# *geoTangle*: Interactive Design of Geodesic Tangle Patterns on Surfaces



GIACOMO NAZZARO, Sapienza University of Rome
ENRICO PUPPO, University of Genoa
FABIO PELLACINI, Sapienza University of Rome

Fig. 1. Models decorated with *geoTangle* (large images) to mimic the style of real-world artisanal objects (upper insets), together with the input meshes (lower insets: teapot 1.5M triangles, elephant 2M triangles). We model decorations by recursively splitting a surface into progressively finer regions, to which we apply material and displacement variations. All patterns were constructed with only four operators that split regions along the isolines or integral curves of scalar fields derived from geodesic computations.

Tangles are complex patterns, which are often used to decorate the surface of real-world artisanal objects. They consist of arrangements of simple shapes organized into nested hierarchies, obtained by recursively splitting regions to add progressively finer details. In this article, we show that 3D digital shapes can be decorated with tangles by working interactively in the intrinsic metric of the surface. Our tangles are generated by the recursive application of only four operators, which are derived from tracing the isolines or the integral curves of geodesics fields generated from selected seeds on the surface. Based on this formulation, we present an interactive application that lets designers model complex recursive patterns directly on the object surface without relying on parametrization. We reach interactive speed on meshes of a few million triangles by relying on an efficient approximate graph-based geodesic solver.

CCS Concepts: • **Computing methodologies → Graphics systems and interfaces**; **Shape modeling;**

Additional Key Words and Phrases: User interfaces, geometry processing

**ACM Reference format:**
Giacomo Nazzaro, Enrico Puppo, and Fabio Pellacini. 2021. *geoTangle*: Interactive Design of Geodesic Tangle Patterns on Surfaces. *ACM Trans. Graph.* 41, 2, Article 12 (November 2021), 17 pages.
https://doi.org/10.1145/3487909

This work was partially supported by MIUR under grants PRIN DSurf and Dipartimenti di Eccellenza.
Authors' addresses: G. Nazzaro and F. Pellacini, Sapienza University of Rome; emails: nazzaro@di.uniroma1.it, pellacini@di.uniroma1.it; E. Puppo, University of Genoa; email: enrico.puppo@unige.it.


0730-0301/2021/11-ART12 $15.00
https://doi.org/10.1145/3487909

## 1 INTRODUCTION

Over the past 20 years, digital sculpting has been employed extensively in the creation of 3D shapes, allowing artists to easily design highly detailed objects with a natural look. While sculpting, artists work seamlessly on a high-resolution model, disregarding the underlying mesh topology or any UV-unwrapping. All the fine details are directly embedded in the mesh, which is automatically refined as needed during editing. From a technical standpoint, this is made possible by keeping sculpting operations interactive on high-resolution meshes, with a typical target size between one and 10 million triangles. In this article, we propose a similar approach to address the decoration of surfaces with complex patterns.

*Raster* decorations can be added to a surface using texture mapping, which involves surface unwrapping and parametrization. Raster decorations can be painted directly on the object, while the decor is sampled and stored in a texture. Alternatively, procedural textures can be generated in parametric space and then mapped to the object.

When it comes to the creation of *vector* graphics on a surface, these approaches become unwieldy. Vector primitives are specified by control points and mathematical equations. If such primitives are evaluated in parametric space, then computations can be done in closed form, but the result needs to be mapped through parametrization, which involves unavoidable distortions and seams, together with other drawbacks and limitations that we further discuss in Section 2.1.

However, if primitives are specified and evaluated directly on the surface, they must undergo the geodesic metric. Until now, the latter approach was considered prohibitive due to the cost of geodesic computations. In this article, we advocate that producing vector graphics directly in the intrinsic metric of the surface, and embedding the result in the underlying mesh, can lead to an efficient and practical solution that supports interaction while removing all drawbacks of parametrization.

While our approach to vector graphics in the geodesic metric is fairly general, here we focus on the specific domain of *tangles*, a form of decorative art, which subsumes many of the challenges that are not addressed properly in current methods. Tangles consist of complex arrangements of simple shapes, which are organized into nested hierarchies, obtained by recursively splitting regions to add progressively finer details. Due to their detailed, repetitive, and recursive nature, tangles need to be produced with the aid of procedural methods. Besides, they must consist of exact geometries, specified with vector graphics, since the boundaries of each element recursively define regions to be filled with further patterns.

*geoTangle.* We address the interactive design of tangles on surfaces by presenting a general framework of subdivision and grouping operators that act on manifolds. Based on this framework, we implement a prototype system, called *geoTangle*, which supports the interactive design of tangle patterns directly on the object's surface. With *geoTangles*, users can easily design tangles mimicking real-world styles, shown, for example, in Figure 1, Figure 17, Figure 18, and the supplemental video.

We model tangles by the recursive applications of just four subdivision operations, which cut regions along the isolines and the integral curves of geodesic fields computed from appropriately selected and arbitrarily complex seeds. Our system supports user interaction on meshes of millions of triangles, thanks to methods for fast and robust geodesic computation, which are one of the main technical contributions of this article.

To the best of our knowledge, there exists no other method to interactively design tangles on surfaces. While we do not claim to support all possible patterns, our framework is general, compact, efficient, and easily extensible without major modifications to support the design of most geometrically defined patterns in tangle art. Possible extensions are discussed in Section 6.

We validate how well *geoTangle* models real-world decorations in three manners. First, we model decorations to match the styles of real-world artisanal objects. Second, we measure the accuracy and speed of our geodesic solver and of the overall editor to show that we can maintain both accuracy and interactivity while modeling. Third, we run a user study where we validate whether users can easily produce complex patterns with our application.

*Contribution.* Our first main contribution comes form a framework and a prototype system to create easily and interactively complex structured vector patterns on the surface of 3D objects by working directly in the intrinsic metric of the surface. From the point of view of interactive pattern design, this possibility is radically different from previous approaches.

Our second main contribution is a fast geodesic solver that is interactive on meshes with millions of triangles, requires no expensive pre-computation, can be updated efficiently upon mesh editing, and is accurate enough at high triangle count. No other existing geodesic solver exhibits all such characteristics together.

More importantly, we believe that our work demonstrates a new perspective, which is alternative to texture mapping, for the interactive design of complex patterns directly on surfaces. We believe this work is the first that shows how complex patterns can be formalized in the geodesic metric, also showing that interactive vector graphics on detailed surfaces is possible.

## 2 RELATED WORK

We first review related works in general, next we discuss in more detail the limitations of approaches based on parametrization and how we differentiate from the 2D *gTangle* system [Santoni and Pellacini 2016] that inspired our work.

*Digital Sculpting.* In digital sculpting systems, users displace surfaces by applying local edits and paint them to change the appearance [Autodesk 2020; Pilgway 2019; Pixologic 2021]. Generating our patterns using sculpting would take hours of work, even for a skilled artist [Santoni et al. 2016]. Besides, if a pattern needs adjustments, then the entire sculpt needs to be redone, while we can just apply it with modified parameters. The Overcoat system [Schmid et al. 2011] and Model-guided 3D Sketching [Xu et al. 2018] can decorate surfaces with brush strokes, but support only raster decorations. The Polygonal Patterns [Jiang et al. 2015] creates patterns directly with polygon shapes, but lacks the control needed to address different styles.

*Procedural Patterns.* Complex patterns might be generated with procedural methods, for which there exists significant literature.

Ebert et al. [2002] present the basic methods for procedural texturing, which is at the base of most pattern synthesis techniques. All these methods require the user to describe the pattern using either code or visual languages, and pattern generation requires evaluating a function at each point in either 2D or 3D textures. Conversely, we model patterns interactively.

A second class of methods generates new patterns from example images via non-parametric texture synthesis [Sendik and Cohen-Or 2017; Wei et al. 2009; Zhou et al. 2018]. While these methods are remarkably reliable for unstructured patterns, they often fail to capture complex structural properties. This is especially true for recursive patterns, as discussed in Santoni and Pellacini [2016]. Chen et al. [2016] address the problem of packing repeated decorations on a surface, but do not provide the needed structure to produce recursive patterns. Compared to ours, all these approaches generate textures rather than vector graphics primitives on surfaces.

The last class of methods, and the one more closely related to our work, is based on stochastic grammars, traditionally used for modeling procedural architecture [Schwarz and Wonka 2015], which recursively split shapes into smaller components. Santoni and Pellacini [2016] show that group grammars can be used to describe tangle patterns in the 2D domain, and Carra et al. [2019] extend their use to procedural animation. Li et al. [2011] use grammars guided by vector fields to place external details on surfaces, but cannot handle recursive patterns. An alternative approach to grammars is to use a custom programming language to express stationary discrete textures [Loi et al. 2017]. Our system is inspired by Santoni and Pellacini [2016], but it is targeted at the non-Euclidean manifold setting, and at interactive editing, instead of writing grammars or programs. We discuss the main differences in Section 2.2.

*Geodesics.* In our work, we need a geodesic solver that: can compute geodesic distance fields from multiple sources, on meshes up to one or few million triangles, in a time compatible with interaction (i.e., about 0.1–0.2 second per operation); can be updated efficiently upon mesh editing; and is accurate enough at high triangle count. We discuss the literature under this perspective. Broadly speaking, there exist three classes of methods for computing geodesic distance fields and paths. Surveys can be found in Bose et al. [2011]; Crane et al. [2020].

Exact methods for polyhedral surfaces stem from the works of Mitchell et al. [1987] and Chen and Han [1990]. Even the most recent methods in this line [Qin et al. 2016; Ying et al. 2019] are too slow to support interaction on moderately large meshes.

Graph-based methods provide approximated solutions by restricting possible paths to chains of arcs in a graph. The sole network of edges provides poor distance estimation and wiggly paths: Single paths can be straightened by computing shortcuts on-the-fly [Campen et al. 2013; Sharp and Crane 2020], but the global distance field cannot be improved efficiently. Several methods have been proposed, which precompute either an extended graph adding Steiner nodes [Lanthier et al. 1997, 2001] or a subgraph of the complete graph connecting all vertices [Adikusuma et al. 2020; Wang et al. 2017; Ying et al. 2013]. Generally speaking, there exist a tradeoff between the complexity of the graph and of its maintenance, and the accuracy of the approximation. We rely on a simple graph-based method, discussed in Section 4.2, which scales well to large meshes and can be maintained very easily and efficiently upon mesh refinement.

PDE methods define the geodesic distance problem in terms of partial differential equations and compute approximate geodesic distances on a smoothed surface. The Fast Marching Method [Bronstein et al. 2009; Kimmel and Sethian 1998] requires no preprocessing and could be adapted well to local computations and dynamic mesh refinement, but it is overly slow for our needs. A parallel version of FMM [Romero Calla et al. 2019] greatly improves time performances, yet remains slower than our method, with a comparable accuracy. The heat method of Crane et al. [2013] requires resolving sparse linear systems of the size of the mesh. It runs efficiently on relatively large meshes by exploiting prefactorization, but it cannot be extended easily to manage local computations or dynamic mesh refinement. Only the former issue can be partially addressed by incorporating the methods proposed in Herholz and Alexa [2018]; Herholz et al. [2017a]. A very recent parallel version of the heat method of Tao et al. [2021], which does not require any pre-computation, can scale well to large meshes, but it still remains too slow for our purposes.

