A Programmable Model for Designing Stationary 2D Arrangements

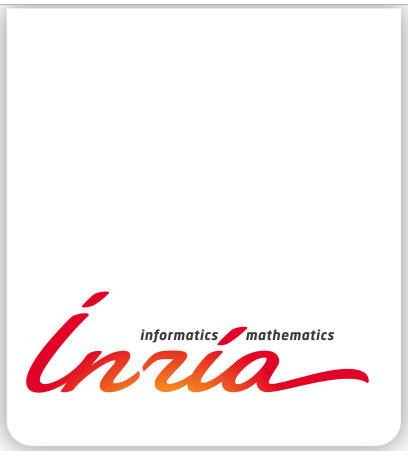
Hugo Loi, Thomas Hurtut, Romain Vergne, Joëlle Thollot

A Programmable Model for Designing Stationary 2D Arrangements

Hugo Loi, Thomas Hurtut, Romain Vergne, Joëlle Thollot

RESEARCH REPORT
N° 8713
February 2015
Project-Team Maverick

ISSN 0249-6399 ISRN INRIA/RR--8713--FR+ENG



A Programmable Model for Designing Stationary 2D Arrangements

Hugo Loi*, Thomas Hurtut†, Romain Vergne*, Joëlle Thollot *

Project-Team Maverick

Research Report n° 8713 — February 2015 — 25 pages

Abstract: This paper introduces a programmable method for designing stationary 2D arrangements for element textures, namely textures made of small geometric elements. These textures are ubiquitous in numerous applications of computer-aided illustration. Previous methods, whether they be example-based or layout-based, lack control and can produce a limited range of possible arrangements. Our approach targets technical artists who will design an arrangement by writing a script. These scripts are using three types of operators: *partitioning operators* for defining the broad-scale organization of the arrangement, *mapping operators* for controlling the local organization of elements, and *merging operators* for mixing different arrangements. These operators are designed so as to guarantee a stationary result meaning that the produced arrangements will always be repetitive. We show that this simple set of operators is sufficient to reach a much broader variety of arrangements than previous methods. Editing the script leads to predictable changes in the synthesized arrangement, which allows an easy iterative design of complex structures. Finally, our operator set is extensible and can be adapted to application-dependent needs.

Key-words: Programmable approach, element texture, spatial arrangement,texture design, texture synthesis

* Inria-LJK (UGA, CNRS)
† École Polytechnique de Montréal

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Un modèle programmable pour la conception d'arrangements 2D stationnaires

Résumé : Cet article introduit une méthode programmable de conception d'arrangements stationnaires d'éléments 2D pour les textures à base d'éléments. Ces textures sont omniprésentes dans de nombreux domaines de l'illustration assistée par ordinateur. Les méthodes précédentes, qu'elles soient pilotées par un exemple ou par un arrangement prédéfini, manquent de controlabilité et ne peuvent produire qu'un ensemble limité d'arrangements. Notre approche cible les artistes techniques qui concevront un arrangement en écrivant un script. Ces scripts utilisent trois types d'opérateurs : les opérateurs de *partition* qui définissent l'organisation globale de l'arrangement, les opérateurs d'*association* qui controlent son organisation locale, et les opérateurs de *fusion* qui mélangent des arrangements entre eux. Ces opérateurs garantissent des résultats stationnaires qui auront ainsi toujours le meme aspect quelque soit le contexte d'utilisation. Nous montrons que cet ensemble simple d'opérateurs est suffisant pour atteindre une variété d'arrangements bien plus large que les approches précédentes. L'édition des scripts amène des changements prévisibles dans l'arrangement synthétisé, ce qui permet de concevoir des structures complexes facilement, de manière itérative. Enfin, notre ensemble d'opérateurs est extensible et peut etre adapté à des besoins dépendants des applications.

Mots-clés : Approche programmable, texture à base d'éléments, arrangement spatial, conception de textures, synthèse de textures

1 Introduction

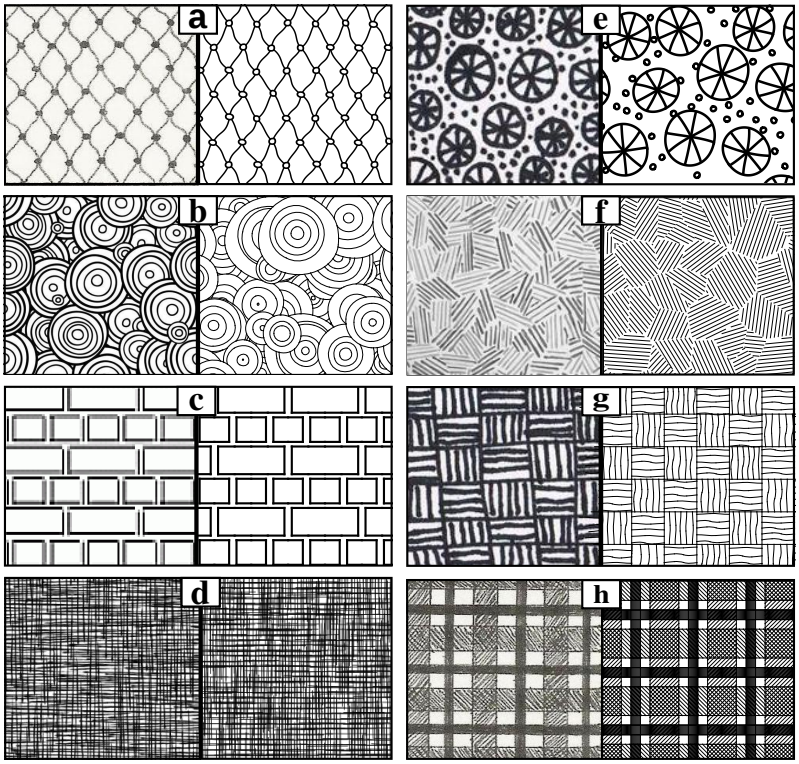


Figure 1: **Element textures commonly used.** These textures can be found in professional art (d,g,h), casual art (a,e,f), technical productions such as Computer-Assisted Design illustration tools (c), and textile industry (b). For each example, we show a hand-drawn image (left), and our synthesized reproduction of its geometric arrangement (right). **(a,b,c)** Classic regular distributions with contact, overlap and no adjacency between elements respectively. **(d)** Overlap of two textures creating cross hatching. **(e)** Non overlapping combination of two textures. **(f,g,h)** Complex element textures with clusters of elements. — Image credit: (d,g,h) [15]; (a,e) Profusion Art [`profusionart.blogspot.com`]; (f) Hayes' Art Classes [`hayesartclasses.blogspot.com`]; (c) CompugraphX [`www.compugraphx.com`]; (b) 123Stitch [`www.123stitch.com`].

This paper introduces a programmable method for designing repetitive arrangements of geometric elements. Such repetitive arrangements are good candidates for element texture design (Figure 1). Throughout this paper, we distinguish arrangements from textures in that textures can be stylized, using for instance colors and varying stroke widths. Element textures are a fundamental aspect of illustration. They add complexity to a drawing and support many artistic effects. They also depict important information such as materials in architectural plans, fabric in clothes, terrain type in topographic maps, biological materials in medical illustrations, etc. Producing element textures is therefore mandatory for many illustration systems and application fields, such as 2D animation, cartography, and other computer-assisted design tasks like pattern creation for textile or wallpaper industry. These applications often need to synthesize a large amount of each element texture either because the target image is very large (geographic map, wallpaper) or because the same texture is used in many images (comics, 2D animation). In

this context, manual authoring is tedious which motivates the need for a completely automated production pipeline. Several problems need to be solved to achieve this goal: the synthesis of various geometric elements, their arrangement on the plane, the choice of style attributes for each element and the rendering of the final element texture. Each of these topics is a research problem in itself. Existing works address extensively the tasks of creating elements [5, 20, 6] as well as stylizing and rendering geometrical data [17, 11, 14, 9]. In this article we address the problem of spatially **arranging** existing elements into a texture so as to fill a given region.

Targeted Properties. A computer-aided design tool for the production of texture arrangements should meet several requirements which we formalize in the following targeted properties:

- *Predictability.* Iterations between clients and technical artists involve numerous edits of the produced arrangements, which is feasible only through a controllable synthesis engine with predictable results.
- *Expressiveness.* The design tool must be expressive enough to allow the creation of classic layouts used by technical artists (see Figure 1 for an overview). When looking at manually drawn patterns, we observe that artists use regular and non-regular elements distributions and control elements adjacency such as contact or overlap. Complex arrangements are obtained by composing multiple distributions, the composition rule being generally a non-overlap superposition of these distributions. Some arrangements are also structured into clusters of elements and can be thought as being based on multi-scale arrangements.
- *Stationarity.* The main property of a texture is to be repetitive, enforcing its perception as a whole [34]. This characteristic can be formalized as the result of a stationary process. It means that the spatial statistics of the arrangement should not depend on the spatial location. Since this property is required for every texture, the design tool has to guarantee that all produced arrangements are stationary.
- *Extensibility.* Some specific projects might need arrangements that cannot be initially produced by the design tool, despite its native expressiveness. It must then provide a way to add new components for synthesizing these arrangements, while still guaranteeing stationary results.

