MesoGen: Designing Procedural On-Surface Stranded Mesostructures

Élie Michel
LTCI, Télécom Paris, IP Paris
Adobe Research
Paris, France
emichel@adobe.com

Tamy Boubekeur
Adobe Research
Paris, France
boubek@adobe.com

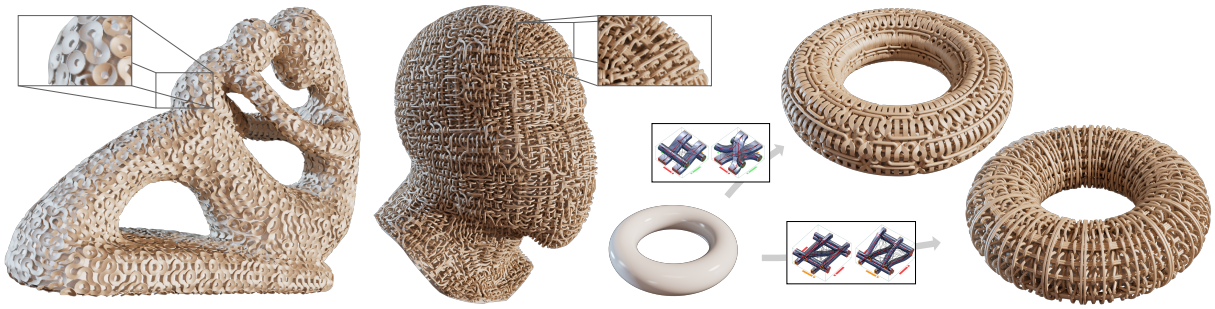

Figure 1: Mesostructures produced using MesoGen. The design of very intricate topology can be achieved rapidly (left and middle) and a given input macrosurface results in various styles (right) depending on the tile set created with our workflow.

ABSTRACT

Three-dimensional mesostructures enrich coarse macrosurfaces with complex features, which are 3D geometry with arbitrary topology in essence, but are expected to be self-similar with no tiling artifacts, just like texture-based material models. This is a challenging task, as no existing modeling tool provides the right constraints in the design phase to ensure such properties while maintaining real-time editing capabilities. In this paper, we propose MesoGen, a novel tile-centric authoring approach for the design of procedural mesostructures featuring non-periodic self-similarity while being represented as a compact and GPU-friendly model. We ensure by construction the continuity of the mesostructure: the user designs a set of atomic tiles by drawing 2D cross-sections on the interfaces between tiles, and selecting pairs of cross-sections to be connected as strands, i.e., 3D sweep surfaces. In parallel, a tiling engine continuously fills the shell space of the macrosurface with the so-defined tile set while ensuring that only matching interfaces are in contact. Moreover, the engine suggests to the user the addition of new tiles whenever the problem happens to be over-constrained. As a result, our method allows for the rapid creation of complex, seamless procedural mesostructure and is particularly adapted for wicker-like ones, often impossible to achieve with scattering-based mesostructure synthesis methods.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGGRAPH '23 Conference Proceedings, August 6–10, 2023, Los Angeles, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0159-7/23/08…$15.00
https://doi.org/10.1145/3588432.3591496

CCS CONCEPTS

• **Computing methodologies** → **Graphics systems and interfaces**; **Mesh models**.

KEYWORDS

Interactive shape modeling, 3D mesostructure, tilling, shell mapping

ACM Reference Format:
Élie Michel and Tamy Boubekeur. 2023. MesoGen: Designing Procedural On-Surface Stranded Mesostructures. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings (SIGGRAPH '23 Conference Proceedings), August 6–10, 2023, Los Angeles, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3588432.3591496

1 INTRODUCTION

Geometric surface enrichment is often achieved using displacement mapping, for which content can easily be authored using standard (2D) painting tools. However, the content injected onto the macrosurface has fixed disk topology and cannot represent complex structures, such as tunnels and handles. To overcome this issue, generalized displacement mapping and shell mapping are explicitly modeling the surrounding space of the macrosurface and use various mechanisms to instantiate complex shapes in it. Unfortunately, this comes at the cost of tedious authoring, as the mesostructure shall still behave like a mappable object, conforming to tilability constraints and deforming following the macrosurface curvature. As a consequence, only complex preprocessings [Zhou et al. 2006] acting on preexisting geometry have been developed so far to transform a 3D surface into a proper mesostructure. In practice, interactive mesostructure synthesis is often achieved using primitive scattering over the macrosurface, which is efficient at

Our implementation is available at https://github.com/eliemichel/MesoGen

reproducing dense packings, but not convenient for stranded ones e.g., tubular networks or frayed scaffolds (see Fig. 1). We propose MesoGen, a novel approach to such self-similar mesostructure design built upon a tiling algorithm [Wang 1961] and adopting an interface-centric workflow, where the user creates mesostructure 3D tiles through the 2D cross-sections they form at their interfaces. MesoGen runs in real-time on any quad-based surface domain and allows for the creation of complex mesostructures using a few brush strokes. Just like displacement maps, the resulting model can be reused across macrosurfaces – with minimal tile set adjustments – and is architectured to be compact and GPU-friendly.

Contributions. Our main contributions are: **(i)** a mesostructure design method centered on *continuity by construction* and built upon a tiling engine; **(ii)** a tiling engine, evolving state-of-the-art with user-prescribed constraints and a tile suggestion mechanism, that we evaluate; **(iii)** a compact procedural mesostructure model feeding this engine, architectured for real-time user feedback and delivering hundreds of millions of polygons per-second on standard GPUs; and **(iv)** a mapping mechanism leveraging the procedural nature of our mesostructure model to fill the macrosurface shell space. Our method allows designing a variety of mesostructure types, and is particularly well-suited for stranded structures like wicker, which contain long connected features; it completes approaches based on the scattering of mesoelements.

2 RELATED WORK

Tile-based generation. Inspired by the tiling problem formalized by Wang [1961], the field of computer graphics uses tiling solvers to address one of its everlasting bottlenecks: authoring in a reasonable human time the amount of graphic content that a machine can process. Wang tiling was first applied to texture generation, with the *aperiodic texture mapping* of Stam [1997] showing how laying out multiple patches of texture break the visual repetitivity that strikes the human eye when naively repeating the same image.

