Fabricable Tile Decors

WEIKAI CHEN*, The University of Hong Kong, USC Institute for Creative Technologies
YUEXIN MA*, The University of Hong Kong
SYLVAIN LEFEBVRE, INRIA
SHIQING XIN, Shandong University
JONÀS MARTÍNEZ, INRIA
WENPING WANG, The University of Hong Kong

Fig. 1. Given a target surface and a set of user-specified tiles (top left), our method produces a dedicated packing of tiles that is optimized for fabrication (bottom left). The results are printed as independent flat patches (second column) with integrated hinges and snap-fit joints. The patches are folded and assembled into the final object (third column); in this case a functional handbag that can carry light objects.

Recent advances in 3D printing have made it easier to manufacture customized objects by ordinary users in an affordable manner, and therefore spurred high demand for more accessible methods for designing and fabricating 3D objects of various shapes and functionalities. In this paper we present a novel approach to model and fabricate surface-like objects composed of connected tiles, which can be used as objects in daily life, such as ornaments, covers, shades or handbags.

Our method is designed to maximize the efficiency and ease of fabrication. Given a base surface and a set of tile elements as user input, our method generates a tight packing of connected tiles on the surface. We apply an efficient and tailored optimization scheme to pack the tiles on the base surface with fabrication constraints. Then, to facilitate the fabrication process, we use a novel method based on minimal spanning tree to decompose the set of connected tiles into several connected patches. Each patch is articulated and can be developed into a plane. This allows printing with an inexpensive FDM printing process without requiring any supporting structures, which are often troublesome to remove. Finally, the separately printed patches are reassembled to form the final physical object, a shell surface composed of connected user-specified tiles that take the shape of the input base surface. We demonstrate the utility of our method by modeling and fabricating a variety of objects, from simple decorative spheres to moderately complex surfaces, such as a handbag and a teddy bear. Several user controls are available, to distribute different type of tiles over the surface and locally change their scales and orientations.

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

Additional Key Words and Phrases: fabrication, tiles, packing, foldable.

ACM Reference format:
Weikai Chen, Yuexin Ma, Sylvain Lefebvre, Shiqing Xin, Jonàs Martínez, and Wenping Wang. 2017. Fabricable Tile Decors. *ACM Trans. Graph.* 36, 6, Article 175 (November 2017), 15 pages.
https://doi.org/10.1145/3130800.3130817

*Joint first author

1 INTRODUCTION

Additive manufacturing is paving the way to mass customization, enabling ordinary users to create their own version of a base product and customizing it in creative and beautiful ways. With the increasing availability of 3D printers in FabLabs and homes, users can fabricate their customized objects directly in a wide variety of colors and materials. However, design customization remains a difficult task for non-expert users. Hence, there has been significant research dedicated to providing computational support for customization of shapes, materials, and functionality of 3D objects. For instance, researchers have focused on shape optimization for balancing [Prévost et al. 2013], creating spinnable objects [Bächer et al. 2014], turning surfaces into physical filigrees [Chen et al. 2016], designing wind instruments [Li et al. 2016; Umetani et al.

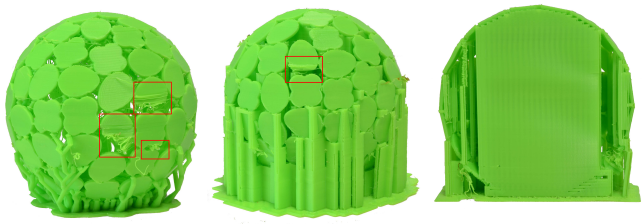
2016], lampshades [Zhao et al. 2016] and helping users maintain fabricability during modifications [Shugrina et al. 2015].

In this paper we study the modeling and fabrication of customized decoration patterns obtained by packing together distinct decorative elements, or *tiles*. Such patterns are ubiquitous in fashion and design; hobbyists often create them by juxtaposing existing decorations, e.g. stickers or decals. Our goal is to enable non-expert users to digitally model such decoration on a base surface with a set of flat tiles. Taken together, these tiles are joined by hinges and connectors to form an articulated object that takes the shape of the input surface and can be fabricated using a home printer (see Figure 1).

This is a challenging problem as the tiles must be closely packed and properly connected to each other in order to define a shell-like object. Furthermore, fabrication is made difficult by the hollow nature of the shape. Printing directly with a filament printer requires a large amount of support structures. Besides wasting time and material, the supports are particularly difficult to remove as they end up enclosed within the shape itself (see Figure 2). Even on a high-end powder-based system (e.g. SLS), the part occupies a large volume while using little material. It is thus not very efficient to print, as only few objects can be made in a single batch. Therefore, there is no easy option to efficiently print such objects on widely accessible printers.

Our approach directly takes fabrication constraints into account. The constituent tiles of the synthesized object are joined by hinges instead of rigid connectors. This allows neighboring tiles to rotate with respect to one another. Instead of producing a single final object for printing, we propose to decompose the synthesized shell-like object into a small number of connected tiles subsets, which we call *patches*. We carefully place the hinges to ensure that the patches are developable, and output each patch in its developed (flat) configuration. This has a major advantage: all the tiles of a patch as well as their connecting hinges can be printed *flat* on a low-cost filament printer, in a single print session. No support is required. After printing, all patches are folded and assembled back together, using *snap-fit* connectors printed along their boundaries. The result, locked in its final position, is the desired shell-like object.

The main contributions of this paper are:



(a) Tree supports (b) Standard supports (c) Inner view of (b)

Fig. 2. Directly printing tile decors on a filament printer requires support. (a) Tree support structure. Several tiles are incorrectly printed as highlighted within the red rectangles. (b) Standard, downward extrusion support structure. (c) Inner view of (b). A large amount of support is required, and it is not easily accessible for removal.

- A new approach to fabricate shell-like flexible objects composed of user-specified tiles in an affordable and easy way, using a standard filament printer. Such objects can be used as objects in daily life, such as ornaments, covers, handbags, etc. (see Figures 1, 19).
- A novel packing algorithm on surfaces that is specifically tailored to take into account connectivity constraints and fabrication considerations.
- A new segmentation method that decomposes the network of connected tiles into a small number of developable patches in order to facilitate the fabrication of these patches without supports on a standard filament printer.

2 PREVIOUS WORK

Art and design. As early adopters of additive manufacturing, artists are constantly pushing the boundaries of fashion and design. Among the many examples of this trend, *Nervous System* [Rosenkrantz and Louis-Rosenberg 2007] has produced beautiful flexible designs made of many interconnected triangles – the resulting objects are bracelets, necklaces or even full dresses. The flexibility is obtained by placing hinges in between the triangles of a densely tessellated initial flat surface. As a result, the designs can be printed flat, pre-assembled, on low-cost printers.

While this is a key inspiration for us – in particular showing the importance of taking into account printability concerns in the design itself – our goal is different. We seek to produce 3D structures made of packed, arbitrary shaped tiles. We do not have a base flat contour to work from, and we approximate a target 3D surface.

Pattern synthesis. Several recent works consider the problem of decorating surfaces to turn them into printable objects, with the same purpose of customization for 3D printing. Zhou et al. [2014] synthesize connected patterns along curves, and use their approach to model fabricable 3D objects. Dumas et al. [2015] modify texture synthesis to account for structure and rigidity, and synthesize printable 3D patterns that cover a base surface. Chen et al. [2016] synthesize filigree-like structures along surfaces. Their approach is close to ours since the input is a set of base elements and a target surface. However, they relax the packing problem by allowing appearance-preserving overlaps between elements. This produces appealing patterns, but cannot guarantee their base shape is preserved, and thus forbids the use of patterns with a semantic meaning (e.g. a fish, a heart, a letter). Zehnder et al. [2016] explore the interactive design of curve networks onto surfaces. The user positions curve elements (visually similar to a bent wire) onto a surface. The curves are simulated as elastic rods, giving a very natural feel to their deformations, while intersections are disallowed to preserve their appearance. Tight packings are achieved thanks to deformations. This approach produces beautiful, airy curve networks that can be fabricated on high-end printers.

All the aforementioned techniques output complex yet hollow 3D geometries that are challenging to print. In contrast, our approach strives to produce an easy and efficient to print output.

Packing onto surfaces. The packing of shapes into the plane is an important topic of research with many industrial applications (e.g. textile). However, packing on surfaces has been less explored. Lai et

al. [2006] and Dos Passos et al. [2009; 2008] proposed methods for creating mosaics on surfaces using convex planar tiles. Hu et al. [2016] recently proposed an approach for surface mosaics that supports *irregular* planar tiles. It operates through iterated continuous and combinatorial steps. Ma et al.[2011] and Roveri et al.[2015] consider a data-driven synthesis of packings from exemplar elements. Schumacher et al. [2016] present stenciling, a complementary work to ours, which brings together element packing and structural optimization in an elegant, joint optimization. Stenciling approximates each tile as a circle in a packing objective, making it solvable by a gradient-based method. That is however not ideal in our scenario as we seek for dense packing. The approximation would lead to difficulties for joint placement along concave tiles, as the actual tile boundary differs significantly from a circle. Other techniques consider pattern distributions by locally mapping 2D elements (i.e. *decals*) onto surfaces, e.g. [Lefebvre et al. 2005; Schmidt et al. 2006; Wang et al. 2016a]. These are less suited to our needs as we seek to preserve the planarity of the packed elements.

In this work we use the approach of Hu et al. [2016] as a comparison baseline regarding packing quality; please refer to supplemental material. It is worth noting, however, that the problem settings are different. The aforementioned techniques are geared towards mosaicking, where large numbers of relatively small tiles are placed. We instead have to use fewer tiles, large with respect to object curvature, to accommodate for fabrication constraints.