Our Approach. We propose a programming-based method where each arrangement is represented by a user-written script. Programmable approaches have been proven useful for many designing tasks in computer graphics, including shading [7], modeling [29], stylized rendering of 3D scenes [14, 11] and motion effects [32]. As in these works, we target artists having programming abilities such as technical directors. Our contribution is therefore to provide the first programmable design tool dedicated to arrangements creation while simultaneously satisfying the four properties defined above.

Technical Contribution. To build our programmable method, we define a set of predictable operators that allow to produce a wide variety of arrangements while ensuring their stationarity. For that we take inspiration from programmable raster textures design[1]. In these methods, the design process (1) starts with an initialization such as Perlin's noise, (2) involves a number of filters such as color mapping, and (3) uses combining operations such as blending to mix multiple textures. Instead of a pixel grid, we store our arrangements in a high-level structure that stores adjacency and geometric information. Then, similarly to raster texture, we introduce three types of operations for the design process: (1) the structure is initialized with stationary partitions

[1] `www.allegorithmic.com`

such as a grid; (2) instead of filters, local geometric transformations are next applied to refine the partitions; (3) merging operators finally allow to combine multiple arrangements into complex ones. These three categories of operators are sufficient to achieve expressiveness, while creating a modeling scheme where stationarity is guaranteed at all stages. The synthesis is controlled step-by-step, which allows to edit the script with predictable effects. Finally, our method can easily be extended by adding new operators as long as they satisfy locality conditions in order to preserve stationarity.

2 Related Work

We focus our study on object-based texture representations, such as vector graphics representations, rather than their raster counterpart. Indeed, even if some efficient methods have been devised in the context of raster textures design [10], pixel-based textures loose geometric and connectivity information of the elements at stake, preventing further stylization or editing. In the context of object-based texture representation, existing computer-aided solutions for element placement fall into two main categories: example-based approaches which have seen a recent increased interest, and layout-based solutions usually proposed in commercial software. After reviewing these two classes of approaches that allow to produce stationary arrangements, we will review other procedural modeling approaches that are more expressive or predictable but loose stationarity.

2.1 Example-based Element Texture Synthesis

Most methods in the literature of element textures synthesis are dedicated to example-based approaches. They propose an artist-friendly interface where a small user-drawn exemplar is analyzed and synthesized over a larger domain. These approaches produce stationary arrangements and are easy to use for casual users. However they have a limited use in industrial contexts due to their lack of predictability and expressiveness.

Predictability. Describing the texture through a single examplar brings an ambiguity between desired invariants (such as "all elements must touch each other at their ends") and variable properties (such as "elements can have random orientations"). Furthermore, small modifications in the examplar may produce large unpredictable changes in the output. Besides, the exemplar needs to be stationary. So any modification has to be spread all over the exemplar meaning that the user has to rearrange the entire exemplar at each design iteration.

Expressiveness. None of the existing example-based methods succeeds to cover all classic layouts presented in Figure 1. We tested three recent methods [21, 19, 27] and we observed limitations controlling contact or overlap (Figure 2(a)), regularities such as alignments (Figures 2(b) and 2(c)) and multi-level arrangements (Figure 2(d)). These limitations come from two fundamental issues. First, *approximate representations* limit the types of elements and adjacencies that can be handled. For example, a centroidal element model [21] is not adapted to strongly anisotropic elements. Similarly bounding boxes [19] or sampling [27] reduce control on adjacency (Figure 2(a)). The proxy geometries introduced in [25] help to control more precisely elements adjacency. However, it does not solve overlapping cases due to an inaccurate similarity measure of overlapping relations. Second, the lack of *high-level information* makes it hard to detect geometrical constraints at variable scales such as alignments and clusters. It has been done for specific applications, like in [37] for arrangements of tiles, but we are looking for a more general approach.

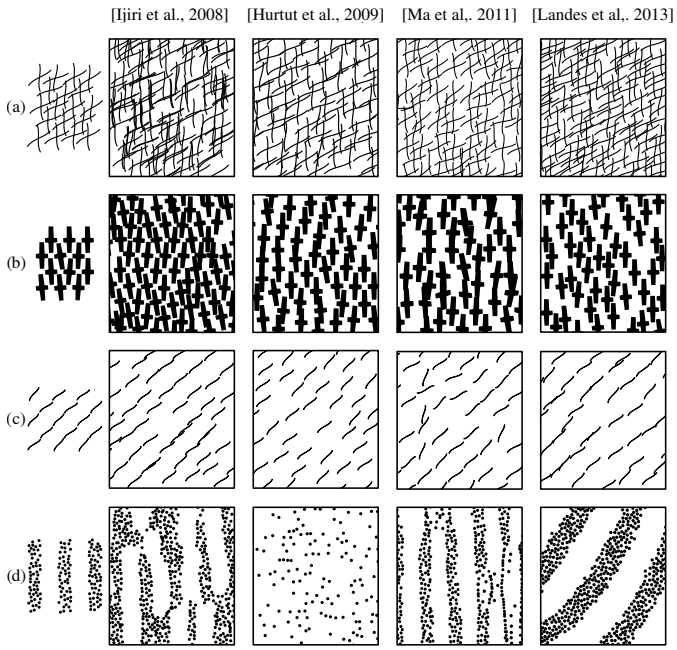


Figure 2: **Example-based methods' limitations.** Input examplars are shown on the left. Synthesized results from previous methods [21, 19, 27, 25] are shown on the four right columns. **(a)** A bimodal hatching, explicitly cited as one of the last limitations in [19]. While each interior hatch drawn in the exemplar crosses exactly three other hatches, no method preserves this property. In the case of the Expectation-Maximization algorithm of [27], possibly unwanted overlaps created during the patch-based initialization tend to be corrected during the optimization step thanks to the deformations handled. However, desired overlaps are still not ensured. **(b,c)** Regular structures with respect to three and one axis of alignment. The growing Delaunay-based approach of [21] achieves to reproduce these regularities in some ways. Yet the heuristics used to preserve the local graph structure also tend to create some gaps. Dense packing is challenging for Monte-Carlo statistical approaches such as [19, 25]. Indeed, although running 10^8 iterations, the (b) example outputs for these two methods still have some density variations. **(d)** A simple case of element clusters that no method succeeds to reproduce faithfully.

2.2 Predefined Layouts

Vector graphics software such as Adobe Illustrator or Inkscape propose predefined layouts to arrange user-drawn elements. The most common example of such layouts is the grid. With the same approach, recent online tools[2] propose methods for tiling small user-drawn arrangements. More complex stand-alone algorithms can synthesize uniform distributions efficiently [18, 24]. All these methods produce stationary results and are straightforward to use for obtaining a single particular layout. Their effect is predictable but their expressiveness is limited to a single kind of arrangement and they usually are not easily extendable. Typically Figure 2(d) would be hard to do with such approaches because it mixes regular and random distributions.

[2] `www.colourlovers.com/seamless`

2.3 Procedural Modeling

In this section, we present several inspiring procedural methods, coming from other fields than texture synthesis. We share some properties with these approaches, but none of them is well suited to element texture production because they have not been designed to ensure stationary outputs.

Historically, L-Systems [26] were used early in computer graphics to model plants [31]. Being originally dedicated to the generation of one-dimensional patterns, they cannot enforce a stationarity property in a two-dimensional domain. This is also the case for their extensions: parametric, timed and open L-Systems.

Shape grammars are another renowned procedural modeling approach [33, 36, 29]. Like more general context-dependent growth systems [35, 28], they use an axiom that is either a single element or the domain boundary. User-programmed growth rules must handle the propagation (or the subdivision) into the entire domain. Consequently, users would have to make a careful, non-intuitive use of each rule to obtain stationary arrangements.

Other arrangement transformations have been studied such as parquet deformations [23] and Escher construction operators [16]. These models are very specific to their respective application fields, which limits their expressiveness. However they are similar to our approach in the sense that they locally apply geometric transformations to an initial partition. Our approach targets general stationary arrangements.

3 Overview

Design principles. In a programmable approach, the task of the user is to build the algorithm that will produce his envisioned result. For that we provide the user with three types of operators, each of them responsible of a specific task: partitioning operators initialize an arrangement, mapping operators refine it and merging operators create combinations of arrangements. All of these operators have to guarantee the stationarity of the resulting arrangement. The texton theory [22] states that the appearance of an arrangement emerges from the broad-scale organization of micro-patterns called "local texture elements". Therefore stationarity occurs at broad-scale whereas local texture elements do not need to be constrained. Following this theory each type of operator will guarantee stationarity at its own scale:

- *Partitioning operators.* The design of an arrangement starts with a stationary partition. It ensures stationarity at broad-scale while letting the user choose between a regular or non-regular global arrangement structure.

- *Mappers.* The initial partition is locally refined using mappers. Mappers are user-programmed functors and control both local geometry and adjacency, by for instance placing an imported element and transforming it depending on its neighbors. A mapper is always applied everywhere on the arrangement via a *mapping operator*. Whereas no specific property has to be satisfied by elements, this is the locality and the homogeneous application of the mapper all over the arrangement that will ensure stationarity. Note that mappers can also call a partitioning operator in order to create a subscale arrangement. This can be useful to create texture elements made of a stationary arrangements (see for instance the subscale stripe arrangements in Figure 1(g)).