In Section 5.2, we further compare our method to prior art, and we provide a quantitative evaluation of its performances.

### 2.1 Comparison with Texture Mapping

In tangle generation, each target region, which contains a given pattern, can be arbitrarily extended, complex and curved. To gain an understanding of our typical target shapes and decorations, consider, for instance, the teapot in Figure 1, which is a replica of a real piece of pottery. Each leg of the teapot, as well as its main body, is considered as a single piece to be filled with a pattern that covers it uniformly and seamlessly.

Procedural textures cannot be used, since they do not induce a subdivision of the object, which is needed to create further patterns recursively. Vector tangles could be generated procedurally in parametric space and then mapped to the target object via parametrization. In the latter case, any editing is indirect, hence unnatural and hard to control: The artist edits the decor in parametric space while checking the outcome in object space.

Besides the unwieldy user interface, unwrapping and parametrization must cope with two limitations. From a topological standpoint, the unwrapped surface must have the topology of a punctured disc. Any surface with a different topology needs to be cut open to be unwrapped by introducing *seams*. From a metrics standpoint, any parametrization will induce some *distortion*. Further seams can help to reduce distortion, but this is a typical no-free-lunch problem [Li et al. 2018; Poranne et al. 2017].

Distortion alters the appearance of vector primitives once they are mapped to the target surface, making them hard to control. Seams hinder the creation of primitives that go across them or make them discontinuous. Several methods have been proposed to alleviate the effect of seams. Seamless parametrization methods [Campen et al. 2020; Levi 2021] enforce the parametrization to be smooth across seams. However, this constrains further the already difficult task of defining a usable parametrization for a given surface and increases distortion, concentrating it around "cones." Also, this does not solve the problem entirely, as 2D primitives

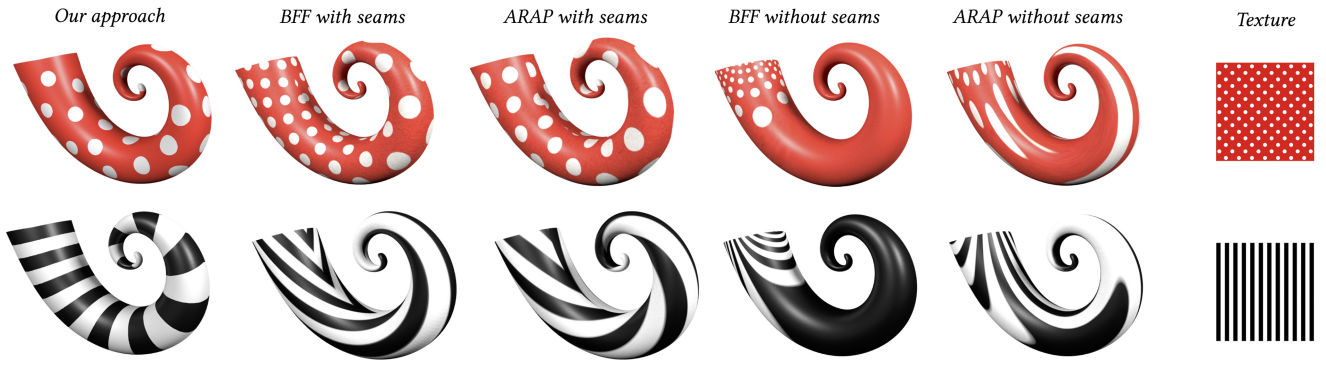


Fig. 2. *Comparison with parametrization.* Polka dots and stripe patterns can be applied to surfaces by tiling textures in the parametric domain. Here, we show parametrization with and without seams generated with BFF [Sawhney and Crane 2017] and ARAP [Liu et al. 2008]. A parametrization with seams introduces discontinuities in the pattern that become visible on the shape, while a parametrization without seams causes strong distortions. The same patterns obtained with our method shows no such artifacts, since we work directly in the geodesic metric.

must still be disallowed to cover a cone. More general techniques that try to achieve inter-chart continuity lead to fairly complicated solutions [Prada et al. 2018].

In Figure 2, we show the effect of performing straight mapping of simple patterns through parametrization, either with or without seams. Our method, which builds patterns directly on the surface by generating geodesic fields on-the-fly, could be seen as a way to obtain locally a proper parametrization for the pattern at hand without the limitations of seams and distortion.

### 2.2 Comparison with *gTangle*

Our work is inspired by *gTangle* [Santoni and Pellacini 2016], which showed that 2D tangles can be modeled using stochastic grammars, where a few operators are applied recursively to obtain complex patterns. Similarly to them, we define our algebra of tangles as the recursive combination of few subdivision and grouping operators. However, there are two fundamental differences between *gTangle* and our approach.

First, *gTangle* is an inherently 2D method, which cannot be extended directly to the geodesic metric. Figure 3 shows a comparison of our method with patterns generated with operators equivalent to *gTangle*'s ones and applied to the surface via parametrization. Patterns generated with *gTangle* are properly defined in the 2D domain, but, when applied to surfaces via parametrizations, they show the same discontinuities and strong distorsions that we have previously discussed. These artifacts derive from the fact that our operators are defined directly in the geodesic metric, while *gTangle*'s operators are defined in the Euclidean 2D metric.

Second, *gTangle* patterns are hard to design, since users have to write a formal grammar. While the authors demonstrate a simple user interface, this is just used to pick from a set of manually edited grammars. Writing grammars is notoriously complex, even when using visual programming languages, like node graphs. Program parameters may be found by optimization [Shi et al. 2020], but editing the procedural program itself is still not possible automatically. Our system has the entirely different focus of designing patterns via user interaction and fully supports selection, grouping, and joining operators.

## 3 DRAWING TANGLES ON SURFACES

In *geoTangle*, patterns are generated by recursively applying a small set of operators to regions of a manifold. Tangles are defined as hierarchies of regions, and operators take as input a tangle, as well as a set of parameters, and produce a modified tangle consisting of refined regions. Figure 4 shows an example of a pattern generation sequence in *geoTangle*.

In Appendix A, we formally define an algebra of tangles over a manifold $\mathcal{M}$, upon which our formulation is built. Informally, the algebra is summarized as follows: (1) The atomic elements of the algebra are regions of $\mathcal{M}$; a tangle $\mathcal{T}$ is a tree of nested regions; the leaves of $\mathcal{T}$ form a partition of $\mathcal{M}$ and are ultimately used to draw the pattern. (2) Subdivision operators modify a tangle $\mathcal{T}$ by partitioning selected regions; subdivision operators are specific and characterize the different patterns that can be generated. (3) Grouping and joining operators permit to treat groups of regions as unique regions to ease editing operations; such operators are generic and defined once for the algebra.

While the algebra is intentionally general and extensible, *geoTangle* is based on a small set of specific subdivision operators. Our purpose here is to demonstrate that a large variety of patterns can be generated on the basis of such operators, while leaving the system open to further extensions.

The main challenge in lifting tangle patterns to the manifold setting is that the 2D Euclidean domain and its global reference system altogether are lost. To support the necessary computations, we need to design our operators upon local, relative reference systems that are suitable for the pattern at hand. Our choice is to rely on patterns that can be defined in terms of distances and directions. We thus review the most common patterns adopted in tangle art in 2D, showing that the underlying geometric concepts can be formalized in that way. Based on such observations, we devise our operators for the manifold setting.

### 3.1 2D Patterns Revisited

A pattern in the 2D plane is defined with an arrangement of lines that partition a region $r$. We assume lines to come from the common arsenal of 2D vector graphics: polylines and scribbles, arcs of

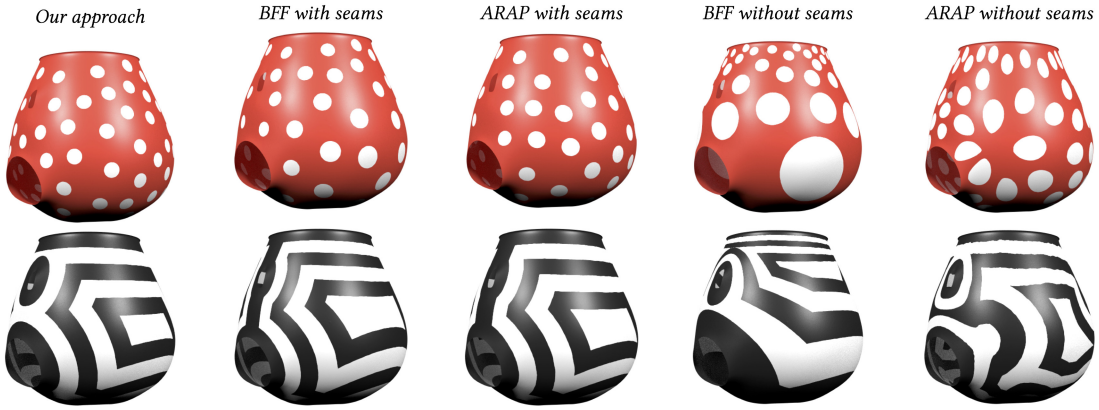


Fig. 3. *Comparison with gTangle.* Patterns generated in the parametric domain for a complex patch with operators equivalent to *gTangle*, parametrized with and without seams, exhibit either artifacts close to the seams or large distortions. When parametrizing without seam, the introduced distortions are too relevant to properly reproduce the *gTangle* pattern on the surface. When parametrizing with seam, the *gTangle* pattern is necessarily altered, since the shape of the domain has changed (see the missing dots in the red example, and the additional stripe in the black-and-white example). Our method provides always correct results, since it computes the pattern directly in the intrinsic metric of the surface providing a correct result.

Fig. 4. Complex recursive patterns are obtained in just 11 steps (left to right, top to bottom) starting at a base shape (upper left). At each step, we apply a basic operation, which recursively splits the surface into finer decorations. The generated patterns naturally follow the shape of the surface, as they are based on geodesic distance fields.

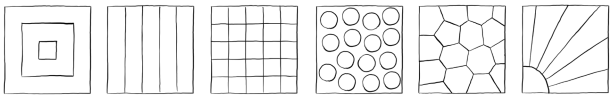

Fig. 5. Examples of typical patterns used in 2D tangles. From the left: outline, hatch, grid, polka dots, Voronoi, radial.

circles and ellipses, and splines. Patterns can be either *absolute,* i.e., defined with respect to the embedding plane, and clipped to the region of interest $r$, or *relative* to the region $r$. In the latter case, the boundaries of $r$ play some relevant role in the definition of the pattern. We list here a few common examples that are also depicted in Figure 5.

*Outlines* provide a typical example of relative patterns. Frames consisting of offset copies of the boundary are drawn towards the interior of the region. The amount of offset and the number of frames define the arrangement. Note that the result consists of contour lines of the distance from the boundary.

*Hatches* are defined by providing a reference line and a displacement direction. Copies of the reference line are displaced for a given offset in the given direction. The pattern is either relative or absolute, depending on whether the reference line belongs to the boundary of $r$ or not. Again, offset lines are contour lines of the distance from the reference line.

*Grids* are obtained by the repeated application of hatches along different reference lines. Using two nearly orthogonal directions defines a quadrangular grid, which may also provide a scaffold to place repeated copies of isolated shapes arranged in a regular lattice. Note also that each cell from hatching or grid implicitly defines a local coordinate frame, which can be used to define further distance-based patterns within it.