Fabrication from sheets and wires. Several recent works focus on helping users fabricate shapes from simpler materials. Miguel et al. [2016] optimize wire sculptures that approximate input surfaces. Iarussi et al. [2015] present an approach for wire jewelry design. Other approaches consider fabrication from paper or similar planar sheets. The original surface is unfolded into charts [Shatz et al. 2006] or strips [Mitani and Suzuki 2004; Takezawa et al. 2016] which can be cut, folded and assembled to approximate the input. Fabrication from flat surfaces has also been studied for modeling large objects from laser cut wood pieces [Cignoni et al. 2014], for modeling tight-fit cloths from 3D scanners by automatic flattening [Zhang et al. 2016], to fabricate surfaces that self-fold under the action of heat [Kwok et al. 2015], to fabricate iris folding patterns [Igarashi et al. 2016] and to fabricate wire mesh sheets [Garg et al. 2014] and auxetic planar materials [Konaković et al. 2016].

While our approach shares the idea of assembling from planar patches, our problem setting is very different as we consider networks of rigid planar tiles along the surface, and exploit 3D printed hinges to fold patches into shape.

3D printing large objects. Solutions have been proposed for printing large objects, which either do not fit the printer, or occupy a large volume compared to their material use. A first set of methods decompose a shape into smaller parts, for later assembly [Luo et al. 2012]. Other methods additionally consider how to pack the parts together for printing [Attene 2015; Chen et al. 2015; Vanek et al. 2014]. Wang et al. [2016b] decompose objects into smaller parts for maximizing print quality, changing the print direction of each subpart. Finally, Song et al. [2016] decompose large objects into small pieces that are 3D printed and then fixed to an internal, laser-cut structure.

The tile packings generated by our approach are not an ideal input for the aforementioned methods, as cross sections are thin everywhere. By proposing a method specifically tailored to our outputs, we are able to introduce rotational degrees of freedoms between tiles and can fully unfold large patches, maximizing print efficiency.

3 OVERVIEW

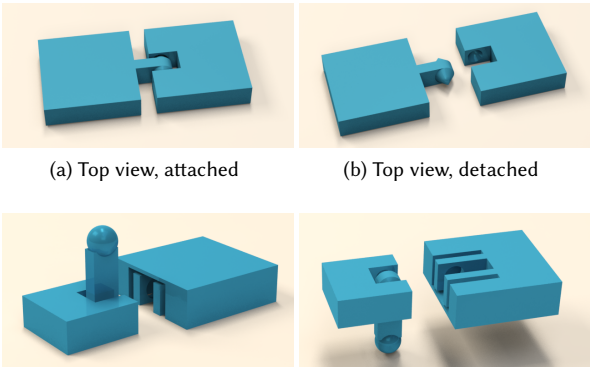
Our approach starts from a set of user-specified tiles, and a target surface represented by a triangular mesh $\mathcal{S}$. The output is a set of *foldable patches* that (1) can be printed flat without support and (2) are made of tiles interconnected by hinges with a rotational degree of freedom. Each patch can be folded and assembled with other patches in order to approximate $\mathcal{S}$ (see Figure 1). We refer to the surface assembled from the synthesized tile patches as the *tile network* (see Figure 11a). The tile network is optimized to approximate the target surface. The user can optionally specify a varying tile scale and orientation field along the surface, as well as choose different tiles in different parts of the surface. Note that our objective is to provide an efficient approach for fabricating tile decors that can stand on their own (e.g. bag, vase, teddy bear), as well as form covers over objects (lampshade). We have incorporated a simple physical simulation that considers both gravity and common pinch forces (Section 5.3.1), but the tile decors in our case are indeed not appropriate to carry large loads. Nevertheless, our experiments have shown that the printed results maintain their shape in daily use (Fig. 19).

Fabrication Constraints. In order to connect adjacent tiles, parts of the tiles need to be carved out so as to insert joints (see Figure 3). However, the tiles may contain narrow features and concavities. As illustrated in Figure 4, we need to take care to position the tiles such that they face each others along sufficiently large areas, allowing for creating cavities to insert joints. In addition, a tile typically has multiple connectors and we have to avoid conflicts – overlaps – between different cavities (see Figure 14). Our approach does not support tiles that are thin everywhere, e.g. thin wire–like structures, as they offer no space for connectors.

3.1 Pipeline overview

Our framework consists of two main steps: *tile packing* and *patch extraction*. During tile packing, tiles are densely packed over the base surface. The tiles are kept aligned with the tangent plane passing through their centroid so that the result covers the base surface, which is convenient to produce tile networks that cover a given object.

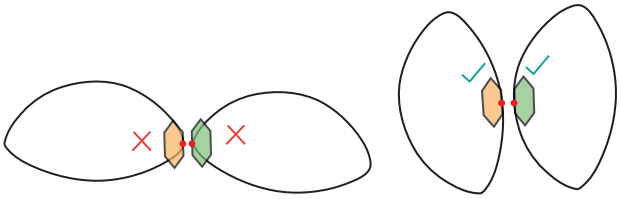
Unlike conventional packing methods which solely focus on maximizing surface coverage [Hu et al. 2016], we seek to produce a dense packing of irregular tiles that 1) satisfies fabrication constraints and 2) follows the target surface as closely as possible. This results in a complex optimization problem, with both non smooth and combinatorial aspects (number of tiles, interlocking of concavities). We therefore propose a dedicated optimization that can jointly optimize the positions, scales and orientations of the tiles in order to enforce fabrication constraints and approximate the base surface. We detail the tile packing in Section 4.



(a) Top view, attached (b) Top view, detached

(c) Top view (d) Bottom view

Fig. 3. (a,b) Hinge joint. Note that (b) is only for showing the inner structure of hinge joint as it is always printed attached. (c,d) Snap-fit joint. Note that for those snap-fit joints that are internal to a patch, their bar will be printed vertically but would be folded down to connect adjacent tile during assembly.



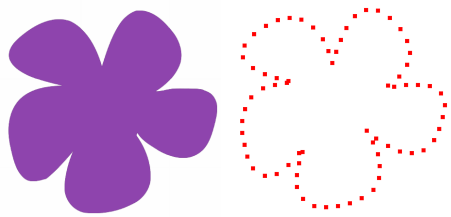
(a) Failure case (b) Correct configuration

Fig. 4. Illustration of fabrication constraints. The nearest points between two neighboring tiles will host the joint. (a) Failure case. Neither of the hosting points has sufficient space to be curved out in order to insert a joint. (b) Correct configuration. A joint can be safely inserted as either of the hosting points can provide sufficient cutting area.

After packing, we divide the tile network into several flat, foldable patches. We resort on two types of joints for connecting the tiles: hinge joints (Figure 3a), which allow rotations between two connected tiles, keeping their inter-distance constant; and snap-fit hinge joints (Figure 3c), which are printed disconnected and later assembled to connect tiles within and across patches. For the sake of clarity, we refer to the later type of joints simply as snap-fits. After assembly, hinges and snap-fits impose distance constraints between the tiles, locking the assembled patches into a stable surface in most cases. The details of patch extraction are discussed in Section 5.

4 TILE PACKING

In this section, we detail our tailored optimization tile packing algorithm. Dense packings of irregular flat tiles (mosaicking) cannot be optimized via traditional gradient based approaches [Hu et al. 2016]. Compared to mosaicking, we also face additional constraints: fabrication imposes the use of fewer tiles that are relatively large



(a) Input tile (b) Sampled points

Fig. 5. The tiles are represented by samples along their contour.

compared to surface curvature (this stems from limitations in minimal printable feature size and maximum object size). This makes packing even more difficult, especially as the tiles remain planar and rigid. We also have to ensure that the tiles are neighboring in a way that allows the insertion of connectors.

Our approach is based on a two-phase *attract-and-repulse* mechanism that iteratively refines an initial layout towards our goal. Figure 6 illustrates the idea behind the proposed mechanism. (1) We first initialize the distribution of tiles by enforcing that each newly placed tile has a limited overlap with existing ones. (2) During the attraction phase, tiles are attracting each other to form near-uniform overlaps. The fabrication constraints are considered during optimization. After tile attraction, empty spaces may appear providing room for adding new tiles. This progressively increases surface coverage (Figure 6c). The attraction phase iterates until no more tiles can be added. (3) The repulsion phase resolves remaining tile overlaps by encouraging tiles to repel their neighbors. This is achieved by adaptively scaling tile sizes and optimizing both tile placement and orientation. We stop the repulsion phase when a uniform inter-spacing among tiles is achieved.

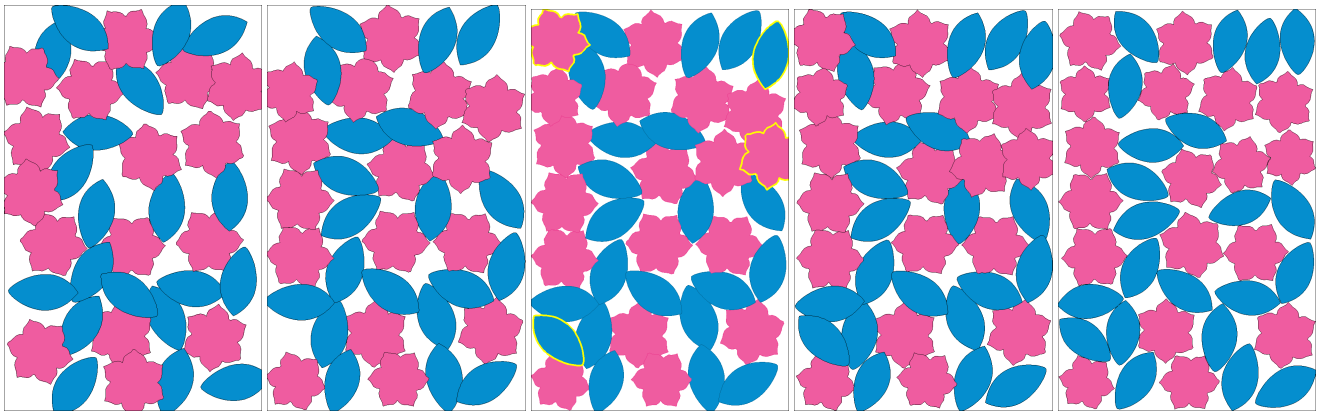
4.1 Tile representation

We input tiles as 2D closed boundary polygons in the XY plane, with the origin at their centroid. During optimization, we only consider a sampling of the contours, as shown in Figure 5b. We use two different samplings for performance reasons: the finest is a sampling with a spacing of 2 mm and the coarsest is a sampling with a constant number of 20 samples per tile. The 3D geometry of the tiles is reconstructed during post-processing. The tiles may only be uniformly scaled during optimization, and are otherwise kept unmodified.