- *Merging operators.* Finally, complex arrangements are sometimes more easily designed when seen as the merge of simple arrangements such as the overlap of two textures. *Merg-*

```
def overview():
    size = 2000
    blob = ImportSVG("data/blob.svg")
    zig  = ImportSVG("data/zig.svg")

    # Mapper: Places a blob in a face.
    def map_blob_to(face):
        new_blob = Rotate(blob,Random(face,0,2*pi,0))
        return MatchPoint(new_blob,BBoxCenter(new_blob),
             Centroid(face))

    # Mapper: Replaces an edge by a curved line.
    def map_curve_to(edge):
        if IsBoundary(edge):
            return ToCurve(edge)
        src_c = PointLabeled(zig,"start")
        dst_c = PointLabeled(zig,"end")
        src_v = Location(SourceVertex(edge))
        dst_v = Location(TargetVertex(edge))
        return MatchPoints(zig,src_c,dst_c,src_v,dst_v)

    # Mapper: Generates an arrangement in a face.
    def map_arrangement_to(face):

        # Grid partition with randomized orientations
        theta     = Random(face,0,2*pi,1)
        width     = BBoxWidth(face)/5
        lines1    = StripesProperties(theta,width)
        lines2    = StripesProperties(theta+pi/2,width)
        init      = GridPartition(lines1,lines2,
             CROP_ADD_BOUNDARY)

        # Mapping operator: maps a curve on each edge
        arrangement = MapToEdges(map_curve_to,init)
        return arrangement(face)

    # Init: Uniform partition
    props    = IrregularProperties(10/(size*size))
    init_tex = UniformPartition(props,KEEP_OUTSIDE)

    # Mapping operator: maps a blob in each face
    blob_tex  = MapToFaces(map_blob_to,init_tex)

    # Mapping operator: maps an arrangement in each face
    final_tex = MapToFaces(map_arrangement_to,blob_tex)

    # Export final arrangement
    ExportSVG(final_tex,size)
```

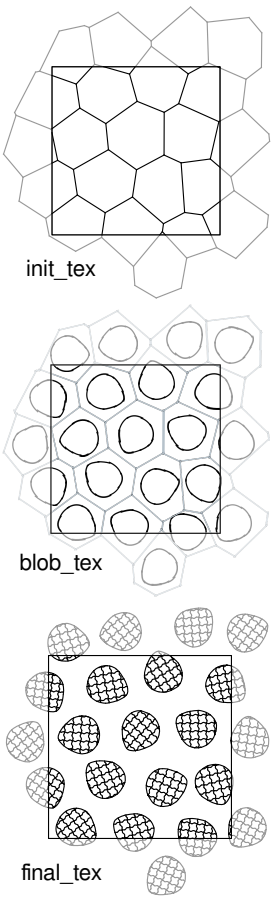


Figure 3: **An example of a script and its output. Left:** A script based on two imported SVG elements (a blob-like shape and a small stroke) and three user-defined local mappers to control local features. **Right:** The output is a two-scale arrangement. We show here two intermediary results and the final output.

ing operators provide such mechanisms by performing overlap, inclusion and exclusion operations. They do not change the stationarity of their input arrangements.

Functional programming. Our approach is entirely functional. We define an arrangement as a function that takes as input a region and returns a collection of curves. All our operators, regardless of their category (partitioning, mapping, merging), output an arrangement. On the input side, partitioning operators take in a region whereas merging operators take in two arrangements to be combined. Mapping operators take in both a mapper and an arrangement to be mapped. We use Python as the programming user interface, its syntax being simple and intuitive to most programmers and well designed for functional programming.

Data representation. The collection of curves returned by an arrangement function is embedded into a planar map: A structure containing vertices (intersection points), edges (pieces of curves linking vertices) and faces (2D domains enclosed by edges) [4]. Spatial adjacency between these three types of component can be precisely handled by the user with mappers, allowing contact or overlap control.

Overview example. Figure 3 gives an example of the synthesis of a two-scale arrangement. Three mappers are first designed in this script. The first two ones map an SVG element on a face (L.9) and an edge (L.19). The last one creates a regular partition (L.29) and calls the second mapper (L.32) to map a curve on each of its edges. Once these mappers are defined, a uniform partition is created (L.37). A blob shape is mapped on each of its faces using the first mapper (L.40). The third mapper then maps a regular arrangement on the resulting faces which are now blobs (L.43). Induced edges and faces are exported respectively as open and closed SVG polylines (L.46).

4 Planar Maps

Planar maps are a topological modeling tool introduced in [4] for representing drawings. All the curves manipulated by a user (imported SVG elements, operators outputs) are automatically converted into planar maps. Each of them internally represents an arrangement and provides easy accesses to precise topological relations between cells such as intersections, contacts, and inclusions.

Definition. The planar map induced from a collection of curves $\mathcal{C}$ is defined as a set of cells partitioning the plane (Figure 4). Cells are of three types: edges, vertices and faces. Edges are the set of maximal pairwise disjoint subcurves of $\mathcal{C}$. Vertices are the set of limit end points of edges. Faces are the set of maximal parts of $\mathbb{R}^2 - \mathcal{C}$. An incidence graph completes the representation allowing access to all types of adjacencies in the planar map.

Cell labels. On top of the planar map, we add a set of labels to each cell, accessible to mapping operators. They will typically be used to select a subset of cells when needed. We let the user define the (finite) set of possible labels by either giving labels to partitions' cells or adding labels to imported SVG curves.

Face labels reconstruction. When modifying or combining planar maps, labelling has to be conserved. We adopt the same solution as [3]. Since planar maps are induced by curves, labels should be stored only on curves. Faces labels are thus stored on their adjacent edges and reconstructed each time a new planar map is induced.

Implementation. Practically, in our implementation, planar maps are based on the CGAL arrangement structure [12] and use exact arithmetic and geometry with rational coordinates to avoid any topological artifacts due to numerical imprecisions.

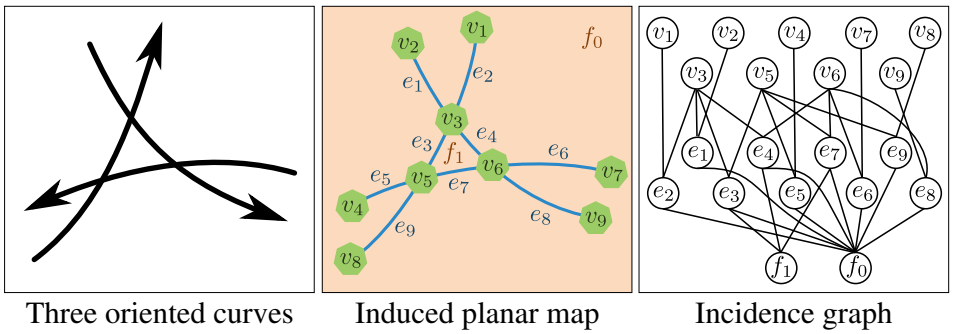


Figure 4: **Planar map representation. Left:** Three oriented curves. **Middle:** Induced planar map composed of nine vertices, nine edges and two faces. The face f_0 is unbounded. Edges are oriented accordingly to their originating curves. For example, e_1's source vertex is v_2 and its target vertex is v_3. **Right:** The incidence graph of the planar map denotes the relationships between vertices, edges and faces. We did not include arcs between vertices and faces for clarity, they can be deduced by transitivity.

5 Partitioning Operators

The first step in the design of an arrangement is to choose a partition to determine its global structure. Such partitions must be stationary and should hold a regular or non-regular global structure. These partitions will be extensively refined by defining local transformations using mappers, as presented in Section 6. If required, more operators could be easily added to adapt to specific needs. Our goal in this section is therefore to provide operators that ensure a stationary partition, simple enough to begin the design, but subsequently flexible enough to allow all possible refinements.

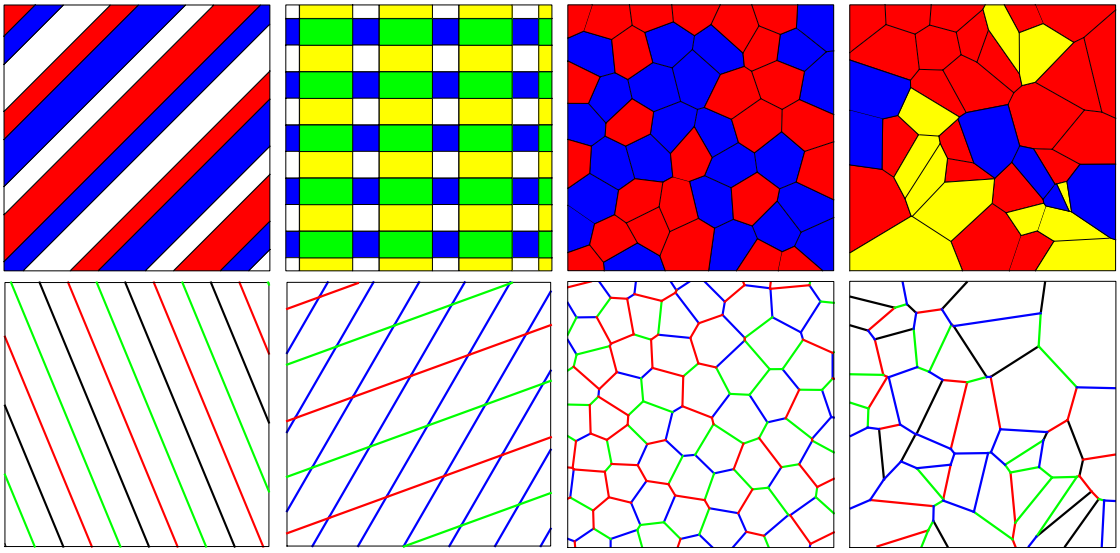
5.1 Regular Vs Non-regular Partitions

We propose four partitioning operators that allow to design regular and non-regular partitions of the input region. These operators, in addition to a a few others that let specify partition labels and properties, are recalled in the Table 1 of the appendix.