*Shape placements* consist of multiple copies of a base shape, placed inside a region $r$ with a prescribed density. Typical shapes are roughly approximated with their minimal enclosing circles, which are virtually placed with some stochastic process, such as Poisson sampling. Then the basic shape is drawn inside each circle, with either a prescribed or a random orientation. Polar coordinates with respect to the enclosing circle may provide a local frame to draw the shape, which is again purely distance-based. Polka dots are the simplest form of placement. Voronoi patterns are a special instance of placement: In this case, just seed points are placed inside $r$ with a sampling process, then the pattern is defined by their Voronoi diagram, which is again defined upon distance.

*Radial patterns* consist of lines (or placements of basic shapes along them) emanating from a focal point or shape. The density

of lines about the focal shape defines the pattern. In this case, the pattern can be formalized as made of integral curves of the gradient of distance from the focal shape.

While the above list is certainly not exhaustive, most geometric patterns can be obtained with combination and generalization of such patterns. See, for instance, Figure 13 for examples of a few macros included in our system, and Figure 9 for more complex examples involving scaffolds. Some relevant classes of patterns that we do not address in our work are discussed in Section 6.

### 3.2 Lifting to the Manifold Setting

Straight-line segments and circles admit straightforward extensions to the geodesic metric as geodesic lines and geodesic circles (i.e., contour lines of the distance from a point), respectively. In *geoTangle*, we stick to such lines and their generalizations that will be described in the following. In Section 6, we discuss how the system could be extended to include splines and ellipses, too.

Our subdivision operators trace lines that are either contour lines of a scalar field $f$ defined over $\mathcal{M}$ or integral curves of its gradient $\nabla f$. We define our scalar fields upon geodesic distances, depending on both the manifold $\mathcal{M}$, and a seed set made of points and lines on $\mathcal{M}$ from which distances are computed. Much flexibility of our operators stems from the possibility to set seeds in a proper way.

We consider two types of scalar fields, shown in Figure 6: The geodesic distance $dist_A(x)$ of surface point $x$ from a seed set $A$, and the blend between geodesic distance fields from two independent seed sets, defined as

$$blend_{A,B}(x) = \frac{dist_A(x)}{dist_A(x) + dist_B(x)}.$$

The blend field, already used in Campen and Kobbelt [2011] to derive parametrizations, has some nice properties: Its values are in the range $[0, 1]$, where $blend_{A,B}(A) = 0$ and $blend_{A,B}(B) = 1$; and it is anti-symmetric in the sense that $blend_{A,B}(x) = 1 - blend_{B,A}(x)$. Moreover, if $A$ and $B$ are lines, then its contour lines are parallel to both of them in their vicinity, and the integral curves of its gradient meet them orthogonally.

Note that the simple case of geodesic lines and circles corresponds to integral curves and contour lines, respectively, of the field $dist_x$ from a source point $x$. Allowing for the computation of distance fields from a source set $A$ different from a single point, and to blend fields, greatly extends the patterns that we can obtain.

### 3.3 The Subdivision Operators of *geoTangle*

We define just four operators, which are exemplified in Figure 7. Our operators are not direct translations of the 2D patterns listed in Section 3.1; rather, their combinations with proper parameters encompass such patterns.

*Contour operator.* The *contour* operator traces the contour lines of the geodesic field. It is defined as

$$(\mathcal{T}, r) \overset{Contour(A,[B],\delta,k)}{\mapsto} \mathcal{T}',$$

where $A$ is the seed set; $B$ is an optional secondary seed set and either the $blend_{A,B}$ or the $dist_A$ function is used, depending on whether or not $B$ is specified; $\delta$ is an offset for the isovalues to be sampled; and $k$ is the number of contours to trace. The operator computes the proper distance field from the seed set(s) provided, then it traces $k$ contour lines of such fields at multiples of $\delta$ and embeds such lines in the underlying mesh representing $\mathcal{M}$.

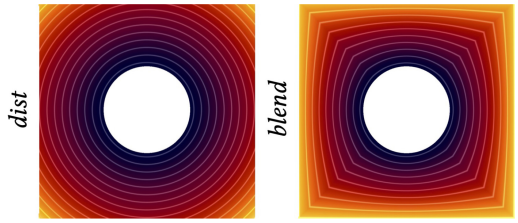


Fig. 6. Field visualization: *dist* field from the center and *blend* field from the center to outer border.

Depending on the selected parameters, this operator can be used to construct a variety of patterns, including contours, polka dots, hatching, and grids (see Figure 7 top row). The contour operator has been used to obtain most of the decorations on the teapot in Figure 1.

*Stream operator.* The *stream* operator cuts regions by tracing the integral curves of the gradient of the geodesic field. It is defined as

$$(\mathcal{T}, r) \overset{Stream(A,[B],\delta)}{\mapsto} \mathcal{T}',$$

where $A$ and $B$ are the seed sets, each of which must be a line or loop, and the field used is defined as in the previous operator; $\delta$ is an offset for sampling $A$. The operator computes the proper distance field from the seed set(s), then it samples points along $A$ uniformly at distances $\delta$ and, for each sampled point $p$, it traces the integral curve of the field gradient, starting at $p$ until reaching $B$ or the boundary of $r$. As before, the traced lines are embedded in $\mathcal{M}$.

The stream operator supports various forms of radial patterns. When applied between the base rings of a cylindrical region it produces longitudinal hatchings and it can be used in combination with the contour operator to produce a grid (see Figure 13). The stream operator has been used to obtain the main pattern of the rug on the back of the elephant in Figure 1, as well as in several parts of the rolling teapot in Figure 17.

*Voronoi operator.* The *Voronoi* operator partitions a region into the cells of a Voronoi diagram in the geodesic metric. It is defined as

$$(\mathcal{T}, r) \overset{Voronoi(A)}{\mapsto} \mathcal{T}',$$

where $A$ is a set of points sampled inside region $r$. The user controls the seeds in the region, which can be either manually selected or generated with Poisson sampling, as described in Section 4.2, to simulate the look of centroidal Voronoi tessellation. We rely on Poisson sampling rather than the CVD to achieve interactivity. We use the Voronoi operator to obtain cellular-like patterns or as a drawing scaffold to place finer details. See examples in Figure 12, as well as on the fertility in Figure 17.

*Polyline operator.* Finally, the *polyline* operator draws a single geodesic polyline on the surface. It is defined as

$$(\mathcal{T}, r) \overset{Polyline(A)}{\mapsto} \mathcal{T}',$$

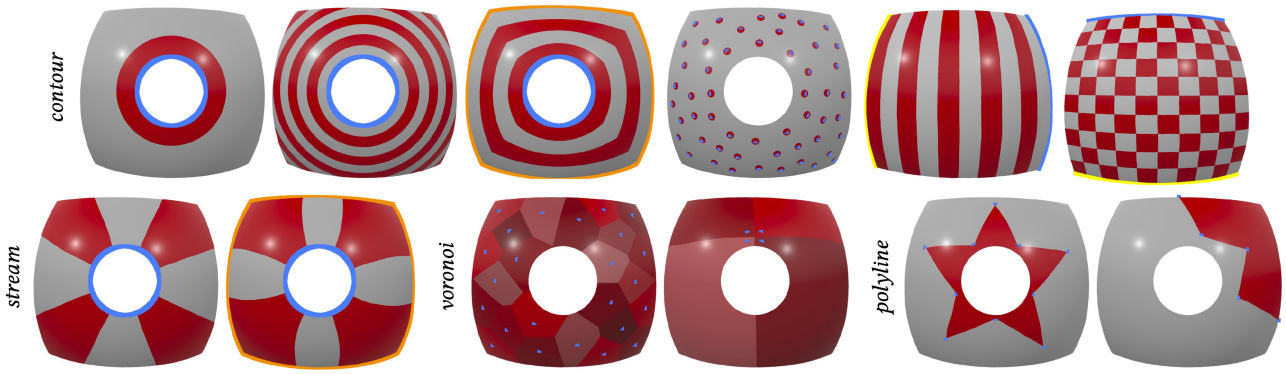


Fig. 7. Example decorations created by applying operators with different fields and seed sets. (Top, left to right) *Contour* operator with (1) *dist* field from the inner boundary (in blue) cutting along a single isovalue; and (2) cutting along multiple isovalues; (3) *blend* field between inner (blue) and outer (orange) boundary, cutting along multiple isovalues; (4) *dist* field from Poisson sampling of points on the surface; (5) *blend* field between left (yellow) and right (blue) boundaries with multiple isovalues; (6) *blend* field between lower (yellow) and upper (blue) boundaries on the same region, already refined with (5). (Bottom left, left to right) *Stream* operator with (7) *dist* field from inner boundary; (8) *blend* field between inner and outer boundary. (Bottom middle, left to right) *Voronoi* operator with (9) *dist* field from points selected with Poisson sampling; and (10) from manually selected points. (Bottom right, left to right) *Polyline* operator with (11) closed polyline on a multiply connected region; (12) open polyline connecting two points on the same boundary loop.

where $A$ is a sequence of seed points that lie inside or on the boundary of region $r$. If the polyline forms a closed loop or it connects two points from the same boundary loop, then region $r$ is split into two sub-regions; otherwise, it is embedded into the region's boundary. For instance, two lines connecting different boundaries of a cylindrical region can be used to cut a longitudinal strip, e.g., as on the spout of the teapot in Figure 1.

The polyline operator is most often used to cut large, meaningful patches to be further refined and decorated, or to provide a scaffold for the application of other operators.

### 3.4 Applying Operators

*Seed selection.* In our prototype, the user can select region boundaries, either completely or partially, as well as points on the surface. A seed set is made of an arbitrary collection of such selections. Points can be either selected individually or sampled according to a Poisson distribution in the geodesic metric. Significantly different patterns stem from different selections, as already shown in Figure 7.

*Grouping and hierarchy.* Combining group tagging and the possibility to apply operators to internal nodes of the tree allows us to easily obtain different patterns and assign colors and materials. See examples in Figure 8 and Figure 13 (grid). This operation is natural in our framework, since field computation and split operators are independent. This same scenario required specialized operations in the group grammars presented in Santoni and Pellacini [2016]. This extension also implies that we are less sensitive to the order in which operations are applied, a common concern in split grammars [Schwarz and Wonka 2015], thus leaving more freedom to the user.

*Scaffolds.* A *scaffold* is a dummy pattern encoded in the tangle tree, which partitions a region without producing any change of appearance, namely, the internal boundaries of this pattern are just virtual and the pattern behaves as a single region. A scaffold is used as a base structure to produce further patterns, e.g., as in the examples shown in Figure 9 and Figure 12.

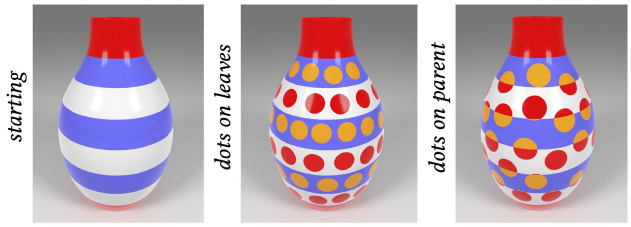


Fig. 8. (Left) The body of the vase has been partitioned into stripes with a contour operator. (Middle) A polka dots pattern is obtained with a contour operator applied to Poisson point sampling in each stripe region separately. (Right) The same pattern is applied on the parent region of the stripes, which is the body of the vase; in this case, the whole surface between the red stripes is considered as a single piece for point sampling, distance computation, and region splitting.