4.2 Initialization

We initialize the tile distribution following multi-class blue noise sampling [Wei 2010]. We made this choice as it provides a reasonably good initialization while accommodating for simple controls regarding mixing different tiles and considering overlaps.

New tiles are placed around the boundary of existing tiles, following a dart-throwing procedure. We accept a new tile position if it satisfies both of the following criteria: (1) it slightly overlaps with existing tiles; and (2) the overlapping ratio with each existing tile is below a given threshold. This encourages a dense initialization –



(a) Initialization (b) Tile attraction (c) Add new tiles (d) Tile attraction again (e) Repulsion phase

Fig. 6. Tile packing algorithm. (a) Initialization. **(b-d) Attraction Phase**. During the attraction phase, we encourage each tile to stay close to neighbors while maintaining near-uniform overlaps (b). This strategy makes room for adding new tiles so as to increase surface coverage (c). In this case, four new tiles (highlighted with yellow) are added in now empty regions. (d) We iterate until no more tiles can be added. **(d) Repulsion phase**. We encourage each tile to repel its neighbors to resolve any remaining overlap. This is achieved via adaptively scaling the tile sizes and optimizing both the tile positions and orientations.

overlaps are resolved later. This process is repeated until a maximum number of trials has been reached (1500 in our implementation).

If no control field is provided, i.e. orientation is free, we optimize the orientation of the newly positioned tile so as to increase the possibility of placing hinges between neighbors, as will be detailed in Section 4.3.2. If, however, a control field is provided, the new tile is scaled and orientated as dictated by the control field at the tile insertion position. The tile position is then immediately optimized to maximize its hinge area, as detailed in Section 4.3.1. Immediately optimizing inserted tiles limits the creation of bad local cases for the following global optimization steps.

Using different tiles. When using different tiles, we found it necessary to explicitly encourage mixing. Specifically, for tiles belonging to different classes the overlap area should not exceed 20% of the area of the tiles, while the threshold is reduced to 5% for tiles of the same class. This encourages tiles from different classes to be neighbors while prohibiting tiles of a same class from staying too close, which is demonstrated in Figure 6a.

4.3 Attraction phase

The attraction phase distributes and optimizes the position of the tiles to bring them closer together. The optimization variables are the number of tiles, the positions of their centroids along the surface, their sizes (uniform scaling), and their orientations (angle around normal).

4.3.1 Objective function. The objective function is a combination of three terms, designed to encourage the tiles to come closer together (neighborhood distance) while fitting the surface (local surface approximation) and allowing for hinge placement (hinge area).

Neighborhood distance. We define the set of direct neighbors $\mathcal{N}_i$ of a tile T_i as the set of tiles that are overlapping T_i, e.g. $\mathcal{N}_i = \{T_j | T_i \cap T_j \neq \emptyset, j \neq i\}$. The intersection is tested after projecting each tile to a common projection plane, which is orthogonal to the bisector plane of the support planes of T_i and T_j (dashed line in Figure 7).

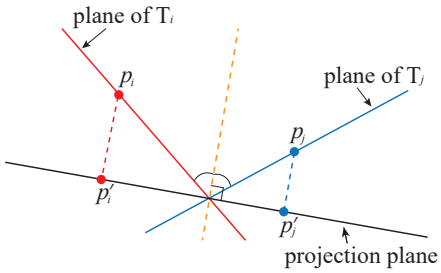


Fig. 7. Common projection plane of T_i and T_j (side view).

We define the neighborhood distance between a tile T_i and its neighborhood $\mathcal{N}_i$ as:

$$\text{dist}(T_i, \mathcal{N}_i) = \max_{p_j \in \mathcal{P}_{\mathcal{N}_i}} (\text{dist}(p_j, T_i)) \quad (1)$$

where $\mathcal{P}_{\mathcal{N}_i}$ is the point set formed by the union of all the sample points of the tiles in $\mathcal{N}_i$ that lie inside T_i (green dots in Figure 8). $\text{dist}(p_j, T_i)$ returns the shortest distance of point p_j to the boundary of T_i. Since T_i and the tiles in $\mathcal{N}_i$ are not coplanar, we project the points in $\mathcal{P}_{\mathcal{N}_i}$ onto the plane of T_i to perform all the distance computations. $\text{dist}(T_i, \mathcal{N}_i)$ evaluates to a large value if $\mathcal{N}_i = \emptyset$ (i.e. T_i has no overlaps). Intuitively, minimizing $\text{dist}(T_i, \mathcal{N}_i)$ encourages T_i to be attracted to adjacent tiles while preserving a minimum overlap with each neighbor when tiles come together.

Local surface approximation. The tiles are aligned with the tangent plane of the base surface $\mathcal{S}$ at their centroid. In high-curvature

regions this quickly leads to large deviations from $\mathcal{S}$. We penalize such configurations by defining the following approximation error for a tile T_i:

$$\text{approx}(T_i, \mathcal{S}) = \max_{p_i \in T_i} (\text{dist}(p_i, \mathcal{S})) \quad (2)$$

where $\text{dist}(p_i, \mathcal{S})$ returns the (Euclidean) distance from point p_i to the closest point on $\mathcal{S}$. Low values of $\text{approx}(T_i, \mathcal{S})$ indicate a better local approximation of $\mathcal{S}$ by T_i.

Hinge area. The tile positions and orientations influence where hinges can be added between neighboring tiles. We measure the capacity of holding a hinge by using the *local shape thickness* – a notion similar to the shape diameter defined in [Shapira et al. 2008]. For a 2D shape, we define the shape thickness at a boundary point p as the diameter of the maximal ball centered at the medial axis and tangential to p (as shown in Figure 9a). Larger values indicate that it is more likely that an hinge can be placed at this location.

Whenever considering adjacent tiles (overlapped or not), we would like them to be translated or oriented such that the potential connection points have large local shape thicknesses. For the non-overlapping tiles (Figure 9a), the connection points to place an hinge are the two nearest points between the tile contours. As tiles may be overlapping during optimization, we predict the connection points of such cases as the middle point of the curve segment contained in the other tile (Figure 9b). We denote the hinge area for a tile T_i as $\Theta(T_i, \mathcal{N}_i)$: it is computed as the sum of the local shape thicknesses for all potential contact points between T_i and its neighbors.

Objective function. We finally define the global attraction objective function $E_{attract}$:

$$E_{attract} = \alpha \sum_i \text{dist}(T_i, \mathcal{N}_i) + \beta \sum_i \text{approx}(T_i, \mathcal{S}) - \Theta(T_i, \mathcal{N}_i) \quad (3)$$

$E_{attract}$ is a weighted sum of the neighboring distance, local approximation and hinge area terms, where α and β are controlling the tradeoff. We set $\alpha = 1.0$ and $\beta = 1.5$ as we found this achieved a good tradeoff between packing performance and local approximation. β is set larger than α to penalize positions in high-curvature regions.

During the attraction phase, our goal is to find a tile configuration with the lowest value of $E_{attract}$.

4.3.2 Minimization. $E_{attract}$ is a complex objective function with a mixture of continuous and combinatorial terms: it cannot be optimized directly by gradient-based methods. In addition, the solution space for $E_{attract}$ contains a vast number of possible configurations, which emerge from combinatorial aspects (number of tiles, interlocking of concavities), and its energy landscape is riddled with local minima. Fortunately, many of these local minima will produce configurations that are good enough to be used in practice, and our algorithm is designed to find them efficiently. We rely on a greedy strategy to locally optimize the configuration of each tile. The pseudo-code for minimization is detailed in Algorithm 1. It takes the following steps:

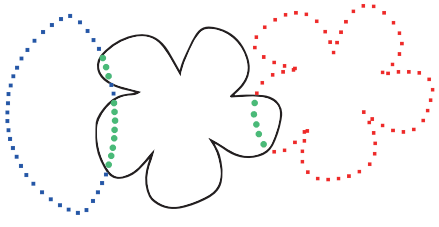
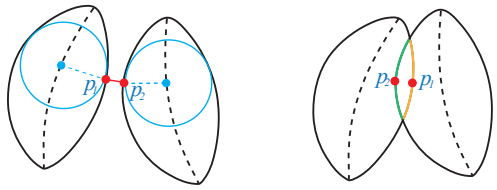


Fig. 8. Samples for the neighborhood distance (green dots).



(a) Non-overlapping tiles (b) Overlapping tiles

Fig. 9. Potential joint positions. (a) Local shape thickness. The medial axis of each shape is outlined by a dashed line. The medial balls at p_1 and p_2 are drawn in blue. (b) In case of overlap p_1 and p_2 are the midpoints of the orange and green segment, respectively.

Position update. In algorithm PositionUpdate each tile is translated to a number of candidate positions (including the current position), searching for a displacement producing a lower value of $E_{attract}$. The candidate positions are randomly sampled within the minimal circle centered on the centroid and fully enclosing the tile. We test 400 positions in our implementation. The position with lowest value is used to update the tile in the current iteration. Note that if the user provided a scale control field, the tile is resized before computing $E_{attract}$ at each tested position.

In regions of high-curvature it is likely that no good update can be found, as the tile plane misaligns with the surface. When all the candidate positions for a tile T_i produce an approximation error $\text{approx}(T_i, \mathcal{S})$ greater than a threshold τ_{approx}, the tile is shrunk at its current position. This favors smaller tiles in high curvature regions. A minimum size constraint prevents tiles from becoming too small. τ_{approx} is determined before starting the attraction phase. The approximation errors of all tiles are computed and sorted. τ_{approx} is the average error of the top 20%.

Angle update. Algorithm AngleUpdate optimizes the orientation of each tile. Each tile is tested with different orientations while its centroid remains fixed. If an orientation field is provided, we only consider ±5 degrees deviations from the prescribed orientation (10 candidate angles); otherwise, we test all rotations with 1 degree stepping (360 candidate angles). Among the candidate rotation angles, we consider the top n with the lowest value of $E_{attract}$ as the candidate orientations ($n = 10$ in our implementation). Among these, we select the one that provides the best opportunity to insert an hinge, that is the angle that maximizes the minimum of local shape thicknesses between the tile and its neighbors.