Regular partitions. The "StripesPartition" operator partitions the domain with a distribution of parallel lines. This operator is defined with the stripe angle and the width between two successive lines. Optionally, the user may define a cycle of widths that will be repeated periodically until all lines are placed. For instance in Figure 5(a) the top image shows a cycle with two alternating width values (20 units and 10 units), while the bottom image uses only one width value (15 units). These parameters are set by the "StripesProperties" function that takes a variable number of arguments. Labels might also be associated to faces and/or edges using the same cycle process. In that case, all partition's faces/edges are labelled by successively picking the corresponding value in the label list (Figure 5(a)). "GridPartition" partitions the domain with two distributions of parallel lines and is thus obtained by specifying two stripes partitions.

Note that when faces are labelled for both stripes, each single face receives a total of two labels. For instance in the top image of Figure 5(b), the green color denotes the presence of both labels “yellow” and “blue”.

Non-regular partitions. “UniformPartition” and “RandomPartition” operators are computed using Voronoi diagrams, respectively based on Poisson-Disk and Poisson distributions. In both cases, the user needs to specify a density value that defines the number of samples per unit area via the "IrregularProperties" function. Labels might also be attached to faces and edges of these partitions. In that case, the user defines a list of labels and apparition probabilities used to randomly assign each face and/or edge (Figure 5(c,d)).



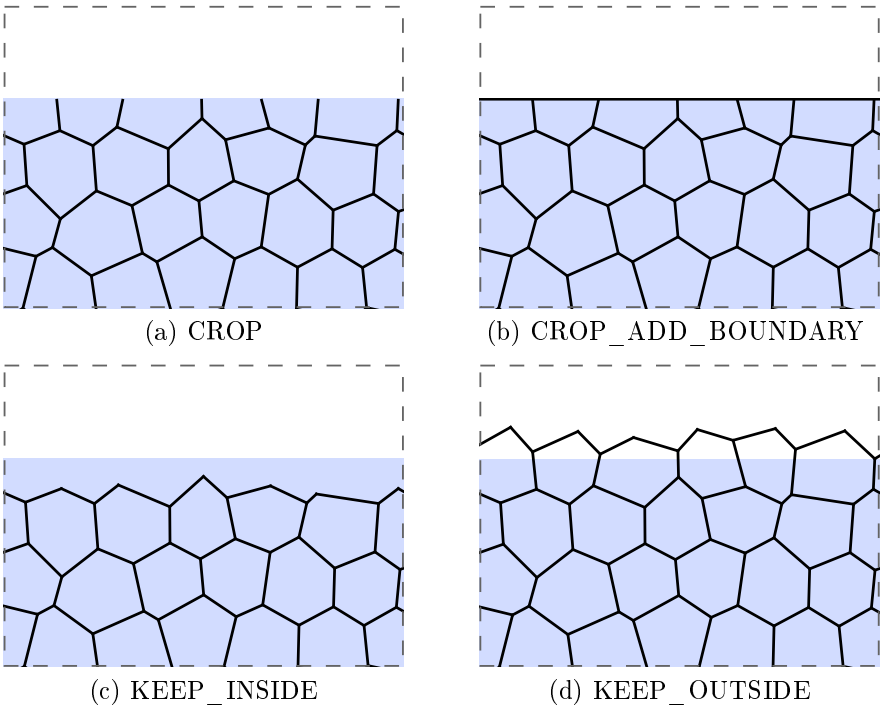
(a) Stripes (b) Grid (c) Uniform (d) Random

Figure 5: **Available types of partition.** When designing an arrangement, the first step is to choose a type of partition among four possible ones, whether it is a regular (a,b) or a non-regular partition (c,d). Colors denote assigned labels to faces (top) or edges (bottom). We vary the width between lines of regular partitions using periodic cycles of values. The same process is used to assign labels. The density of irregular partitions is controlled by a single parameter. Labels may also randomly be assigned according to user-defined probabilities. Faces and edges may contain multiple (cycling) labels to precisely control the final arrangement.

5.2 Border management

When partitioning a face, the user may want various behaviors at its boundary. We provide four border management options that cover all the cases we encountered (Figure 6). The CROP option cuts the partition at the boundary of the face. The CROP_ADD_BOUNDARY option does the same except that it adds the outline curve of the face. For these two options, the resulting planar map usually ends up with faces with a different shape on the border than in the middle of the original face. If one prefer to keep constant face shapes, like to keep constant grid cells, he can choose between two other options: KEEP_INSIDE or KEEP_OUTSIDE. The first option keeps only the cells that are strictly included in the original face whereas the latter keeps

all the cells that intersect the original face. The resulting cells can thus overlap the face border. Note that stripes partitions are always cropped as their faces are infinite.



(a) CROP (b) CROP_ADD_BOUNDARY

(c) KEEP_INSIDE (d) KEEP_OUTSIDE

Figure 6: **Border behaviors.** The user can choose between four partition behaviors at the domain's border. In these illustrations, a uniform partition is created on a light blue domain. One can first simply crop the partition along the border (a), with possibly adding the domain's outline curve (b). The cells of the partition that overlap the border can also be discarded (c) or kept (d).

6 Mappers

Mappers are a central feature of our approach. As previously mentioned, a texture is a *large-scale* stationary arrangement of *small-scale* elements. Contrary to partitioning operators that create the broad-scale structure of the arrangement, mappers are targeting small-scale elements. In practice, a mapper is a function that takes as input a single cell of a planar map. It applies (almost) arbitrary code written by the user so as to create, combine, transform and place curves on a particular location according to the cell's information (position, incident vertices, edges, faces, etc.). The mapper finally outputs a collection of curves.

In order to preserve stationarity, mappers must be executed homogeneously on all the cells of a planar map. Since the initial planar map comes from a stationary partition, this property is preserved, formalizing the large-scale repetitive aspect of textures. This homogeneous execution is handled by mapping operators. We provide one mapping operator per cell's type: "MapToVertices", "MapToEdges" and "MapToFaces" (Table 2 in appendix). The mapping operator takes as arguments the arrangement to be mapped and a user-programmed mapper. Its output is a new set of (stationary) curves. It is worth noting that the resulting arrangement can in turn be

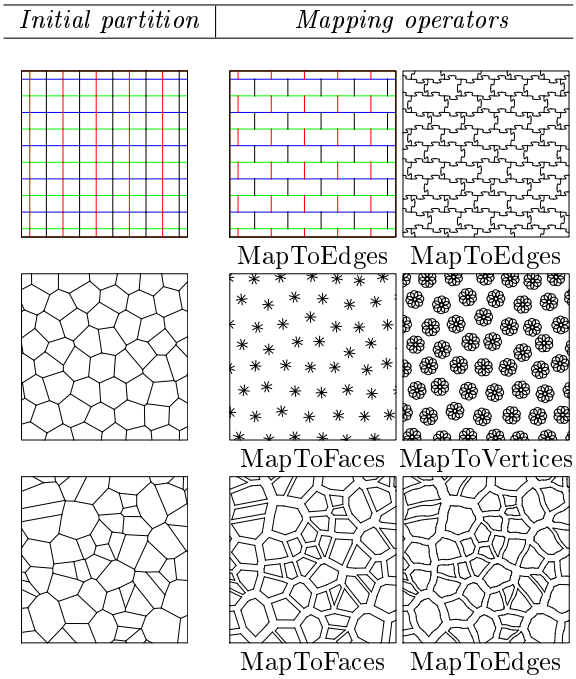


Figure 7: **Using mappers.** Three examples of mappers effect on initialized partitions. **Top:** A grid is first initialized using the "GridPartition" operator (left). Each edge is labelled using user-defined value cycles (shown with colors in this example). Based on the "HasLabel" operator, a mapper that keeps edges in staggered rows is applied on each edge of the grid (middle). The final puzzle pattern is obtained by a mapper using the "MatchPoints" operator that places a simple curved line on each edge (Right). **Middle:** The planar map is initialized with a uniform partition (left). Four lines are matched to the centroid of each face of the partition to build a new set of construction lines (middle). Overlapping circles are mapped on the resulting vertices to create rosette flowers (right). **Bottom:** Starting from a random initial partition (left), the induced faces are slightly scaled down (middle). Some curves, picked from a limited example set are finally mapped on each induced edge using the "MatchPoints" operator (right).

used as input to another mapping operator in order to generate more complex patterns.