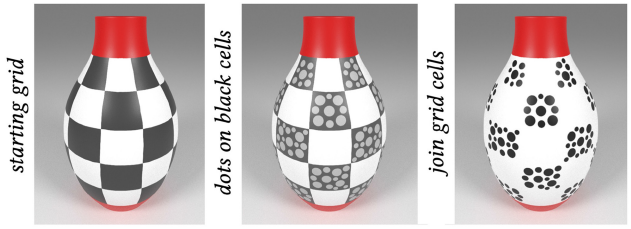


Fig. 9. The initial grid (left) is used as a scaffold to sample seeds in the dark regions (middle), while the final decor consists of just the polka dots (right).

*Perturbation and displacement.* To obtain a more "handcrafted," organic look, the user can perturb geodesic fields to simulate the imprecision of artists' hands, as shown in Figure 10. We do so by offsetting the vertices of the mesh along their normals, using 3D Perlin noise, before computing geodesic distances. This

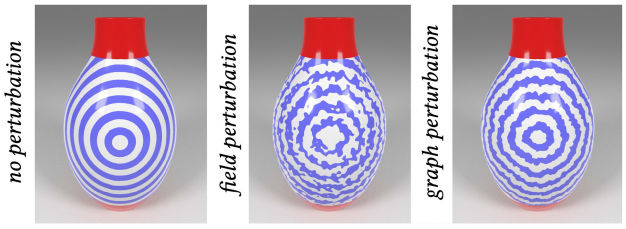


Fig. 10. The same pattern is generated without perturbation, with perturbation of the scalar field and with perturbation applied to the surface before geodesic field computation. Note how perturbing the surface instead of the resulting geodesic field preserves the topology of the generated regions, e.g., the connected components.

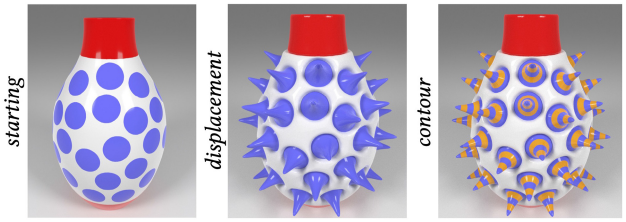


Fig. 11. Example pattern generated applying strong surface displacement using different profiles to create bulges and horn-like structures. Note how we can coherently edit the horns with additional decorations, since distance fields are computed on the updated geometry.

perturbation is used only by the geodesic solver, while the actual mesh is not changed. Geodesic distances computed on such warped geometry are therefore transformed accordingly to a coherent metric and produce irregular shapes, providing organic results. This works significantly better than perturbing the scalar field directly, since the latter can cause unwanted topological changes to the perturbed regions due to the fact that the perturbed field is no longer a distance field. Perturbation can be controlled by setting the gain and frequency of the Perlin noise. For instance, all the irregular blobs on the body of the teapot in Figure 1 are actually circles from a perturbed metric; likewise, the irregular stripe patterns on the spherical bodies forming the lid of the same teapot are obtained with perturbation.

Finally, we allow the user to apply arbitrarily large surface displacement after each operator is applied. The displacement is proportional to the geodesic distance from the region boundaries, being null at the boundaries to avoid discontinuities. Its profile is controlled by applying gain and bias operators from Schlick [1994] to create bumps, pits, ridges, or spikes. This feature has been used extensively in most of our results; see Figures 1, 11, 12, and 17.

*Macros.* Overall, we found that by combining selection, operators, and hierarchy, we can create very complex patterns with ease. We also found that some specific combinations of operators and seed sets were often chosen together to create distinctive and recognizable patterns. To reduce manual work, we bundle these configurations into "macros" in a manner similar to 3D editors. In our application, each macro is represented with just a button in the interface. Figure 13 shows two simple sequences of editing using the eight macros we found most useful. See figure caption for a description.

Fig. 12. Left: The application of the first operators is used to create stripes and grids, which are used as structures to apply further operators and create higher-order patterns. Center: Result obtained with application of three macros: *outline* on the whole surface, *cells* on the two resulting regions, and *outline* again on all cells, which are then displaced. Right: Complex patterns composed of many regions, which are colored following the parity of their group tag.

*User interface.* To create and edit surface tangles, we have implemented a user interface that presents to the user the selection methods and split operators exactly as described previously. Figure 14 shows the user interface that is also demonstrated in the supplemental video. To further simplify editing, we support hierarchy tree navigation and undo. We have used this interface to generate all results in this article.

## 4 IMPLEMENTATION

In the discrete setting, we encode the manifold $\mathcal{M}$ as a finely tessellated triangle mesh $M$. We refine the mesh after applying each operator to precisely embed region boundaries as chains of edges, allowing us to represent regions exactly as groups of triangles. Since we target design applications, we support real-time interaction on commodity hardware on meshes up to a few million triangles.

Our formulation relies on geodesic distance fields, which are notoriously expensive to compute, especially for large meshes. We use a simple graph-based geodesic solver that is efficient, scalable, accurate enough at our tessellation level, and easy to update as the mesh is refined and displaced. We achieve efficiency and scalability by encoding the mesh and the graph with compact and tightly coupled data structures. This data-oriented design has other benefits, since it easily supports undo, serialization, and rendering. We will not discuss these aspects in this article, though.

In the remainder of this section, we discuss implementation details to aid in reproducibility. We will also release an open-source implementation upon article acceptance.

### 4.1 Representation

*Mesh data structure.* We encode triangle meshes with an indexed data structure augmented with face adjacencies, a.k.a. winged data structure [Paoluzzi et al. 1993], compactly stored in three arrays: positions (`float[3]`), triangles (`int[3]`), and face adjacencies (`int[3]`). The indexed format provides a minimal representation of geometry and connectivity, while adjacencies provide support for efficient line tracing, region flooding, and boundary

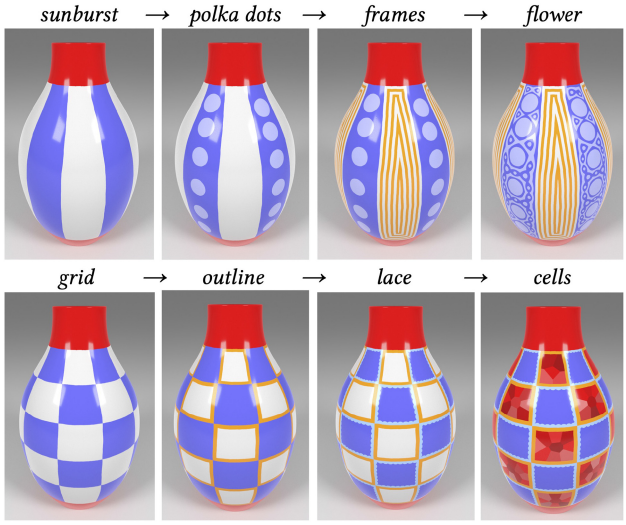


Fig. 13. Example decoration created by applying four macros successively. (First row, left to right): Example decoration applying four macros successively. (1) The *sunburst* macro creates longitudinal stripes or wedges, connecting the two borders of a cylindrical region by applying the *stream* operator and the two boundaries as seed sets. (2) The *polka dots* macro, in the blue region, creates dots by applying the *contour* operator to a set of Poisson-sampled seed points from the selected region. (3) The *frames* macro generates concentric outlines by applying the *contour* operator recursively to the boundary of the selected region, which in this case is the white one. (4) The *flower* macro creates flowing decorations by applying the *contour* operator to the *blend* between the distance from the region boundaries and the distance from Poisson-sampled points. (Second row from the left) Example decoration created by applying four macros successively. (5) The *grid* macro generates checkerboards by recursively applying the *contour* and *stream* operators between the opposite boundaries of a cylindrical region. (6) The *outline* macro creates outlines by applying the *contour* operator to all the boundaries of the selected boundaries. (7) The *lace* macro creates embroidery-like decorations by applying the *contour* operator to seed points uniformly sampled from the boundaries of the selected region. (8) The *cells* macro creates cellular patterns by applying the *Voronoi* operator to Poisson-sampled seed points.

computation. We have preferred this simple triangle-based data structure over more general and popular edge-based data structures, such as the half-edge [Weiler 1985], just because it is more compact and easier to update upon mesh editing, yet sufficient for our needs.

*Pattern representation.* We explicitly encode the tree of patterns $\mathcal{T}$, as described in Appendix A, which maintains the hierarchy as well as the tags of all regions. The overhead of this data structure is negligible, as it contains at most a few thousand nodes even on our most complex results, as shown in Figure 1.

We embed all boundaries of regions in the underlying mesh by cutting it each time an operator is applied. This is very convenient, because we can store the boundaries of each region $r_i \in \mathcal{T}$ as chains of edges. Also, each region is represented exactly as a collection of triangles of $M$: We thus label each triangle with the region identifier of the region containing it.

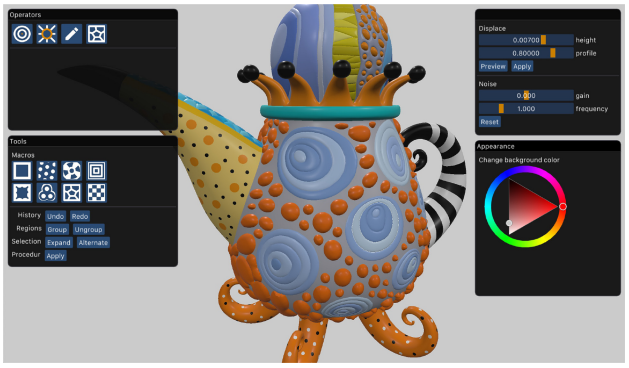


Fig. 14. We implemented an application for real-time editing that let the user decorate an input model applying the operators described in Section 3.3 directly. The user interface also features utilities for all the operations described in Section 3.4, such as surface displacement, perturbation, and *macros*.

Compared to storing a hierarchy of meshes, this representation is both significantly more compact and does not need to be updated as the mesh is refined. It also supports efficiently editing operations such as serialization, undos, and so on.

*Field representation.* Scalar fields are encoded at mesh vertices and extended by linear interpolation, while their gradient field is piecewise-constant on triangles. Operators compute either contour lines or integral curves, including geodesic paths, which cut the mesh along polylines. We split all triangles intersected by such polylines every time an operator is applied.

## 4.2 Geodesic Computations

*Geodesic Graph.* Our solver is implemented using the graph exemplified in Figure 15. Nodes correspond to mesh vertices, while arcs correspond to mesh edges as well as dual edges, i.e., arcs connecting pairs of vertices opposite to an edge. The length of each arc is computed as the exact polyhedral distance between the vertices it connects. Arc lengths are stored in single precision to reduce memory usage, as our method is not prone to high error propagation—distances are just summed during graph visit—while most approximation error stems from discretizing geodesic paths along arcs of the graph.

We store the graph as adjacency lists with a simple array of arrays data structure, where we employ small vector optimizations for the adjacency lists. This solution ensures that most of the graph is laid out on a single contiguous chunk of memory, which reduces heap pressure and improves cache locality during the graph visit.