Updating sequence. We process tiles in parallel, updating the tile positions in independent sets similarly to the mechanism in [Chen et al. 2016].

Increasing the surface coverage. The minimization of $E_{attract}$ leads to a compact packing of existing tiles, uncovering other parts of the surface. Therefore, after position and orientation optimization we add tiles to uncovered regions in AddMoreTiles. This is done similarly to the initialization. The attraction phase stops if no more tiles can be added.

Algorithm 1 Attraction

Input:
Target Surface $\mathcal{S}$; Initial placement of tile set $\mathcal{T}$ on S; Input scale field $\mathcal{F}$ on $\mathcal{S}$.

Output:
Optimized tile configuration $\mathcal{T}_O$ that follows $\mathcal{F}$ on S.

```
repeat
    T ← PositionUpdate(T, F);
    T ← AngleUpdate(T);
    stop ←AddMoreTiles();
until stop
return T_O = T;
```

4.4 Repulsion phase

The second phase of the tile packing focuses on resolving overlaps that remain after the attraction phase. In particular, we adaptively scale each tile according to its distance to adjacent tiles, and encourage repulsion among neighboring tiles to evenly distribute tile inter-spacing. A target gutter size (the gap between the nearest two tiles) d is given as input to the algorithm. Our algorithm iterates until there is no overlap between any two tiles and the minimal gutter distance is larger than or equal to d (1 mm in our implementation).

4.4.1 Objective function. The objective function is designed to push the tiles away from each others. We define a repulsion term between two tiles T_i and T_j:

$$\text{repulse}(T_i, T_j) = \min\{\min_{p_i \in T_i} \text{sdist}(p_i, T_j), \min_{p_j \in T_j} \text{sdist}(p_j, T_i)\} \quad (4)$$

where $\text{sdist}(p_i, T_j)$ is the *signed* closest distance from point p_i to the point set sampled from T_j's contour:

$$\text{sdist}(p_i, T_j) = \begin{cases} -\min_{p_j \in T_j} \|\Gamma(p_i) - \Gamma(p_j)\| & \text{if } p_i \text{ inside } T_j \\ \min_{p_j \in T_j} \|\Gamma(p_i) - \Gamma(p_j)\| & \text{otherwise} \end{cases} \quad (5)$$

where $\Gamma(p)$ projects the point onto the common projection plane of T_i and T_j, as illustrated in Figure 7.

The sign of $\text{repulse}(T_i, T_j)$ indicates whether T_i and T_j overlap with each other (negative values imply an overlap). The absolute value of repulse measures how deeply T_i and T_j penetrate each other if they overlap, or how far they are separated from each other if they do not.

During the repulsion phase we seek to maximize the value of repulse between each tile T_i and its *adjacent neighbors* $\mathcal{A}_i$. These are the neighbors that are spatially close to the current tile and have to be pushed away. Note that this neighborhood is different from the one used during the attraction phase, denoted by $\mathcal{N}_i$ (Section 4.3.1). $\mathcal{A}_i$ is defined as $\mathcal{A}_i = \{T_j | \text{repulse}(T_i, T_j) < \sigma, j \neq i\}$, that is, the set of tiles closer to T_i than a threshold distance σ. This is illustrated in Figure 10 ($\sigma = 6$ mm in our implementation).

We then define the final objective function as:

$$E_{repulse} = \sum_i \min_{T_j \in \mathcal{A}_i} \text{repulse}(T_i, T_j) \quad (6)$$

Maximizing this objective increases the distance between each tile and its adjacent neighbors.

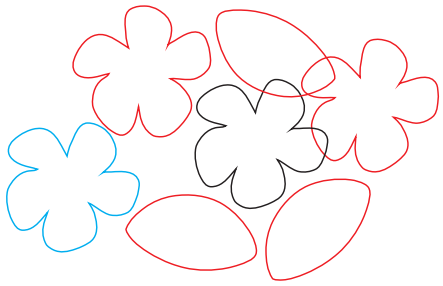


Fig. 10. The adjacent neighborhood (in red) of the black tile are all the tiles surrounding it while being closer than a threshold.

4.4.2 Maximization. We resort on a similar strategy as described in Section 4.3.2 to maximize $E_{repulse}$. Algorithm 3 details the procedure, which executes the following steps:

Scale update. To eliminate gaps and overlaps between tiles we adaptively scale them according to the distances to their adjacent neighbors. The pseudo code for ScaleUpdate is given in Algorithm 2. We measure the size of a tile as the diameter of the smallest enclosing circle centered on the centroid.

ComputeDistance returns d_i, the smallest value of repulse (Equation 4) between a tile and its adjacent neighbors. Assuming that the current size of T_i is s_i, we compute in ComputeNewScale the expected size of T_i in the next iteration as $k_i = s_i + (d_i - d)/2.0$, where d is the target gutter distance. We divide $(d_i - d)$ by two as T_i and its nearest neighbor change in size simultaneously. After the expected size is computed for all tiles, ScaleToExpectedSizes applies the actual scaling operations.

Position and orientation update. We follow a similar strategy as in Algorithm 1 to locally update the positions and orientations of the tiles. However, during each iteration, we seek to search for updates that *maximize* Equation 6.

Termination. At each iteration we track the minimum distance between closest neighbors d_{min}. If this value is larger than or equal to the target gutter distance d the process terminates – all tiles are far enough from their neighbors. Otherwise, the iterations continue up to a maximum value. The overall process is illustrated in Figure 6e.

While there is no strict guarantee that the process converges, we never observed cases where the maximum number of iterations is reached: after initialization and attraction phase there is only a

Algorithm 2 ScaleUpdate

Input:
Input tile layout $\mathcal{T}$ on S, target interspacing distance d;
Output:
Updated tile layout $\mathcal{T}_O$ with size adaptively scaled.

$K := \emptyset$; // *vector that stores expected size of each tile*
for each tile $T_i \in \mathcal{T}$ **do**
  $\mathcal{A}_i \leftarrow$ FindAdjNeighbors(T_i);
  $d_i \leftarrow$ ComputeDistance(T_i, $\mathcal{A}_i$);
  $k_i \leftarrow$ ComputeNewScale(T_i, d_i);
  $K \leftarrow K \cup \{k_i\}$;
end for
$\mathcal{T}_O \leftarrow$ ScaleToExpectedSizes($\mathcal{T}$, K);
return $\mathcal{T}_O$;

limited amount of overlap, and these can be resolved without having to apply excessive scaling to the tiles.

Algorithm 3 Repulsion

Input:
Target Surface $\mathcal{S}$; Input tile distribution $\mathcal{T}$ on S; Target interspacing distance d; Input magnitude field $\mathcal{F}$ on $\mathcal{S}$.
Output:
Optimized tile configuration $\mathcal{T}_O$ that follows $\mathcal{F}$ on S.

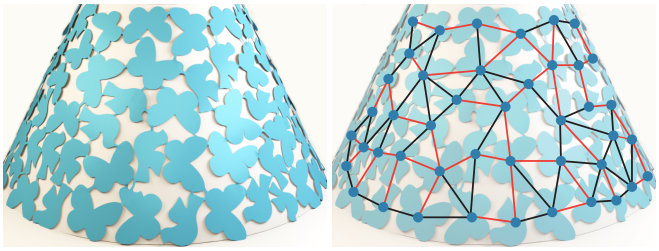
repeat
  $\mathcal{T} \leftarrow$ ScaleUpdate($\mathcal{T}$);
  $\mathcal{T} \leftarrow$ PositionUpdate($\mathcal{T}$, $\mathcal{F}$);
  $\mathcal{T} \leftarrow$ AngleUpdate($\mathcal{T}$);
  $d_{min} \leftarrow$CheckDistance();
until $d_{min} \geq d$ **or** maximum iterations reached
return $\mathcal{T}_O = \mathcal{T}$;

5 PATCH EXTRACTION

We now consider the problem of connecting tiles with hinges and snap-fit joints, thus forming patches that can be fabricated flat, folded and assembled to form the final 3D shape. The set of potential hinges are the connections between a tile T_i and its adjacent neighbors in $\mathcal{A}_i$ (see Section 4.4.1 and Figure 10). For two neighboring tiles, the potential location of a hinge is in-between the two closest points along the contours. We use the neighborhood information to create a graph G that captures the tile network. Each tile T_i is a node, and one edge is added for each adjacent neighbors T_j in $\mathcal{A}_i$ (see Figure 11b). This graph is very densely connected since the tiles are packed.

This section focuses on two questions: 1) how to segment the tile network into foldable patches (Sections 5.1 and 5.2) and 2) how to assign and optimize the hinge placement so that the final assembled printout is fabricable and possibly stable (Section 5.3).

What we mean by *stable* is that the tiles are all locked in place with respect to one another. This possibility stems from the fact that taken all together the joints create a dense set of distance constraints between the rigid tiles. These constraints hold the tiles into a stable configuration – a property which is expected for a closed convex polyhedron through Cauchy's rigidity theorem, but may not generally be true (e.g. a flat surface). Our algorithm attempts to preserve this property while trying to reduce the number of joints (Section 5.3.1). Note that if the shape is not stable even when inserting all joints, the final result can still be fabricated but will exhibit unconstrained degrees of freedom.

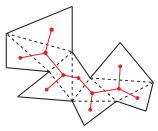


(a) Tile network (b) Tile graph

Fig. 11. Tile network and its corresponding graph. Note that (b) only shows the graph for the tiles that are visible in (a). The red edges in (b) form a spanning tree of the graph.

5.1 Extracting foldable patches

Decomposing a surface into developable/flattenable patches is a well-studied field [Julius et al. 2005; Tang et al. 2016; Wang 2008]. However, conventional methods segment the surface into near-developable patches and then deform the patches into fully developable ones. This may introduce distortions and mismatches between the boundaries of adjacent patches. We propose a method better suited to our application, based on the extraction of spanning trees. It finds a segmentation that does not produce distortions when flattening patches, and that maintains corresponding boundaries between adjacent patches.