6.1 Mapper definition

Formally, a mapper is a user-programmed functor m that maps a single cell c of a planar map $\mathcal{P}$ to a new collection of curves $\mathcal{C}$. The key idea of our model is that the programmed functor m will automatically be executed on each cell $c \in \mathcal{P}$ by a mapping operator. To ensure that the mapping of m on $\mathcal{P}$ preserves stationarity, the following conditions must be respected:

- m is local and depends only on cells of $\mathcal{P}$ inside a given bounded neighborhood. Only a bounded number of incidence queries should then be called inside a given functor.
- m does not depend on a particular execution order. It means that global variables are read-only and should not be overwritten.
- m does not depend on global coordinates to avoid specific mappings to be applied at

particular locations in the plane. Consequently, only relative cell's coordinates are available from the user point of view.

These conditions ensure that a functor will have the same behavior everywhere in the input planar map.

6.2 Programming Mappers

We provide a set of low-level built-in operators specifically designed to program mappers, given in Table 3 of the appendix. All the examples shown in this paper have been created with this simple operator set:

- *Incidence* operators are dedicated to access all the information stored is the incidence graph of the planar map.
- *Adjacency* operators are used to place elements while controlling their adjacency either to one or two vertices, or in a face.
- *Geometry* operators retrieve information of the input cell such as its location, contour, centroid, etc.
- *Labels* operators are dedicated to the management of labels.
- *Random values* operators allow to easily vary the properties of the mapping inside each cell.

We also provide a set of useful utility functions that yield simple geometric affine transformations, bounding box information as well as the loading of an SVG element. These functions are accessible from everywhere in user-scripts (see Table 5 of the appendix).

6.3 Using mappers

A typical use of mapping operators is to modify an original partition, for instance by removing of modifying some cells, then placing some imported elements possibly controlling their adjacency. One can stop here or continue to map elements until reaching the desired arrangement. We show three examples in Figure 7 to illustrate the variety of effects a mapper allows to create on the partitions. The top example leverages labels and the precise matching of curve endpoints to create a puzzle-like brick wall arrangement. The middle example uses single point matching and location operators to create a uniform distribution of rosette shapes. In the third example, the faces of a uniform partition are first rescaled before replacing each of their edge by a new smooth curve. More complex arrangements can be created by calling partitions operators into mappers as shown in the overview (Figure 3).

7 Merging Operators

Merging operators take two arrangements as inputs, and return one arrangement (Figure 8). They provide a simple way to mix simple arrangements to obtain complex patterns. We propose three different merging operators (Table 4):

- *Union* computes a new arrangement that results from the collection of all the edges produced by the two inputs arrangements. It is used to group multiple distributions (Figure 8(d)).

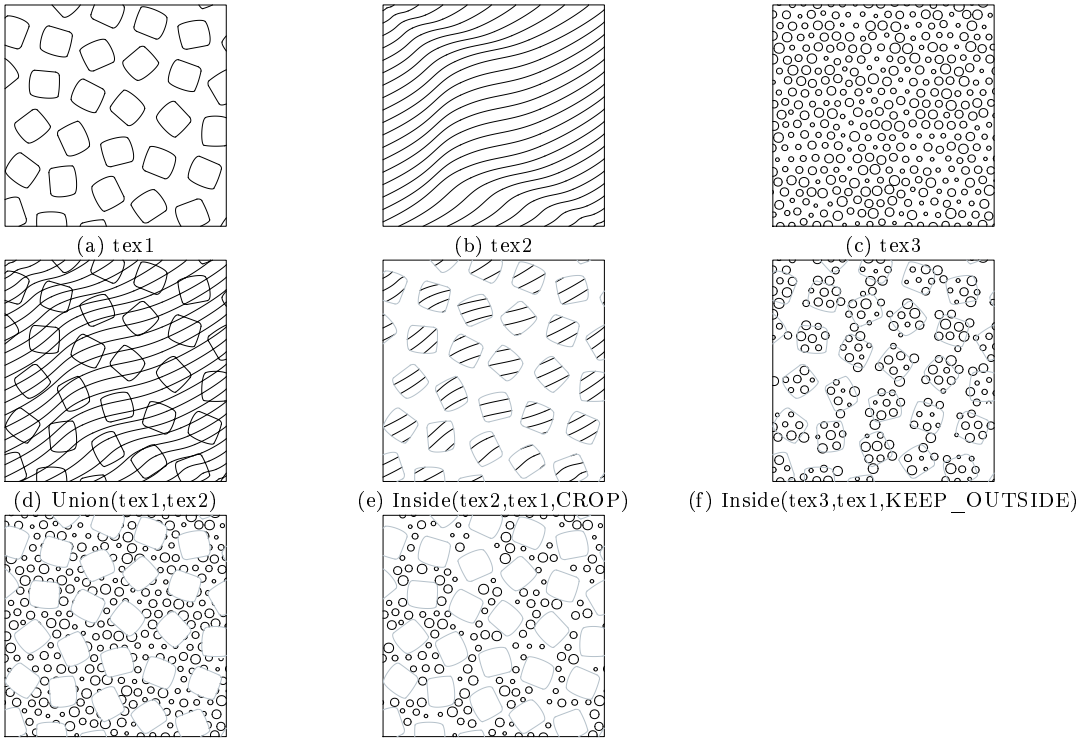
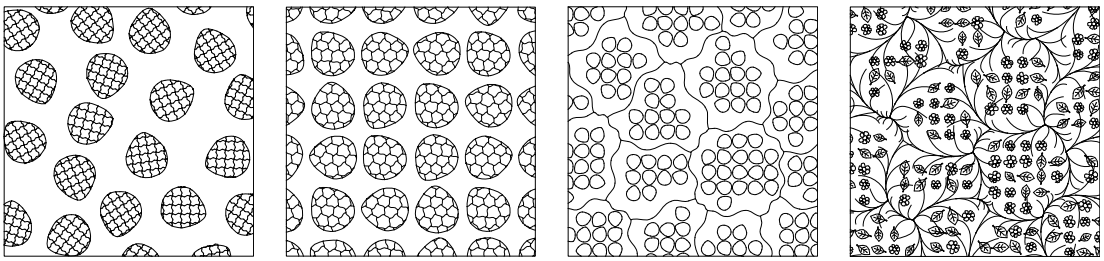


Figure 8: **Merging multiple distributions. (a,b,c)** Three simple arrangements obtained via partitioning and mapping operators. **(d)** The *Union* operator overlaps its two input arrangements. **(e,f)** The *Inside* operator behaves like a mask, keeping only the edges from a first arrangement that fall inside the faces of a second one. The same border management options are proposed as for partitions (see Figure 6). **(g,h)** The *Outside* operator also behaves like a mask, keeping only the edges from a first arrangement that fall outside the faces of a second one. Same border management options are available.

- *Inside and Outside* are masking operators. They keep only the edges produced by a first arrangement that are falling inside and outside the bounded faces, respectively, of a second arrangement. A border management option is mandatory for these operators. It allows to precisely define if cells have to be kept-in, kept-out or cropped along the first arrangement boundaries (Figure 8(e-h)).

8 Results

Along the paper we have shown that our method guarantees stationary outputs by construction. We also have highlighted how it is extensible at all stages. Here we present practical modeling sessions that demonstrate its *predictability* and *expressiveness*. Designing an arrangement is an iterative process. The user progressively finds the set of successive rules that leads to the result he has in mind. As shown along the paper, the basic strategy is to design simple arrangements to be combined. A general structure is chosen for each one, and further refined. All the scripts and execution times used to produce the images of this paper are included in the supplemental



(a) Original script (b) Switch partitions (c) Switch arrangements (d) Vary mappings

Figure 9: **Script edition.** A design strategy can be to edit iteratively a starting arrangement. **(a)** Original two-scale arrangement from Figure 3. **(b)** The grid partition previously applied to the lower scale is exchanged with the uniform partition from the blob distribution. **(c)** Another inversion: blobs are now regularly distributed inside the uniformly distributed cells. **(d)** A twig is mapped to the smooth stroke of (c), and flowers or leafs with varying scales are now mapped to the blob shape.

materials. Most results were generated in a few seconds (except Figures 1(b,d) and 12(d) that needed more than one minute).

Script Editing. In terms of interaction, our modeling approach is very similar to node-based material shaders commonly used in the 3D pipeline: (1) partitioning operators correspond to initialization nodes, (2) mapping operators correspond to filtering nodes, and (3) merging operators correspond to the $2 \rightarrow 1$ combination nodes. This interaction scheme has been used during the last 30 years since the seminal work on Shade Trees [7]. It is commonly acknowledged to be efficient. In particular, it favors iterative design processes as well as the exploration of various combinations at the artist's whim.

Figure 9 shows the kind of variations that are produced during such an exploratory usage of our tool. Each image shows the result obtained by a slight modification of the script presented in the overview (Figure 3). These variations are predictable because the script is composed of small understandable chunks of code (partitions and mappers) linked together by simple merging operators. A regular user of our tool should be able to foresee how these edits in the script will influence the execution of the other chunks left unchanged.