Following Stam's, multiple other approaches explored tile-based texturing. Neyret and Cani [1999] used tiles for on-surface synthesis rather than for paving a plane, thus performing seamless texturing. They took some liberties with the original Wang tile framework, changing the tile shape to triangles, and introducing the need to orient tile edge labels, which we also adopt in our method.

Our tile-based approach draws inspiration from the work of Cohen et al. [2003], which highlights that the construction of tile's interior content is a key bottleneck. Like ours, their procedure for building this content leverages the awareness of interface assignments to ensure the continuity of the result, but they focus primarily on automated tile filling, either from examples (for texture generation) or using a scattering process (Poisson disc distributions). We provide more control by letting the users interactively author tile's interior while preserving interface-aware constraints. This possibility is briefly mentioned by Cohen et al., but their stamp-based approach still requires manual adjustment to ensure continuity (see supplementary document). Instead of letting the user laying out existing primitives over the tiles, we enable them to design shapes by first defining their cross-section at each interface and then connecting them. Not only this avoids the problem of overlapping multiple edges, but it also generalizes to 3D (and so 2D cross-sections).

Cohen et al. also show that, unlike other texture synthesis techniques such as *image quilting* [2001], tile-based texture generation limits the computational workload involved in blending texture patches: once the graph cut sewing is prepared for each tile, the synthesis itself is very fast – it only consists in laying out tiles – and can be done on the fly during real time rendering [Wei 2004]. Our method also benefits from this factorization, in contrast to the *mesh quilting* method [Zhou et al. 2006] for instance.

In their representation of forest scenes, Decaudin and Neyret [2004] present tiling as a mean to compactly encode the geometry of all trees, since they precompute light transport only for a set of tiles before instantiating them on the fly at render time. We adopt a similar strategy in our mesostructure representation, using tiles to share memory. Although others on-surface texture generation [Turk 2001] address surface parameterization artifacts as much as tile-based methods, tiles provide a better control on structures spanning across long distances. Yarn modeling techniques are good examples of how tile-based geometry constitutes a rich design space [Leaf et al. 2018; Narayanan et al. 2019; Nimkulrat et al. 2017; Yuksel et al. 2012], but these focus on fabricability constraints and/or the coupling with powerful physical simulations for predefined tiles with simple cross sections while we focus on designing custom tilesets.

Tiling engines. Stam [1997] uses a predefined tile set for which a constructive algorithm for aperiodic tiling is known to always work [Grünbaum and Shephard 1987]. Neyret and Cani [1999] consider the exhaustive tile set where all combinations of Wang labels are available, thus tiling is always solvable. The approach of Cohen et al. [2003] is more flexible than Stam's, as the tile sets are generated depending on the number of Wang labels such that they can use a tiling algorithm that always succeeds. Also using a restricted family of tile sets, Fu et al [2005] generalize the solver to non-regular slot grids. Lastly, some tiling methods do not rely on Wang's framework [Chen et al. 2017] but this leads to a heavier solving operation. Since all these approaches rely on the tile set being an internal entity hidden from the user, we turned to more generic tiling engines based on *constraint propagation*, which have been used for graphics content by Merrell [2007] and Gumin [2016].

Surface amplification. Enriching a coarse surface with subpolygonal details has been a subject of interest quite since mesh-based representations are used. These details are typically mapped like textures; the first example is displacement mapping [Cook 1984], which deforms polygons along their normal, and then came iterations like View-Dependent Displacement Mapping [Wang et al. 2003] and Generalized Displacement Maps [Wang et al. 2004]. As displacement-based methods are constrained to the topology of the original macrosurface, Porumbescu et al. [2005] proposed *shell maps* which use surface meshes as 3D texture data of arbitrary topology along a macrosurface. This work led to a number of follow-ups, adapting it for real-time mapping [Ritsche 2006], mitigating deformation artefacts [Jeschke et al. 2007] or using it for geometry transfer [Takayama et al. 2011]. In between lies hybrid approaches like *relief mapping* of complex mesostructure topology [Policarpo and Oliveira 2006] (but limited to a few overhanging layers). A radically different approach to geometric texture mapping is to leverage an implicit representation of the macrosurface [Brodersen

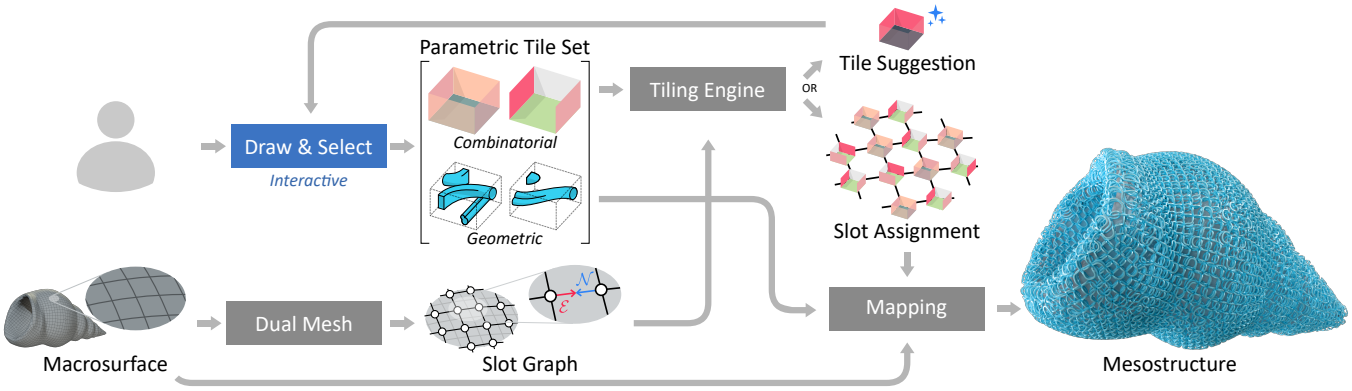

Figure 2: Overview of MesoGen. The user controls the synthesis process with the Draw & Select mechanism, detailed in Figure 4.

et al. 2008]. Our approach makes no exception to the overall surface amplification scheme: the meso-scale geometric content is defined in a few unit cubes – the tiles – and then mapped onto the target surface.