In particular, let us consider a case where the tile graph G is a tree – it is easy to observe that it could be flattened/unfolded: each hinge separates the graph in two distinct parts which can rotate freely around the hinge axis with respect to one another. This notion is similar to the foldability of the *triangulation dual*, where triangles are nodes and edges are hinges [O'Rourke 1998], as illustrated in the inset. The flattening/unfolding might however produce overlaps in the plane.

We build upon this idea and formulate patch extraction as a *graph partitioning* problem: we seek to partition the graph G into a set of spanning trees such that each tree can be unfolded.

5.2 Graph partitioning

We now describe how to partition the graph G (Figure 11b) into a set of spanning trees. There are two fabrication requirements for the graph partitioning problem. First, the size of the unfolded geometry of each patch cannot exceed the extent of the printer bed; second, the unfolding should not produce overlaps between tiles. We propose a method to grow each patch in sequence while considering the properties of its unfolded geometry.

We decorate the graph with edge weights. Each edge weight indicates the capability $c(i, j)$ to insert a hinge between T_i and T_j at two points p_i and p_j as described in Section 4.3.2 and Figure 9. $c(i, j)$ is equal to the average local shape thickness. A larger $c(i, j)$ indicates a larger space for inserting a joint.

In practice the hinges require smaller footprints than the snap-fits (see Figure 3). Therefore, we encourage edges with lower $c(i, j)$ to become hinges. We thus seek to extract a *minimal spanning tree* from G such that the edges with lower $c(i, j)$ belong to the tree.

In the following, whenever selecting an edge to become a hinge (or snap-fit) we first check whether the value of $c(i, j)$ is large enough to host the joint, and ignore the edge otherwise. Note that most edges can host a joint: the packing algorithm specifically optimizes tile orientations to achieve this.

We grow patches with the following process. We start from a tile selected randomly. We then locally grow a subgraph by breadth-first expansion. At each step of the growth, we extract a minimal spanning tree and verify whether the flattening/unfolding of tree is valid, in which case we continue growing the subgraph. The unfolding is valid as long as it does not produce an overlap between tiles and it fits the printer bed size. If the unfolding is not valid, we return the last valid spanning tree as the next patch, and start extracting a new patch. This is done until all tiles have been covered.

Algorithm 4 PatchGrow

Input:
Undirect weighted graph G; Index of starting tile $\mathcal{T}_k$; Size of printing bed: w - width; l - length

Output:
A minimal spanning tree that is grown from tile $\mathcal{T}_k$.

```
M = ∅; M.add(k); // vector that stores the patch nodes
q.push(k); // queue that stores pending nodes
F_out; // output spanning tree
while !q.empty() do
    t = q.pop();
    N ← GetNeighbors(t, G);
    for each j ∈ N do
        M.add(j);
        F ← MinimalSpanningTree(M);
        pass ← Check2DPattern(F, w, k);
        if pass then
            q.push(j); F_out = F;
        else
            M.pop_back();
        end if
    end for
end while
return F_out;
```

5.3 Snap-fits optimization

After patch extraction the set of hinges is fully determined (edges of the spanning trees). However, the number and placement of snap-fit joints is not fixed yet. This affects both the model stability and its printability.

Ideally, we would like to minimize the number of snap-fits: they have to be manually assembled, and using too many is likely to result in non-printable configurations as they cannot be hosted in the tiles. However, each snap-fit introduces a new distance constraint in the final assembly, working towards locking the result in a stable configuration. We thus seek for a proper amount of snap-fit joints so that the final printout is both fabricable and stable.

Note that some tile networks simply cannot be made stable: this depends on the input surface properties (see discussion in the introduction of Section 5). In this section we assume that the tile network is stable if all possibly snap-fit joints are inserted – thus, our objective during the snap-fit joint selection process is to preserve this property. If a surface cannot be made stable, all possible snap-fit joints will be inserted.

5.3.1 Snap-fit edges selection. A necessary condition for the assembled printout to lock in a stable configuration is to have loops in the connection graph – otherwise any leaf tile would be able to rotate freely along its hinge. In addition, smaller loops increase the set of constraints, further reducing the number of degrees of freedom in the assembly. Based on these simple observations, we optimize the selection of snap-fits to ensure each tile is captured inside constraint loops of small size.

During the algorithm we check the stability of the assembly by using a physics simulation (based on *Bullet Physics* [Coumans 2009]), verifying whether the tiles remain fixed in space under the effect of gravity and common pinch forces. The lowest point of the object is fixed to the ground as we are not interested in checking for balance. If the largest displacement exceeds 5 mm we consider the model unstable.

Algorithm 5 details the process. The input consists of the graph G and the set of edges that are already selected as hinges. The remaining unselected edges become a pool for selecting snap-fits, see Figure 13a. We filter out any edge in which tiles could not host a snap-fit. We refer to edges that could become snap-fits as *available*.

The first step in Algorithm 5 is to construct initial loops so that each tile node is captured within a certain cycle. This is implemented in ConnectLeafNodes. We search all leaf nodes (degree 1) and add an additional available edge to them, favoring closest neighbors. As the graph is densely connected, and as the packing is optimized to maximize the possibility to place joints, there is a very high likelihood that such an edge exists. This is illustrated by the green dashed segments in Figure 13b.

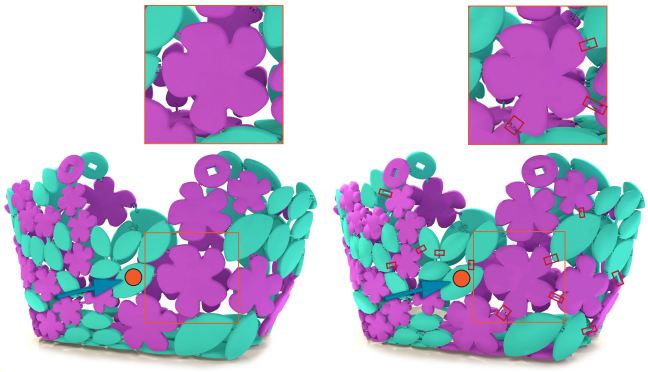
After removing leaf nodes, all tiles belong to cycles. The next step is to achieve a denser connectivity. DetectMinimalCycles detects the minimal cycles $\{C_i\}$. The value of k determines the maximal accepted length for the cycles. If a cycle C_i is longer than k, more edges are added to it by searching for available edges connecting pairs of nodes in the cycle, favoring closest neighbors. Thanks to the high degree of connectivity in G many such choices exist. This process, performed by EnhanceLargeCycles, is illustrated in Figures 13c and 13d. Finally, TestStability checks whether the model is stable with the current set of snap-fits. If not, the value of k is decreased and another iteration adds more snap-fits. At worst the algorithm terminates when all available snap-fits are added. Our algorithm is capable to adapt the number and placement of snap-fits according

Algorithm 5 SelectSnapFits

Input:
Graph G; A set of hinge edges E; Snap-fit hinge pool P;
Output:
Result stability f_{stb}; A set of edges F in G that will be realized by snap-fits.

```
k ← 15; f_stb ← true;
G' ← ConnectLeafNodes(E);
while true do
  {C_i} ←DetectMinimalCycles(G');
  (F, G') ← EnhanceLargeCycles({C_i}, k, P);
  stable ← TestStability(F, E);
  if stable then
    break;
  else
    if F == P then
      f_stb ← false;
      break;
    else
      k−−;
    end if
  end if
end while
return (f_stb, F);
```

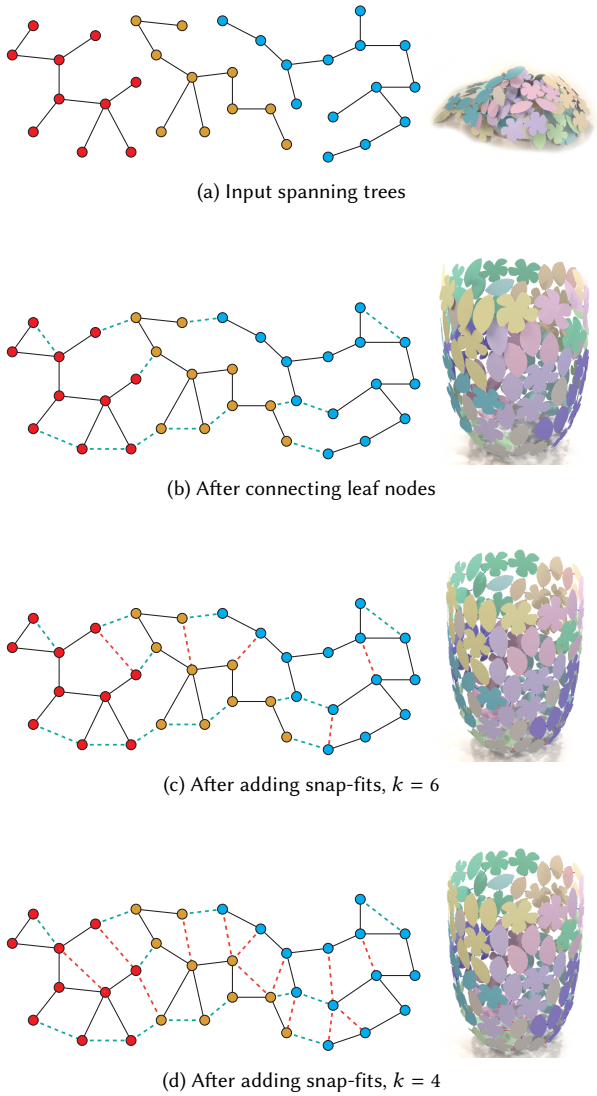
to different input external forces. As shown in Figure 12, when larger external forces are imposed, our approach will automatically enhance the structure with more connections.



(a) Input smaller external forces (b) Input larger external forces

Fig. 12. Results of inputting different external forces. $5N$ and $15N$ pinching forces are imposed in (a) and (b), respectively. The forces are imposed on both sides of bag symmetrically. The orange dot dictates the pressure point on the front side while the blue arrow shows the force direction. As the joints are designed to be embedded in an inconspicuous way, we highlight the additional joints resulted from larger forces in the red rectangles in (b). A closeup shows how a flower tile is strengthened.

