1138461. 2

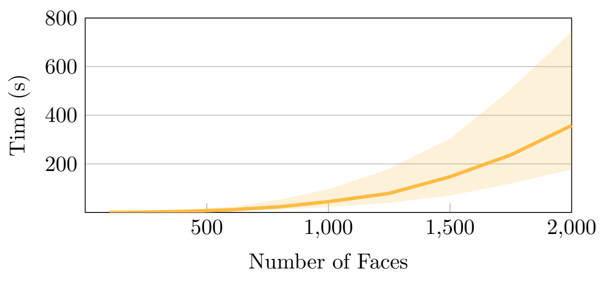
[KTGW20] Korpitsch T., Takahashi S., Gröller E., Wu H.-Y.: Simulated annealing to unfold 3d meshes and assign glue tabs. *Journal of WSCG 28*, 1-2 (2020), 47–56. doi:10.24132/JWSCG.2020.28.6. 2

[KXL17] Kim Y., Xi Z., Lien J.: Disjoint convex shell and its applications in mesh unfolding. *Computer-Aided Design 90* (September 2017), 180–190. doi:10.1016/j.cad.2017.05.014. 2

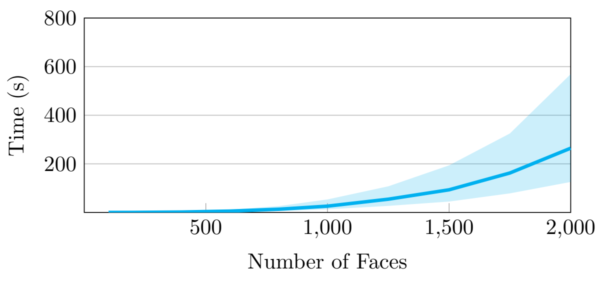
[MG80] Mitchell A. R., Griffiths D. F.: *The Finite Difference Method in Partial Differential Equations*. Wiley, 1980. 6

[MS04] Mitani J., Suzuki H.: Making papercraft toys from meshes using strip-based approximate unfolding. *ACM Transactions on Graphics 23*, 3 (August 2004), 259–263. doi:10.1145/1015706.1015711. 2

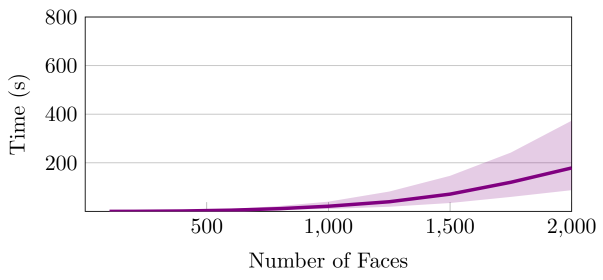
[SA07] Sorkine O., Alexa M.: As-rigid-as-possible surface modeling. In *Proceedings of EUROGRAPHICS/ACM SIGGRAPH Symposium on Geometry Processing* (2007), pp. 109–116. doi:10.2312/SGP/SGP07/109-116. 5