Empirically, a limiting factor when using tile-based modeling is the creation of tile content that remains seamless at any time. Many approaches are data-based, taking an example as input [Bhat et al. 2004; Cohen et al. 2003; Gumin 2016; Merrell 2007; Zhou et al. 2006]. Although this works well for 2D raster images, it is much harder to define in the case of 3D vector content laid out on irregular grids [Merrell and Manocha 2008] so, in practice, tile based 3D mesh generation uses manually crafted atoms. For 2D vector tiles, Bian et al [2018] propose an editor in which, while drawing on tiles, the user sees an onion skin of the continuation lines of neighboring tiles. Porting this approach to 3D content is not straightforward, and our work draws from this spirit of attributing a predominant role to interfaces during authoring. When not based on arbitrary examples, detail generation methods can also be domain-specific [Landreneau and Schaefer 2010]. De Toledo et al.[2008] provides a comparison of various mesostructure techniques.

3 MESOGEN

Our workflow (Sec. 3.1), summarized in Fig. 2, is based on a factorized, highly structured representation of the mesostructure (Sec. 3.2) which feeds a tiling engine exposing a feedback loop to the user for efficient authoring (Sec. 3.3) and for which we propose dedicated mapping (Sec. 3.5) and real-time rendering (Sec. 3.6) procedures.

3.1 Design workflow

MesoGen takes a mesh representing the macrosurface as input, along which the mesostructure is to be generated. Basically, the user designs the mesostructure by creating progressively a set of tiles, while a tiling engine covers the macrosurface by instantiating consistently and rendering a tile arrangement on-the-fly (Fig. 4).

In the tile set, a tile is defined by **(i)** a geometric content and **(ii)** adjacency rules, with the geometric content being instanced each time the tile is used by the engine. Adjacency rules are specified by labeling the four sides of a tile [Wang 1961]: two tiles can be located next to each other onto the macrosurface if and only if they share the same *interface label*. Like Neyret and Cani [1999], we also add an orientation flag to these interface labels, and only interfaces that are a mirror of each other may be juxtaposed.

The main friction when defining the content of a tile is to ensure that it is consistent with the content of any other tile that the tiling engine could place next to it. This is why we take the problem the other way around: in our approach, users author geometric content by drawing 2D cross-sections on the tile's interfaces. As such, the tile's geometric content is entirely defined by (i) assigning interfaces to the four sides of the tile and (ii) selecting pairs of cross-sections to connect using a *sweep surface*. The continuity of the mesostructure across interfaces is thus ensured by construction.

Another source of friction in the creative process relates to tiling engine failures. Since we let the user design arbitrary tile sets, and since the tiling problem is NP-hard in general, failures happen frequently, even with the best in class tiling engine. Consequently, we designed our tiling engine to suggest the addition of a new tile to the user whenever failure occurs, prescribing which configuration of interfaces could have enabled it to pave the whole macrosurface.

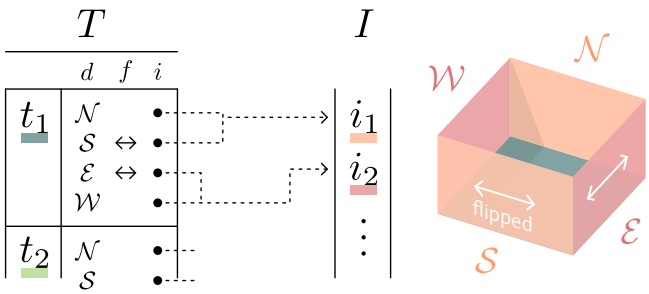

Figure 3: Wang labelling: A tile references, for each direction $d \in D$, one of the interfaces $i \in I$ whose geometric content must comply with. The interface can be horizontally flipped.

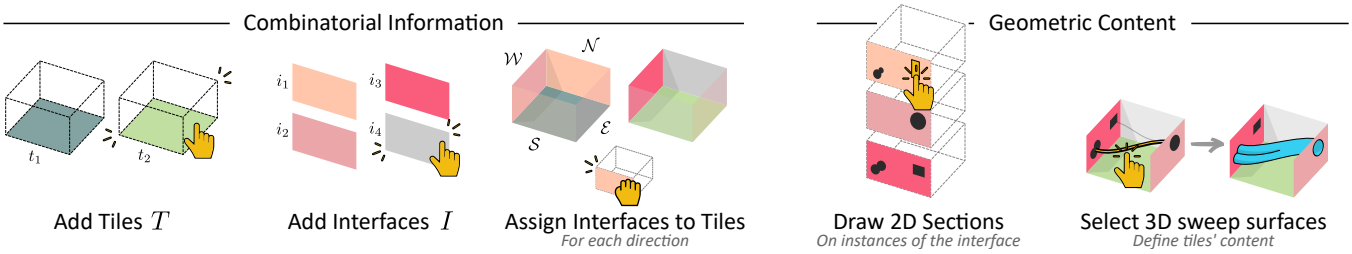

Figure 4: Tile set authoring involves setting combinatorial information (interfaces) and geometric content (2D cross-sections and 3D sweeps).

3.2 Procedural mesostructure model

Our mesostructure model compactly represents its surface elements in a factored way. Essentially, it takes the form of a tuple (T, I, M, A) composed of a tile set (T, I), a macrosurface M and an assignment A of tiles to the macrosurface, as summarized in Figure 2.

The *tile set* is formed by (i) a set I of interfaces containing 2D cross-sections, as well as (ii) a list T of tiles. Following the usual model of Wang tiles, a tile contains for each of its four sides – identified by its direction $d \in D = \{\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}\}$ for *north*, *south*, *east* and *west* – a reference to one of the interfaces (Fig. 3). We also add a flipping flag and note $i_k^{\leftrightarrow}$ the flipped version of an interface i_k. A tile also contains a geometric content, given as a list of 3D sweep surfaces, each referencing a pair of 2D cross-sections interpolated along a procedural 3D Bézier curve. The alignment between the source and target cross-sections is guessed by minimizing the distance on 20 points sampled along their contours, and can be manually tuned by the user if they want more torsion (see Fig. 5). Additional geometric content may be injected into the tile, provided it is entirely contained within the extent of the tile, i.e., it does not interact with the interfaces. Lastly, a tile contains a set of flags indicating whether the tiling engine is allowed to flip and rotate it.

The 2D content of each interface – instantiated on each tile side that references this interface – is modelled as a binary space occupancy function over the unit square. During the design phase, connected components are dynamically detected and constitute the cross-sections that can be selected for generating sweep surfaces. Shading information (albedo, roughness, metallicness) is assigned to each cross-section and interpolated along sweeps.