We provide in the supplemental material an experiment showing the effects of adding different numbers of snap-fit edges.



(a) Input spanning trees

(b) After connecting leaf nodes

(c) After adding snap-fits, $k = 6$

(d) After adding snap-fits, $k = 4$

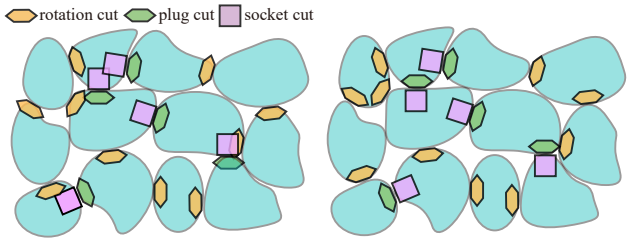
Fig. 13. Adding snap-fit joints. The left column illustrates the graph, the right column shows the physical simulation as more snap-fits are added.

5.3.2 Final joint assignments. After selecting the hinges and snap-fit joints, there might exist overlaps between their geometries (see Figure 14a). Hinges require space only for one of their ends (the other simply protrudes out of the tile). Snap-fits require space on both ends, one being slightly larger as shown in Figure 3d. This results in a combinatorial problem where we attempt to resolve for all conflicts, while swapping the joint ends assignment.

We proceed in two steps described in Algorithm 6. First, the joint ends are evenly distributed by function EvenlyDistribute. We define the "load" of a tile as the ratio between the number of joint ends it hosts and its area. We distribute the joint ends while attempting to achieve an even load across all tiles. We process the tiles in a priority queue in order of decreasing load value. Each time a tile is visited, the neighboring tiles are checked to assign the joint

ends such that the load is kept minimal.

This initial assignment might however create conflicts. To resolve these we process the tiles in a priority queue by decreasing order of number of conflicts. A conflict can be removed in two ways in function ResolveConflicts: the first attempted is to swap the ends of a joint; the second is to slightly move the joint attachment along the tile boundary. These changes can produce a conflict on a neighboring tile, and we therefore add any new conflict to the queue. In rare cases the conflicts cannot be resolved: this is detected whenever neither swapping nor moving the attachment works, or when a new conflict is produced on an already processed tile. In such a case we cancel the edge – a situation that occurred only on one edge in all our experiments (Lamp in Figure 19). Figure 14 shows the effect of joint assignment optimization.



(a) Before optimization (b) After optimization

Fig. 14. Effect of joint assignment optimization.

6 IMPLEMENTATION AND FABRICATION

This section gives details regarding implementation and fabrication.

6.1 3D model generation

To generate model for fabrication, we first unfold each patch into a set of 2D contours. The 3D model is constructed via adding a thickness to the 2D pattern. Joints are embedded into the model via boolean operations [Jacobson et al. 2016; Wang 2014].

Snap-fits internal to a patch are printed vertically (Figure 3) – which avoids having to consider potential collisions. Other snap-fits (across patches) are printed horizontally whenever possible, such that the part inserted into the tile is hidden from view (this can be seen on the printed patches in the second column of Figure 1).

6.2 Optimizations

We achieve better performance by using a coarser sampling. In particular, the attraction phase resorts on a coarse sampling of the tiles (20 samples per tile), followed by a first repulsion phase using a coarse sampling. After this point the packing is almost finalized, and we perform a final repulsion phase using the finest sampling (2 mm spacing). This second repulsion phase terminates quickly as only small overlaps – missed by the coarse pass – are resolved. We provide timings in Section 7.4.

Algorithm 6 DistributeJoints

Input:
Initial placement of joints $\mathcal{H}$; graph G;
Output:
An optimized distribution of joints $\mathcal{H}_{out}$.
1: $\mathcal{H}' \leftarrow$ EvenlyDistribute($\mathcal{H}$, G);
2: $\mathcal{H}_{out} \leftarrow$ ResolveConflicts($\mathcal{H}'$);
3: **return** $\mathcal{H}_{out}$;

7 RESULTS

In Section 7.1 we show several results produced with our method. In Section 7.2 we present UI controls provided to help users curstomize their designs. In Section 7.3 we compare our algorithm with different state-of-the-art optimization algorithms. We provide execution timings in Section 7.4. In supplemental material, we discuss convergence rate and surface packing quality.

7.1 Fabrication

We test our algorithm on a variety of models, from a simple sphere to a teddy bear with high curvatures. Figure 1 and Figure 15 show our fabrication results without control field and their corresponding simulations. Tiles of various shapes are used to decorate the base surfaces, including round convex shapes (e.g. sphere and egg) and concave contours with thin features (e.g. the bird and fish pattern). We fabricate all the results using a filament printer (Flash Forge *Creator Pro*) with ABS filament. All of the assembled models correspond to their simulation and approximate the target surface well. Some low amount of distortion can be seen, which is due to necessary tolerances when fabricating pre-assembled joints.

Our fabrication results have a variety of uses, for instance acting as lamp shades, vase decor (Figure 19) or even as a bag that can carry light objects (Figure 1). On the bag, two ring tiles are manually placed and fixed during optimization in order to attach the handles.

Table 1 summarizes the statistics for each fabricated result: size of assembled printout, number of tiles and the number of fabricated patches. The number of patches depends on both the model size and the surface complexity. Models with high curvatures require more patches (e.g. teddy bear). Note that Handbag1, Hangbag2, Lamp1 and Lamp2 refer respectively to the results in Figure 1, Figure 16e, Figure 19 and Figure 16f. Lamp1 is also visible in the supplemental material.

7.2 UI Control

We provide several controls to help users stylize their designs: orientation and scale as well as using different tiles in different regions.

7.2.1 Scale and orientation fields. Our method allows the user to edit both the size and orientation of the tiles. Figure 16 demonstrates results in which the tiles are adapted to an input control field, which consists of a scaling field and an orientation field, both specified by the user. In the figure the scale field is color coded, and ranges from 0.5 to 1.0. The streamlines in Figure 16a and 16b reveal the orientation fields used on the two results. The user edits the orientation field by sketching lines on the surface. A smooth orientation field is generated by treating the sketch lines as constraints, using

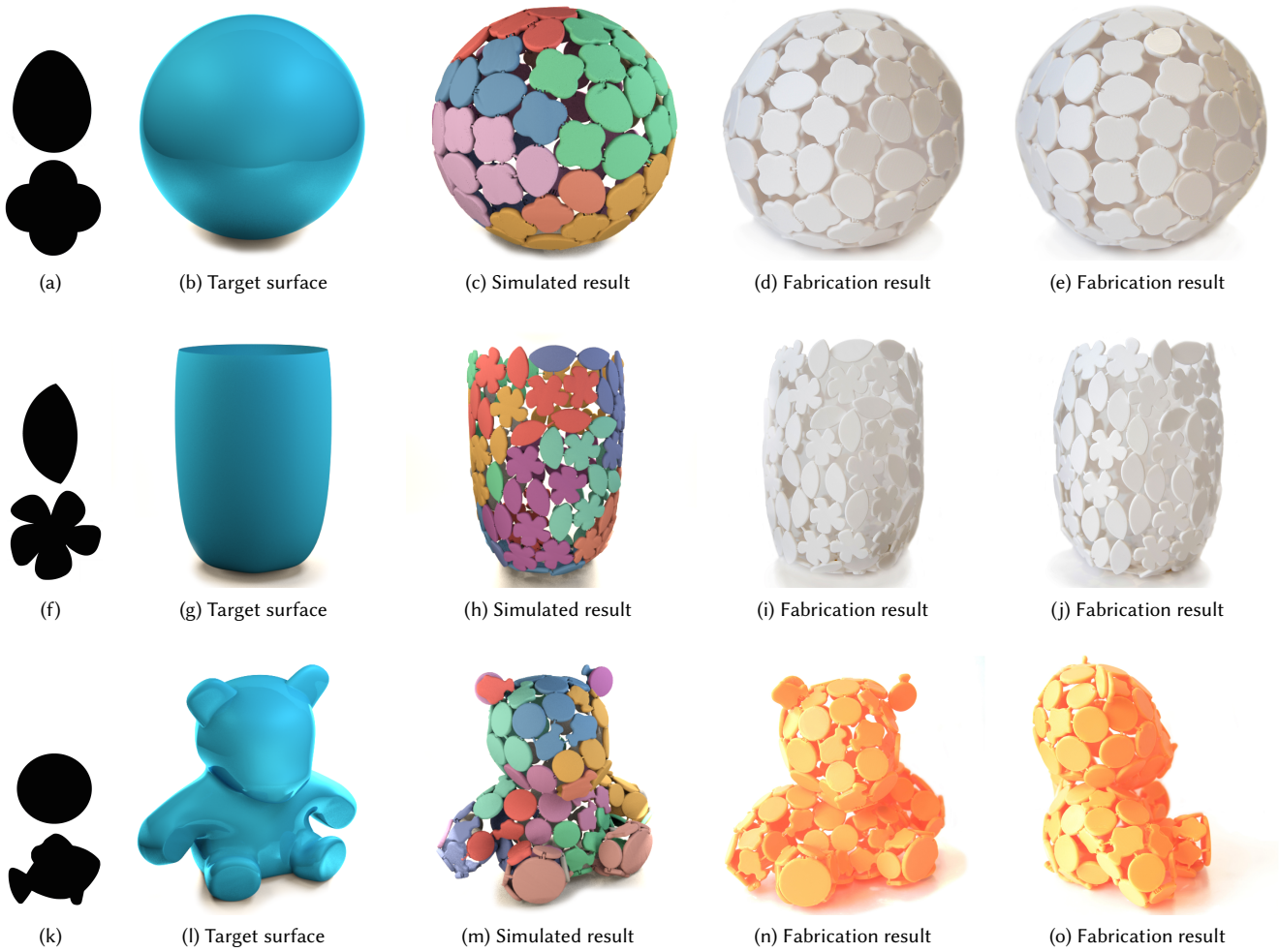


Fig. 15. **From left to right:** input tiles, target surfaces, simulated results with each patch color coded and fabricated results (two views).

the implementation provided by [Diamanti et al. 2014]. The user can paint desired scales directly onto the base surface using a brush tool.