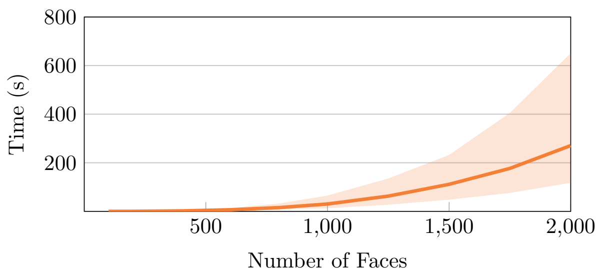
(a) *CCF0*



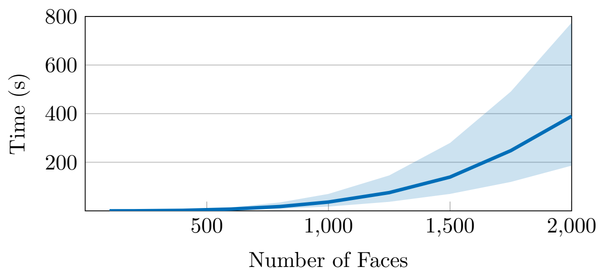
(b) *CCF1*



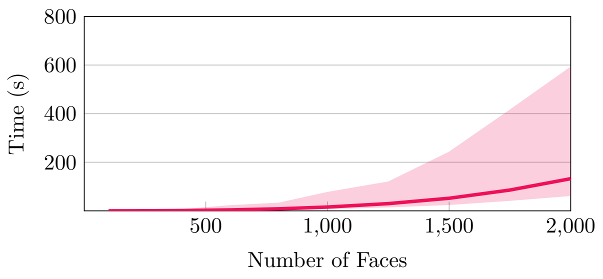
(c) *CCF2*



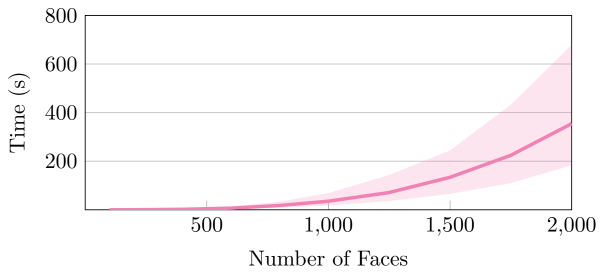
(d) *cMCF*



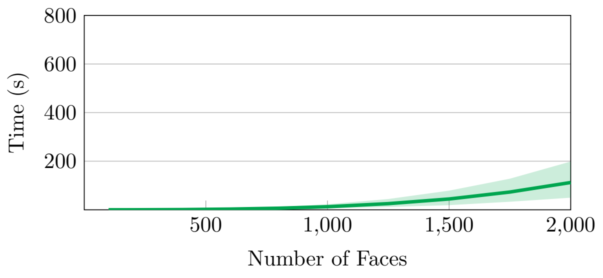
(e) *IIF*



(f) *ENAF*



(g) *AEF*



(h) *TU*

Figure 22: *A side-by-side comparison of the timings for all flavors of our algorithm, and Tabu Unfolding, including distribution-indicators. The line for each graph is the median time, the area around the line marks the value at 25% and 75% of the recorded timings.*

[Sch97] SCHLICKENRIEDER W.: *Nets of Polyhedra*. Diploma thesis, Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, 1997. 3

[SGC18] STEIN O., GRINSPUN E., CRANE K.: Developability of triangle meshes. *ACM Transactions on Graphics 37*, 4 (July 2018), 77:1–14. doi:10.1145/3197517.3201303. 2

[SLMB05] SHEFFER A., LÉVY B., MOGILNITSKY M., BOGOMYAKOV A.: Abf++: Fast and robust angle based flattening. *ACM Transactions on Graphics 24*, 2 (April 2005), 311–330. doi:10.1145/1061347.1061354. 2

[SP11] STRAUB R., PRAUTZSCH H.: Creating optimized cut-out sheets for paper models from meshes. *Karlsruhe Reports in Informatics 36* (2011), 1–15. doi:10.5445/IR/1000025577. 2

[SSGH01] SANDER P. V., SNYDER J., GORTLER S. J., HOPPE H.: Texture mapping progressive meshes. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (August 2001), Association for Computing Machinery, pp. 409–416. doi:10.1145/383259.383307. 5

[STL06] SHATZ I., TAL A., LEIFMAN G.: Paper craft models from meshes. *The Visual Computer 22*, 9 (September 2006), 825–834. doi:10.1007/s00371-006-0067-6. 2

[Tac09] TACHI T.: Origamizing polyhedral surfaces. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (2009), 298–311. doi:10.1109/TVCG.2009.67. 2

[TWS*11] TAKAHASHI S., WU H.-Y., SAW S. H., LIN C.-C., YEN H.-C.: Optimized topological surgery for unfolding 3d meshes. *Computer Graphics Forum 30*, 7 (2011), 2077–2086. doi:10.1111/j.1467-8659.2011.02053.x. 2

[XKKL16] XI Z., KIM Y.-H., KIM Y. J., LIEN J.-M.: Learning to segment and unfold polyhedral mesh from failures. *Computers & Graphics 58*, C (August 2016), 139–149. doi:10.1016/j.cag.2016.05.022. 2