The *macrosurface* M is the domain where tiles are instantiated, extending the usual case of grid generation. It takes the form of a quad surface mesh and defines the associated *slot graph* (S, E), namely the undirected dual graph of the quad mesh connectivity where a vertex s of the slot graph (a *slot*) corresponds to a face of the quad mesh. Each half-edge $(s, e) \in S \times E$ of the slot graph is labelled with a direction $d \in D$, with at most one use of a given direction per slot. This indicates how a tile should be instantiated on this slot. Vertex positions and normal vectors define the shell space [Porumbescu et al. 2005] in which the mesostructure lives.

The *slot assignment* $A : S \to T \times P$ provides for each slot a *transformed tile*, namely a tile index and a tile transform indicating whether the tile should be rotated and/or flipped. This transform takes the form of a permutation $p \in P$ of its four base corners. An assignment is *valid* if the interface assigned to an half-edge is always the flipped version of the interface of its opposite half-edge. Formally, let s_1 and s_2 be two slots connected by an edge e in the slot graph, and d_1 (resp. d_2) the direction labeling the half-edge (s_1, e) (resp. (s_2, e)). Two (transformed) tiles t_1 and t_2 can be assigned to s_1 and s_2 only if the interface attached to the tile t_1 in direction d_1 is the flipped version the one attached to the tile t_2 in direction d_2.

Note that when the slot graph is a regular grid, a half-edge labelled with a *north* direction always faces a half-edge labelled with a *south* direction, but for an arbitrary graph, it is not necessarily the case (which is why we reason based on half-edges).

3.3 Tiling

Constraint solving. Given the tile set and the slot graph, the tiling engine produces a valid slot assignment. Our tiling engine, summarized in Alg. 1, is largely based on the Wave Function Collapse algorithm [Gumin 2016], itself following mostly the engine proposed by *Model Synthesis* [Merrell 2007]. It proceeds by progressive reduction of the possibility space, alternating two steps (see Fig. 6). Initially, the set of all tiles under all transforms is assigned to each of the slots, then it greedily propagates constraints through a depth-first traversal of the slot graph. Each time this recursive propagation (*collapse*) step reaches a fixed point, the possibility set of one of the slots is arbitrarily reduced to a single tile (*observe* step). In order to reduce the chance of leading to a dead-end – a case where the possibility set of a slot is empty – the observed slot is chosen so as to minimize the amount of information removed from the system.

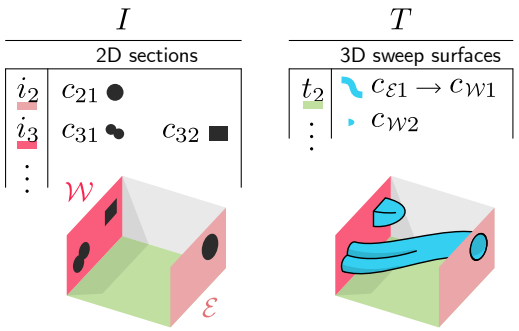

Figure 5: The geometric content of tiles is defined by sweeping across 2D cross-sections drawn on interfaces. The same cross-section may be used by more than one sweep, and if they are not used by any, a *cap* surface is automatically added.

In the case of equiprobable tiles, this simply means we observe the slot with the smallest possibility set (that has more than 1 tile). When stuck, the algorithm restarts with a different random seed.

The tiling problem being NP-hard, this algorithm does not magically handle all cases, but benefits from some nice properties. First, it is easy to implement, and has proven to be useful in practice, especially for video games [Stalberg 2018]. Then, it is not tied to the regular grid structure on which tiling algorithms are usually applied; we were able to adapt it to the arbitrary slot graph derived from our input macrosurface with minimal modification. Lastly, reasoning about possibility spaces is a flexible framework in which it is easy to encode extra constraints, like forcing some interfaces to occur only on the boundaries of the macrosurface. Other work even ensure path finding or other non-local constraints [Sandhu et al. 2019], and these could be ported to our use case.

ALGORITHM 1: Outline of the tiling solver. Pink underlined items show our additions to the typical Wave Function Collapse algorithm [Gumin 2016]: (1) `RecordNeighbors` saves the cause of the dead-end for the tile suggestion mechanism, and (2) we traverse an arbitrary slot graph rather than a regular grid.

```
Data: Slot graph G = (S, E) and tile set T
Result: Slot assignment A : S → P(T)
fn Solve G, T:
    A_0 ← InitialConstraints(G, T);
    A ← A_0;
    repeat
        try:
            s_0 ← Observe(A);
            Collapse(s_0);
        catch Finished:
            return A;
        catch DeadEnd:
            A ← A_0;
fn Observe A:
    if exists s ∈ S such that |A[s]| = 0 then
        RecordNeighbors(s) ;                                   (1)
        throw DeadEnd;
    if not exists s ∈ S such that |A[s]| > 1 then
        throw Finished;
    m ← min(|A[s]|, for slots s ∈ S such that |A[s]| > 1);
    s_0 ← random slot such that |A[s_0]| = m;
    A[s_0] ← { random tile from A[s_0] };
    return s_0;
fn rec Collapse s_1:
    foreach direction d_1 do
        (s_2, d_2) ← neighbor half-edge of (s_1, d_1) ;        (2)
        ResolveConflicts(s_1, d_1, s_2, d_2);
        if s_2 changed then
            Collapse(s_2);
```

Boundary constraints. The user may annotate tile interfaces with two flags: *boundary exempt* and *boundary only*. The first one specifies that the interface must never occur in a direction that is connected to no other slot. This is typically used for any non empty interface when the user does not want open ended sweeps. The second flag tells that the interface must never be connected, it is allowed only on boundaries. This can be used to ensure that the generated shape is made of one single piece, without including empty interfaces. These flags do not really interfere with the solving algorithm, they can be fully applied as a preprocessing of the possibility space (Alg. 2). For each slot that has no half-edge labelled with a given direction d, we initially remove all tiles whose interface in direction d is *boundary exempt*. And for each direction for which there exists a half-edge, we remove tiles whose corresponding interface is *boundary only*. Then we feed the tiling engine with this initial possibility space (Fig. 6, center).