As can be seen in the simulation and printed results, the tiles follow both the sizing and orientation fields while the result is a dense, fabricable packing.

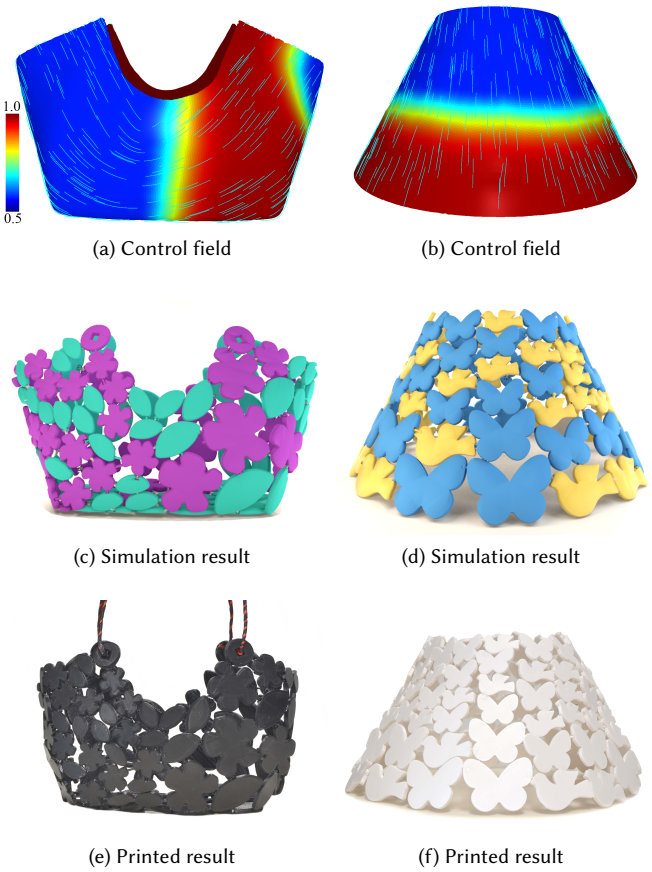
7.2.2 Global tile class control. Our approach lets the user specify the ratios of different tile classes that appear in the result. Figure 17 shows a case where the ratio is changed to achieve different results. This control is mainly achieved during the initialization step, when the tile distribution is generated. We employ the strategy described in [Wei 2010] for sampling from different classes.

7.2.3 Local tile class control. We let the user control the choice of tiles locally, as illustrated in Figure 18. A brush tool allows the user to paint the desired tile class id directly onto the surface. Tiles can only appear in a region with a same class id. This allows to customize the tile decors even further, as shown in Figure 18.

7.3 Comparison with gradient-free methods

We compare our approach to state of the art gradient-free optimization algorithms in Figure 20. In particular, we test two global optimization algorithms (ISRES [Runarsson and Yao 2005] and MLSL [Rinnooy Kan and Timmer 1987]) and two local optimization approaches (COBYLA [Powell 1994] and SBPLX [Rowan 1990]). The results of these algorithms are produced using the implementations of the *NLopt* library [Johnson 2014]. We use the same initialization (Figure 20a) and the same objective functions for all tested algorithms.

As seen in Figure 20, these generic optimizers perform poorly in our case (Figure 20b and Figure 20c). As none of the optimizers reached termination (i.e. a packing where the spacing constraint is achieved), we stopped them after 2 hours of running time. The local optimization methods perform better than their global counterparts



(a) Control field (b) Control field

(c) Simulation result (d) Simulation result

(e) Printed result (f) Printed result

Fig. 16. Results of control field editing. The user can edit both the size and orientation of tiles.

Parameters	Sphere	Vase	Bear	Lamp1
Size	(17,17,16)	(17,17,27)	(22,13,21)	(27,27,17)
#T	102	126	147	83
#P	10	13	22	14
Parameters	Lamp2	Handbag1	Handbag2	
Size	(27,27,17)	(25,13,18)	(25,13,18)	
#T	90	122	101	
#P	15	17	16	

Table 1. Statistics of fabricated results (Figure 15 and Figure 16). Statistics (from top to bottom) include size of assembled printout, number of tiles packed and number of patches used for assembly. The size is represented as $(length, width, height)$, all measured in centimeters.

Time	Sphere	Vase	Bear	Lamp1
t_{pack}	8.15	9.23	15.16	6.57
t_{patch}	0.25	0.27	0.35	0.28
t_{print}	36.1	52.0	42.6	37.1
t_{asm}	8.5	13.2	18.7	9.1
Time	Lamp2	Handbag1	Handbag2	
t_{pack}	7.12	11.32	10.14	
t_{patch}	0.29	0.30	0.27	
t_{print}	37.5	37.8	36.9	
t_{asm}	9.9	10.6	10.1	

Table 2. Timing of fabricated results (Figure 15 and Figure 16). For each result, we show the timing of tile packing t_{pack}, patch extraction t_{patch}, printing t_{print} and assembling t_{asm}. All timings are listed in minutes expect t_{print} which is in hours.

(see Figure 20d and Figure 20e) since the initial tile layout provides a good starting point. However, without an effective search strategy, it is hard for these methods to find valid solutions and there remain many unresolved tile overlaps as well as uncovered regions. Our algorithm achieves a much better result thanks to dedicated heuristics. We provide more details on the evolution of the objective functions during optimization in supplemental material.

7.4 Timings

Table 2 summarizes the timing of our results. All the results are produced on computer with an Intel i7-4770 CPU and 16GB RAM. The tile packing dominates the runtime. The timing is mostly influenced by the number of tiles and the complexity of the target surface. The printing time is mainly related to the surface area of target surface. For instance, the vase took around 7 more hours to print than the bear due to its larger surface.

8 LIMITATIONS AND CONCLUSIONS

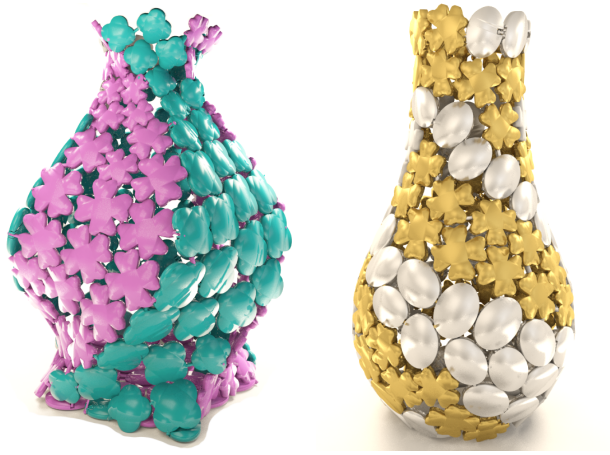
Our approach lets anyone fabricate visually interesting objects by decorating a surface with tiles. This mimics a popular way of improving a surface appearance by applying stickers and decals. Rather than synthesizing a complex 3D model difficult to print, our technique is designed to allow for efficient fabrication: the final surface is assembled from articulated patches that print flat, without support. This makes them fabricable on home filament printers, and easy to pack which maximizes utilization of powder-based printers, and reduces shipment costs.

Due to the fabrication constraints, our method cannot handle tiles which are thin everywhere (as shown in the inset). These do not offer sufficient space for inserting joints. Our method tends to ignore small scale surface details, e.g. the scales of a dragon, as our tiles are relatively large to allow for fabrication.

While the assembly step is left to the user – and it does take some time – we find the assembly to be an enjoyable process, that gives the user a better sense of ownership on the part she customized. However, it would be interesting, as future work, to attempt to further simplify this stage. Also, the main structural fragilities are the articulated hinges. To obtain stronger objects it would be interesting to investigate whether we could print solid connectors and deform them with heat, as in [Sageman-Furnas et al. 2015]. As the scale of input surface grows, the number of patches increases

(a) Input tiles (b) Scaling field (c) Equal ratio (d) More leaves (e) More clovers

Fig. 17. Result of varying the percentage of each tile class. (a) and (b) shows the input tile set (from top to bottom: leaf, flower and clover) and the underlying scaling field. (c) is a result where each class appears equally, while in (d) leaves appears with 70% probability and in (e) clovers appear with 70% probability.



(a) (b)

Fig. 18. Results of user-specified tile mixture. The user can restrict certain tile class to grow only on target regions.

Fig. 19. Fabrication results used for home decor.

– since the printer bed size limits the maximal extent of a patch. Tolerance required in the fabrication of joints may accumulate and increasingly lead to larger distortion in the final assembly. We will also explore the optimization of joint length for reducing global deformation. Finally, it would be interesting to study whether a final assembly could be made stable by gluing or constraining the motion of a small subset of hinges.

Using our technique, users without prior expertise can model objects that fully exploit advanced possibilities of 3D printing: embedding pre-assembled hinges and snap-fit joints in a model, as well as producing freeform, unusual geometries. We hope our approach will find a wide audience, and we will make the application available for everyone to enjoy.

ACKNOWLEDGMENT

We would like to thank Lei Chu for photographing the printouts and anonymous reviewers for their insightful comments. This work is supported by Hong Kong RGC (GRF 17208214), ERC grant SmartForge (StG-2012-307877) and NSFC (61772016).