In Figures 11 and 12 we show iterative design sessions where the user envisions a particular arrangement and edits the result towards this objective. Our method allows to proceed step by step and to display the arrangements produced at each step. This helps making sure that the edits converge towards the envisioned result. These two figures display the temporary steps of the design sessions as well as the results finally obtained. They showcase how this script-editing scheme is helpful for quickly designing complex arrangements.

Expressiveness. The examples shown in Figures 10, 11 and 12 illustrate how the properties usually required by artists can be obtained (see Section 1): regular and non-regular arrangements, various elements adjacency relations such as contact or overlap, compositions of several arrangements and clusters of elements. All the scripts producing these examples have less than 55 human-readable lines and they use only the operators given in appendix. Our method is therefore able to achieve the target level of expressiveness. It is the first approach that completes this objective since it overcomes the limitations of the existing techniques as shown in Figure 10. Let's note that our feasible set of arrangements does not strictly include those of the previous methods. In particular, our proposed operators may not be able to reproduce all the subtle

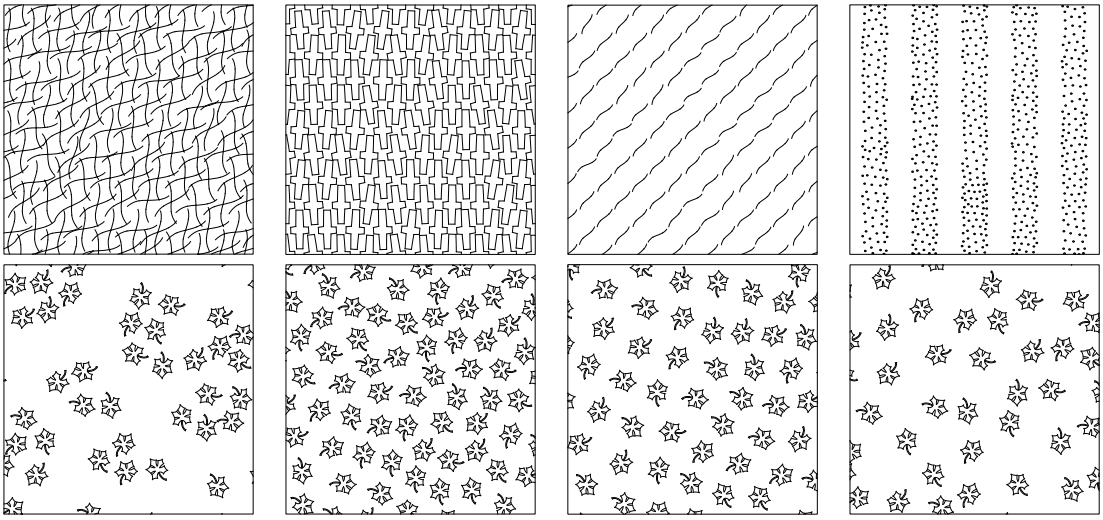


Figure 10: **Comparison with examplar-based approaches. Top:** we go beyond by-example methods' limitations from Figure 2 by faithfully reproducing the target arrangements with our set of operators. **Bottom:** The evaluation protocol developed in [2] showed that even expert designers do not usually agree on what should be the output arrangement based on one examplar. We show here that we can reproduce the four different expert manual arrangements gathered in the second figure from AlMeraj's study, which all subtly vary from the given input examplar.

variations synthesized by example-based methods. However our solution is able to span a variety of arrangements that was not feasible before, which was the goal of this article.

These results demonstrate that expressiveness is achievable with a restrained set of operators. It also validates our insight of separating the design tasks between a very small set of partitioning operators and an unlimited set of possible refinements. It is particularly visible in Figure 11 where a variety of arrangements are designed based on simplistic regular partitions. These results validate as well our choice of manipulating all kinds of primitives (vertices, edges and faces), whereas alternative strategies based only on dot anchors or edge distributions would limit the possible spectrum of edits.

Figure 13 shows a complete vector drawing, textured with seven scripts using our approach. These seven textures cover a variety of effects. Rough natural materials are depicted using various hatching (wood) and stippling (teapot) techniques. The man-made parquetry arrangement is based on specific adjacency relations. Finally, the tapestry uses the two-scale texture from Figure 9(d).

9 Discussion

9.1 Limitations

Continuous variations. Figure 10(d) could be seen as a distribution of dots following a periodic step density function, alternating blank regions with null density and crowded regions with high density. One could imagine a variation with a sinusoidal density function instead. This variation would be unfeasible in our system. The only possible way to do something close to it would be to

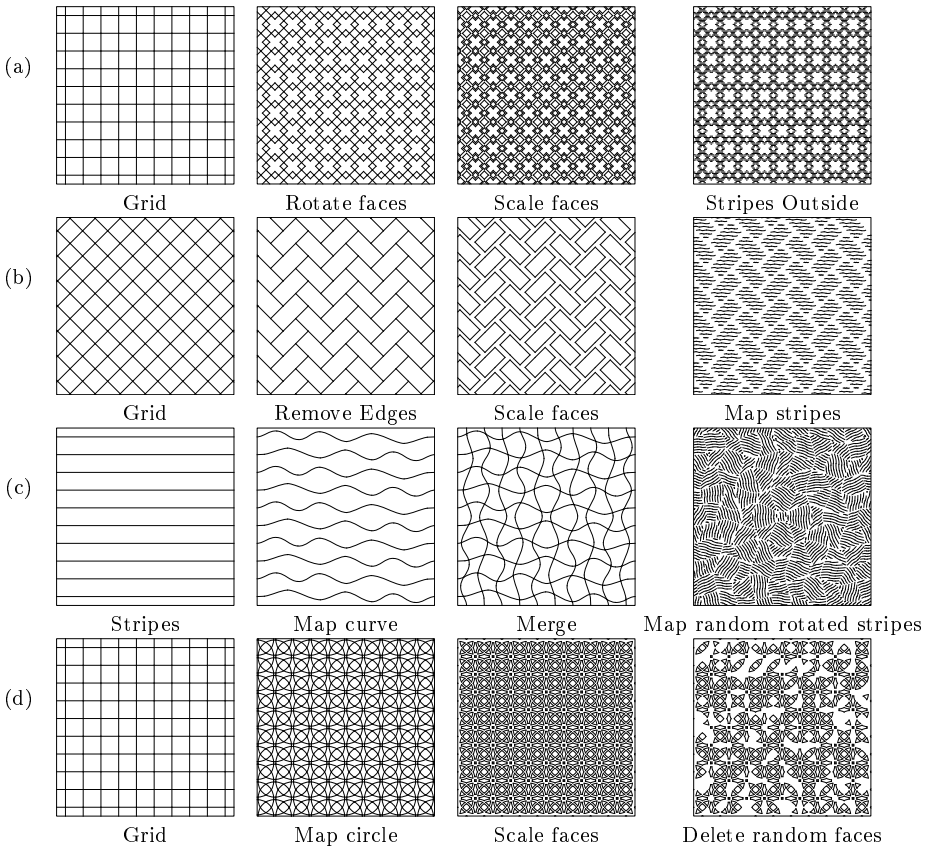


Figure 11: **Creating complex structures starting from regular partitions.** Each row shows some iterative design steps, starting from an initial regular partition. **(a,b,c)** Two-scale examples where the initial partition is refined in different ways, and the resulting regions filled with various stripe patterns to produce hatching effects. **(d)** A mosaic-like partition is made using a grid and mapped circles. A kind of aging effect is finally obtained by deleting some faces randomly.

generate a very fine StripesPartition, and then to fill the faces obtained with constant densities that would make a piecewise-constant approximation of the sine function. This limitation is due to our choice of generating the arrangement's properties from discrete predicates only, for example the value of a label. As discussed in future works, continuous control maps would be helpful in such cases.

Implicit control. In our approach the user explicitly controls all spatial relations in the arrangement. Unlike most by-example approaches where targeted properties are given as input of the synthesis algorithm, our input is the construction script. As a consequence our approach allows to precisely control element adjacencies but does not help producing arrangements that exhibit implicit behaviors such as the ones resulting of physical simulation or other global optimization processes. A typical example is a zebra texture that is hard to design with our approach.

Operators. We designed our operator set in order to allow a wide variety of arrangements.