ALGORITHM 2: Initialization of the possibility space prior to running the tiling engine. The pink underlined section shows how border exempt/only interfaces can easily be integrated.

```
Data: Slot graph G = (S, E) and tile set T
Result: Slot assignment A : S → P(T)
fn InitialConstraints T:
    foreach slot s ∈ S do
        A[s] ← T;
        foreach direction d ∈ D do
            if s has no half-edge labelled d then
                A[s] ← A[s] − {t ∈ T | the interface of t in
                direction d is border exempt };
            else
                A[s] ← A[s] − {t ∈ T | the interface of t in
                direction d is border only };
        if |A[s]| = 0 then
            RecordNeighbors(s);
    foreach slot s ∈ S do
        Collapse(s);
```

3.4 Tile suggestion

Our tile suggestion mechanism is executed during the feedback loop between the model and the user, in order to address the fact that tiling can be arbitrarily hard or even not possible for a given tile set on a given macrosurface. When stuck for too long, the tiling engine provides the user with a new tile, specifying the configuration of interfaces that would have helped it. To do so, we introduce a suggestion algorithm, listed in Alg. 3, based on a voting scheme. More precisely, we consider the set L of all tile side configurations that can be generated from the set of interfaces I. Each time the possibility set of a slot becomes empty – forcing the engine to backtrack – a vote is cast for all configurations of L that are compatible with the possibility sets of its neighbors. All possible transformations (rotation, flip) are applied to a configuration of L when checking that it can fit, and when the dead-end was reached while applying the initial constraints, we filter L with border constraints. For instance, an element of L labeled with an interface i in the *north* direction may receive a vote only if there is at least one tile in the possibility set of the *north* neighbor that exposes an interface $\overset{i}{\leftrightarrow}$ towards the empty slot, and if i is not border-only. The algorithm then suggests the tile that received the highest number of votes.

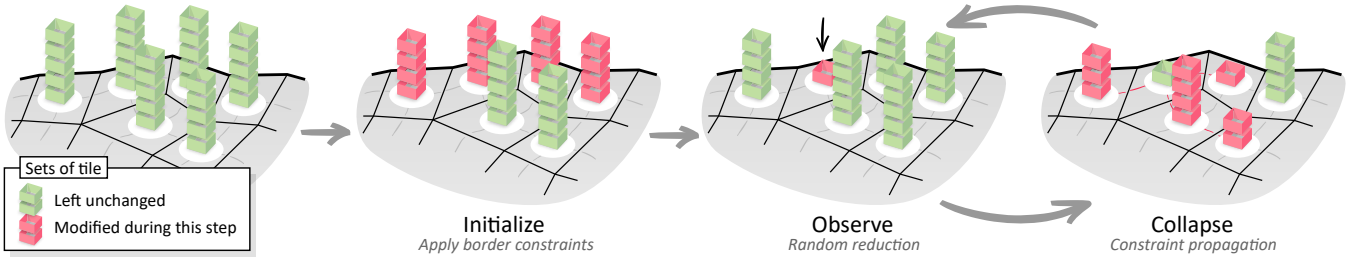

Figure 6: Outline of the tiling solver described in Algorithm 1. The possibility space of each slot is initialized to the set of all tiles, then our border exempt/only interfaces imposes some initial constraint, and the remainder of the algorithm is an alternation of arbitrary local choices and depth-first constraint propagation.

ALGORITHM 3: Our tile suggestion algorithm is based on a voting system. In practice, we also label votes with the transform p and break ties in the argmax by maximizing the number of identity transforms.

Data: Dead end neighborhoods N recorded during solving. A neighborhood $n \in N$ gives for each direction $d \in D$ a set of possible transformed interfaces $n_d = \{i_1, \underset{\leftrightarrow}{i_2}, \dots\}$ (where $\underset{\leftrightarrow}{i_2}$ means that interface i_2 is flipped).
Result: An interface i_d for each direction $d \in D$ of the new tile
fn SuggestNewTile N:
  Initialize votes: $I^4 \to \mathbb{N}$ to **0**;
  foreach *neighborhood* $n \in N$ **do**
    foreach $\boldsymbol{i} \in n_{\mathcal{N}} \times n_{\mathcal{S}} \times n_{\mathcal{E}} \times n_{\mathcal{W}}$ **do**
      foreach *tile transform* p **do**
        $\boldsymbol{i'} \leftarrow \text{inverse}(p) \cdot \boldsymbol{i}$;
        $\text{votes}(\boldsymbol{i'}) \leftarrow \text{votes}(\boldsymbol{i'}) + 1$;
  return argmax(votes);

3.5 Shell Mapping

Once a transformed tile is assigned to a slot, the last stage of our framework aims at mapping it to the actual shell space of the macro-surface for rendering. We cast this mapping problem as a deformation one [Porumbescu et al. 2005], from the mesostructure normalized space to the shell one, which thickness is controlled by the user. We express the geometric content of a tile w.r.t. the 8 corners of its slot's bounding box and use these local coordinates to reexpress it w.r.t. the extruded quad, taking inspiration from cage-based deformation.

The Shell Mapping approach [Porumbescu et al. 2005] can be reformulated in our case by replacing the barycentric interpolation performed over a tetrahedralization of a prism extruded from a triangle with a generalization of barycentric coordinates [Ju et al. 2005; Langer et al. 2006] computed within the hexahedron extruded from a quad. However, although this yields smoother deformation than dicing the hexahedron in tetrahedrons and applying Porumbescu et al. scheme, significant distortion still subsists.

To contain it, we use the parametric nature of the tile's geometric content, i.e., sweep objects, and deform their trajectory curves first, before the sweeping step. Doing so, the 2D cross-sections preserve their expected shape, e.g., a circle will produce a perfect tube, not an ellipsis-based one (see Fig. 7). Note that this may be opted out if one aims at deforming cross-sections as well, but we found empirically that cross-section preservation is often the expected behavior.

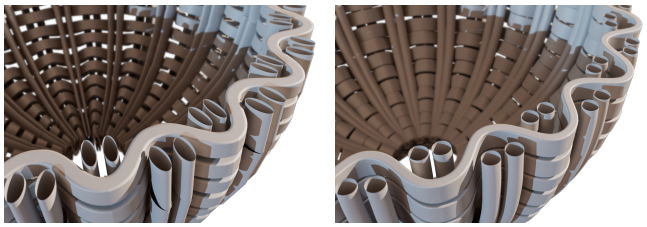

Figure 7: When mapping a tile's content into an hexahedron of the shell, deforming each point of the generated surface (left) leads to more distortion than applying the deformation to the underlying curves, prior to sweeping (right, ours).