[XL17] XI Z., LIEN J.-M.: Polyhedra fabrication through mesh convexification: A study of foldability of nearly convex shapes. In *Volume 5B: 41st Mechanisms and Robotics Conference* (August 2017). doi:10.1115/DETC2017-67212. 2

[Zaw23] ZAWALLICH L.: *Unfolding Polyhedra via Tabu Search*. Tech. rep., University of Zurich, 2023. 2, 3, 7, 15

[ZJ16] ZHOU Q., JACOBSON A.: Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797* (July 2016). doi:10.48550/arXiv.1605.04797. 8

[ZS00] ZHOU T., SHIMADA K.: An angle-based approach to two-dimensional mesh smoothing. In *Proceedings of the 9th International Meshing Roundtable* (October 2000), pp. 373–384. 6

Appendix A: Pseudo-Code

Algorithm 1 Tabu Search for Unfolding Polyhedra

```
Require: $T$                      ▷ $T$ is the transformer, applying a flow
Require: $S$        ▷ $S$ is the 2D overlap solver, i.e. the Tabu Unfolder
Require: $U$                                      ▷ $U$ is some unfolding
  function SURFACEFLOWUNFOLDING($T, S, U$)
    $H \leftarrow stack()$
    $t \leftarrow T.getMinimalStepSize()$
    while $!T.isConverged()$ do                              ▷ Section 4.2
      $T.performStep(t)$
      $H.push(T.currentVertices())$
      $t \leftarrow \min(2t, T.getMaximalStepSize())$
    end while
    $U \leftarrow createInitialUnfolding(H.pop())$           ▷ Section 4.3
    if $!S.resolveOverlaps(U)$ then                             ▷ [Zaw23]
      return $-1$                                           ▷ Failed case
    end if
    while $!H.isEmpty()$ do                                  ▷ Section 4.4
      $U' \leftarrow U$
      $U.setVertices(H.pop())$
      if $!S.resolveOverlaps(U)$ then                           ▷ [Zaw23]
        $U \leftarrow U'$                            ▷ Restore last valid state
        return 1                                 ▷ Approximative result
      end if
    end while
    return 0                                                  ▷ Success
  end function
```

Appendix B: Success Rates

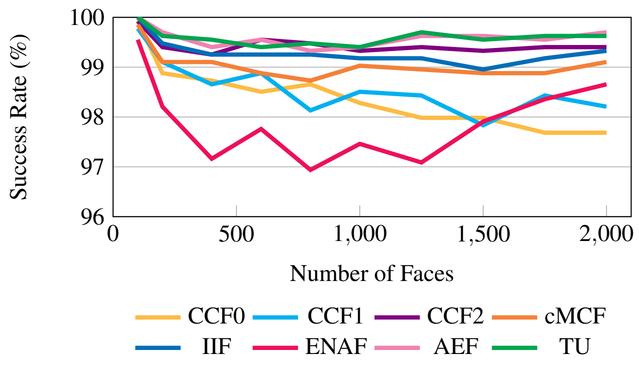


Figure 23: *The success rates of our algorithm in all variants compared to Tabu Unfolding excluding approximative results.*