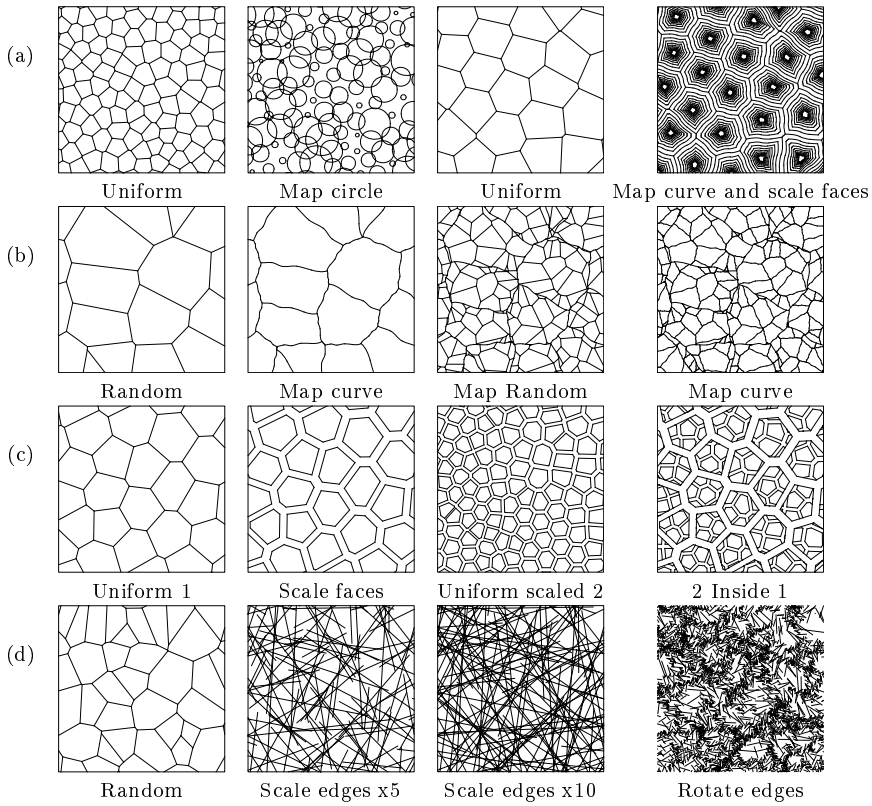


Figure 12: **Creating complex structures starting from non regular partitions. (a)** First row shows two examples starting from a uniform partition, yet with radically different final arrangements. **(b)** A random partition, after having mapped its edges with a curve, is recursively applied to its own regions, achieving a two-scale cracks effect. **(c)** Another two-scale arrangement, based on an inside merging operator, leading to a turtle shell effect. **(d)** The arrangements can quickly depart from the initial partition, even with simple refinements: the edges of a random partition are directly scaled then rotated to produce various random lines distributions.

More operators could be added for specific needs. As an example, we currently control adjacency based on one or two contact points. It may be interesting to increase the number of constraint points to create more constrained arrangements. This requires non-rigid transformations and interpolation, which is left for future work.

Planar maps. Since our internal representation is a planar map, we inherit all the limitations of this model. In particular, there is no simple way to determine which faces of the planar map are intended to be the interior of the elements. For instance, the drawing of a ring is constituted of two concentric circles. This induces two faces considered at the same level by our operators, whereas the user might want to process them separately. Labels can be used to resolve some ambiguities but not all of them. Other representations could be investigated to solve this problem such as Vector Graphics Complexes [8].

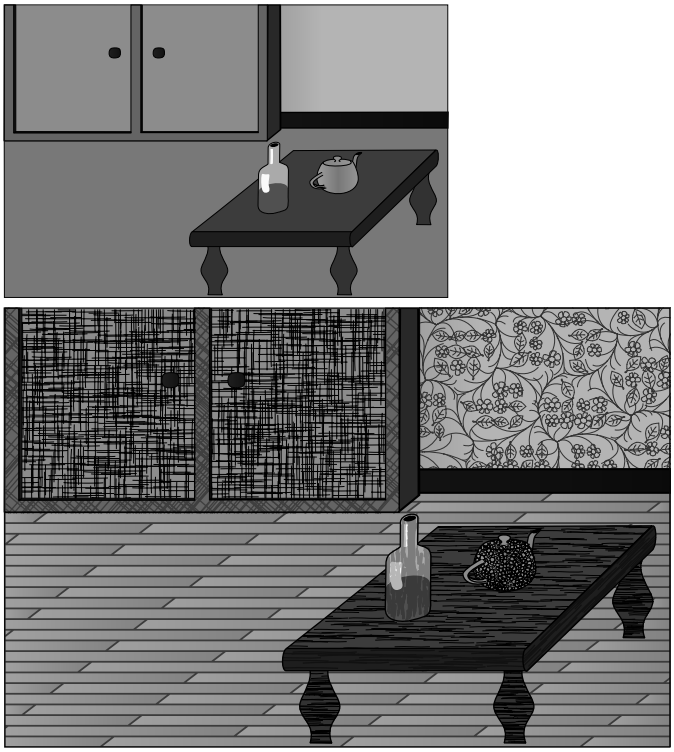


Figure 13: **Texture-based illustration. (top)** A simple SVG drawing. **(bottom)** The drawing is textured using seven scripts. The created paths can be grouped and imported in any vector drawing software. They can then be processed as any other SVG element. For example here, a single action was needed for filling all the parquetry slats with a horizontal linear gradient.

9.2 User interface

We have shown that our programmable approach yields predictable and controllable results. However the interaction scheme offered by a programming language is not suitable for non-programmers. A way to broaden the audience of our method is to offer more intuitive user interfaces. This should be possible thanks to the combination scheme of our operators which is natively nodal. Formally, operations are organized as a Directed Acyclic Graph where nodes are operations and pointers are planar maps (we call them "nodes" and "pointers" to avoid confusion with planar map cell types). It means that a straightforward node-based graphical interface such as in [1] would be sufficient to wrap our operator combination scheme. However the (almost) arbitrary code in our mappers is much more difficult to represent graphically. A simple solution could be to abstract these mappers as operation nodes. Users with programming skills would then create such nodes using a regular text editor and share these nodes to the non-programming community.

An interesting issue to pursue could be to propose inverse procedural modeling such as in [13]. A full inverse *programmable approach* is probably too difficult since it would boil down to go back to the limitations of by-example approaches. Yet one could target just a few operators' parameters such as density and cycles, or more global characteristics such as the type of partition. These could be learned from simple examples or user given sketches.

9.3 Towards a complete programmable illustration pipeline

Our programmable approach addresses the problem of *placing* elements in a texture. This problem is part of a complete texture synthesis pipeline. We discuss here how the remaining parts could be combined with our approach.

Elements synthesis. Our current system in able to import existing elements. A straightforward extension could be to add an import operator that pick random elements produced by existing algorithms of element synthesis such as [5]. However if one may want to stay in a programmable pipeline, operators may be devised to increase the control on each element shape. Procedural modelling already offers numerous methods for context-dependent element synthesis that we could use to extend our model [28].

Control maps. It is sometimes suitable to add control maps to guide the global behavior of the arrangement by locally varying its density or orientation. We plan to extend our approach in that direction by devising new operators that would give mappers the ability to query external data. These data could locally deform partitions or impact elements spatial properties. Starting from our stationary distributions and depending on the control map, the resulting element textures would exhibit a repetitive aspect even if not strictly stationary.

Stylization. The stylization step can be done manually by loading SVG exported by our system in commercial vector graphics software. However, it would make sense to stay in a programmable approach for this step because style attributes could be linked with placement data via specific operators. A similar approach has been applied to the stylisation of line-drawn 3D models [14, 11]. This method would be a good candidate to extend our approach to stylization.

Rendering. Currently, we produce simple SVG outputs containing only polylines. As our internal representation is a planar map, the resulting SVG file does not contains stacked polygons. In order to extend the vector formats handled by our approach, new operators should be defined. For example, stacked polygons would need ordering operators on top of the planar map. We could also produce other types of vector formats such as diffusion curves [30] by adding color points mappers.

We give here the list of our operators in respective tables: partition operators (Table 1), mapping operators (Table 2), mappers' built-in operators (Table 3), merging operators (Table 4), and other useful functions available anywhere in user scripts (Table 5).

Table 1: **Partition operators**.

Regular partitions	
StripesProperties(Scalar a,Scalar $w1$[,Scalar $w2$,...])	Sets stripes properties
SetEdgeLabels(Properties p, String $l1$[, String $l2$,...])	Adds edges labels to p
SetFaceLabels(Properties p, String $l1$[, String $l2$,...])	Adds faces labels to p
StripesPartition(Properties p)	Creates a stripes partition
GridPartition(Stripes $S1$, Stripes $S2$, Border b)	Creates a grid partition
Irregular partitions	
IrregularProperties(Scalar d)	Sets the partition density
SetWeightedVertexLabels(Properties p, String $l1$, Scalar $w1$[, String $l2$, Scalar $w2$...])	Adds vertices labels to p
SetWeightedEdgeLabels(Properties p, String $l1$, Scalar $w1$[, String $l2$, Scalar $w2$...])	Adds edges labels to p
SetWeightedFaceLabels(Properties p, String $l1$, Scalar $w1$[, String $l2$, Scalar $w2$...])	Adds faces labels to p
UniformPartition(Properties p, Border b)	Creates a uniform partition
RandomPartition(Properties p, Border b)	Creates a random partition

Table 2: **Mapping operators**.

MapToVertices(Mapper m, Arrangement A)	Applies m to all vertices of A
MapToEdges(Mapper m, Arrangement A)	Applies m to all edges of A
MapToFaces(Mapper m, Arrangement A)	Applies m to all edges of A

Table 3: **Mappers built-in operators**.