As our trajectories are cubic Bézier curves, our mapping problem now boils down to the positioning, for each of them, their four control points $(p_0, ..., p_3)$ in shell space. Our solution is meant to **(i)** ensure tangential continuity of the trajectories across interfaces and **(ii)** get the trajectories as close as possible to sections of circle when possible. The point p_0 (resp. p_3) is moved to its corresponding interface and expressed using bilinear interpolation over its 4 corners. Tangential continuity is ensured by placing the remaining control points using the normal of the corresponding hexahedron face n_0 (resp. n_3), namely $p_1 = p_0 + m_0 n_0$ (resp. $p_2 = p_3 + m_3 n_3$). To approach a section of circle, the magnitude m_0 (resp. m_3) of the tangent is defined as follows: $m_i = m\frac{\alpha_i}{\alpha_i + \alpha_{3-i}}$ for $i \in \{0, 3\}$ with $m = 8d\frac{\sqrt{2}-1}{3}$ and $\alpha_i = \angle(p_{3-i} - p_i, n_i)$. When both p_0 and p_3 are on the same interface, the diameter d is set to $||p_1 - p_2||$. When ends are on neighboring interfaces, this distance is multiplied by $\sqrt{2}/2$. When they are on opposite interfaces, we no longer try to match a circle; we use the same value of d but set the weights α_0 and α_3 to 1 since $\alpha_0 + \alpha_3$ is null. We adopted this heuristic for its visual consistency, and the value of m_0 and m_3 can be globally or locally scaled by the user to produce various looks.

3.6 GPU Rasterization

We start by sampling each 2D cross-section of each interface using *Clipper* [Angus Johnson 2014]. These are stored as CSG trees modeling a 2D space occupancy function so that we can change

Table 1: For Fig. 9: GPU memory, drawn triangles and timing.

Example	Memory	Triangles	Render	Tiling
(1a)	5.66 MB	57.9 M	6.2 ms	1132 ms
(1b)	3.06 MB	44.7 M	6.0 ms	3212 ms
(1c)	6.30 MB	70.0 M	7.9 ms	708 ms
(2a)	2.96 MB	24.3 M	5.1 ms	87 ms
(2b)	1.83 MB	18.8 M	5.1 ms	135 ms
(2c)	4.18 MB	29.8 M	5.8 ms	279 ms
(3a)	1.76 MB	6.42 M	3.6 ms	17.8 ms
(3b)	1.33 MB	7.19 M	4.0 ms	43.3 ms
(3c)	3.49 MB	6.18 M	3.1 ms	39.1 ms

the discretization dynamically. The resulting points sets are then stored in 1D texture maps using a *repeat* wrap mode.

Second, for each sweep surface in the tile set – not for each instance – we allocate a VAO modeling a regular grid mesh. Its horizontal resolution is the maximum of the size of the start and end cross-section textures. The vertical resolution is a user defined parameter driving smoothness. We use a compute shader to assign x and y coordinates to each point by interpolating from the start to the end section, taking care of reversing the coordinate at which cross-section textures are sampled from u to $1-u$ when an interface is flagged as *flipped*. This creates base sweeps that will later be deformed per-instance to conform to their target trajectory.

Third, the shell space is represented as a GPU SSBO storing, for each macrosurface quad, the eight corners of its shell hexahedron.

Fourth, we allocate four SSBOs to hold the control points of the Bézier curves, containing one vector per instance of a sweep. A compute shader uses the shell space SSBO and the slot assignments to fill these control point buffers.

Finally, one draw call is issued for each sweep surface, and hardware-instanced as many times as there are uses of the corresponding tile in the slot assignment. We deform the VAOs at the vertex shader stage to follow the Bézier trajectory. Any shading method can be used on the rasterized fragments.

4 RESULTS

The renderings from Figure 1 have been computed using a third party render engine. The performances of our C++/OpenGL prototype are reported in Tab. 1, measured on an Intel Core i5 CPU, with 16GB Ram and an NVidia GeForce Titan RTX. As an element of comparison to show the compactness of our representation, the example (1c) occupies 1.75 GB when exported as a binary PLY file. Figure 9 shows that once a tile dictionary has been defined, it may easily be used across multiple macrosurfaces, applying a similar style to various shapes, requiring the user to only create missing tiles corresponding to unseen topological configurations. We observed that the most pleasant mesostructures are obtained when the macrosurface is coarse, with convex quads aligned along curvature flows and moderate elongation, but these are not strict requirements. More examples can be found in the supplementary material.

Figure 7 illustrates the interest of manipulating a procedural representation of tile content when it comes to mapping the content into a cell of the macrosurface's shell. Rather than blindly deforming the synthesized mesostructure, we deform the input

Figure 8: Macrosurface with boundaries: enforcing an empty interface at boundaries to prevent open geometry (middle and right). One can also prevent this empty interface from occurring away from boundaries (right).

Table 2: Success rate R and amount N of new tiles needed by different suggestion strategies to complete the tiling (clamped to 10, beyond which it is considered unsuccessful). Averaged over $14k$ runs (100 times 20 random tile sets and 7 macrosurfaces). For each of the 140 scenarios, a strategy counts as best (b) when its R is higher, or for equal R its N is lower. It counts as exclusive best (B) when it is the only best.

Strategy	R	N	b	B
Fully Random	5.3% ± 14.9%	6.7 ± 0.2	7.1%	0.0%
Guided Random	75.9% ± 34.3%	4.1 ± 1.3	34.9%	23.0%
Voting (ours)	**85.3% ± 30.9%**	**3.7 ± 0.7**	**77.0%**	**65.1%**

of the procedural construction, namely the control points of the sweep's trajectories. This leads to a more natural deformation, that conserves the aspect ratio of user-drawn 2D cross-sections. Fig. 8 is a typical use case of our border constraints. Without them, open ended surfaces appear on the boundaries (fig. 8.a). The user can then add an empty interface and flag all the other ones as *border exempt*, so only the empty interface is used at boundaries (fig. 8.b), and prevent disconnection using the *border only* flag (fig. 8.c).