We build the graph once at the beginning of the editing session by using face adjacencies to construct dual edges. Then, we *locally* update the graph after each mesh refinement operation: Updates only involve nodes incident at split triangles, which are retrieved easily and efficiently, thanks to the implicit connection between vertices and nodes, and edges, adjacencies, and arcs. Updates upon displacement are also efficient, since they only require recomputing edge lengths without modifying the graph topology.

*Geodesic Solve.* We compute geodesic distance fields by wavefront expansion over the graph, shown as pseudocode in

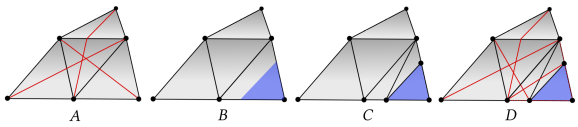


Fig. 15. (A) Our graph has one node for each vertex in the mesh (black dots), a bidirectional arc for each edge in the mesh (black lines) and for each pair of vertices that are opposite to an edge (red lines). The weight of each arc is equal to the geodesic distance between the connected vertices. (B) When an operator is applied, the surface is partitioned into new regions (blue and grey). The boundary that separates the new regions is a polyline crossing the edges of the triangles in the mesh. (C) The mesh is refined to embed the polyline, so each triangle belongs to one region. (D) Only arcs in the graph traversing the split triangles are updated.

Algorithm 1. We adopt the **small-label-first (SLF)** and **large-label-last (LLL)** as search heuristics [Bertsekas 1998]. Although the worst-case time complexity of such techniques is suboptimal, on mesh-like graphs they perform significantly faster than a standard propagation method, like a Dijkstra search or the FMM [Wang et al. 2017]. A Dijkstra-like search visits the same node as few times as possible, but it requires a priority queue, which is slower to maintain than the simple double-ended queue used with the SLF and LLL heuristics. We implement this queue efficiently with a circular buffer, which can be accessed and updated with almost the same speed as a plain array. This has a dramatic impact on practical performance, as we show by comparing the two algorithms in Table 2.

Note also that, differently from Dijkstra search, our algorithm lends itself to parallelization, e.g., by using a double-ended queue with concurrent access [Graichen et al. 2016]. Such an extension is beyond the scope of this article, but it might improve the performance of our system to work on even larger meshes.

Our implementation aggressively exploits computation locality by applying early exits when bounding the graph search to a region. To limit the geodesic computation to a portion of the surface, before the visit, we set the initial distance field to an infinite value only in the area of interest and set it to zero everywhere else and in the source vertices. The solver computes distance fields from any given set of source vertices, while lines are just sampled at their vertices. Note that we use our solver just to compute the geodesic distance, while we do not trace geodesic paths with sequences of arcs in the graph, as the latter would result in wiggly polylines.

*Poisson sampling.* We use Poisson sampling, shown as pseudocode in Algorithm 2, to generate seed sets for various operators. We adopt a farthest point sampling technique [Eldar et al. 1997] under the geodesic metric. This scheme requires computing a distance field for each sampling point, which becomes too expensive for interaction with more than a few samples. We take advantage of the wavefront nature of graph search to significantly reduce the computation time.

We begin by computing the distance field from the region boundary. Then, we iteratively select the vertex with maximum distance and add it to the seed set. We update the same distance field by expanding from the new seed without resetting the already computed distances. This makes sure that each new visit expands only in the proximity of the new seed, since it stops when hitting nodes

**ALGORITHM 1:** Visit of geodesic graph (`visit_graph`)

**Input**: Graph $G$, Sources $S$, Initial geodesic distance field $D$, Distance threshold $r$
**Output**: Updated geodesic distance field $D$

```
Q ← S  // initialize node queue
w ← 0  // total weight of the queue
while Q ≠ ∅ do
    // LLL heuristic: pick the first node with
    // weight less than the queue average
    node ← pop_front(Q)
    while D[node] > w/|Q| do
        push_back(Q, node)
        node ← pop_front(Q)
    w ← w − D[node]
    // check for early exit
    if D[node] > r then continue
    // visit neighbors to propagate distances
    foreach neighbor, length ∈ G[node] do
        d_old ← D[neighbor]
        d_new ← D[node] + length
        if d_new ≥ d_old then continue
        if neighbor ∈ Q then
            // node already in queue, update total weight
            w ← w − d_old + d_new
        else
            // add neighbor to queue with SLF heuristic
            if d_new < D[front(Q)] then
                push_front(Q, neighbor)
            else
                push_back (Q, neighbor)
            w ← w + d_new
        D[neighbor] ← d_new  // propagate distance
return D
```

with a smaller distance from the previous set. Each new sample is increasingly cheaper to compute, resulting in a sub-linear time complexity, as shown in Figure 16. After a certain number of samples, computation time increases linearly but very slowly, becoming dominated by the cost of computing *argmax* in the distance field array. This is another advantage of a graph-based solver over alternatives that are non-local.

*Voronoi.* Generating a Voronoi diagram requires the computation of a distance field for each element in the seed set, which is too expensive. Again, we speed up this operation by exploiting early exits in graph search. As shown in Algorithm 3, we first compute the distance field from all seeds together to find its maximum value. We then set this distance as bound for early exit when computing the field for each seed. Intuitively, this ensures that each mesh vertex is visited roughly twice, so computation time is roughly equal to twice the cost of a full visit, regardless of the number of seeds. Figure 16 shows that the increase in time is about linear, but the slope is very small: Note that with 130 generators the time is about only three times the cost for a complete solve from a single source.

For each vertex of the region of interest, we collect the distance from its three closest seeds and we generate the Voronoi

**ALGORITHM 2:** Poisson sampling

**Input**: Graph $G$, Selected region id $R$, Minimum distance between samples $r$, Max number of samples $N$
**Output**: Set of sampled vertex ids $S$
**begin**
  $S \leftarrow \emptyset$
  B $\leftarrow$ `boundary_vertices(`$R$`)`
  $D \leftarrow$ `init_field(`$R$, B`)` // init field to confine visit to region
  **if** B $\neq \emptyset$ **then**
    // offset sampled points away from the boundary
    $D \leftarrow$ `visit_graph(`$G$, B, $D$, $+\infty$`)`
    **foreach** d $\in D$ **do** d $\leftarrow$ d $+ r/2$
  **while** *true* **do**
    v $\leftarrow$ `argmax(`$D$`)` // choose new sample
    **if** $D$[v] $< r$ **then break**
    $S \leftarrow S \cup$ {v} // add new sample to result
    **if** $|S| = N$ **then break**
    // update distance field with new sample
    $D$[v] $\leftarrow 0$
    $D \leftarrow$ `visit_graph(`$G$, {v}, $D$, $+\infty$`)`
  **return** S

**ALGORITHM 3:** Voronoi field generation

**Input**: Graph $G$, Selected region id $R$, Set of seed vertex ids $S$
**Output**: Array of triples of closest generators $I$, Array of generator distances $H$
**begin**
  $H$[:] $\leftarrow (+\infty, +\infty, +\infty)$ // clear all entries
  $I$[:] $\leftarrow (-1, -1, -1)$ // clear all entries
  $D \leftarrow$ `init_field(`$R$, $S$`)` // confine visit to $R$ from $S$
  $D \leftarrow$ `visit_graph(`$G$, $S$, $D$, $+\infty$`)` // distance field from $S$
  $r \leftarrow$ `max(`$D$`)` // bound visit to small distance
  // find distance for each generator
  **foreach** $s \in S$ **do**
    $D \leftarrow$ `init_field(`$R$, {$s$}`)` // confine visit to $R$ from $s$
    $D \leftarrow$ `visit_graph(`$G$, {$s$}, $D$, $r$`)` // distances from $s$
    // find three closest generators and their distance
    **foreach** $v \in R$ **do** update $I[v]$ and $H[v]$
  **return** $I$, $H$

diagram by splitting all triangles that have vertices lying in different Voronoi regions, as in Herholz et al. [2017b].

*Line tracing.* Operators require extracting contour lines and integral curves, as well as cutting the mesh with such lines. Contour lines are extracted per triangle by linear interpolation. Integral curves and geodesic paths are computed by tracing the piecewise-constant gradient per triangle. Each triangle intersected by one such line is split along the corresponding segment, forming three new triangles. When a triangle is split, arcs and nodes are added to the graph to represent new vertices and edges, and the adjacency of nodes in the split triangle is updated accordingly, as shown in Figure 15.

*Operators.* The operators described in Section 3 are implemented on top of the functionality described before. The *contour* operator requires one or two geodesic solves, for *dist* and *blend*, respectively, followed by the extraction of isolines and cuts along them. The *stream* operator requires the same solves, this time followed by cuts along integral curves. In both cases, computation is bound to the region in which the operator is applied. The *polyline* operator is implemented by tracing geodesic paths between each pair of successive points. Each segment is computed starting at a vertex and ascending the gradient of the distance field generated from the previous vertex. This operation requires one solve per segment: Early exit occurs as soon as the target vertex is reached, so the computation is bound to the intersection between the selected region and a geodesic circle having the segment as radius. Finally, shape perturbation and displacement are trivially implemented by updating the edge lengths in the graph and the positions of vertices, respectively.

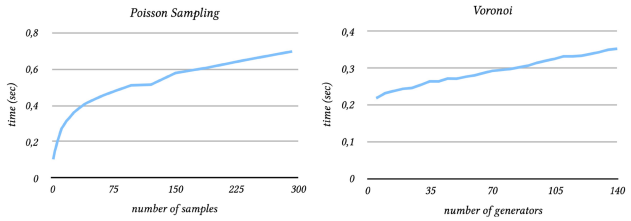


Fig. 16. Computation time for Poisson point sampling and Voronoi on the elephant model (2M triangle) as the number of samples increases. By exploiting the locality of the graph-based distance computation, we drastically reduce the time complexity.

*Discussion.* Our system is compact and coherent, because all operations rely on geodesic distance computation, as well as on few other straightforward operations, namely, line tracing and mesh cutting. Our choices were motivated by the constraints inherent to interactive editing of large meshes: time efficiency, scalability, and ease of update upon mesh modifications.

We experimented with several other geodesic solvers in the literature, but our specific graph solver provides an optimal tradeoff under a variety of aspects, such as accuracy, speed, scalability, simplicity, dynamic update, and early exits. The graph already provides a compact "precomputed" structure that can be quickly updated. On the contrary, computation of a geodesic field from a given source in general provides no hint on computation of the field from a different source. Scalability, simplicity, and ease of update descend from using just the vertices of the mesh as nodes and relations from local mesh topology as arcs. Nodes in our graph have an average degree of 12, hence for a mesh with $N$ vertices our graph has $N$ nodes and about $6N$ bidirectional arcs.

In contrast, graph-based methods using Steiner nodes [Lanthier et al. 1997, 2001] are much more complex to maintain upon dynamic updates and have a larger memory footprint. Their number of arcs increases quadratically with the number $k$ of Steiner nodes per edge: for $k = 3$ there are $\sim 10N$ nodes and $\sim 84N$ arcs, making these graphs impractical for large meshes even with moderately low values of $k$.

The DGG [Adikusuma et al. 2020; Wang et al. 2017] and SVG [Ying et al. 2013] methods use graphs that do have just the vertices as nodes, but each node has a high degree (order $10^2 - 10^3$).