Incidence	
IncidentFaces(Vertex v)	Faces connected to v
IncidentEdges(Vertex | Face c)	Edges connected to c
IncidentVertices(Face f)	Vertices connected to f
SourceVertex(Edge e)	Source vertex connected to e
TargetVertex(Edge e)	Target vertex connected to e
LeftFace(Edge e)	Left face connected to e
RightFace(Edge e)	Right face connected to e
Adjacency	
MatchPoint(Curves c, Point s, Point t)	Translates curves in the direction $t - s$
MatchPoints(Curves c, Point $s1$, Point $s2$, Point $t1$, Point $t2$)	Applies the rigid transformation $(s1, s2) \rightarrow (t1, t2)$ to c
MatchFace(Curves c, Face f)	Scales and Translates c in f
Geometry	
Location(Vertex v)	Position of vertex v
LocationAt(Edge e, Scalar s)	Position on e, according to $s \in [0, 1]$
Centroid(Face f)	Centroid position of face f
Contour(Face f)	Boundary of face f
Append(Curves $c1$, Curves $c2$)	Appends $c2$ to $c1$ and returns the new set
ToCurve(Edge e)	Transforms edge e into a curve
Labels	
HasLabel(Cell | Cells c,String l)	Tests if cell(s) c contain the label l
IsBoundary(Cell c)	Tests if c is adjacent to the unbounded face
PointLabeled(Curves c,String l)	Returns the location in c labelled by l
CurveLabeled(Curves c,String l)	Returns the curve c labelled by l
Random values	
Random(Scalar min,Scalar max)	Random value $\in [min, max]$
Random(Cell c,Scalar min, Scalar max,Scalar n)	Deterministic random value. This function always returns the same value for a given cell c and scalar n

Table 4: **Merging operators**.

Union(Arrangement $A1$, Arrangement $A2$)	All the curves from $A1$ and $A2$
Inside(Arrangement $A1$, Arrangement $A2$, Border b)	Edges of $A1$ inside $A2$'s faces
Outside(Arrangement $A1$, Arrangement $A2$, Border b)	Edges of $A1$ outside $A2$'s faces

Table 5: **Useful functions available in our scripts**

ImportSVG(String *filename*)	Loads curves from the given SVG file
ExportSVG(Arrangement A, Scalar *size*)	Exports A in SVG
BBoxWidth(Cell | Curves c)	Bounding box width of an element c
BBoxHeight(Cell | Curves c)	Bounding box height of an element c
BBoxCenter(Cell | Curves c)	Bounding box center of an element c
Scale(Curves c,Scalar s)	Scales c by a factor s
Rotate(Curves c,Scalar s)	Rotates c by a factor $s \in [0, 2\pi]$
Translate(Curves c,Vector v)	Translates c in the direction v
Nothing()	Returns an empty set of curves

References

[1] Gregory D. Abram and Turner Whitted. Building block shaders. *Computer Graphics (Proceedings of SIGGRAPH '90)*, 24(4):283–288, September 1990.

[2] Zainab AlMeraj, Craig S. Kaplan, and Paul Asente. Towards effective evaluation of geo-

metric texture synthesis algorithms. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, NPAR '13, pages 5–14, New York, NY, USA, 2013. ACM.

[3] Paul Asente, Mike Schuster, and Teri Pettit. Dynamic planar map illustration. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3), July 2007.

[4] P. Baudelaire and M. Gangnet. Planar maps: An interaction paradigm for graphic design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '89, pages 313–318, New York, NY, USA, 1989. ACM.

[5] William V. Baxter and Ken Ichi Anjyo. Latent doodle space. *Computer Graphics Forum (Proceedings of Eurographics 2006)*, 25(3):477–485, 2006.

[6] Neill D. F. Campbell and Jan Kautz. Learning a manifold of fonts. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)*, 33(4):91:1–91:11, July 2014.

[7] Robert L. Cook. Shade trees. *Computer Graphics (Proceedings of SIGGRAPH '84)*, 18(3):223–231, January 1984.

[8] Boris Dalstein, Remi Ronfard, and Michiel Van De Panne. Vector Graphics Complexes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)*, 33(4):133:1–133:12, July 2014.

[9] Stephen DiVerdi. A brush stroke synthesis toolbox. In *Image and Video-Based Artistic Stylisation*, volume 42, pages 23–44. Springer, 2013.

[10] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 3 edition, 2002.

[11] Elmar Eisemann, Holger Winnemöller, John C. Hart, and David Salesin. Stylized vector art from 3d models with region support. *Computer Graphics Forum (proceedings of the Eurographics Symposium on Rendering)*, 27(4):1199–1207, 2008.

[12] Efi Fogel, Dan Halperin, and Ron Wein. *CGAL arrangements and their applications*. Geometry and Computing. Springer, 2012.

[13] Bruno Galerne, Ares Lagae, Sylvain Lefebvre, and George Drettakis. Gabor noise by example. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012)*, 31(4):73:1–73:9, July 2012.

[14] Stéphane Grabli, Emmanuel Turquin, Frédo Durand, and François X. Sillion. Programmable rendering of line drawing from 3d scenes. *ACM Transactions on Graphics*, 29(2):18:1–18:20, April 2010.

[15] Arthur L. Guptill. *Rendering in Pen and Ink: The Classic Book On Pen and Ink Techniques for Artists, Illustrators, Architects, and Designers (Practical Art Books)*. Watson-Guptill, August 1997.

[16] Peter Henderson. Functional geometry. *Higher Order and Symbolic Computation*, 15(4):349–365, December 2002.

[17] Aaron Hertzmann. Fast paint texture. In *Proceedings of the 2nd International Symposium on Non-photorealistic Animation and Rendering*, NPAR '02, pages 91–ff, New York, NY, USA, 2002. ACM.

[18] Stefan Hiller, Heino Hellwig, and Oliver Deussen. Beyond Stippling – Methods for Distributing Objects on the Plane. *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3):515–522, September 2003.

[19] T. Hurtut, P.-E. Landes, J. Thollot, Y. Gousseau, R. Drouillhet, and J.-F. Coeurjolly. Appearance-guided synthesis of element arrangements by example. In *Proceedings of the 7th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '09, pages 51–60. ACM, 2009.

[20] Thomas Hurtut and Pierre-Edouard Landes. Synthesizing structured doodle hybrids. In *SIGGRAPH Asia 2012 Posters*, SA '12, pages 43:1–43:1, New York, NY, USA, 2012. ACM.

[21] Takashi Ijiri, Radomìr Měch, Takeo Igarashi, and Gavin Miller. An example-based procedural system for element arrangement. *Computer Graphics Forum (Proceedings of Eurographics 2008)*, 27(2):429–436, 2008.

[22] Bela Julesz. Textons, the elements of texture perception, and their interactions. *Nature*, 290(5802):91–97, 1981.

[23] Craig S. Kaplan. Curve evolution schemes for parquet deformations. In *Proceedings of Bridges 2010: Mathematics, Music, Art, Architecture, Culture*, pages 95–102. Tessellations Publishing, 2010.

[24] Ares Lagae and Philip Dutré. A procedural object distribution function. *ACM Transactions on Graphics*, 24(4):1442–1461, 2005.

[25] Pierre-Edouard Landes, Bruno Galerne, and Thomas Hurtut. A shape-aware model for discrete texture synthesis. *Computer Graphics Forum (proceedings of the Eurographics Symposium on Rendering)*, 32(4):67–76, 2013.

[26] Aristid Lindenmayer. Mathematical models for cellular interaction in development: Parts i and ii. *Journal of Theoretical Biology*, 18, 1968.

[27] Chongyang Ma, Li-Yi Wei, and Xin Tong. Discrete element textures. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)*, 30(4):62:1–62:10, July 2011.

[28] Radomír Měch and Gavin Miller. The *Deco* framework for interactive procedural modeling. *Journal of Computer Graphics Techniques (JCGT)*, 1(1):43–99, Dec 2012.

[29] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 25(3):614–623, July 2006.

[30] Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. Diffusion curves: a vector representation for smooth-shaded images. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)*, 27(3):92:1–92:8, August 2008.

[31] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[32] Johannes Schmid, Robert W. Sumner, Huw Bowles, and Markus Gross. Programmable motion effects. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)*, 29(4):57:1–57:9, July 2010.

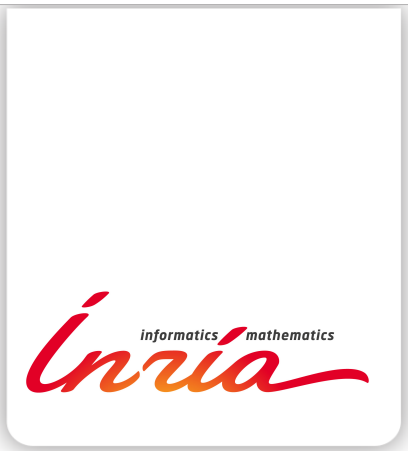
[33] George Stiny, James Gips, George Stiny, and James Gips. Shape grammars and the generative specification of painting and sculpture. In *Proceedings of the Workshop on generalisation and multiple representation, Leicester*, 1971.

[34] A. M. Treisman and G. Gelade. A feature-integration theory of attention. *Cognitive Psychology*, 12:97–136, 1980.

[35] Michael T. Wong, Douglas E. Zongker, and David H. Salesin. Computer-generated floral ornament. In *Proceedings of SIGGRAPH '98*, pages 423–434, New York, NY, USA, 1998. ACM.

[36] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)*, 22(3):669–677, July 2003.

[37] Yi-Ting Yeh, Katherine Breeden, Lingfeng Yang, Matthew Fisher, and Pat Hanrahan. Synthesis of tiled patterns using factor graphs. *ACM Transactions on Graphics*, 32(1):3:1–3:13, February 2013.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399