Tile suggestion ablation study. Our tile suggestion mechanism essentially consists in two parts: **(i)** consolidating a set of candidates based on the solver's failure situations and **(ii)** vote for these candidates depending on their occurrence under any possible transform. Table 2 compares our strategy (*Voting*) with one where the voting part (ii) is replaced by a random draw (*Guided Random*), and one where the solver's failure cases (i) are completely ignored (*Fully Random*). Statistics are detailed in the additional material. We can observe that the awareness of failure cases greatly improves the suggestion over the baseline in all scenarios. Our voting scheme helps further and is, in a large majority of our 140 test cases, the most efficient mean to produce a solvable tile set. Interestingly our method reduces the variance of the results. This and its determinism makes our approach more predictable for the end user. It is nevertheless a heuristic, which may not be optimal in some cases.

Additional examples. Fig. 11 provides additional examples rendered in our real-time viewer, with complex topologies emerging from a few tiles only, per-sweep material properties, boundary-aware behaviors and use cases ranging from basketry to sci-fi. More examples are provided in our supplemental material.

5 DISCUSSION

Limitations & Future Work. MesoGen lacks structural constraints which would be essential for fabrication. Although successful, the mesostucture synthesis may sometimes not be as visually pleasing as expected, as exemplified in Fig 10. Directions for future work include anchoring specific tiles on the macrosurface, reversing the tiling to operate on interfaces rather tiles, fusing suggestion and solving, using macrosurface attributes to locally restrict the use of some tiles, using data-driven schemes [Tu et al. 2020; Zhou et al. 2006] and exploring implicit sweeps [Schmidt and Wyvill 2005] to address smoothly self-intersections in highly curved base domain regions. Last, quad mesh design could be incorporated in MesoGen, either by embedding remeshing tools [Jakob et al. 2015] or by suggesting local remeshing operations as an alternative to new tiles when the tiling fails. On-surface direction fields may also be instrumental in giving more macroscopic control to the user by biasing the initial configuration of the solver.

Conclusion. We proposed MesoGen, a method for authoring and representing rich 3D mesostructures along the surface of a quad mesh. Our approach is efficient at creating filament-like mesostructures, a case which is not covered by scattering-based mesostructure synthesis. We reduced the boilerplate involved in defining the 3D content of tiles by integrating the constraint of continuity at tile interfaces from the very beginning of the design process. And as a by-product, the parametric nature of the content interacts nicely with the mapping into the shell space, mitigating deformations.

REFERENCES

Angus Johnson. 2014. *Clipper*. http://www.angusj.com/delphi/clipper.php

Pravin Bhat, Stephen Ingram, and Greg Turk. 2004. Geometric Texture Synthesis by Example. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing (SGP '04)*. Association for Computing Machinery, New York, NY, USA, 41–44. https://doi.org/10.1145/1057432.1057437

Xiaojun Bian, Li-Yi Wei, and Sylvain Lefebvre. 2018. Tile-Based Pattern Design with Topology Control. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1 (2018), 23–38. https://doi.org/10.1145/3203204

A. Brodersen, K. Museth, S. Porumbescu, and B. Budge. 2008. Geometric Texturing Using Level Sets. *IEEE Transactions on Visualization and Computer Graphics* 14, 2 (March 2008), 277–288. https://doi.org/10.1109/TVCG.2007.70408

Weikai Chen, Yuexin Ma, Sylvain Lefebvre, Shiqing Xin, Jonàs Martínez, and wenping wang. 2017. Fabricable Tile Decors. *ACM Transactions on Graphics* 36, 6 (Nov. 2017), 175:1–175:15. https://doi.org/10.1145/3130800.3130817

Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. 2003. Wang Tiles for Image and Texture Generation. *ACM Transactions on Graphics* 22, 3 (July 2003), 287–294. https://doi.org/10.1145/882262.882265

Robert L. Cook. 1984. Shade Trees. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. Association for Computing Machinery, New York, NY, USA, 223–231. https://doi.org/10.1145/800031.808602

Rodrigo De Toledo, Bin Wang, and Bruno Lévy. 2008. Geometry Textures and Applications†. *Computer Graphics Forum* 27, 8 (2008), 2053–2065. https://doi.org/10.1111/j.1467-8659.2008.01185.x

Philippe Decaudin and Fabrice Neyret. 2004. Rendering Forest Scenes in Real-Time. In *EGSR04: 15th Eurographics Symposium on Rendering*. Eurographics Association, 93.

Alexei A. Efros and William T. Freeman. 2001. Image Quilting for Texture Synthesis and Transfer. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 341–346. https://doi.org/10.1145/383259.383296

Chi-Wing Fu and Man-Kang Leung. 2005. Texture Tiling on Arbitrary Topological Surfaces Using Wang Tiles. *Eurographics Symposium on Rendering (2005)* (2005), 6 pages. https://doi.org/10.2312/EGWR/EGSR05/099-104

Branko Grünbaum and G. C. Shephard. 1987. *Tilings and Patterns* (first ed.). W. H. Freeman and Company, New York.

Max Gumin. 2016. Wave Function Collapse.

Wenzel Jakob, Marco Tarini, Daniele Panozzo, and Olga Sorkine-Hornung. 2015. Instant Field-Aligned Meshes. *ACM Transactions on Graphics* (2015). https://doi.org/10.1145/2816795.2818078

Stefan Jeschke, Stephan Mantler, and Michael Wimmer. 2007. Interactive Smooth and Curved Shell Mapping. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques (EGSR'07)*. Eurographics Association, Goslar, DEU, 351–360.

Tao Ju, Scott Schaefer, and Joe Warren. 2005. Mean Value Coordinates for Closed Triangular Meshes. *ACM Trans. Graph.* 24, 3 (2005), 561–566.

Eric Landreneau and Scott Schaefer. 2010. Scales and Scale-like Structures. *Computer Graphics Forum* 29, 5 (2010), 1653–1660. https://doi.org/10.1111/j.1467-8659.2010.01774.x

Torsten Langer, Alexander Belyaev, and Hans-Peter Seidel. 2006. Spherical Barycentric Coordinates. In *SGP*.