Their memory footprint is high and they are slow to update upon mesh refinement. Finally, the methods proposed in Campen et al. [2013]; Sharp and Crane [2020] rely just on the graph of edges, but computations to straighten paths are done on-the-fly, making them suitable for point-to-point path computation only.

With the PDE method of Crane et al. [2013], each solve implies the solution of a sparse $N \times N$ linear system, which can be pre-factorized for a given mesh, making the solution fast at the price of pre-computation. The main concern in using this method is that every time we change the mesh topology or geometry the expensive factorization step needs to be repeated. See Section 5.2 for a comparative analysis in terms of accuracy and performance.

The sequential FMM [Kimmel and Sethian 1998], which requires no pre-processing, has been compared to the heat method by several authors in the literature [Crane et al. 2013; Romero Calla et al. 2019], always resulting orders of magnitude slower, with a similar accuracy. Even the fastest parallel implementation presented in Romero Calla et al. [2019] resulted slower than the heat method for meshes of moderate size, and just slightly faster for large meshes. By comparing the timings reported in Tables 2 and 3 from Romero Calla et al. [2019] with the timings of our Table 2, it is evident that our simple sequential implementation remains competitive even with respect to their parallel FMM.

Our implementation is already sufficient to support interaction on millions of triangles on commodity hardware. If more efficient solvers are proposed, which are compliant with our constraints, then they could be integrated into our framework without changing it. In particular, if parallel algorithms can be used, then they can surely improve scalability of the system further. However, we consider this contribution to be orthogonal to our framework, which already achieves performance on non-trivial objects.

## 5 RESULTS AND VALIDATION

We validated our work in three manners. First, we modeled decorations that mimic real-world styles. Second, we tested the accuracy and speed of the overall system and of the geodesic solver to show that it remains interactive. Third, we validated our user interface with a user study to show that novices to the system can replicate given patterns.

### 5.1 Editing Sequences

In the beginning, the whole surface is seen as an individual patch to be filled with a pattern. The initial subdivision can be obtained with any of our operators: with closed polylines, with contours or Voronoi patterns from manually selected seeds, through Poisson sampling, with polylines, contours, or streams that connect borders. The subsequent operations will always refer to bordered regions.

Figure 1, Figure 17, and Figure 18 show complex patterns created with our system during interactive sessions. We chose to model patterns that mimic real-world examples with different artistic styles to show that our model can capture intricate decorations made by artists. Table 1 shows statistics of the editing sequences corresponding to such images.

Overall, we found that creating complex patterns is easy with our interface. We used from up to hundreds of single operations to create patterns made of up to 1,800 individual decorations, which correspond to regions in our model. Note that some such operations were applied as macros, simplifying editing further. The number of operations we employ is significantly smaller than using standard modeling tools with either polygonal modeling or sculpting workflows, e.g., see Denning et al. [2011, 2015] for statistics of common modeling sequences.

To gain a better sense of the recursive nature of the decorations, we report the depth of the pattern tree, which reached 21 in our most intricate result. This shows that by applying patterns recursively, even just a few times, we can greatly increase the complexity of the decoration while maintaining the editing manageable for users.

### 5.2 Performance and Accuracy

In our examples, we handle models between 500k and 2M triangles, which are further subdivided during editing. Throughout the modeling sequences, our system remains interactive with computation times of about 0.2 s per operation, including geodesic computation, mesh cutting, and graph update. Memory usage is also compact, never exceeding 300 Mb, which includes the mesh and the geodesic solver, as well as interface support data. Performances were evaluated on a 2.9 GHz laptop with 16 GB of RAM running on a single-core for our application.

Fast geodesic computation is the main technical feature that enables us to model decorations well. We test the performance and accuracy of our solver by computing the geodesic field from a single source to all vertices on a variety of meshes, summarized in Table 2. While we use a very simple graph, our solver remains accurate enough with an error between 1.1% and 1.6% over an exact polyhedral solution. Computation times are between 0.015 s and 2.951 s for a single-core implementation running on meshes between 300k and 28M triangles. This speed is fast enough for all our modeling needs.

Following the discussion in Section 4.2, we compared our solver to a reference implementation of the FMM [Jacobson 2021] and to the author's implementation of the heat method [Crane et al. 2013], also using as a reference the exact polyhedral solver VTP [Qin et al. 2016], which is a state-of-the-art MMP-like method. The accuracy of our solver is just slightly lower than the heat method on relatively small meshes, while being better on large meshes; the FMM method is consistently more accurate, with an error two to five times smaller than our method. However, our solver consistently runs at roughly twice the speed of the heat method and about 40 times faster than FMM. Our method always remains compatible with interaction, even when considering the additional times for update after mesh cutting. Conversely, the FMM method is definitely too slow to support interaction on large meshes; while the heat method cannot be updated after mesh cutting without undergoing the cumbersome pre-processing step.

To stress the performance of our method, we have run experiments on meshes containing up to a few tenths of millions triangles. Although, with such a large meshes, a complete solve over the whole mesh may take more than one second, we remark that our method becomes faster as the target region becomes smaller. Therefore, even with such very large models, we expect it to be

Fig. 17. Results created with our application starting from undecorated models, shown in the insets. Decorations were inspired by real-world examples: The tank-teapot on the left reproduces the playful look of handmade toys, the result in the middle matches the appearance of carnival masks of Venice, the statue on the right is decorated with intricate patterns that imitate tangles on ceramics. Statistics about the models and the editing sessions are reported in Figure 1.

Table 1. Statistics on the Editing Sequences Used for Interactive Decoration

| model name | triangles at start | triangles at end | number of regions | number of operations | tree depth | average time per operation | memory usage at end |
|---|---|---|---|---|---|---|---|
| fertility | 0.50M | 0.90M | 1,810 | 655 | 21 | 0.134s | 96Mb |
| dinopet | 0.60M | 0.81M | 1,304 | 212 | 6 | 0.141s | 110Mb |
| teapot | 1.50M | 1.65M | 511 | 129 | 13 | 0.146s | 173Mb |
| tank | 1.45M | 1.68M | 636 | 145 | 10 | 0.189s | 174Mb |
| mask | 2.00M | 2.15M | 62 | 30 | 5 | 0.232s | 228Mb |
| elephant | 2.00M | 2.61M | 364 | 212 | 13 | 0.241s | 278Mb |

The average time per operation takes into account the time needed to compute the geodesic field, cut the mesh, update the graph and the pattern representation data.

Table 2. Comparison with the Exact VTP [Qin et al. 2016], the FMM [Jacobson 2021; Kimmel and Sethian 1998], and the Heat Method [Crane et al. 2013] (For the Latter, We Use the Implementation Provided by the Author, using Cholmod as Backend)

| model | | VTP | FMM | | heat | | | ours | | | | Dijkstra |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| name | triangles | solve | solve | error | build | solve | error | build | solve | error | update | solve |
| kitten | 300k | 3.0s | 0.813 | 0.5% | 1.49s | 0.068s | 0.5% | 0.061s (24x) | 0.015s (4.5x) | 1.1% | 0.013s | 0.036s |
| elephant | 500k | 10.9s | 1.572 | 0.4% | 3.47s | 0.095s | 0.5% | 0.098s (35x) | 0.027s (3.5x) | 1.1% | 0.023s | 0.063s |
| fertility | 500k | 1.489 | 3.5s | 0.4% | 2.86s | 0.123s | 0.5% | 0.109s (26x) | 0.026s (4.8x) | 1.1% | 0.023s | 0.061s |
| lucy small | 525k | 2.064 | 9.1s | 0.7% | 1.98s | 0.082s | 1.5% | 0.175s (11x) | 0.034s (2.4x) | 1.6% | 0.025s | 0.076s |
| mask | 2.0M | 121.0s | 8.674 | 0.3% | 14.4s | 0.391s | 1.1% | 0.585s (24x) | 0.114s (3.4x) | 1.2% | 0.115s | 0.281s |
| nefertiti | 2.0M | 20.3s | 10.957 | 0.3% | 14.8s | 0.345s | 0.9% | 0.730s (20x) | 0.149s (2.3x) | 1.5% | 0.139s | 0.329s |
| dragon | 7.2M | 79.9s | 121.300 | 0.4% | 59.6s | 1.500s | 3.0% | 2.613s (24x) | 0.446s (3.9x) | 1.5% | 0.571s | 1.110s |
| thai statue | 10.0M | 83.1s | 196.519 | 0.6% | 99.0s | 2.724s | 6.9% | 5.317s (18x) | 0.935s (2.9x) | 1.6% | 0.770s | 1.927s |
| lucy | 28.0M | – | – | –% | – | – | – | 16.09s (– x) | 2.639s (– x) | – | 2.951s | 6.091s |

Columns *build* report pre-processing times to pre-factor the system and to build the graph, respectively. Build times for our solver also include the time to compute the triangle adjacency needed to build the graph. Columns *solve* report average time for computing the distance field from a single point source, where average is taken over 100 random samples. We report root-mean-square errors between results of approximated methods and the polyhedral solution from Qin et al. [2016]. Speedup factors reported between brackets are related to the heat method, which is the fastest of competitors. Column *update* reports the average time to update our graph after mesh split, where the average is taken over 100 different long slices that roughly cut the mesh in half. The last column reports times to run a Dijkstra search on our graph. The last three models from the Stanford 3D scanning repository (http://graphics.stanford.edu/data/3Dscanrep/) have been used only for timing experiments. In the cases of very large models, VTP and FMM fail to complete, while the heat method generates unrecoverable numerical errors.

Fig. 18. Decoration of an organic surface obtained using noise and displacement. Note how operations can be applied over already displaced regions.

slow for the first few operations, then accommodate a more responsive frame rate.

In terms of pre-computation times, the heat method needs factorizing a sparse matrix of the same size of the mesh. Such high times suggest that it would be hard to try a dynamic update after mesh edit. Conversely, the time spent to build our graph is consistently shorter than the time to load a mesh from disk and build a standard data structure.

We also compared our solver to a straightforward implementation of Lanthier's graph [Lanthier et al. 2001] with just one Steiner point per edge, using the same graph traversal algorithm. In terms of accuracy, results are comparable to our solver. We cannot objectively compare solve times, as the two implementations were not equally optimized, but on average our solver was about 10 times faster. Regardless of low-level optimizations, we assume our method to be more efficient, since Lanthier's graph with one Steiner node per edge is more than three times larger, as explained in Section 4.2. Beyond accuracy and performance, our graph is much easier to maintain upon mesh cutting, and that was the determining factor, which made it a better choice for our application.

Finally, we evaluate the performance of running a standard Dijkstra search on our graph, rather than using the SLF/LLL heuristics of our solver. The timings of the Dijkstra search result consistently more than twice slower than ours.

### 5.3 User Study

We ran a user study to validate whether our prototype system is easy to use, whether it allows users to model tangle patterns.

*Experimental procedure.* We asked 17 subjects with different degrees of expertise, ranging from novices to professional 3D artists, to use our prototype after a short training and an unguided editing session on a model of their choice. We asked subjects to perform three matching tasks of increasing difficulty, in which they had to use the application to reproduce a target image, shown in a picture. For these tasks, we chose the mask model as input mesh, which has non-trivial topology but is also easy to navigate in a 3D viewer. Images are provided in the additional material and in Figure 19.