Appendix C: Detailed Results

Value	|F|	CCF0	CCF1	CCF2	cMCF	IIF	ENAF	AEF	TU
Min Time (s)	all	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Median Time (s)	100	0.084	0.023	0.033	0.026	0.021	0.025	0.031	0.011
	200	0.658	0.183	0.224	0.218	0.205	0.173	0.223	0.110
	400	4.021	1.637	1.610	2.037	1.943	1.265	1.992	0.874
	600	10.894	5.518	4.994	6.678	7.250	3.823	6.841	2.959
	800	23.362	13.118	11.232	15.523	17.623	8.338	17.844	6.578
	1000	44.314	25.879	21.207	30.417	36.217	15.605	35.331	12.586
	1250	78.464	54.462	39.680	62.847	75.035	29.738	71.136	25.345
	1500	146.800	93.135	71.005	111.389	139.063	51.772	133.746	44.485
	1750	236.749	162.307	119.841	176.871	247.880	85.581	224.284	73.079
	2000	357.827	264.432	177.698	270.761	389.460	132.324	354.654	112.325
Mean Time (s)	100	0.271	0.083	0.083	0.058	0.050	0.050	0.056	0.022
	200	2.419	0.720	0.557	0.690	0.615	0.527	0.738	0.252
	400	17.641	6.335	4.181	6.557	5.913	3.717	5.848	2.029
	600	39.680	24.011	11.435	18.355	19.417	12.259	18.345	4.918
	800	101.832	65.042	33.623	51.343	43.395	20.572	42.791	12.210
	1000	207.592	110.041	49.602	102.792	87.162	42.770	90.118	24.838
	1250	377.937	254.690	91.107	185.970	188.924	73.979	169.916	51.944
	1500	690.486	442.203	186.203	313.842	340.770	138.707	306.974	83.320
	1750	1052.158	821.936	293.300	557.080	651.608	237.562	508.521	149.971
	2000	1619.280	1462.094	392.168	859.284	916.174	349.601	808.549	216.275
Max Time (s)	100	9.343	7.597	4.298	3.919	3.758	1.694	3.442	0.847
	200	79.252	47.120	32.263	79.282	56.387	13.023	64.868	23.238
	400	597.214	402.811	447.347	517.408	333.222	126.505	591.949	189.075
	600	1673.838	2079.016	1064.863	1078.352	1890.658	403.559	2321.188	325.489
	800	5110.841	4927.781	4397.848	5020.113	3183.697	935.753	4266.497	838.617
	1000	8605.563	7648.411	8071.711	8054.064	8731.906	1881.454	9524.942	2064.167
	1250	18270.135	15895.157	10796.006	11891.373	10738.786	3397.558	17670.767	6974.261
	1500	31437.614	23212.025	29255.130	28491.302	31639.702	6196.127	27684.283	5967.156
	1750	50000.308	51725.592	37344.659	51444.273	38839.597	9984.051	28138.796	9237.242
	2000	69236.818	78211.479	30525.573	58148.552	43429.453	15665.376	45195.883	10896.299
Success Rate (%)	100	99.925	99.776	99.925	99.851	100.000	99.552	100.000	100.000
	200	98.880	99.104	99.403	99.104	99.477	98.208	99.701	99.627
	400	98.730	98.657	99.253	99.104	99.253	97.162	99.403	99.552
	600	98.506	98.880	99.627	98.880	99.253	97.760	99.552	99.403
	800	98.656	98.134	99.477	98.730	99.253	96.938	99.328	99.477
	1000	98.282	98.506	99.627	99.029	99.178	97.461	99.403	99.403
	1250	97.984	98.432	99.627	98.954	99.178	97.087	99.627	99.701
	1500	97.984	97.834	99.552	98.880	98.954	97.908	99.627	99.552
	1750	97.685	98.432	99.552	98.880	99.178	98.357	99.552	99.627
	2000	97.685	98.208	99.552	99.104	99.328	98.656	99.701	99.627
Success Rate Approx (%)	100	99.925	99.851	99.925	99.925	100.000	99.925	100.000	100.000
	200	99.104	99.403	99.403	99.328	99.701	98.730	99.776	99.627
	400	99.178	99.029	99.253	99.178	99.627	97.984	99.701	99.552
	600	98.954	99.104	99.627	99.253	99.851	98.282	99.627	99.403
	800	98.880	98.656	99.477	99.029	99.776	97.909	99.627	99.477
	1000	98.581	98.581	99.627	99.253	99.851	98.432	99.627	99.403
	1250	98.357	98.805	99.627	99.178	99.552	97.386	99.776	99.701
	1500	98.208	98.133	99.552	99.178	99.552	97.984	99.851	99.552
	1750	98.357	98.880	99.552	99.104	99.447	98.581	99.925	99.627
	2000	98.282	98.581	99.552	99.328	99.447	98.880	99.925	99.627

Table 2: *Detailed timings and success rates for all flavors of our algorithm, as well as Tabu Unfolding.*