Jonathan Leaf, Rundong Wu, Eston Schweickart, Doug L. James, and Steve Marschner. 2018. Interactive Design of Periodic Yarn-Level Cloth Patterns. *ACM Transactions on Graphics* 37, 6 (Dec. 2018), 202:1–202:15. https://doi.org/10.1145/3272127.3275105

Paul Merrell. 2007. Example-Based Model Synthesis. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07)*. Association for Computing Machinery, New York, NY, USA, 105–112. https://doi.org/10.1145/1230100.1230119

Paul Merrell and Dinesh Manocha. 2008. Continuous Model Synthesis. In *ACM SIGGRAPH Asia 2008 Papers (SIGGRAPH Asia '08)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/1457515.1409111

Vidya Narayanan, Kui Wu, Cem Yuksel, and James McCann. 2019. Visual Knitting Machine Programming. *ACM Transactions on Graphics* 38, 4 (Aug. 2019), 1–13. https://doi.org/10.1145/3306346.3322995

Fabrice Neyret and Marie-Paule Cani. 1999. Pattern-Based Texturing Revisited. In *26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. ACM SIGGRAPH, 235. https://doi.org/10.1145/311535.311561

Nithikul Nimkulrat, Janette Matthews, and Tuomas Nurmi. 2017. Tiling Notation as Design Tool for Textile Knotting. In *Bridges 2017 Conference Proceedings*. David Swart, Carlo H. Séquin, and Kristóf Fenyvesi, Waterloo, Canada, 4.

Fabio Policarpo and Manuel M. Oliveira. 2006. Relief Mapping of Non-Height-Field Surface Details. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games (I3D '06)*. Association for Computing Machinery, New York, NY, USA, 55–62. https://doi.org/10.1145/1111411.1111422

Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. 2005. Shell Maps. *ACM Transactions on Graphics* 24, 3 (July 2005), 626–633. https://doi.org/10.1145/1073204.1073239

Nico Ritsche. 2006. Real-Time Shell Space Rendering of Volumetric Geometry. In *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia (GRAPHITE '06)*. Association for Computing Machinery, New York, NY, USA, 265–274. https://doi.org/10.1145/1174429.1174477

Arunpreet Sandhu, Zeyuan Chen, and Joshua McCoy. 2019. Enhancing Wave Function Collapse with Design-Level Constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games (FDG '19)*. Association for Computing Machinery, New York, NY, USA, Article 17. https://doi.org/10.1145/3337722.3337752

Ryan Schmidt and Brian Wyvill. 2005. Generalized Sweep Templates for Implicit Modeling. In *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE '05)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/1101389.1101428

Oskar Stalberg. 2018. Wave Function Collapse in Bad North. (April 2018).

Jos Stam. 1997. Aperiodic Texture Mapping.

Kenshi Takayama, Ryan Schmidt, Karan Singh, Takeo Igarashi, Tamy Boubekeur, and Olga Sorkine-Hornung. 2011. GeoBrush: Interactive Mesh Geometry Cloning. *Computer Graphics Forum (Proc. EUROGRAPHICS 2011)* 30, 2 (2011), 613–622.

Peihan Tu, Li-Yi Wei, Koji Yatani, Takeo Igarashi, and Matthias Zwicker. 2020. Continuous Curve Textures. *ACM Transactions on Graphics* 39, 6 (Nov. 2020), 168:1–168:16.

Greg Turk. 2001. Texture Synthesis on Surfaces. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 347–354. https://doi.org/10.1145/383259.383297

Hao Wang. 1961. Proving Theorems by Pattern Recognition — II. *Bell System Technical Journal* 40, 1 (1961), 1–41. https://doi.org/10.1002/j.1538-7305.1961.tb03975.x

Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. 2003. View-Dependent Displacement Mapping. *ACM Transactions on Graphics* 22, 3 (July 2003), 334–339. https://doi.org/10.1145/882262.882272

Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. 2004. *Generalized Displacement Maps*. The Eurographics Association. https://doi.org/10.2312/EGWR/EGSR04/227-233

Li-Yi Wei. 2004. Tile-Based Texture Mapping on Graphics Hardware. In *Proc. Graphics Hardware*. 55–63.

Cem Yuksel, Jonathan M. Kaldor, Doug L. James, and Steve Marschner. 2012. Stitch Meshes for Modeling Knitted Clothing with Yarn-Level Detail. *ACM Trans. Graph.* 31, 4, Article 37 (2012), 37:1–37:12 pages.

Kun Zhou, Xin Huang, Xi Wang, Yiying Tong, Mathieu Desbrun, Baining Guo, and Heung-Yeung Shum. 2006. Mesh Quilting for Geometric Texture Synthesis. In *ACM SIGGRAPH*. 690–697.

Figure 9: Once designed, the same tile set can be applied to various macrosurfaces.

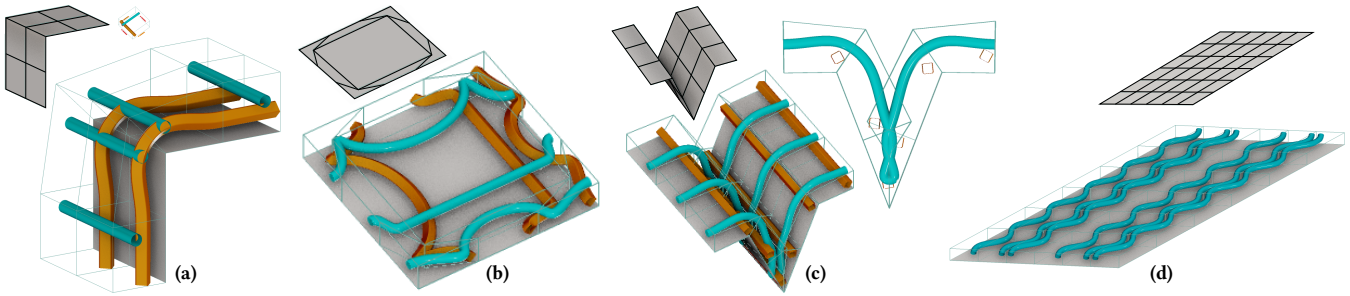

Figure 10: Limitations. Deforming sweep trajectories rather than resulting sweep surfaces conserves cross-sections but may create intersections not present in the unit tiles (a, c). Our construction of sweep trajectories efficiently ensures tangential continuity but is too local to follow alignments at larger scale (b, d). Conservation of straight lines requires macrosurface faces to be close to rectangles.

Figure 11: Additional results illustrating the variety of mesostructures achievable with our system, captured from our real-time generated mesostructures. For each example, we report the amount of synthesized polygons, the GPU memory footprint of our procedural model and the render time on an *Nvidia RTX 3070 Ti* device for 720p images. The last row corresponds to the beauty shots of Fig. 9 and the examples of Tab. 1.