After each task, subjects were asked to rate the similarity of their results with the reference and to evaluate how easy they found it to complete the task, using a scale from 1 to 10. We also asked subjects to rate whether they would have been able to obtain the same kind of results with a different 3D application, whether they found the interface responsive, and whether they were satisfied with the overall experience. We include a copy of the final questionnaire in the supplemental material.

*Quantitative Evaluation.* Figure 19 shows the results of our user study, where for all ratings we rejected the null hypothesis ($p \leq 0.05$), i.e., those results are statistically significant.

All 17 users were able to successfully complete the reproduction tasks, spending different amounts of time in the editing session, but never more than four minutes for each task, out of a maximum task length of five minutes. All users rated their results quite similar, if not identical, to the reference ones. This suggests that our interface provides sufficient control to reproduce given complex patterns. All subjects also found the tasks easy to perform and reported that they would not have been able to obtain the same kind of results with a different 3D editing software.

In general, all subjects also found the interface responsive and were satisfied with the overall experience with the application, confirming that our implementation remains interactive at all times.

In conclusion, the user study demonstrated that users agree that our application is expressive, easy to use, and can produce results that match the look of real decorated objects.

*Qualitative Feedback.* Some non-expert users informally reported that they were surprised by the complexity of the results they managed to obtain with the application and the ease with which they were able to control the editing operations. We think that this cannot be explained only by the usability of the interface, but rather it is a direct consequence of the intuitive design of our editing operators, which requires no expertise to be understood. These facts suggest that the editing workflow of our application is well-suited for non-technical artists and designers, too.

Experts users reported that they found the application responsive and pleasant to use. We quote here some informal feedback we collected: "The editing was surprisingly fast and enjoyable. I did not have to think about triangles, edge loops or topology issues as in Maya or Blender; I could just focus on the result."

## 6 FUTURE WORK AND LIMITATIONS

*Future Work.* The architecture of *geoTangle*, which is based on the tangle algebra and on our geodesic solvers, can be extended in various ways, by plugging more operators and tools into the same framework. All extensions listed below are orthogonal to the contribution of this article, so we propose them for possible future work.

One possible extension consists of supporting more geometric primitives to draw boundary lines. In a companion paper [Authors 2021], we show that interactive Bézier splines can be ported to manifolds. Such technology could be seamlessly integrated into

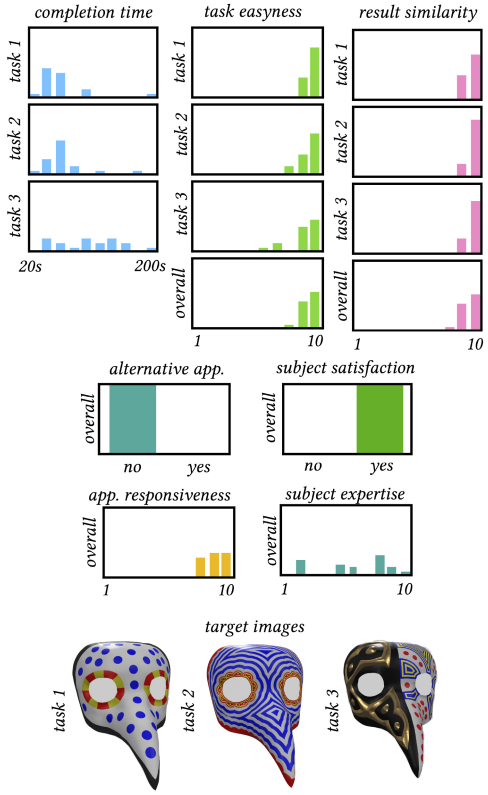


Fig. 19. Result data from the user study. Histograms on the left report the time spent by the subjects to complete each task. The remaining bar charts show subjects' feedback about their tasks: Users with different amount of expertise found the tasks easy to perform, their results similar to the reference ones and the application responsive. All subjects were satisfied with their editing experience with the application and do not think that using another digital tool would allow them to obtain the same results at the same time.

*geoTangle*, being fully compatible with its data structures, and it can address efficiently meshes of the same size.

Another possible extension is the placement of multiple instances of a structured shape, e.g., SVG. The basic shape can be drawn in 2D. The control points of the drawing are mapped to the target regions either with normal mapping (for circular regions) or with bilinear coordinates (for quadrangular regions), and the replicas are drawn directly inside the target regions by using geodesic primitives. Note that we do not need to compute a dense mapping (local parametrization) from the reference region to the target region, since a sparse mapping of control points is sufficient, which can be computed efficiently on-the-fly with simple extensions of our geodesic computations.

Directional fields on surfaces have been extensively studied in the literature, and their interactive design and control can be effectively supported [Vaxman et al. 2017]. Directional fields can be used to generate patterns based on further streamlines, which can be easily combed and constrained to lines and boundaries in the decor. In this case, we just substitute such fields to the gradients of our geodesic fields. Frame fields can be used to warp the geodesic metric to an anisotropic metric, thus allowing for a seamless and controlled perturbation of geodesic fields.

Finally, diffusion curves [Orzan et al. 2008] can be naturally incorporated, too, to support smooth-shaded coloring of regions.

*Limitations.* While we introduce a model for tangles on surfaces, not all patterns can be easily reproduced. For example, recursive patterns based on tight packing of arbitrarily-shaped elements cannot be easily reproduced in terms of geodesics. While there is a large literature on this, we remind the reader that packing is NP-hard in general, so all methods proposed so far are necessarily strong approximations.

Another type of pattern that we did not specifically include is floral decorations. One possibility would be to adopt a region-growing model similar to the one used in procedural trees [Longay et al. 2012] and express it in terms of geodesic paths. The main concern though is that controlling floral arrangement over an arbitrary manifold remains hard, since there is no global orientation to use while growing. We leave the investigation of such further patterns to future work.

A possible concern of our implementation is that we do not add mesh details besides the ones arising from splitting regions. This may lead to aliasing artifacts in the generated patterns, for example in the streamlines case. In our experience, this did not happen, since we always use very detailed meshes. If this were to be a problem, since all operations are procedural, we could re-apply them on a subdivided copy of the mesh to get higher resolution without losing edits. This is the approach used in Adobe Substance Designer to solve a similar issue in 2D [Adobe 2020].

## 7 CONCLUDING REMARKS

We have presented an interactive method for generating recursive patterns on surfaces and its use to model real-world decorations. Our model consists of a closed algebra of regions, which can be split by applying four operators at will. Operators are defined upon geodesic fields on the surface and our implementation relies on fast geodesic computations. A user study shows that the resulting application is effective, it is responsive on meshes in the order of one or a few million triangles, and it is easy to use for novices, too. Our system is easily extensible in a variety of ways, while some specific patterns, such as tight packings of shapes and floral decorations, still remain out of its scope and will be addressed in future work.

## APPENDIX

## A THE ALGEBRA OF TANGLES

*Elements.* Let $\mathcal{M}$ be a 2-manifold, possibly with boundary. A *region* $r$ is a connected subset of $\mathcal{M}$ bounded by a finite number of oriented boundary loops $\{l_0, \ldots, l_k\}$; each loop is defined in turn with a sequence of curves. A *tangle* $\mathcal{T}$ is a tree of regions, having its root at $\mathcal{M}$; the children of each node $r$ in $\mathcal{T}$ form a partition of region $r$.

Regions can be arranged in *groups*, which are used to apply operators to multiple regions. A node in $\mathcal{T}$ is identified with a tag $\langle p_i, g_i, r_i \rangle$ where: $p_i$ is its parent region in $\mathcal{T}$; $g_i$ is the group it belongs to; and $r_i$ is its unique region identifier. The root region

corresponding to $\mathcal{M}$ is identified with $\langle \perp, 0, 0 \rangle$, while new group and region identifiers are generated incrementally as operators modify the tangle they are applied to. The *path* to a region $r_i$ in $\mathcal{T}$ is obtained by recursively substituting its parent $p_i$ with its tag.

In summary, a region of $\mathcal{T}$ is fully described by its geometry, namely, the set of its boundary loops, and its tag.

*Subdivision operators.* A generic operator *Op* has the form

$$(\mathcal{T}, r) \overset{Op(P)}{\mapsto} \mathcal{T}',$$

where $\mathcal{T}$ is a tangle, $r$ is a region of $\mathcal{T}$, and $P$ is a set of parameters that are specific of operator *Op*; the result $\mathcal{T}'$ is a modified tangle with additional leaves.

A subdivision operator induces a partition of region $r$. If $r$ is a leaf node, then the result is a collection of regions that become children of $r$ in $\mathcal{T}'$; if $r$ is an internal node, then this partition is intersected with the leaves of the subtree rooted at $r$ to obtain the new leaves in $\mathcal{T}'$.

*Grouping and joining operators.* We support explicit and implicit grouping of regions. Implicit grouping results from the application of a subdivision operator. The default tagging generates two groups: the foreground and the background, which are assigned new group tags; optionally, one may select to apply the same group to all regions, or a different group to each region, or groups assigned cyclically mod $k$. When an operator is applied to an internal node $r_i \in \mathcal{T}$, the group assigned to a new leaf $r_j \in \mathcal{T}'$ will depend both on the group assigned by the operator to the region partitioning $r_i$ and containing $r_j$ and on the group of the parent of $r_j$ in $\mathcal{T}'$: Leaves with the same combination will be assigned the same new group.

Explicit grouping is performed by selecting a collection of regions in $\mathcal{T}$ and assigning the same group $g$ to all of them. We support the following two operators:

$$\mathcal{T} \overset{Group(R)}{\mapsto} \mathcal{T}', \qquad \mathcal{T} \overset{Ungroup(g)}{\mapsto} \mathcal{T}',$$

where $R$ is a set of regions and $g$ is a group tag in $\mathcal{T}$. The *Group* operator assigns a new common group tag to all regions of $R$ in $\mathcal{T}'$. Conversely, the *Ungroup* operator considers all regions tagged with group $g$ in $\mathcal{T}$ and assigns a new different group to each of them in $\mathcal{T}'$. We also support explicit joining of regions:

$$\mathcal{T} \overset{Join(R)}{\mapsto} \mathcal{T}',$$

where $R$ is a set of regions of $\mathcal{T}$ that form a single connected component. The *Join* operator assigns the same region identifier to all regions of $R$ in $\mathcal{T}'$ so they will be treated as a single region. In practice, explicit grouping and joining are performed by selection through user interaction.

## ACKNOWLEDGMENTS

Photos of real objects: teapot by Natalya Sots; elephant by Dayal J. Daryanani. 3D Models: teapot by Michele Serpe, rolling teapot model by Brice Laville (concept by Tom Robinson - RenderMan "Rolling Teapot" Art Challenge), mask by Turbosquid user StriderS, fertility from the Stanford 3D Scanning Repository, dyno by Turbosquid user DANIEL38.

## REFERENCES

Y. Y. Adikusuma, Z. Fang, and Y. He. 2020. Fast construction of discrete geodesic graphs. *ACM Trans. Graph*, 39, 2 (2020), 1–14.

Adobe. 2020. Substance Designer. (2020). Retrieved from https://www.substance3d.com.

Claudio Mancinelli, Giacomo Nazzaro, Fabio Pellacini, and Enrico Puppo. 2021. b/Surf: Interactive B\'ezier Splines on Surfaces. arXiv preprint arXiv:2102.05921.