REFERENCES

J. O'Rourke. 1998. *Computational geometry in C.* Cambridge university press.
M. Attene. 2015. Shapes in a Box: Disassembling 3D Objects for Efficient Packing and Fabrication. *Computer Graphics Forum* 34, 8 (2015), 64–76.
M. Bächer, E. Whiting, B. Bickel, and O. Sorkine-Hornung. 2014. Spin-It: Optimizing Moment of Inertia for Spinnable Objects. *ACM Trans. Graph.* 33, 4 (2014), 96:1–96:10.
W. Chen, X. Zhang, S. Xin, Y. Xia, S. Lefebvre, and W. Wang. 2016. Synthesis of Filigrees for Digital Fabrication. *ACM Trans. Graph.* 35, 4 (2016), 1–13.
X. Chen, H. Zhang, J. Lin, R. Hu, L. Lu, Q. Huang, B. Benes, D. Cohen-Or, and B. Chen. 2015. Dapper: decompose-and-pack for 3D printing. *ACM Trans. Graph.* 34, 6 (2015), 1–12.
P. Cignoni, N. Pietroni, L. Malomo, and R. Scopigno. 2014. Field-aligned mesh joinery. *ACM Trans. Graph.* 33, 1 (2014), 1–12.
E. Coumans. 2009. Bullet Physics. (2009). http://bulletphysics.org/wordpress/.
O. Diamanti, A. Vaxman, D. Panozzo, and O. Sorkine-Hornung. 2014. Designing N-PolyVector Fields with Complex Polynomials. *Computer Graphics Forum* 33, 5 (2014), 1–11.
J. Dumas, A. Lu, S. Lefebvre, J. Wu, and C. Dick. 2015. By-Example Synthesis of Structurally Sound Patterns. *ACM Trans. Graph.* 34, 4 (2015), 12.
A. Garg, A. O. Sageman-Furnas, B. Deng, Y. Yue, E. Grinspun, M. Pauly, and M. Wardetzky. 2014. Wire Mesh Design. *ACM Trans. Graph.* 33, 4, Article 66 (2014), 12 pages.

(a) Input tile layout (b) ISRES (c) MLSL (d) COBYLA (e) SBPLX (f) Ours

Fig. 20. Comparisons with other generic derivative-free optimization algorithms. We use the same tile initialization (a) as the input. (b) ISRES and (c) MLSL are global-optimization algorithms while (d) COBYLA and (e) SBPLX are algorithms for local optimization.

W. Hu, Z. Chen, H. Pan, Y. Yu, E. Grinspun, and W. Wang. 2016. Surface Mosaic Synthesis With Irregular Tiles. *IEEE transactions on visualization and computer graphics* 22, 3 (2016), 1302–1313.

E. Iarussi, W. Li, and A. Bousseau. 2015. WrapIt: Computer-assisted Crafting of Wire Wrapped Jewelry. *ACM Trans. Graph.* 34, 6, Article 221 (2015), 8 pages.

Y. Igarashi, T. Igarashi, and J. Mitani. 2016. Computational design of iris folding patterns. *Computational Visual Media* 2, 4 (2016), 321–327.

A. Jacobson, D. Panozzo, and others. 2016. libigl: A simple C++ geometry processing library. (2016). http://libigl.github.io/libigl/.

S. G. Johnson. 2014. The NLopt nonlinear-optimization package. (2014). http://ab-initio.mit.edu/nlopt.

D. Julius, V. Kraevoy, and A. Sheffer. 2005. D-Charts: Quasi-Developable Mesh Segmentation. In *Computer Graphics Forum*, Vol. 24. Wiley Online Library, 581–590.

M. Konaković, K. Crane, B. Deng, S. Bouaziz, D. Piker, and M. Pauly. 2016. Beyond developable: computational design and fabrication with auxetic materials. *ACM Trans. Graph.* 35, 4 (2016), 89.

T.-H. Kwok, C. C. Wang, D. Deng, Y. Zhang, and Y. Chen. 2015. Four-dimensional printing for freeform surfaces: design optimization of origami and kirigami structures. *Journal of Mechanical Design* 137, 11 (2015), 111413.

Y.-K. Lai, S.-M. Hu, and R. R. Martin. 2006. Surface mosaics. *The Visual Computer* 22, 9-11 (2006), 604–611.

S. Lefebvre, S. Hornus, and F. Neyret. 2005. Texture Sprites: Texture Elements Splatted on Surfaces. In *ACM Symposium on Interactive 3D Graphics and Games*. ACM SIGGRAPH, ACM Press.

D. Li, D. I. Levin, W. Matusik, and C. Zheng. 2016. Acoustic Voxels: Computational Optimization of Modular Acoustic Filters. *ACM Trans. Graph.* 35, 4 (2016).

L. Luo, I. Baran, S. Rusinkiewicz, and W. Matusik. 2012. Chopper: Partitioning Models Into 3D-Printable Parts. *ACM Trans. Graph.* 31, 6 (2012), 1.

C. Ma, L.-Y. Wei, and X. Tong. 2011. Discrete Element Textures. *ACM Trans. Graph.* 30, 4 (2011), 62:1–62:10.

E. Miguel, M. Lepoutre, and B. Bickel. 2016. Computational Design of Stable Planar-Rod Structures. *ACM Trans. Graph.* 35, 4 (2016), 1–11.

J. Mitani and H. Suzuki. 2004. Making papercraft toys from meshes using strip-based approximate unfolding. *ACM Trans. Graph.* 23, 3 (2004), 259.

V. A. D. Passos and M. Walter. 2009. 3D virtual mosaics: Opus Palladium and mixed styles. *The Visual Computer* 25, 10 (2009), 939–946.

V. D. Passos and M. Walter. 2008. 3D mosaics with variable-sized tiles. *The Visual Computer* 24, 7-9 (2008), 617–623.

M. J. Powell. 1994. A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Advances in optimization and numerical analysis*. Springer, 51–67.

R. Prévost, E. Whiting, S. Lefebvre, and O. Sorkine-Hornung. 2013. Make it stand: balancing shapes for 3D fabrication. *ACM Trans. Graph.* 32, 4 (2013), 81:1–81:10.

A. H. Rinnooy Kan and G. Timmer. 1987. Stochastic global optimization methods Part I: Clustering methods. *Mathematical programming* 39, 1 (1987), 27–56.

J. Rosenkrantz and J. Louis-Rosenberg. 2007. Neverous System. (2007). http://n-e-r-v-o-u-s.com/blog/.

R. Roveri, A. C. Öztireli, S. Martin, B. Solenthaler, and M. Gross. 2015. Example based repetitive structure synthesis. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 39–52.

T. H. Rowan. 1990. Functional stability analysis of numerical algorithms. (1990).

T. P. Runarsson and X. Yao. 2005. Search biases in constrained evolutionary optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 35, 2 (2005), 233–243.

A. O. Sageman-Furnas, N. Umetani, and R. Schmidt. 2015. Meltables: fabrication of complex 3d curves by melting. In *SIGGRAPH Asia 2015 Technical Briefs*. ACM, 14.

R. Schmidt, C. Grimm, and B. Wyvill. 2006. Interactive Decal Compositing with Discrete Exponential Maps. In *ACM Trans. Graph.* ACM, 605–613.

C. Schumacher, B. Thomaszewski, and M. Gross. 2016. Stenciling: Designing Structurally-Sound Surfaces with Decorative Patterns. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 101–110.

L. Shapira, A. Shamir, and D. Cohen-Or. 2008. Consistent mesh partitioning and skeletonisation using the shape diameter function. *The Visual Computer* 24, 4 (2008), 249–259.

I. Shatz, A. Tal, and G. Leifman. 2006. Paper craft models from meshes. *The Visual Computer* 22, 9-11 (2006), 825–834.

M. Shugrina, A. Shamir, and W. Matusik. 2015. Fab forms: customizable objects for fabrication with validity and geometry caching. *ACM Trans. Graph.* 34, 4 (2015), 100.

P. Song, B. Deng, Z. Wang, Z. Dong, W. Li, C.-W. Fu, and L. Liu. 2016. CofiFab: Coarse-To-Fine Fabrication of Large 3D Objects. *ACM Trans. Graph.* 35, 4 (2016), 1–11.

M. Takezawa, T. Imai, K. Shida, and T. Maekawa. 2016. Fabrication of freeform objects by principal strips. *ACM Trans. Graph.* 35, 6 (2016), 225.

C. Tang, P. Bo, J. Wallner, and H. Pottmann. 2016. Interactive design of developable surfaces. *ACM Trans. Graph.* 35, 2 (2016), 12.

N. Umetani, A. Panotopoulou, R. Schmidt, and E. Whiting. 2016. Printone: Interactive Resonance Simulation for Free-form Print-wind Instrument Design. *ACM Trans. Graph.* 35, 6, Article 184 (2016), 14 pages.

J. Vanek, J. A. G. Galicia, B. Benes, R. Měch, N. Carr, O. Stava, and G. S. Miller. 2014. PackMerger: A 3D Print Volume Optimizer. *Computer Graphics Forum* 33, 6 (2014), 322–332.

C. C. Wang. 2008. A least-norm approach to flattenable mesh surface processing. In *Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on.* IEEE, 131–138.

C. C. Wang. 2014. LDNI-based Solid Modeling. (2014). http://ldnibasedsolidmodeling.sourceforge.net/.

W. M. Wang, C. Zanni, and L. Kobbelt. 2016b. Improved Surface Quality in 3D Printing by Optimizing the Printing Direction. *Computer Graphics Forum* 35, 2 (2016), 59–70.

X. Wang, T. H. Le, X. Ying, Q. Sun, and Y. He. 2016a. User controllable anisotropic shape distribution on 3D meshes. *Computational Visual Media* 2, 4 (2016), 305–319.

L.-Y. Wei. 2010. Multi-class blue noise sampling. *ACM Trans. Graph.* 29, 4 (2010), 79.

J. Zehnder, S. Coros, and B. Thomaszewski. 2016. Designing structurally-sound ornamental curve networks. *ACM Trans. Graph.* 35, 4 (2016), 99.

Y. Zhang, C. C. Wang, and K. Ramani. 2016. Optimal fitting of strain-controlled flattenable mesh surfaces. *The International Journal of Advanced Manufacturing Technology* (2016), 1–15.

H. Zhao, L. Lu, Y. Wei, D. Lischinski, A. Sharf, D. Cohen-Or, and B. Chen. 2016. Printed Perforated Lampshades for Continuous Projective Images. *ACM Trans. Graph.* 35, 5 (2016), 1–11.

S. Zhou, C. Jiang, and S. Lefebvre. 2014. Topology-constrained synthesis of vector patterns. *ACM Trans. Graph.* 33, 6 (2014), 1–11.