Autodesk. 2020. Mudbox. (2020). Retrieved from https://autodesk.com/mudbox.

D. P. Bertsekas. 1998. *Network Optimization: Continuous and Discrete Models*. Athena Scientific, Belmont, MA.

P. Bose, A. Maheshwari, C. Shu, and S. Wuhrer. 2011. A survey of geodesic paths on 3D surfaces. *Comput. Geom. Theory Appl.* 44, 9 (2011), 486–498.

A. M. Bronstein, M. M. Bronstein, and R. Kimmel. 2009. *Numerical Geometry of Non-Rigid Shapes*. Springer, New York.

M. Campen, M. Heistermann, and L. Kobbelt. 2013. Practical anisotropic geodesy. *Comput. Graph. Forum* 32, 13 (2013), 63–71.

M. Campen and L. Kobbelt. 2011. Walking on broken mesh: Defect-tolerant geodesic distances and parameterizations. *Comput. Graph. Forum* 30, 2 (2011), 623–632.

M. Campen, H. Shen, J. Zhou, and D. Zorin. 2020. Seamless parametrization with arbitrary cones for arbitrary genus. *ACM Trans. Graph.* 39, 1 (2020), 2:1–2:19.

E. Carra, C. Santoni, and F. Pellacini. 2019. Grammar-based procedural animations for motion graphics. *Comput. Graph.* 78 (2019), 97–107.

J. Chen and Yi Han. 1990. Shortest paths on a polyhedron. In *6th Annual Symposium on Computational Geometry*. Association for Computing Machinery, New York, NY, 360–369.

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.

K. Crane, M. Livesu, E. Puppo, and Y. Qin. 2020. A Survey of Algorithms for Geodesic Paths and Distances. arXiv:cs.GR/2007.10430.

K. Crane, C. Weischedel, and M. Wardetzky. 2013. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Trans. Graph.* 32, 5 (2013), 152:1–152:11.

J. D. Denning, W. B. Kerr, and F. Pellacini. 2011. MeshFlow: Interactive visualization of mesh construction sequences. *ACM Trans. Graph.* 30, 4 (2011), 66:1–66:8.

J. D. Denning, V. Tibaldo, and F. Pellacini. 2015. 3DFlow: Continuous summarization of mesh editing workflows. *ACM Trans. Graph.* 34, 4 (2015), 140:1–140:9.

D. Ebert, K. Musgrave, D. Peachey, K. Perlin, and S. Worley. 2002. *Texturing and Modeling: A Procedural Approach* (3rd ed.). Morgan Kaufmann, San Francisco, CA.

Y. Eldar, M. Lindenbaum, M. Porat, and Y. Y. Zeevi. 1997. The farthest point strategy for progressive image sampling. *Trans. Image Process.* 6, 9 (1997), 1305–1315.

M. Graichen, J. Izraelevitz, and M. L. Scott. 2016. An unbounded nonblocking double-ended queue. In *45th International Conference on Parallel Processing (ICPP)*. 217–226.

Philipp Herholz and Marc Alexa. 2018. Factor once: Reusing Cholesky factorizations on sub-meshes. *ACM Trans. Graph.* 37 (2018), 230:1–230:9.

P. Herholz, T. A. Davis, and M. Alexa. 2017a. Localized solutions of sparse linear systems for geometry processing. *ACM Trans. Graph.* 36, 6 (2017), 183:1–183:8.

P. Herholz, F. Haase, and M. Alexa. 2017b. Diffusion diagrams: Voronoi cells and centroids from diffusion. *Comput. Graph. Forum* 36, 2 (2017), 163–175.

A. Jacobson. 2021. gptoolbox. Retrieved from https://mathworks.com/matlabcentral/fileexchange/49692-gptoolbox.

C. Jiang, C. Tang, A. Vaxman, P. Wonka, and H. Pottmann. 2015. Polyhedral patterns. *ACM Trans. Graph.* 34, 6 (2015), 1–12.

R. Kimmel and J. A. Sethian. 1998. Computing geodesic paths on manifolds. *Proc. Natl. Acad. Sci.* 89, 15 (1998), 8431–8435.

M. Lanthier, A. Maheshwari, and J.-R. Sack. 1997. Approximating weighted shortest paths on polyhedral surfaces. In *ACM Symposium on Computational Geometry*. Association for Computing Machinery, New York, NY, 274–283.

M. Lanthier, A. Maheshwari, and J.-R. Sack. 2001. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica* 30, 4 (2001), 527–562.

Z. Levi. 2021. Direct seamless parametrization. *ACM Trans. Graph.* 40, 1 (2021), 6:1–6:14.

M. Li, D. M. Kaufman, V. G. Kim, J. Solomon, and A. Sheffer. 2018. OptCuts: Joint optimization of surface cuts and parameterization. *ACM Trans. Graph.* 37, 6 (2018), 247:1–247:13.

Y. Li, F. Bao, E. Zhang, Y. Kobayashi, and P. Wonka. 2011. Geometry synthesis on surfaces using field-guided shape grammars. *IEEE Trans. Vis. Comp. Graph.* 17, 2 (2011), 231–243.

L. Liu, L. Zhang, Y. Xu, C. Gotsman, and S. J. Gortler. 2008. A local/global approach to mesh parameterization. *Comput. Graph. Forum* 27, 6 (2008), 1495–1504.

H. Loi, T. Hurtut, R. Vergne, and J. Thollot. 2017. Programmable 2D arrangements for element texture design. *ACM Trans. Graph.* 36, 3 (2017), 27:1–27:17.

S. Longay, A. Runions, F. Boudon, and P. Prusinkiewicz. 2012. TreeSketch: Interactive procedural modeling of trees on a tablet. In *EG Symposium on Sketch-based Interfaces and Modeling*. The Eurographics Association, 107–120.

Joseph S. B. Mitchell, David M. Mount, and Christos H. Papadimitriou. 1987. The discrete geodesic problem. *SIAM J. Comput.* 16, 4 (Aug. 1987), 647–668.

A. Orzan, A. Bousseau, H. Winnemöller, P. Barla, J. Thollot, and D. Salesin. 2008. Diffusion curves: A vector representation for smooth-shaded images. *ACM Trans. Graph.* 27, 3 (2008), 92:1–92:8.

A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci. 1993. Dimension-independent modeling with simplicial complexes. *ACM Trans. Graph.* 12, 1 (1993), 56–102.

Pilgway. 2019. 3D-Coat. Retrieved from https://3dcoat.com.

Pixologic. 2021. ZBrush. Retrieved from https://pixologic.com/zbrush/features/overview/.

R. Poranne, M. Tarini, S. Huber, D. Panozzo, and O. Sorkine-Hornung. 2017. Autocuts: Simultaneous distortion and cut optimization for UV mapping. *ACM Trans. Graph.* 36, 6 (2017), 215:1–215:11.

F. Prada, M. Kazhdan, M. Chuang, and H. Hoppe. 2018. Gradient-domain processing within a texture atlas. *ACM Trans. Graph.* 37, 4 (2018), 154:1–154:14.

Y. Qin, X. Han, H. Yu, Y. Yu, and J. Zhang. 2016. Fast and exact discrete geodesic computation based on triangle-oriented wavefront propagation. *ACM Trans. Graph.* 35, 4 (2016), 125:1–125:13.

L. A. Romero Calla, L. J. Fuentes Perez, and A. A. Montenegro. 2019. A minimalistic approach for fast computation of geodesic distances on triangular meshes. *Comput. Graph.* 84 (2019), 77–92.

C. Santoni, C. Calabrese, F. Di Renzo, and F. Pellacini. 2016. SculptStat: Statistical analysis of digital sculpting workflows. arXiv:1601.07765

C. Santoni and F. Pellacini. 2016. gTangle: A grammar for the procedural generation of tangle patterns. *ACM Trans. Graph.* 35, 6 (2016), 182:1–182:11.

R. Sawhney and K. Crane. 2017. Boundary first flattening. *ACM Trans. Graph.* 37, 1 (2017), 1–14.

C. Schlick. 1994. Fast alternatives to Perlin's bias and gain functions. In *Graphics Gems IV*, P. S. Heckbert (Ed.). Academic Press, Amsterdam, the Netherlands, 401–403.

J. Schmid, M. S. Senn, M. Gross, and R. W. Sumner. 2011. Overcoat: An implicit canvas for 3D painting. *ACM Trans. Graph.* 30, 4 (2011), 28:1–28:10.

M. Schwarz and P. Wonka. 2015. Practical grammar-based procedural modeling of architecture. In *SIGGRAPH Asia 2015 Courses*. Association for Computing Machinery, New York, NY.

O. Sendik and D. Cohen-Or. 2017. Deep correlations for texture synthesis. *ACM Trans. Graph.* 36, 5 (2017), 161:1–161:15.

N. Sharp and K. Crane. 2020. You can find geodesic paths in triangle meshes by just flipping edges. *ACM Trans. Graph.* 39, 6 (2020), 249:1–249:15.

L. Shi, B. Li, M. Hašan, K. Sunkavalli, T. Boubekeur, R. Mech, and W. Matusik. 2020. MATch: Differentiable material graphs for procedural material capture. *ACM Trans. Graph.* 39, 6 (2020), 1–15.

J. Tao, J. Zhang, B. Deng, Z. Fang, Y. Peng, and Y. He. 2021. Parallel and scalable heat methods for geodesic distance computation. *IEEE Trans. Pattern Analysis Mach. Intell.* 43 (2021), 579–594.

A. Vaxman, M. Campen, O. Diamanti, D. Bommes, K. Hildebrandt, M. Ben-Chen, and D. Panozzo. 2017. Directional field synthesis, design, and processing. In *ACM SIGGRAPH 2017 Courses*. Association for Computing Machinery, New York, NY.

X. Wang, Z. Fang, J. Wu, S.-Q. Xin, and Y. He. 2017. Discrete geodesic graph (DGG) for computing geodesic distances on polyhedral surfaces. *Comput.-aided Geom. Des.* 52, C (2017), 262–284.

Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. 2009. State of the art in example-based texture synthesis. In *Eurographics State of the Art Report*. The Eurographics Association, Geneve, Switzerland.

K. Weiler. 1985. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Comput. Graph. Applic.* 5, 1 (1985), 21–40.

Pengfei Xu, Hongbo Fu, Youyi Zheng, Karan Singh, Hui Huang, and Chiew-Lan Tai. 2018. Model-guided 3D sketching. *IEEE Trans. Vis. Comput. Graph.* 25, 10 (2018), 2927–2939.

Xiang Ying, Caibao Huang, Xuzhou Fu, Ying He, Ruiguo Yu, Jianrong Wang, and Mei Yu. 2019. Parallelizing discrete geodesic algorithms with perfect efficiency. *Comput.-aided Des.* 115 (Oct. 2019), 161–171.

X. Ying, X. Wang, and Y. He. 2013. Saddle vertex graph (SVG): A novel solution to the discrete geodesic problem. *ACM Trans. Graph.* 32, 6 (2013), 170:1–170:12.

Y. Zhou, Z. Zhu, X. Bai, D. Lischinski, D. Cohen-Or, and H. Huang. 2018. Non-stationary texture synthesis by adversarial expansion. *ACM Trans. Graph.* 37, 4 (2018), 49:1–49:13.

Received May 2021; revised September 2021; accepted September 2021